amira® 3.1

User's Guide and Reference Manual

including amiraDev and amiraVR

Contents

Part I

amira User's Guide

Chapter 1

Introduction

amira is a 3D visualization and modelling system. It allows you to visualize scientific data sets from various application areas, e.g. medicine, biology, chemistry, physics, or engineering. 3D objects can be represented as grids suitable for numerical simulations, notably as triangular surface and volumetric tetrahedral grids. amira provides methods to generate such grids from voxel data representing an image volume, and it includes a general purpose interactive 3D viewer.

Section 1.1 (Overview) provides a short overview of the fundamentals of amira, i.e. its object-oriented design and the concept of data objects and modules.

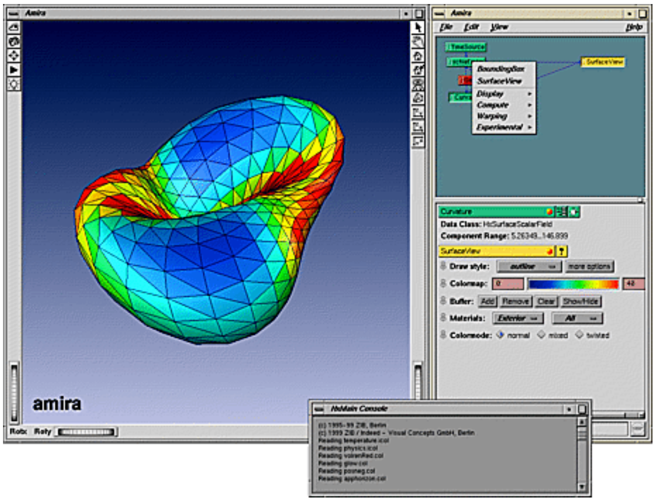
Section 1.2 (Features) summarizes key features of amira, for example direct volume rendering, image processing, and surface simplification.

Section 1.3 (Application Areas) illustrates some typical application areas of amira and shows what kinds of problems can be solved using the system.

Section 1.4 (Extensions) shortly describes optional extension packages available for amira and what they can be used for.

1.1 Overview

amira is a modular and object-oriented software system. Its basic system components are modules and data objects. Modules are used to visualize data objects or to perform some computational operations on them. The components are represented by little icons in the *object pool*. Icons are connected by lines indicating processing dependencies between the components, i.e., which modules are to be applied to which data objects. Data objects of specific types are created automatically from file input data when reading such or as ouput of module computations, modules matching an existing data object are created as instances of particular module types via a context-sensitive popup menu. Networks can be created with a minimal amount of user interaction. Parameters of data objects and modules can be



```
Curvature
Data Class: HxSurfaceScalarField
Component Range: 5.26349...146.899
SurfaceView
Draw style: outline   more options
Colormap: 0   42
Buffer: Add  Remove  Clear  Show/Hide
Materials: Exterior   All
Colormode: ◆ normal  ◇ mixed  ◇ twisted
```

```
(c) 1995-99 ZIB, Berlin
(c) 1999 ZIB / Indeed - Visual Concepts GmbH, Berlin
Reading temperature.icol
Reading physics.icol
Reading volrenRed.col
Reading glow.col
Reading posneg.col
Reading apphorizon.col
```

Figure 1.1: Data objects and modules are represented as little icons in the object pool (top right). In the 3D graphic window a surface colored according to its curvature is shown. Curvature information has been computed by a computational module and is stored as a separate data object. In the mid right window the parameters of selected modules (here: Curvature and SurfaceView) are shown. The window at the bottom provides a Tcl-command shell.

modified in amira's interaction area.

For some data objects such as surfaces or colormaps there exist special-purpose interactive editors that allow the user to modify the objects. All amira components can be controlled via a Tcl command interface. Commands can be read from a script file or issued manually in a separate console window.

The biggest part of the screen is occupied by a 3D graphics window. Additional 3D views can be created if necessary. amira is based on the latest release of the TGS Open Inventor graphics toolkit. In addition, several modules apply direct OpenGL rendering to achieve special rendering effects or to maximize performance. In total, there are more than 120 data object and module types. They allow the system to be used for a broad range of applications. User-defined extensions are facilitated by the amira developer version.

1.2 Features

amira provides a large number of module types allowing you to visualize various kinds of scientific data as well as to create polygonal models from 3D images. All visualization techniques can be arbitrarily combined to produce a single scene. Moreover, multiple data sets can be visualized simultaneously, either in several viewer windows or in a common one. A built-in transformation editor makes it easy to register data sets with respect to each other or to deal with different coordinate systems.

1.2.1 Direct Volume Rendering

One of the most intuitive and most powerful techniques for visualizing 3D image data is *direct volume rendering*. Light emission and light absorption parameters are assigned to each point of the volume. Simulating the transmission of light through the volume makes it possible to display your data from any view direction without constructing intermediate polygonal models. By exploiting modern graphics hardware, amira is able to perform direct volume rendering almost in realtime, even for data volumes of 40 megabytes and more. Volume rendered images can be combined with any type of polygonal display. This improves the usefulness of this technique significantly. Moreover, multiple data sets can be volume rendered simultaneously – a unique feature of amira. Transfer functions with different characteristics required for direct volume rendering can either be generated automatically or edited interactively using an intuitive colormap editor.

1.2.2 Isosurfaces

Isosurfaces are most commonly used for analyzing arbitrary scalar fields sampled on a discrete grid. Applied to 3D images, the method provides a very quick, yet sometimes sufficient method for reconstructing polygonal surface models. Beside standard algorithms, amira provides an improved method which generates significantly fewer triangles with very little computational overhead. In this way large 3D data sets can be displayed interactively even on smaller desktop graphics computers. Like other polygonal models, isosurfaces can be colored in order to visualize a second independent data set. Another highlight comprises the realistic view-dependent way of rendering semi-transparent surfaces. By correlating transparency with local orientation of the surface relative to the viewing direction, complex spatial structures can be understood much more easily.

1.2.3 Segmentation

amira also provides a component for 3D image segmentation with several special-purpose features. This component is called image segmentation editor. It offers a large set of segmentation tools, ranging from purely manual to fully automatic. Among others, the following tools are provided: brush (painting), lasso (contouring), magic wand (region growing), thresholding, intelligent scissors, contour fitting (snakes), contour interpolation and extrapolation, various filters including smoothing, cleaning, and connected component analysis. Although the display is slice-oriented, many tools can be applied in both 2D and 3D. Since the editor does not store contours surrounding regions but region labels, a unique and well-defined classification is guaranteed.

1.2.4 Surface Reconstruction

Once the interesting features in a 3D image volume have been segmented, amira is able to create a corresponding polygonal surface model. The surface may have non-manifold topology if there are locations where three or more regions join. Even in this case the polygonal surface model is guaranteed to be topologically correct, i.e. free of self-intersections. Fractional weights which are automatically generated during segmentation allow the system to produce smooth boundary interfaces. This way realistic high-quality models can be obtained, even if the underlying image data are of low resolution or contain severe noise artifacts. Making use of innovative acceleration techniques, surface reconstruction can be performed very quickly. Moreover, the algorithm is robust and fail-safe.

1.2.5 Surface Simplification

Surface simplification is another prominent feature of amira. It can be used to reduce the number of triangles in an arbitrary surface model according to a user-defined value. Thus, models of finite-element grids, suitable for being processed on low-end machines, can be generated. The underlying simplification algorithm is one of the most elaborate ones available. It is able to preserve topological correctness, i.e., self-intersections commonly produced by other methods are avoided. In addition, the quality of the resulting mesh, according to measures common in finite element analysis, can be controlled. For example, triangles with long edges or triangles with bad aspect ratio can be suppressed.

1.2.6 Generation of Tetrahedral Grids

amira allows you not only to generate surface models from your data but also to create true volumetric tetrahedral grids suitable for advanced 3D finite-element simulations. These grids are constructed using a flexible advancing-front algorithm. Again, special care is taken to obtain meshes of high quality, i.e., tetrahedra with bad aspect ratio are avoided. Several different file formats are supported, so that the grid can be exported to many standard simulation packages. In the developer version additional file formats can easily be added by the user.

1.3 Application Areas

amira is successfully being used in a number of different application areas. Among these are:

- Medicine
- Biology
- Material Sciences
- Computational Fluid Dynamics
- Physics
- Geophysics
- Astrophysics

Examples from these disciplines are illustrated by several demo scripts contained in the online version of the user's guide.

1.4 Extensions

amira extensions are additional sets of modules providing solutions for dedicated application areas. Extensions can be added to a standard amira installation at any time. Usually, for each extension a separate license is required. Currently, the following extensions are available for amira 3.1:

- amiraMol- molecular visualization and biochemical data analysis. Among others, this extension allows you to visualize trajectories (dynamic data), to compute configuration densities, to create molecular surfaces, and to align nuclide or protein sequences. Readers for several different molecular file formats are included.
- amiraDeconv- deconvolution of microscopic images. This extension provides blind and non-blind deconvolution algorithms for enhancing the quality of confocal and widefield 3D image stacks. Several other tools for image preprocessing or computing PSF's from theory or from bead measurements also included.
- amiraVR- virtual reality extension. This extension makes it possible to run amira on a big tiled screen or even in an immersive environment with head-tracking and 3D user interaction. amiraVR supports multi-pipe rendering and multi-threading. 3D floating menus are provided to facilitate user interaction in 3D space.

In addition to these extensions amiraDev is available, an add-on which allows you to develop your own custom modules, file readers, and file writers using the C++ programming language. For more information about amiraDev and amira extensions please refer to the amira web site *www.amiravis.com* or to *www.tgs.com*.

Chapter 2

First steps in amira

This chapter contains step-by-step tutorials illustrating the use of amira. The tutorials are almost independent of each other, so after reading the basics in the Getting Started section it is possible to follow each tutorial without knowing the others. If you go through all tutorials you will get a good survey of amira's basic features. In particular, these topics will be covered:

- Getting started - the basics of amira
- How to load image data - about bounding box and voxel size
- Visualizing 3D images - slices, isosurfaces, volume rendering
- Multi-channel data - grouping multiple 3D images
- Image segmentation - segmentation of 3D image data
- Surface reconstruction - surface reconstruction from 3D images
- Grid generation - creating a tetrahedral grid from a triangular surface
- Warping - how to work with landmark sets
- Slice alignment - how to align physical cross-sections
- 3D image registration - how to register 3D image datasets
- Vector fields - stream lines and other techniques
- Creating animated demonstration - working with the DemoMaker module
- Creating movie files - how to use the MovieMaker module

In all tutorials the steps to be performed by the user are marked by a dot. If you only want to get a quick idea how to work with amira you may skip the explanations between successive steps and just follow the instructions. But in order to get a deeper understanding you should refer to the text.

Note: If you want to visualize your own data, please first refer to Section 4.1. This section contains some general hints on how to import data sets into amira.

2.1 Getting Started

In this section you will learn how to

1. start the program
2. load a demo data set into the system
3. invoke editors for editing the data
4. connect visualization modules to the data
5. interact with the 3D viewer.

The following text has the form of a short step-by-step tutorial. Each step builds on the steps described before. We recommend that you read the text online and carry out the instructions directly on the computer. Instructions are indicated by a dot so you can execute them quickly without reading the explanations between the instructions.

- On a Windows system, select the amira icon from the start menu. On a Unix system, start amira by entering `amira` in a shell window.

If there is no such command, the software has not been properly installed. In this case try to execute the script `bin/start` located in the amira root directory.

When amira is running, windows like those shown in Figure 2.1 appear on the screen. The user interface is divided into four major parts. The 3D viewer window displays visualization results, e.g., slices or isosurfaces. The object pool will contain small icons representing data objects and modules. The working area displays interface elements (*ports*) associated with amira objects. Finally, the console window prints system messages and lets you enter amira commands.

amira provides an integrated hypertext browser. You may use this browser to read the user's guide online. In order to activate the browser, type `help` in the console window or select *User's Guide* from the *Help* menu of amira's main window.

2.1.1 Loading Data

Usually, the first thing you will do after starting amira is to load a data set. Let's see how this can be done:

- Choose *Load ...* from the *File* menu.

After selecting this menu item, the file dialog appears (see Figure 2.2). By default the dialog displays the contents of the first directory defined in the environment variable `AMIRA_DATADIR`. If no such variable exists the contents of amira's demo data directory are displayed. You can quickly switch to other directories, e.g., to the current working directory, using the directory list located in the upper part of the dialog window.

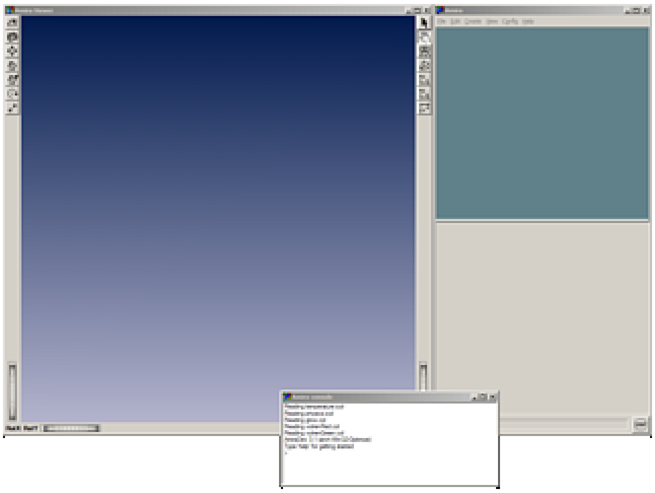


Figure 2.1: The amira user interface consists of four major parts: the 3D viewer (1), the object pool (2), the working area (3), and the console window (4).

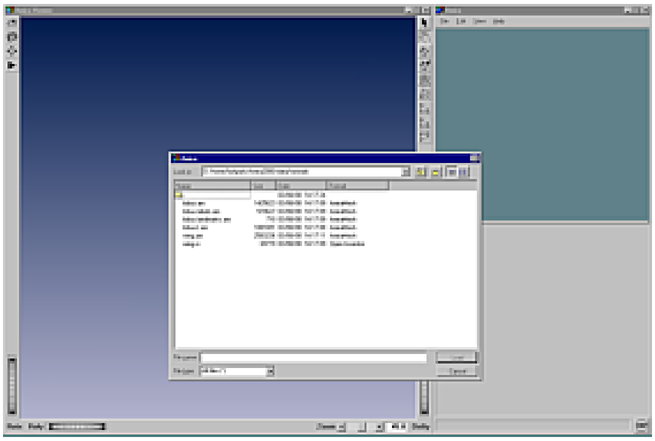


Figure 2.2: Data sets can be loaded into amira using the file browser. In most cases, the file format can be determined automatically. This is done by either analyzing the file header or the file name suffix.

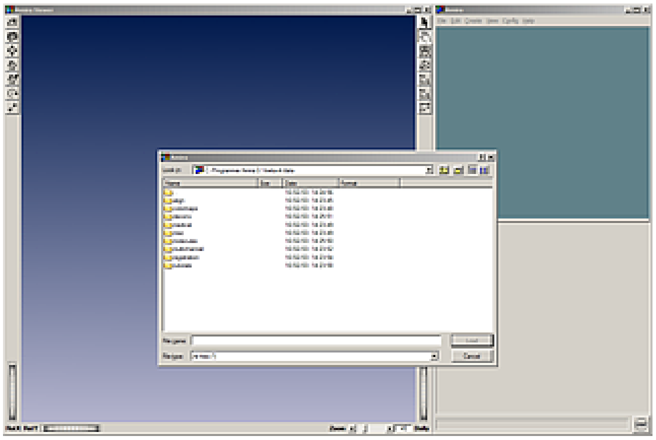


Figure 2.3: Data objects are represented by little green icons in the object pool. Once an icon has been selected information about the data set such as its size or its coordinate type is displayed in the working area.

amira is able to determine many file formats automatically, either by analyzing the file header or the file name suffix. The format of a particular file will be printed in the file dialog right beside the file name.

Now, we would like to load a scalar field from one of the demo data directories contained in the amira distribution.

- Change to the directory `data/tutorials`, select the file `lobus.am` and press *OK*.

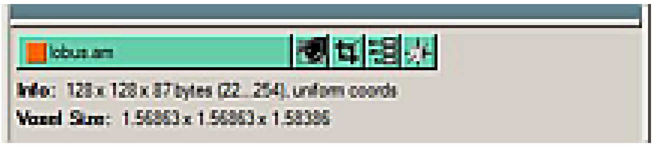
The data will be loaded into the system. Depending on its size this may take a few seconds. The file is stored in amira's native *AmiraMesh* format. The file `lobus.am` contains 3D image data of a part of a fruit fly's brain, namely an optical lobe, obtained by confocal microscopy. This means the data represents a series of parallel 2D image slices across a 3D volume. Once it has been loaded, the data set appears as a little green icon in the object pool. In the following we call this data set "lobus data set".

- Click on the green data icon with the left mouse button to select it.

This causes some information about the data record to be displayed in the working area (Figure 2.3). In our case we can read off the dimensions of the data set, the primitive data type, the coordinate type, as well as the voxel size. To deselect the icon, click on an empty area in the object pool window. You may also pick the icon with the left mouse button and drag it around in the object pool.

2.1.2 Invoking Editors

After selecting an object, in addition to the textual information, some buttons appear in the working area, right next to the data object's name. These buttons represent *editors* which can be used to interactively manipulate the data object in some way. For example, all data objects provide a *parameter editor*. This editor can be used to edit arbitrary attributes associated with the data set, e.g. filename, original size, or bounding box. Another example is the *transform editor* which can be used to translate or rotate the data in world coordinates. However, at this point we don't want to go into details. We just want to learn how to create and delete an editor:



Info: 128 x 128 x 87 bytes (22...254), uniform coords
Voxel Size: 1.56863 x 1.56863 x 1.58386

- Invoke one of the editors by clicking on an editor icon.
- Close the editor by clicking again on the editor icon.

Further information about particular editors is provided in the user's reference manual.

2.1.3 Visualizing Data

Data objects like the lobus data can be visualized by attaching *display modules* to them. Each icon in the object pool provides a popup menu from which matching modules, i.e., modules that can operate on this specific kind of data, can be selected. To activate the popup menu

- click with the right mouse button on the green data icon. Choose the entry called *BoundingBox*.

After you release the mouse button a new *BoundingBox* module is created and is automatically connected to the data object. The *Bounding Box* object is represented by a yellow icon in the object pool and the connection is indicated by a blue line connecting the icons. At the same time, the graphics output generated by the *BoundingBox* module becomes visible in the 3D viewer. Since the output is not very interesting, in this case we will connect a second display module to the data set:

- Choose the entry called *OrthoSlice* from the popup menu of the lobus data set.

Now a 2D slice through the optical lobe is shown in the viewer window. Initially, a slice oriented perpendicular to the z-direction and centered inside the image volume is displayed. Slices are numbered 0, 1, 2, and so on. The slice number as well as the orientation are parameters of the *OrthoSlice* module. In order to change these parameters, you must select the module. Like for the green data icon, this is done by clicking on the *OrthoSlice* icon with the left mouse button. By the way, in contrast to the *BoundingBox*, the *OrthoSlice* icon is orange, indicating that this module can be used for clipping.

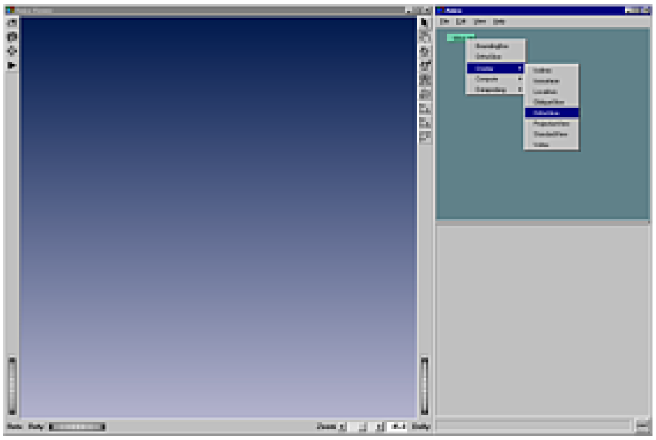


Figure 2.4: In order to attach a module to a data set, click on the green icon using the right mouse button. A popup menu appears containing all modules which can be used to process this particular type of data.

- Select the *OrthoSlice* module.

Now you should see various buttons and sliders in the working area, ordered in rows. Each row represents a *port* allowing you to adjust one particular control parameter. Usually, the name of a port is printed at the beginning of a row. For example, the port labeled *Slice Number* allows you to change the slice number via a slider.

- Select different slices using the *Slice Number* port.

By default, *OrthoSlice* displays slices with axial orientation, i.e., perpendicular to the z-direction. However, the module can also extract slices from the image volume perpendicular to x- and y-direction. These two alternate orientations are referred to as *sagittal* and *coronal* (these are standard phrases used in radiology).

2.1.4 Interaction with the Viewer

The 3D viewer lets you look at the model from different positions. Moving the mouse inside the viewer window with the left mouse button pressed lets you rotate the object. With the middle mouse button you can translate the object. For zooming press both the left and the middle mouse button at the same time and move the mouse up or down.

Notice that the mouse cursor has the shape of a little hand inside the viewer window. This indicates that the viewer is in viewing mode. By pressing the `ESC` key you can switch the viewer into interaction mode. In this mode, interaction with the geometry displayed in the viewer is possible by mouse operations. For example, when using *OrthoSlice* you can change the slice number by clicking on the

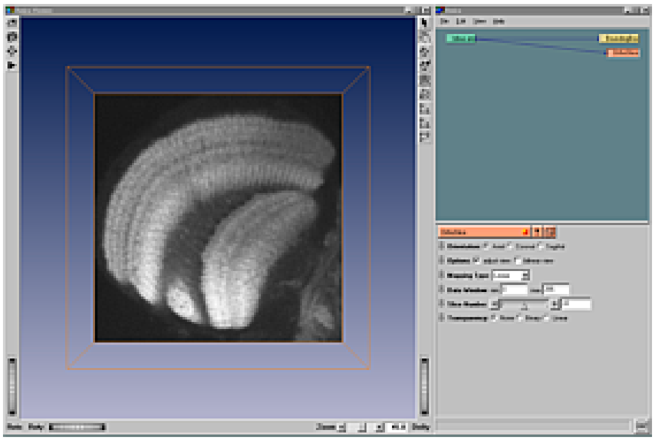


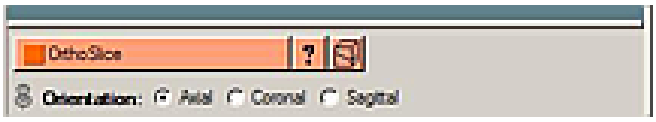
Figure 2.5: Visualization results are displayed in the 3D viewer window. Parameters or ports of a module are displayed in the working area after you select the module.

slice and dragging it.

- Select different buttons of the *Orientation* port of the *OrthoSlice* module.
- Rotate the object in a more general position.
- Disable the *adjust view* toggle in the *Options* port.
- Change the orientation using the *Orientation* port again.
- Choose different slices using the *Slice Number* port or directly in the viewer with the interaction mode described above.

Each display module has a *viewer toggle* by which you can switch off the display without removing the module. This button is attached to the colored bar where the module name is shown as illustrated below.

- Deactivate and activate the display of the *OrthoSlice* or *BoundingBox* module using the *viewer toggle*.



If you want to remove a module permanently, select it and choose *Remove* from the edit menu. Choose *Remove All* from the same menu to remove all modules.

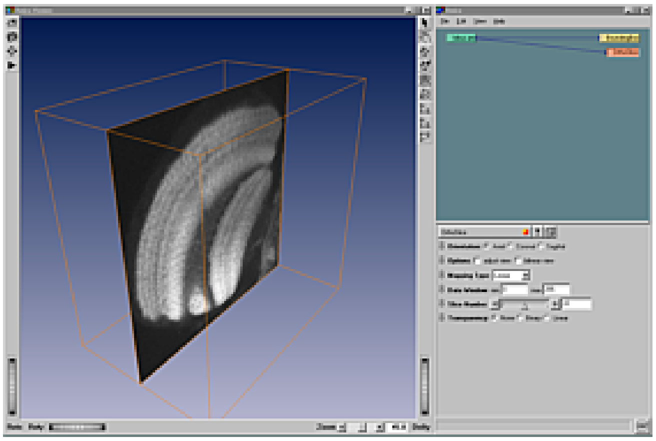


Figure 2.6: The *OrthoSlice* module is able to extract arbitrary orthogonal slices from a regular 3D scalar fi eld or image volume.

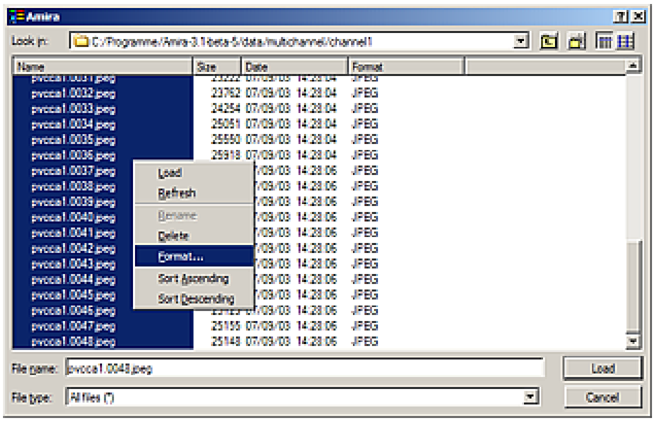
- Remove the *BoundingBox* module by selecting its icon and choosing *Remove* from the *Edit* menu.
- Remove all remaining modules by choosing *Remove All* from the same menu.

Now the object pool should be empty again. You may continue with the next tutorial, i.e., the one on scalar field visualization.

2.2 How to load image data

Loading image data is one of the most basic operations in amira. Other than with 2D images, there are not many standardized file formats containing 3D images. This tutorial guides you by means of examples on how to load the different kinds of 3D images into amira. In particular this tutorial covers the following topics:

1. Using the *File* / *Load* browser and setting the file format.
2. Reading 3D image data from multiple 2D slices.
3. Setting the bounding box or voxel size of 3D images.
4. The *Stacked Slices* file format.
5. Working with LargeDiskData.



Name	Size	Date	Format
pvcca1.0031.jpeg	[illegible]	[illegible]	JPEG
pvcca1.0032.jpeg	23762	07/09/03 14:28:04	JPEG
pvcca1.0033.jpeg	24254	07/09/03 14:28:04	JPEG
pvcca1.0034.jpeg	25051	07/09/03 14:28:04	JPEG
pvcca1.0035.jpeg	25550	07/09/03 14:28:04	JPEG
pvcca1.0036.jpeg	25918	07/09/03 14:28:04	JPEG
pvcca1.0037.jpeg		/09/03 14:28:06	JPEG
pvcca1.0038.jpeg		/09/03 14:28:06	JPEG
pvcca1.0039.jpeg		/09/03 14:28:06	JPEG
pvcca1.0040.jpeg		/09/03 14:28:06	JPEG
pvcca1.0041.jpeg		/09/03 14:28:06	JPEG
pvcca1.0042.jpeg		/09/03 14:28:06	JPEG
pvcca1.0043.jpeg		/09/03 14:28:06	JPEG
pvcca1.0044.jpeg		/09/03 14:28:06	JPEG
pvcca1.0045.jpeg		/09/03 14:28:06	JPEG
pvcca1.0046.jpeg	[illegible]	/09/03 14:28:06	JPEG
pvcca1.0047.jpeg	25155	07/09/03 14:28:06	JPEG
pvcca1.0048.jpeg	25148	07/09/03 14:28:06	JPEG

Figure 2.7: The *Format* option of the fi le browser

2.2.1 The amira file browser

Image data is loaded in amira with the *File/Load* dialog. All file formats supported by amira are recognized automatically either by a data header or by the file name suffix. What follows is only of concern in these cases:

- The automatic file format detection fails.
- 3D image data is stored in several 2D files.
- The data is larger than the available main memory.

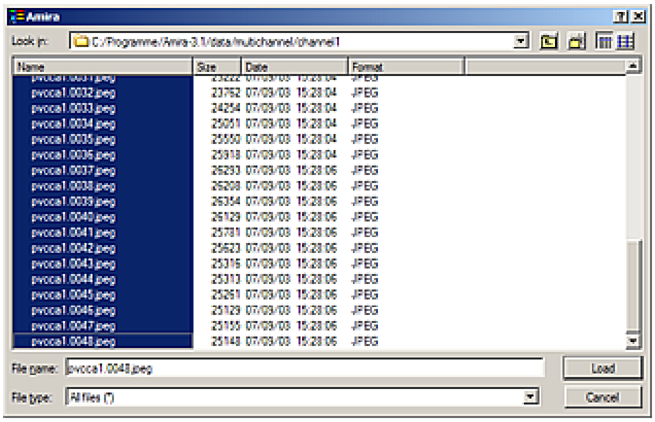
2.2.1.1 Setting the file format

In most cases the format of a file is determined automatically, either by checking the file header or by comparing the file name suffix with a list of known suffixes. In the load dialog the file format is displayed in a separate column in detail view.

Example:

- Files containing the string `AmiraMesh` in the first line are considered AmiraMesh files.
- Files with the suffix .stl are considered STL files.

If automatic file format detection fails, e.g. because some non-standard suffix has been used, the format may be set manually using the *Format* entry in the pop-up menu of the *Load* dialog (right mouse button).



```
Look in: C:/Programme/Amira-3.1/data/multichannel/channel1

Name               Size   Date              Format
pvcca1.0031.jpeg   25222  07/09/03 15:28:04 JPEG
pvcca1.0032.jpeg   23762  07/09/03 15:28:04 JPEG
pvcca1.0033.jpeg   24254  07/09/03 15:28:04 JPEG
pvcca1.0034.jpeg   25051  07/09/03 15:28:04 JPEG
pvcca1.0035.jpeg   25550  07/09/03 15:28:04 JPEG
pvcca1.0036.jpeg   25918  07/09/03 15:28:04 JPEG
pvcca1.0037.jpeg   26293  07/09/03 15:28:06 JPEG
pvcca1.0038.jpeg   26208  07/09/03 15:28:06 JPEG
pvcca1.0039.jpeg   26354  07/09/03 15:28:06 JPEG
pvcca1.0040.jpeg   26129  07/09/03 15:28:06 JPEG
pvcca1.0041.jpeg   25781  07/09/03 15:28:06 JPEG
pvcca1.0042.jpeg   25623  07/09/03 15:28:06 JPEG
pvcca1.0043.jpeg   25316  07/09/03 15:28:06 JPEG
pvcca1.0044.jpeg   25313  07/09/03 15:28:06 JPEG
pvcca1.0045.jpeg   25261  07/09/03 15:28:06 JPEG
pvcca1.0046.jpeg   25129  07/09/03 15:28:06 JPEG
pvcca1.0047.jpeg   25155  07/09/03 15:28:06 JPEG
pvcca1.0048.jpeg   25148  07/09/03 15:28:06 JPEG

File name: pvcca1.0048.jpeg
File type: All files (*)
```

Figure 2.8: Loading multiple 2D images

2.2.2 Reading 3D image data from multiple 2D slices

A common way to store 3D image data is to write a separate 2D image file for each slice. The 2D images may be written in TIFF, BMP, JPEG, or any other supported image file format. In order to load such data in amira, all 2D slices have to be selected simultaneously in the file browser. This can be done by clicking the first file and shift clicking the last one.

- Open the *File* / *Load* dialog.
- Browse to the `/Amira-3.1/data/multichannel/channel/` directory.
- Select the first file `pvcca1.0001.jpeg`
- `Shift`-click the last file (`pvcca1.0048.jpeg`).
- Click *OK*.

2.2.3 Setting the bounding box

When loading a series of bitmap images, usually the physical dimensions of the images are not known to amira. Therefore an *Image Read Parameter* dialog appears that prompts you for entering the physical extent of the so-called *bounding box*. Alternatively, the size of a single voxel can be set. In amira the bounding box of an object is the smallest rectengular, axes aligned volume in 3D space that encompasses the object. *Note that in amira the bounding box of a uniform data set extends from the center of the first voxel to the center of the last one. For example, if you have 256 voxels and you know the voxel size to be 1 mm the bounding box should be set to 0 - 255 (or to some shifted range).*

- Enter 0.85 in the 1st and 2nd text field and 3.5 in 3rd text field of the *Voxel Size* port.
- Click *OK*.

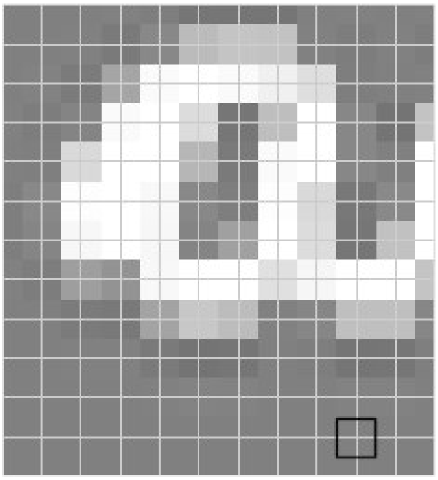


Figure 2.9: The defi nition of the bounding box in amira. Different grey shades depict the intensity values defi ned on the regular grid (white lines). The black square depicts the extent of one voxel. The outer frame depicts the extent of the bounding box.

This method will always create a data set with uniform coordinates, i.e., uniform slice distance. In case of variable slice distances, the so-called StackedSlices format should be used.

2.2.4 The Stacked Slices file format

Especially with histological serial sections it often happens that slices are lost during preparation. To handle such cases, amira provides a special data type corresponding to a file format, called Stacked Slices. This file format allows to read a stack of individual image files with optional z- values for each slice. The slice distance is not required to be constant. The images must be one-channel or RGBA images in an image format supported by amira (e.g. TIFF). The reader operates on an ASCII description file, which can be written with any editor. Here is an example of a description file:

```
# Amira Stacked Slices
# Directory where image files reside
pathname C:/data/pictures
# Pixel size in x- and y-direction
pixelsize 0.1 0.1
# Image list with z-positions
picture1.tif 10.0
picture7.tif 30.0
picture13.tif 60.0
colstars.jpg 330.0
```

```
end
```

Some remarks on the syntax:

- `# Amira Stacked Slices` is an optional header that allows amira to automatically determine the file format.
- `pathname` is optional and can be included if the pictures are not in the same directory as the description file. A space separates the tag "pathname" from the actual pathname.
- `pixelsize` is optional, too. The statement specifies the pixel size in x- and y-direction. The bounding box of the resulting 3D image is set from 0 to pixelsize*(number_of_pixels-1).
- `picture1.tif 10.0` is the name of the slice and its z-value, separated by a space character.
- `end` indicates the end of the description file.
- Comments are indicated by a hash-mark character (#).

2.2.5 Working with Large Disk Data

Sometimes image data are that large that they do not fit into the main memory of the computer. Since the amira visualization modules rely on the fact that data are in physical memory this would mean that such data cannot be displayed in amira. To overcome this, a special purpose module is provided that leaves most of the data on disk and retrieves only a user specified subvolume. This subvolume can then be visualized with the standard visualization modules in amira.

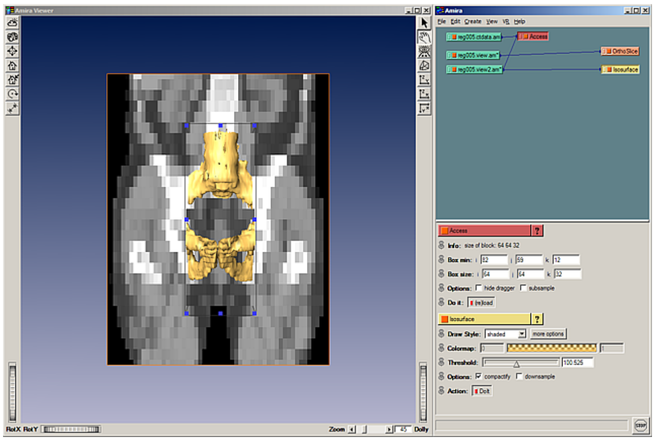
- Use the *Load* dialog and go to c:/Program Files/Amira-3.1/data/medical/
- Right-click on the reg005.ctdata.am and select the *Format* entry from the pop-up dialog
- Select *AmiraMesh as LargeDiskData* as format and confirm your choice with *OK*.
- Press the *Load* button.

The data will be displayed in the object pool as a regular green data icon. The info line indicates that it belongs to the data class *HxStackedSlicesAsExternalData*.

- Right mouse click, attach a *BoundingBox* module.
- Right mouse click, attach an *Access* module.
- Select the *Access* module in the object pool and enter 224, 161, and 59 into the *BoxSize* text fields.
- Check *Subsample* and enter 4 4 2 into the *Subsample* fields and hit the *(re)load* button.

This retrieves a down-sampled version of the data. Disconnect the *reg005.view.am* data icon from the *Access* module and use it as an overview (e.g. with *OrthoSlice*).

- Select the *Access* module in the object pool and deselect the *subsample* check box.



Access

Info: size of block: 64 64 32

Box min: i 82 j 59 k 12

Box size: i 64 j 64 k 32

Options: ☐ hide dragger ☐ subsample

Do it: (re)load

Isosurface

Draw Style: shaded more options

Colormap: 0 1

Threshold: 100.525

Options: ☑ compactify ☐ downsample

Action: DoIt

Figure 2.10: The usage of *AmiraMesh* as *LargeDiskData*. For instantaneous update, the *(re)load* and *DoIt* buttons have been locked

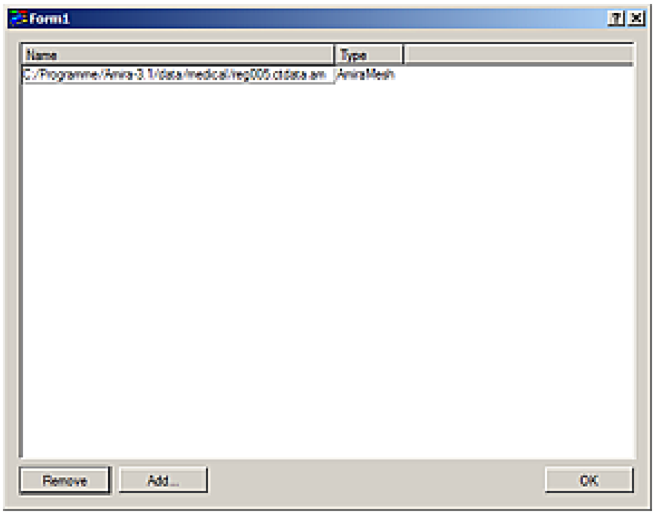
- Use the dragger box in the viewer to resize the subvolume.
- Press the *(re)load* button.
- Attach an *Isosurface* module to the *reg005.view2.am* (set threshold set to 100).

Tip: To browse the data lock the *(re)load* and *DoIt* button of the *Access* and *Isosurface* module, respectively. This is done by clicking the tiny red square in the button. Now each time the blue subvolume dragger is repositioned, the visualization is updated automatically.

Loading *AmiraMesh*, *StackedSlices*, and *Raw* "*asLargeDiskData*" is a convenient and fast way of exploring data that exceed the size of system memory. However, especially with *StackedSlices* it is not always the most efficient way of doing this. amira can store the image data in a special format that facilitates the random retrieval of data from disk.

- Choose from the *Create/Data* menu *ConvertToLargeDiskData*
- Click *Browse* from the *Input* port.
- Go to `/Amira-3.1/data/medical/` and select `reg005.ctdata.am` and click *Add*.
- Click *Browse* from the *Output* port.
- Go to `C:/tmp/` and enter a filename of your choice.
- Click the *DoIt* button.

Although you will most likely not notice any difference with the small image data used in this tutorial, this method for retrieving large data significantly accelerates the *(re)load* operation.



Name	Type
C:/Programme/Amira-3.1/data/medical/reg005.ctdata.am	AmiraMesh

Figure 2.11: The *Input* dialog of the *ConvertToLargeDiskData* module.

2.3 Visualizing 3D Images

This chapter provides a step-by-step introduction to the visualization of regular scalar fields, e.g., 3D image data. amira is able to visualize more complex data sets, such as scalar fields defined on curvilinear or tetrahedral grids. Nevertheless, in this section we consider the simplest case, namely scalar fields with regular structure. Each step builds on the step before. In particular, the following topics will be discussed:

1. orthogonal slices
2. simple threshold segmentation
3. resampling the data
4. displaying an isosurface
5. cropping the data
6. volume rendering

We start by loading the data you already know from Section 2.1 (Getting Started): a 3D image data set of a part of a fruit fly's brain. The data set has been recorded with a confocal laser scanning microscope at the University of Wuerzburg.

- Load the file `lobus.am` located in subdirectory `data/tutorials`.

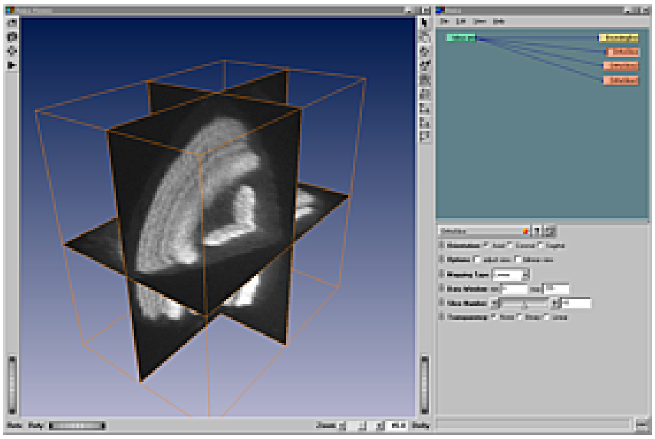


Figure 2.12: Lobus data set visualized using three orthogonal slices.

2.3.1 Orthogonal Slices

The fastest and in many cases most "standard" way of visualizing 3D image data is by extracting orthogonal slices from the 3D data set. amira allows you to display multiple slices with different orientations simultaneously within a single viewer.

- Connect a *BoundingBox* module to the data (use right mouse on lobus.am).
- Connect an *OrthoSlice* module to the data.
- Connect a second and third *OrthoSlice* module to the data.
- Select *OrthoSlice2* and press *coronal* in the orientation port.
- Similarly, for *OrthoSlice3* choose *sagittal* orientation.
- Rotate the object in the viewer to a more general position.
- Change the slice numbers of the three *OrthoSlice* modules in the respective ports or directly in the viewer as described in section Getting Started.

In addition to the *OrthoSlice* module, which allows you to extract slices orthogonal to the coordinate axes, amira also provides a module for slicing in arbitrary orientations. This more general module is called *ObliqueSlice*. You might want to try it by selecting it from the lobus data popup menu.

2.3.2 Simple Data Analysis

The values of the *data window* port of the *OrthoSlice* module determine which scalar values are mapped to black or white, respectively. If you choose a range of e.g., 30...100, any value smaller or equal to 30 will become black, and all pixels with an associated value of more then 100 will become

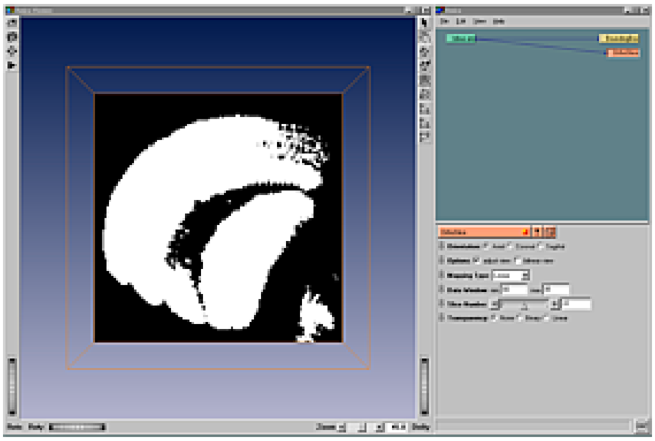


Figure 2.13: By adjusting the data window of the *OrthoSlice* module a suitable value for threshold segmentation can be found. Intensity values smaller than the min value will be mapped to black, intensity values bigger than the max value will be mapped to white.

white. Try modifying the range. This port provides a simple way of determining a threshold, which later can be used for segmentation, e.g., in biology or medicine to separate background pixels from anatomical structures. This can be most easily done by making the minimum and maximum values coincide.

- Remove two of the *OrthoSlice* modules.
- Select the remaining *OrthoSlice* module.
- Make sure that the *mapping type* is set to *linear*.
- Change the minimum and maximum values of the data window port until these values are the same and a suitable segmentation result is obtained. For this data set 85 should be a good threshold value.

A more powerful way of quantitatively examining intensity values of a data set is to use a data probing module *PointProbe* or *LineProbe*. However, we will not discuss these modules in this introductory tutorial.

2.3.3 Resampling the Data

Now we are going to compute and display an *iso-surface*. Before doing so, we will resample the data. The resampling process will produce a data set with a coarser resolution. Although this is not necessary for the iso-surface tool to work, it decreases computation time and improves rendering performance. In addition, you will get acquainted with another type of module. The *Resample* module is a computational module. Computational modules are represented by red icons and typically have a *DoIt* button to start the computation. After you press this button they produce a new data object containing the result.

- Connect a *Resample* module to the data and select it.
- Enter values for a coarser resolution, e.g., x=64, y=64, z=43.
- Press the *DoIt* button.

A new green data icon representing the output of the resample computation named *lobus.Resampled* is created. You can treat this new data set like the original lobus data. In the popup menu of the resampled lobus you will find exactly the same attachable modules and you can save and load it like the original data.

You may want to compare the resampled data set with the original one using the *OrthoSlice* module. You can simply pick the blue line indicating the data connection and drag it to a different data source. Whenever the mouse pointer is over a valid source, the connection line appears highlighted in yellow.

2.3.4 Displaying an Isosurface

For 3D image data sets, isosurfaces are useful for providing an impression of the 3D shape of an object. An isosurface encloses all parts of a volume that are brighter than some user-defined threshold.

- Turn off the viewer toggle of the *OrthoSlice* module.
- Connect an *Isosurface* module to the resampled data record and select it.
- Adjust the threshold port to 85 or a similar value.
- Press the *Do It* button.

2.3.5 Cropping the Data

Cropping the data is useful if you are interested in only a part of the field. A crop editor is provided for this purpose. Its use is described below:

- Remove the resampled data *lobus.Resampled*.
- Activate the display of the *OrthoSlice* module.
- Select the *lobus.am* data icon.
- Click on the green cropping icon in the working area (🟩).

A new window pops up. There are two ways to crop the data set. You can either type the desired ranges of x, y, and z coordinates into the crop editor's window or put the viewer into interaction mode and adjust the crop box using the green handles directly in the viewer window.

- Put the viewer into interaction mode.

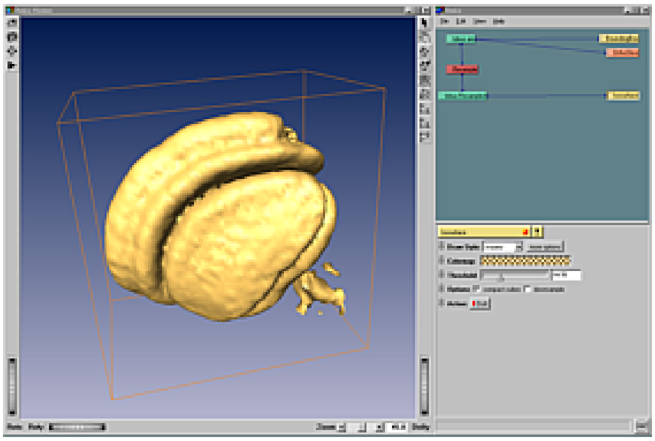


Figure 2.14: Lobus data set visualized in 3D using an isosurface.

- With the left mouse button, pick one of the green handles attached to the crop volume. Drag and transform the volume until the part of the data you are interested in is included.
- Press *OK* in the crop editor's dialog window.

The new dimensions of the data set are given in the working area. If you want to work with this cropped data record in later sessions you should save it by choosing *Save Data As ...* from the *File* menu.

As you already might have noticed, the crop editor also allows you to rescale the bounding box of the data set. By changing the bounding box alone, no voxels will be cropped. You may also use the crop editor to enlarge the data set, e.g., by entering a negative value for the *k min* number. In this case the first slice of the data set will be duplicated as many times as necessary. Also, the bounding box will be updated automatically.

2.3.6 Volume Rendering

Volume Rendering is a visualization technique that gives a 3D impression of the whole data set without segmentation. The underlying model is based on the emission and absorption of light that pertains to every voxel of the view volume. The algorithm simulates the casting of light rays through the volume from pre-set sources. It determines how much light reaches each voxel on the ray and is emitted or absorbed by the voxel. Then it computes what can be seen from the current viewing point as implied by the current placement of the volume relative to the viewing plane, simulating the casting of sight rays through the volume from the viewing point.

You can choose between two different methods for these computations: *maximum intensity* projections or an ordinary emission-absorption model.

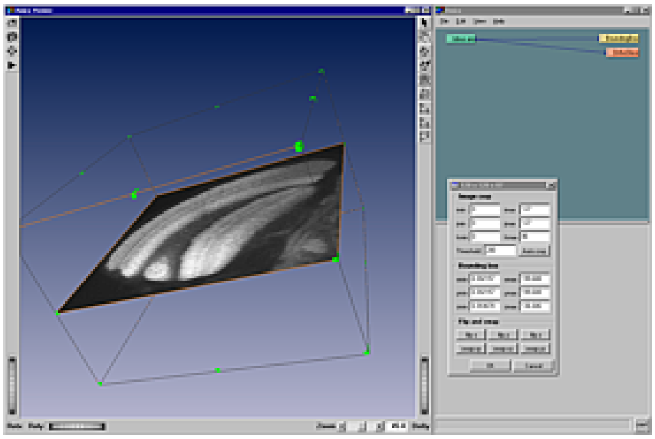


Figure 2.15: The crop editor works on uniform scalar fi elds. It allows you to crop a data set, to enlarge it by replicating boundary voxels, or to modify its coordinates, i.e. to scale or shift its bounding box.

- Remove all objects in the Object Pool other than the *lobus.am* data record.
- Connect a *Voltex* module to the data.
- Select the data icon and read off the range of data values printed on the first info line (22...254).
- Select the *Voltex* module and enter the range in the *Colormap* port.
- Click the *DoIt* button in order to perform some texture preprocessing which is necessary for visualizing the data.

By default emission-absorption volume rendering is shown. The amount of light being emitted and absorbed by a voxel is taken from the color and alpha values of the colormap connected to the *Voltex* module. In our example the colormap is less opaque for smaller values. You may try to set the lower bound of the colormap to 40 or 60 in order to get a better feeling for the influence of the *transfer function*. In order to compute *maximum intensity* projections, choose the *mip* option of port *Mode*.

Internally, the voltex module makes heavy use of OpenGL texture mapping. Both textures modes, 2D and 3D, are implemented. 3D textures yield slightly better results. However, this mode is not supported by all graphics boards. The 3D texture mode requires you to adjust the number of slices cut through the image volume. The higher this number the better the results are.

Alternatively, 2D textures can be used for volume rendering. In this case slices perpendicular to the major axes are used. You may observe how the slice orientation changes if you slowly rotate the data set. The 2D texture mode is well suited for mid-range graphics workstations with hardware accelerated texture mapping. If your computer does not support hardware texture mapping at all, you should use visualization techniques other than volume rendering.

- Set the *em/ab* button of port *Mode*.

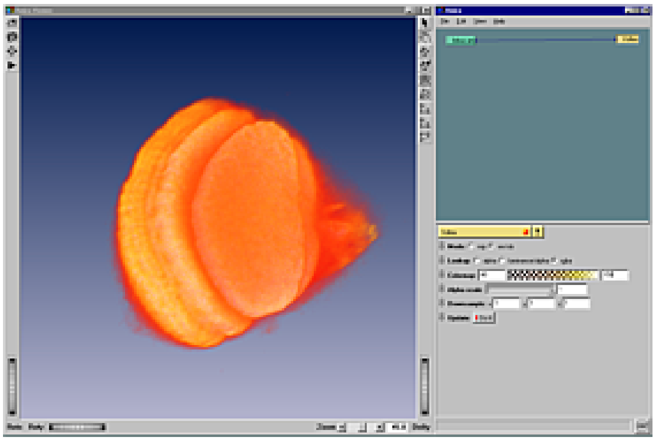


Figure 2.16: The *Voltex* module can be used to generate maxmimum intensity projections as well as volume renderings based on an emission-absoprtion model. In both cases, 2D or 3D texture mapping techniques can be applied applied.

- If you are using 3D texture mode, choose about 200 slices.
- Click with the right mouse button on port *Colormap* and choose *volrenRed.icol*.
- Set *Lookup* to RGBA and change the min and max values of the colormap to 40 and 150.
- Finally, press *Do It* in order to initialize the *Voltex* textures.

Whenever you choose a different colormap or change the min and max values of the colormap, you must press the *Do It* button again. This causes the internal texture maps to be recomputed. An exception are SGI systems with Infinite Reality graphics. On these platforms a hardware-specific OpenGL extension is exploited, causing colormap changes to take effect immediately.

2.4 Working with Multi-Channel Images

This is a step-by-step tutorial on how to visualize multi-channel image data. To follow this tutorial you should be familiar with the basic concepts of amira. In particular you should be able to load files, to interact with the 3D viewer, and to connect display modules to data modules. All these issues are discussed in the getting started section.

We are going to load a set of multi-channel images into the workspace, attach a *MultiChannelField* group object to the data and visualize it with several display modules. The steps are:

1. Load data into amira.
2. Create a *MultiChannelField* and attach channels to it.
3. Using OrthoSlice with a *MultiChannelField*.

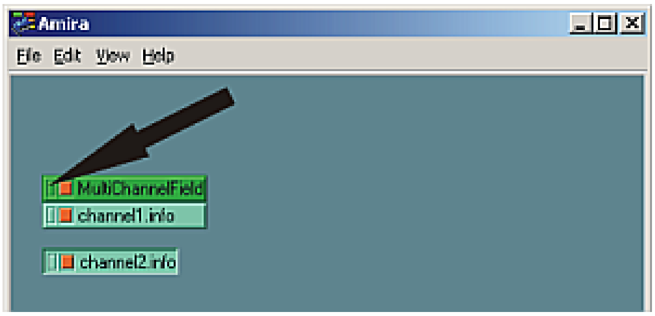


Figure 2.17: Data objects are connected to the *MultiChannelField* object with a right mouse click on the rectangular field indicated by the arrow.

4. Using ProjectionView with a *MultiChannelField*.
5. Using Voltex with a *MultiChannelField*.
6. Saving multi-channel images in a single AmiraMesh file.

2.4.1 Loading Multi-Channel Images into amira

The data we will be working with in this tutorial are confocal stacks of the prothoracic ganglion of the locust *Locusta migratoria*. They were kindly provided by Dr. Paul Stevenson, University of Leipzig, Germany. Two different channels were recorded and stored as separate files.

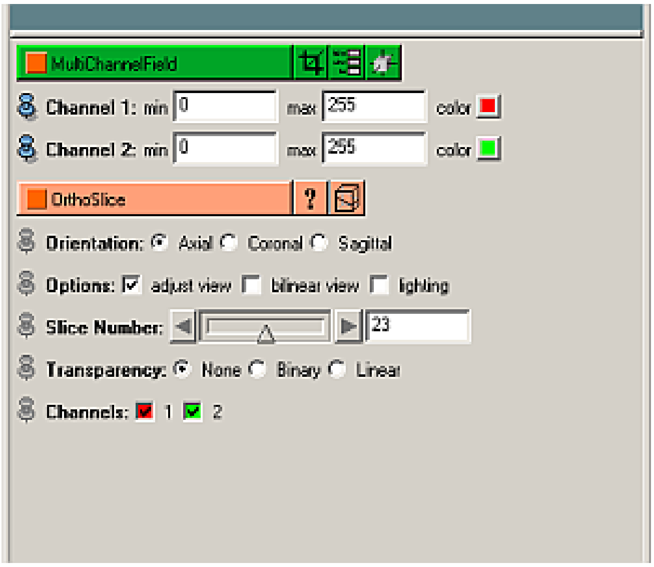
amira supports a number of proprietary multi-channel formats of several microscope manufacturers (e.g., Leica and Zeiss). In such formats all channels are stored in a single file. Therefore the first steps described in this tutorial, namely grouping the channels manually, can often be omitted.

- Load channel 1 data by selecting the file `/data/multichannel/channel1.info`
- Load channel 2 data by selecting the file `/data/multichannel/channel2.info`
- Create a *MultiChannelField* object by selecting *MultiChannelField* from the *Edit Create* menu of the amira main window.

A dark green icon is displayed in the object pool. After selecting the object an info port is displayed saying ”no channels connected”.

- Connect channel1.info to the *MultiChannelField* by selecting ’Channel 1’ from the *MultiChannelField’s* connection menu (right mouse click in the small field on the left side of the icon) and releasing it on the channel1.info icon.
- Repeat the above procedure with channel2.info

When the *MultiChannelField* is selected you will note that two entries, *channel 1* and *channel 2*,


MultiChannelField

Channel 1: min 0 max 255 color 🟥

Channel 2: min 0 max 255 color 🟩

OrthoSlice

Orientation: [✓ Axial] Coronal Sagittal

Options: [✓ adjust view] bilinear view lighting

Slice Number: 23

Transparency: [✓ None] Binary Linear

Channels: ✓ 1 ✓ 2

Figure 2.18: When connected to a *MultiChannelField* object the *OrthoSlice* module has additional check boxes whose number depend on the number of connected channels.

appeared in the module's control panel. Each entry has two range text fields and a color button. The range text fields work very much like those in OrthoSlice. Pressing the color button pops up a color dialog that lets you freely define the color of each channel. Now perhaps it is a good idea to activate the pins corresponding to Channel 1 and Channel 2 in the working area. This will keep the control elements of the *MultiChannelField* module permanent in the working area.

2.4.2 Using OrthoSlice with a MultiChannelField

- Connect an OrthoSlice module to the MultiChannelField by right clicking on the icon and selecting OrthoSlice from the context menu.

When selecting the *OrthoSlice* module you will see that there are two additional check boxes in its control panel corresponding to the two channels. Clicking these check boxes lets you selectively switch on/off each channel. First we adjust the intensity mapping of each channel separately.

- Switch off channel 2 by deselecting its check box.
- Enter 23 and 200 in the min and max range fields of channel 1.

As a result weak stainings - potentially unspecific staining - disappears and those structures that exhibit good staining become even more intense.

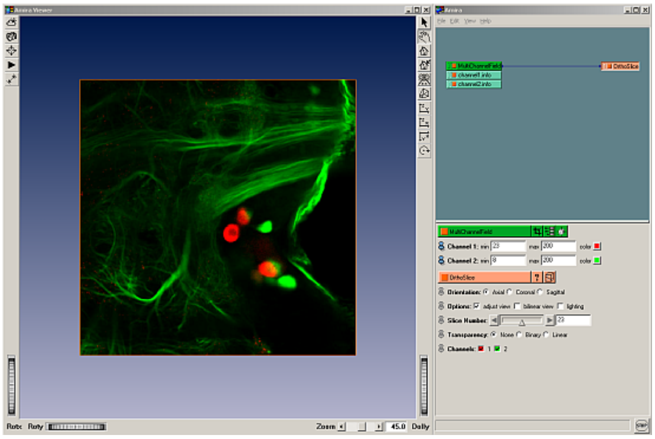


Figure 2.19: Multi channel data visualized using the *OrthoSlice* module.

- Click off channel 1 and click on channel two.
- Enter the values 8 and 200 in the min and max text fields of channel 2. Move through the slices and control the result.

2.4.3 Using ProjectionView with a MultiChannelField

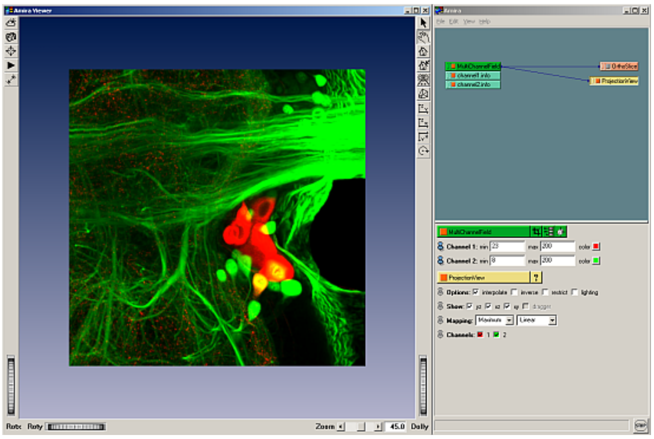
- Switch off the *OrthoSlice* by clicking on the viewer toggle of its icon (orange rectangle).
- Connect a ProjectionView module to the MultiChannelField by right clicking on the icon and selecting ProjectionView from the Display sub-menu.

Similar as with the *OrthoSlice* two new check boxes are shown in the module's control panel which can be used to display channels separately or simultaneously. In this way you may efficiently adjust color and intensity of each channel before diplaying them simultaneously.

2.4.4 Using Voltex with a MultiChannelField

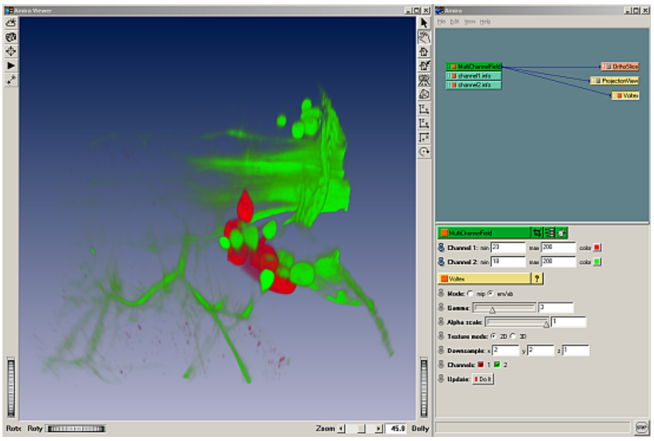
- Switch off the ProjectionView by clicking on the viewer toggle of its icon (orange rectangle).
- Connect a Voltex module to the MultiChannelField by right clicking on the icon and selecting Voltex from the Display sub-menu.

Also here, two channel check boxes are available. In addition to that the familiar colormap field is missing. Instead there is a slider labelled *Gamma*. Now the color of each channel is determined by that defined in the *MultiChannelField* and the *Gamma*-slider controls the steepness of the alpha value (opacity) mapping used for volume rendering. Because volume rendering makes intensive use



MultiChannelField
Channel 1: min 23 max 200 color 🟥
Channel 2: min 8 max 200 color 🟩
ProjectionView
Options: ☑ interpolate ☐ inverse ☐ restrict ☐ lighting
Show: ☑ yz ☑ xz ☑ xy ☐ dragger
Mapping: Maximum Linear
Channels: ☑ 1 ☑ 2

Figure 2.20: Multi channel data visualized using the *ProjectionView* module.



MultiChannelField
Channel 1: min 23 max 200 color 🟥
Channel 2: min 18 max 200 color 🟩
Voltex
Mode: mip [✓ em/ab]
Gamma: 3
Alpha scale: 1
Texture mode: [✓ 2D] 3D
Downsample: x 2 y 2 z 1
Channels: ☑ 1 ☑ 2
Update: Do It

Figure 2.21: Multi channel data visualized using the *Voltex* module.

of hardware texture mapping and most consumer graphics adapter are limited in texture memory size it is recommended to enter at least factors of 2 2 1 in the *Downsample* text fields of the Voltex module.

- Press the *DoIt* button.

Now each time you want to display another channel you have to press the *DoIt* button again.

2.4.5 Saving a MultiChannelField in a Single AmiraMesh File

When the *MultiChannelField* icon is selected in the object pool choose *Save Data As* from the File menu, enter a filename and click OK. The data will be stored in AmiraMesh format so that each time you load the data the two channel stacks and the *MultiChannelField* group object will be restored.

2.5 Segmentation of 3D Images

By following this step-by-step tutorial you will learn how to interactively create a segmentation of a 3D image. A segmentation assigns to each pixel of the image a label describing to which region or material the pixel belongs, e.g., bone or the kidney. The segmentation is stored in a separate data object called a LabelField. A segmentation is the prerequisite for surface model generation or accurate volume measurement.

This tutorial comprises the following steps:

1. Creation of an empty LabelField.
2. Interactive editing of the labels in the Image Segmentation Editor.
3. Measuring the volume of the segmented structures.
4. An alternative segmentation method: Threshold segmentation.

2.5.1 Interactive Image Segmentation

- Load the *lobus.am* data file from the directory *data/tutorials*.
- Right click on the green icon and choose *LabelField* from the Labelling section.

A new green icon appears, the LabelField that will hold the segmentation results. Simultaneously, a new window opens: The image segmentation editor.

- Use the slider on the top, to scroll through the slices. Go to slice 20. You see two bigger structures and one structure just appearing on the top.
- Click on the leftmost button under the label *Tools:*, the brush.
- Mark the rightmost structure with the mouse. Hold down the control button to unselect wrongly selected pixels if necessary.

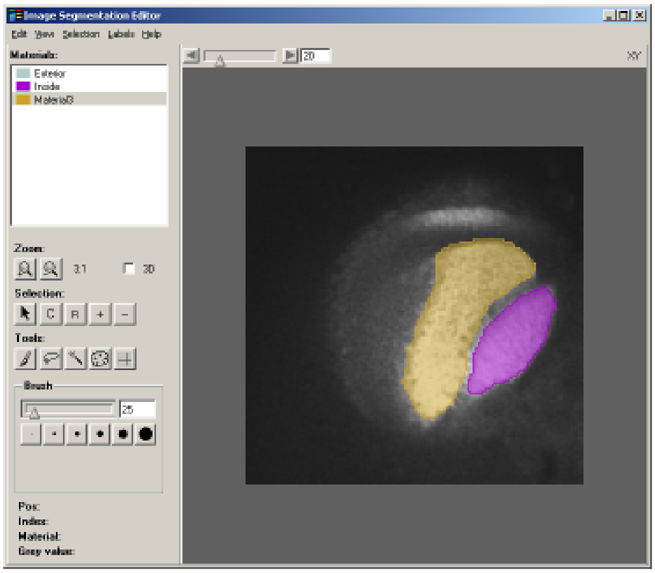


Figure 2.22: Image segmentation editor after selecting and assigning pixels for two structures in one slice

- When done, select the entry *Inside* in the *Materials:* list. Then hit the + button under the *Selection:* label.

The previously selected pixels are now assigned to the material Inside. You can right click on the entry *Inside* in the *Materials:* list and choose a different draw style.

- Click into the material list and choose *New Material* from the right button menu.
- Mark the middle structure using the brush, select the new material in the Materials list and assign the pixels to that structure.
- Go to slice 21 and practice by segmenting the two structures.

If a structure does not change a lot from slice to slice, you can use interpolation.

- Go to slice 22 and mark the right structure using the brush. Go to slice 31 and mark the same structure.
- Choose from the menu bar: Selection/Interpolate.
- Scroll through the data set. You should see that the in between slices 23 to 30 are selected too.
- In order to assign the selected pixels in all slices to the Inside material, select the 3D toggle near the *Zoom* buttons, select the Inside material in the list, and click the + button. Then untoggle the 3D button again.

- Repeat the procedure between slice 32 and 50.
- Repeat the procedure for the middle structure.

Hints:

- It is highly recommended to frequently save the segmentation results while working. In order to do so, select the label field in the amira main window and choose *Save* or *Save As...* from the amira File menu.
- The brush is only the most basic segmentation tool. The segmentation editor provide many more functions, that are described on its reference page.
- There are many useful key bindings, including SPACE and BACKSPACE to change the slice number or 'd' to toggle the draw style.
- Of course you can give the materials more meaningful names or colors using the context menu (right mouse button in the list).

At this point you may want to close the editor by choosing *Close* from the *Edit* menu. Save the label field. In the next tutorial you will learn how to create a 3D surface model from the segmentation results.

2.5.2 Volume Measurement

Once one or multiple structure is segmented, you easily measure its volume:

- Right click on the LabelField's green icon. Choose *Measure/TissueStatistics*.
- Click DoIt. A new icon appears.
- Select this icon and hit show.

The units in the volume column depend on the units you have specified the voxel size. In case of the *lobus.am*, the voxel size is in μm, therefore the volume is in μm^3.

2.5.3 Threshold Segmentation

We now describe an alternative way of segmentation, which can require less manual interaction, but only works for images with good quality.

In some cases a satisfying segmentation can be achieved automatically purely based on the grey values of the image data set.

The first step is to separate the object from the background. This is done by segmenting the volume into exterior and interior regions on the basis of the voxel values.

- Load the *lobus.am* data record from the directory *data/tutorials*.
- Attach a LabelVoxel module to the data icon and select it.

- Type 85 into the text field of port *Exterior-Inside*. You may also determine some other threshold that separates exterior and interior as described in the tutorial on Image Data Visualization.
- Press the *DoIt* button of port *Action*.

By this procedure each voxel having a value lower than the threshold is assigned to *Exterior* and each voxels whose value is greater than or equal to the threshold is assigned to *Interior*. This may, however, cause artifacts that are not part of the object, but have voxel values above the threshold, being assigned to the interior. This can be suppressed by setting the *remove couch* option which assures that only the biggest coherent area will be labeled as the interior and all other voxels are assigned to the exterior.

After pressing the *DoIt* button a new data object is computed and its icon appears in the Object Pool. The data object is denoted *lobus.Labels*. It is of type LabelField, represents a cubic grid with the same dimensions as *lobus.am*, and contains an interior or exterior label for each voxel according to the segmentation result.

2.5.4 Refining Threshold Segmentation Results

You can visualize and manually modify a LabelField by using amira's image segmentation editor. A more detailed description of this tool is contained in the User's Reference guide. Here, we use the image segmentation editor to smooth the data in order to get a nicer looking surface of the object.

- Select the *lobus.Labels* icon and click on the icon in the green title bar in the Working Area that shows a pencil.

In response the image segmentation editor is popped up.

- Change the *slice slider* in the upper part of the editor's window to slice 39.
- Choose a magnification ratio of 4:1 by pushing the zoom-up button in the left part of editor's window.

The image segmentation editor shows the image data to be segmented (*lobus.am*) as well as green contours representing the borders between interior and exterior regions as contained in the *lobus.Labels* data object. As you can see, the borders are not so smooth and there are many little islands, bordered by green contours. This is what we want to improve now.

- Choose *Remove Islands* from the editor's *Label Filter* menu. In response, a little dialog window appears.
- In the dialog window select the *all slices* mode. Then press *Remove* in order to apply the filter in all slices. Note how the segmentation results become less noisy.
- To further clean up the image, choose *Smooth Labels* from the editor's *Label Filter* menu. Another dialog box appears.
- Select the *3D volume* mode and push the *Apply* button in order to execute the smoothing operation.

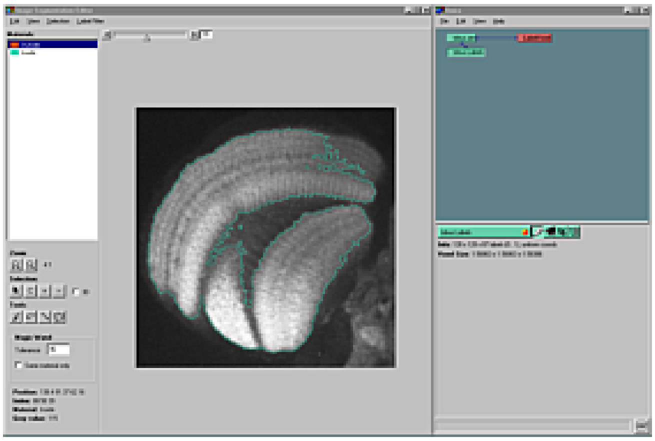


Figure 2.23: Data from confocal microscopy is segmented using amira's image segmentation editor.

- To examine the results of the filter operations, browse through the label field slice by slice. In addition to the slice slider you may also use the cursor-up and cursor-down keys for this.
- Click onto the pencil icon in the Working Area to close the image segmentation editor.

2.6 Surface Reconstruction from 3D Images

By following this step-by-step tutorial, you will learn how to generate a triangular surface grid for an object embedded in a voxel data set. A surface grid allows for producing a 3D view of the object's surface and can be used for numerical simulations.

The generation process consists of these steps:

1. Extracting Surfaces from Segmentation Results
2. Simplifying the Surface

As a prerequisite for the following steps, you need a label field, which holds the result of a previous image segmentation. You can either use the label field which you created in the previous tutorial or load the provided *lobus.labels* data set from the *data/tutoirals* directory.

2.6.1 Extracting Surfaces from Segmentation Results

Now we let amira construct a triangular surface of the segmented object.

- Connect a SurfaceGen module to the *lobus.Labels* data.

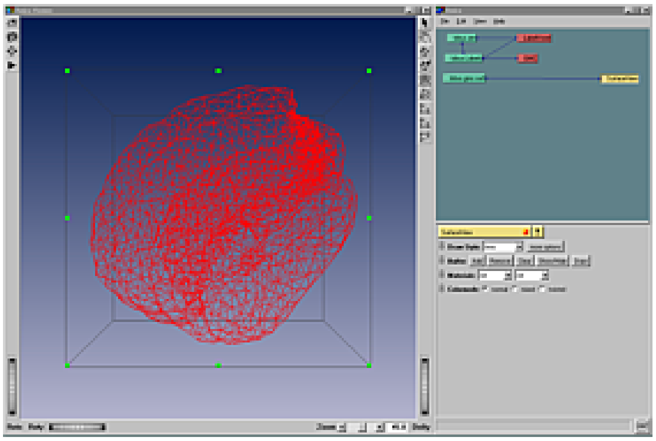


Figure 2.24: Surface representation of optical lobus as triangular grid

- Press *Triangulate* in the *Action* port.

The option *add border* ensures that the created surface be closed. A new data object *lobus.surf* is generated. Again, it is represented by a green icon in the Object Pool.

2.6.2 Simplifying the Surface

Usually the number of triangles created by the *SurfaceGen* module is far too large for subsequent operations. Thus, the number of triangles must be reduced in a *surface simplification* step. In amira a Surface Simplification Editor is provided for this purpose.

- Select the surface *lobus.surf*.
- Click on the triangle mesh icon (first from the right in the title bar) in the Working Area.
- Set the desired number of faces to 3500 in the *Simplify* port.
- Turn on the *fast* toggle in the *Options* port. This option disables some time-consuming intersection tests.
- Push the *Simplify now* button in the *Action* port.

The number of triangles is reduced to about 3500 now. The progess bar tells you how much of the simplification task has already been done.

To examine the simplified surface, attach a SurfaceView module to the *lobus.surf* data object.

The *SurfaceView* module maintains an internal buffer and displays all triangles stored in this buffer. By default the buffer shows all triangles forming the boundary to the exterior. If you change the selection at the *Materials* port, the newly selected triangles are highlighted, i.e., they are displayed using a red wireframe representation. The *Add* and *Remove* buttons cause the highlighted triangles to be added to or removed from the buffer, respectively. You may easily visualize a subset of all triangles using a 3D selection box or by drawing contours in the 3D viewer.

2.7 Creating a Tetrahedral Grid from a Triangular Surface

By following this step-by-step tutorial, you will learn how to generate a volumetric tetrahedral grid from a triangular surface as created in the previous tutorial. A tetrahedral grid is the basis for producing various views of inner parts of the object, e.g., cuts through it, and is frequently used for numerical simulations.

The generation process consists of these steps:

1. Simplifying the Surface
2. Editing the Surface
3. Generating a Tetrahedral Grid

As a prerequisite for the following steps, you need a triangular surface, which is usually the result of a previous surface reconstruction. Load the supplied *lobus.surf* data set from the *data/tutorials* directory.

2.7.1 Simplifying the Surface

Usually the number of triangles created by the *SurfaceGen* module is far too large for subsequent operations, e.g., for a numerical simulation. Thus, the number of triangles should be reduced in a *surface simplification* step. In amira a Surface Simplification Editor is provided for this purpose. There may be different goals for the simplification:

- In *computer craphics*, one wants to prescribe just the number of faces, because this determines the rendering speed.
- For a *numerical simulation*, one often wants to specify the maximum edge length occuring in the grid model.

This tutorial shows how the maximum edge length can be controlled during simplification.

- Select the surface *lobus.surf*.
- Click on the triangle mesh icon (first from the right in the title bar) in the Working Area.
- Set the desired number of faces to 1000 and the desired maximal distance (i.e. edge length) to 10 in the *Simplify* port.
- Leave the *fast* toggle turned off in the *Options* port. This will cause intersection tests to be performed during simplification, which will considerably reduce the probability that the simplified surface contains self intersections.

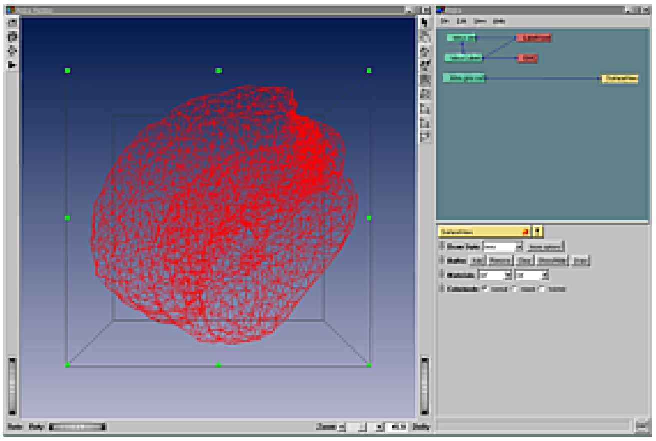


Figure 2.25: Surface representation of optical lobus as triangular grid

- Push the *Simplify now* button in the *Action* port.

Simplification terminates when either of the limits given by the number of faces or the maximum distance is reached. The progress bar tells you how much of the simplification task has already been done. In this example the maximum distance will be the limiting factor, and the resulting surface will contain about 6000 faces.

Besides the maximum edge length, the minimum edge length occuring in the surface should also be controlled, because the ratio of maximum and minimum edge length will influence the quality of the resulting tetrahedral grid. This ratio should not be much larger than 10. If edges that are too short occur in the simplified surface, they can be removed as follows.

- Set the desired minimum distance to 2 in the *Simplify* port.
- Observe the number of faces as shown at the *Surface* port, and press the *Contract edges* button in port *Action*. All edges shorter than 2 will be contracted. In this example about 30 small edges will be detected. You will observe that the number of faces slightly decreases.

2.7.2 Editing the Surface

As a second step of preparation for tetrahedral grid generation, invoke the Surface Editor.

- Select the surface *lobus.surf*.
- Leave the *Surface Simplification Editor* by again clicking on the triangle mesh icon.
- Enter the *Surface Editor* by clicking on the wrench icon (second from the right in the title bar) in the Working Area.

Automatically, a SurfaceView module will be attached to the *lobus.surf* surface. For details about that module see its description.

When the *Surface Editor* is invoked, some menus and additional buttons appear on the top of amira's *Viewer window*. The *Tests* menu contains 5 specific tests which are useful for preparing a tetrahedral grid generation. Each of the tests creates a buffer of triangles which can be cycled through using the back and forward buttons.

- Select *Intersection test* from the *Tests* menu. The total number of intersecting triangles is printed in the console window. Intersections shouldn't occur too often if toggle *fast* was switched off during surface simplification. In case they occur, the first of the intersecting triangles and its neighbors are shown in the viewer window.
- You can manually repair intersections using four basic operations: *Edge Flip*, *Edge Collapse*, *Edge Bisection*, and *Vertex Translation*. See the description of the Surface Editor for details.
- After repairing, invoke the intersection test again by selecting it from the *Tests* menu or by pressing the *Compute* button.
- When the intersection test has been successfully passed, select the *Orientation test* from the *Tests* menu. After surface simplification, the orientation of a small number of triangles may be inconsistent, resulting in a partial overlap of the materials bounded by the triangles. In case of such incorrect orientations, which should occur quite rarely, there is an automatic repair. If this fails, the detected triangles will be shown, and you can use the above mentioned manual operations for repair. **Note:** There are two prerequisites for the orientation test: the surface must be free of intersections, and the outer triangles of the surface must be assigned to material *Exterior*. If the surface does not contain such a material or if the assignment to *Exterior* is not correct, the test will falsely report orientation errors.

A successful pass of the intersection and orientation test is mandatory for tetrahedral grid generation. These tests are automatically performed at the beginning of grid generation. So you can directly enter the *TetraGen* module (see below) and try to create a grid. If one of the tests fails, an error message will be issued in the console window. You can then go back to the *Surface Editor* and start editing.

The remaining three tests analyze the surface mesh with respect to different quality measures. These tests have only to be performed if the tetrahedral quality of the volumetric grid plays an important role, e.g., if the grid will be used for a numerical simulation.

- Select *Aspect ratio* from the *Tests* menu. This computes the ratio of the radii of the circumcircle and the incircle for each triangle. The triangle with the worst (i.e. largest) value is shown first, and the actual value is printed in the console window. The largest aspect ratio should be below 20 (better below 10). Fortunately there is an automatic tool for improving the aspect ratio included in the *Surface Editor*.
- Select *Flip edges* from the *Edit* menu. A small dialog window appears. Set the limit for triangle quality to 10. Select mode *operate on whole surface*. Press button *Flip*. All triangles with an aspect ratio larger than 10 will be inspected; if the aspect ratio can be improved via an edge

flip, this will be done automatically. The console window will tell you the total number of bad triangles and how many of them could be repaired. Press the *Close* button to leave the *Flip edges* tool.

- Select again *Aspect ratio* from the *Tests* menu. Only a small number of triangles with large aspect ratio should remain after applying the *Flip edges* tool.
- Select *Dihedral angle* from the *Tests* menu. For each pair of adjacent triangles, the angle between them at their common edge will be computed. The triangle pair including the worst (i.e. smallest) angle is shown in the viewer, and the actual value is printed in the console window. The smallest dihedral angle should be larger than 5 degrees (better larger than 10).
- For a manual repair of a small dihedral angle proceed as follows: select the third points of both triangles (i.e. the points opposite to the common edge) and move them away from each other. For moving vertices you must enter *Vertex Translation* mode by clicking on the first icon from the right on the top of the viewer window or by pressing the `"t"` key. If the viewer is in viewing mode, switch it into interaction mode by pressing the `ESC` key or by clicking on the arrow icon (the first icon from the top) on the right of the viewer window. Click on the vertex to be moved. At the picked vertex a point dragger will be shown. Pick and translate the dragger for moving the vertex.
- In some cases an edge flip might also improve the situation. Enter *Edge Flip* mode by clicking on the fourth icon from the right on the top of the viewer window or by pressing the `"f"` key. Switch the viewer into interaction mode. Click on the edge to be flipped.
- Select *Tetra quality* from the *Tests* menu. For each surface triangle the aspect ratio of the tetrahedron which would probably be created for that triangle will be calculated. The aspect ratio for a tetrahedron is defined as the ratio of the radii of the circumsphere and the inscribed sphere. The triangle with the worst (i.e. largest) value is shown in the viewer, and the actual value is printed in the console window. The largest tetrahedral aspect ratio should be below 50 (better below 25). If all small dihedral angles have already been repaired, the tetra quality test will mainly detect configurations where the normal distance between two triangles is small compared to their edge lengths. Again, the *vertex translation* and the *edge flip* operation are best suited for a manual repair of large tetrahedron aspect ratios.
- Leave the *Surface Editor* by again clicking on the wrench icon in the Working Area.

2.7.3 Generation of a Tetrahedral Grid

The last step is the generation of a *volumetric tetrahedral grid* from the surface. This means that the volume enclosed by the surface is filled with tetrahedra.

Because the computation of the tetrahedral grid may be time consuming it can be performed as a *batch job*. You can then continue working with amira while the job is running. However, for demonstration purposes we want to compute the grid right inside amira.

- Connect a TetraGen module to the *lobus.surf* surface by choosing *Compute TetraGen* from the popup menu over the *lobus.surf* icon.

- Leave toggle *improve grid* switched on and toggle *save grid* switched off at the *Options* port. The *improve grid* option will invoke an automatic post-processing of the generated grid, which improves tetrahedral quality by some iterations that move inner vertices and flip inner edges and faces. See the description of the Grid Editor for details.

  If toggle *save grid* is selected, an additional port *Grid* appears, where you can enter a filename. The resulting tetrahedral grid will be stored automatically under that name. If you want to run grid generation as a batch job, you must select the *save grid* option.
- Push button *Meshsize* at port *Action*. An editor window will appear. It allows you to define a desired mesh size, i.e., mean length of the inner edges to be created, for each region. For this you must enter the bundle of that region, and select parameter *MeshSize*. Then you can change the value in the text field at the lower border of the editor. There are some predefined region names in amira for which a default mesh size will be automatically set. Make sure that the default values are suitable for your application. If you are not sure about a suitable value, set the desired mesh size to 0. In this case the mean edge length of the surface triangles will be used.
- Push the *Run now* button at port *Action*. A pop-up dialog appears asking you whether you really want to start the grid generation. Click *Continue* in order to proceed.

Once grid generation is running, the progress bar informs you about the number of tetrahedra which already have been created. In some situations grid generation may fail, for example, if the input surface intersects itself. Then an error message will occur at the *Console Window*. In this case go back to the Surface Editor to interactively fix any intersections.

After the tetrahedral grid has been successfully created, a new icon called *lobus.grid* will be put in the Object Pool. You can select this icon in order to see how many tetrahedra the created grid contains. If grid generation takes too long, you may also load the pre-computed grid *lobus.grid* from the *data/tutorials* directory.

As the very last step you may want to have a look at the fruits of your work:

- Attach a GridVolume module to the *lobus.grid*.
- Select the GridVolume icon and push the *Add to* button of the *Buffer* port.

The GridVolume module maintains an internal buffer and displays all tetrahedra stored in this buffer. By default the buffer is empty, but all tetrahedra are highlighted, i.e., they are displayed using a red wireframe representation. The *Add to* button causes the highlighted tetrahedra to be added to the buffer. You may easily visualize a subset of all tetrahedra using a 3D selection box or by drawing contours in the 3D viewer.

Similar to the *Surface Editor*, there is a Grid Editor which can be invoked by selecting the green icon of the tetrahedral grid and clicking on the pencil icon (first from the right in the title bar) in the Working Area. The editor allows for selecting tetrahedra with respect to different quality measures, e.g., aspect ratio, dihedral angles at tetrahedron edges, solid angles at tetrahedron vertices, and edge length. The editor contains several modifiers that can be applied for improving mesh quality.

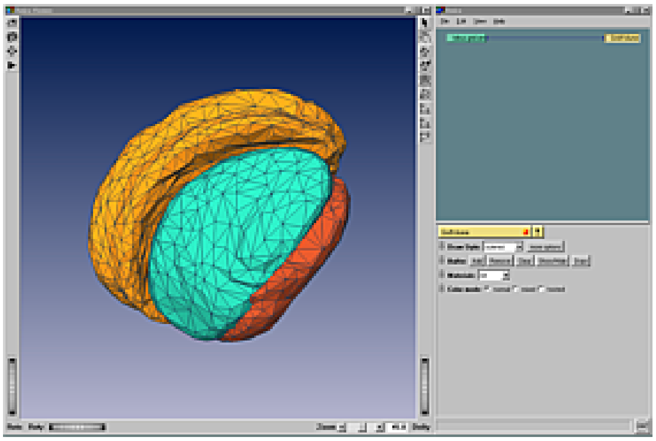


Figure 2.26: Volumetric representation of optical lobe as tetrahedral grid

2.8 Warping and Registration Using Landmarks

This is an advanced tutorial. You should be able to load files, interact with the 3D viewer, and be familiar with the 2-viewer layout and the viewer toggles.

We will transform two 3D-objects into each other by first setting landmarks on their surfaces and then defining a mapping between the landmark sets. As a result we shall see a rigid transformation and a warping which deforms one of the objects to match it with the other. The steps are:

1. Displaying Data Sets in Two Viewers.
2. Creating a Landmark Set.
3. Alignment via a Rigid Transformation.
4. Warping Two Image Volumes.

2.8.1 Displaying Data Sets in Two Viewers

The data we will be working with in this tutorial are of the same kind you have already seen before: Two optical lobes of a drosophila's brain.

- Load the two lobes by executing the script share/examples/landmark.hx.

This script will load two data sets called *lobus.am* and *lobus2.am*. In addition, two isosurface modules connected to each of the data sets will be created. In the viewer the two lobes are visualized by isosurfaces, the first in yellow and the second in blue. As we can see, the lobes are orientated differently. We want to look at each lobe in its own viewer.

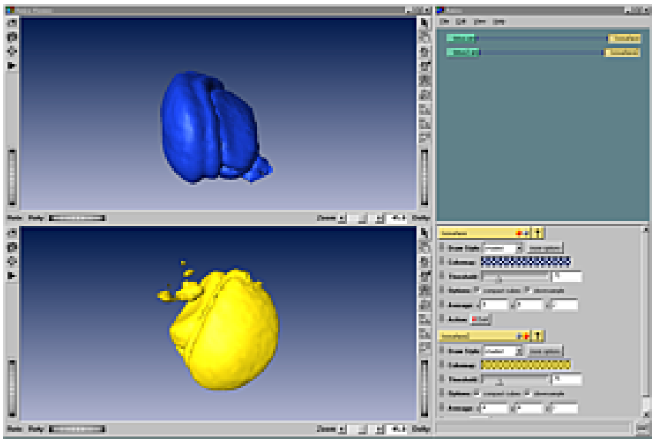


Figure 2.27: Two lobes visualized with isosurfaces in 2-viewer layout.

- Choose *2 Viewers* from the *View Layout* menu.

You can see the two lobes in both viewers.

- Visualize the first lobe (yellow) in the first viewer and the second lobe (blue) in the second viewer by switching off the viewer toggles in the *isosurface* modules.

2.8.2 Creating a Landmark Set

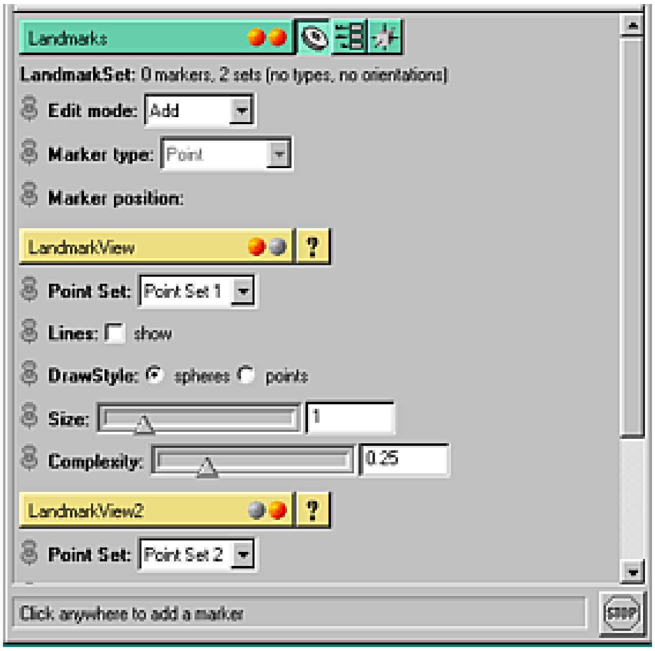
Now, let us create a landmark set object.

- From the main window's *Edit* menu select *Create Landmarks*

in order to create an empty set of landmarks. The new object will show up in the object pool. We are going to match two objects by means of corresponding landmarks, i.e., we actually have to produce two landmark sets. Make this number known to the *Landmarks* object.

- Select object *Landmarks*.
- Type *Landmarks setNumSets 2* in the console window. Instead of manually typing the name of the object you want to send commands to, e.g., *Landmarks*, you may simply press the tab key on the empty command line.
- Start the Landmark Editor by clicking on the button with the disk-shaped icon.

When starting the editor, a *LandmarkView* module is automatically created and connected to the *Landmarks* data object. As indicated on the info line, two empty landmark sets are available now. We use


Landmarks

LandmarkSet: 0 markers, 2 sets (no types, no orientations)

Edit mode: Add

Marker type: Point

Marker position:

LandmarkView

Point Set: Point Set 1

Lines: show

DrawStyle: spheres points

Size: 1

Complexity: 0.25

LandmarkView2

Point Set: Point Set 2

Click anywhere to add a marker

Figure 2.28: The image shows how the viewer toggles and *Point Set* ports should be set.

the editor to define some markers in both objects. For the following, it is useful to pin the three ports on the landmark editor. In order to do so, select the grey pin toggles left from the port's labels.

- Connect a second *LandmarkView* module to the *Landmarks* object.
- Select the first *LandmarkView* module and choose *Point Set*: *Point Set 1*.
- Shift-select the second *LandmarkView* module and choose *Point Set*: *Point Set 2*.
- Adjust the viewer toggles of the two display modules such that the first is visible in the first viewer and the second in the second viewer.

Before starting to set landmarks it is helpful to rotate the two lobes in their viewers such that they are approximately aligned. This will make it easier to locate corresponding features in the two objects and to select reasonable positions for landmarks.

- Rotate the two lobes to align them roughly.

Now, we are ready to define and set corresponding landmarks. Select the *LandmarkSet* object if necessary and choose the option

Edit mode: *Add*.

To set corresponding landmarks simply click with the left mouse button anywhere on the surface in the first viewer first and click on the surface in the second viewer subsequently. The landmarks are visualized as small spheres, the first landmark in yellow and the corresponding landmark in blue. Make sure to always set the first landmark on the first (yellow) surface and the second landmark on the second (blue) surface!!

If you want to change the position of an existing landmark set

> *Edit mode*: *Move*

and select the respective landmark (blue or yellow) by clicking on it with the left mouse button. Then just click at the desired position.

You can also delete existing landmarks by setting

> *Edit mode*: *Remove*

and clicking on the respective landmark. Both corresponding landmarks (blue and yellow) will be removed, no matter which one was selected.

You should now be able to create several landmarks. You may want to change the view of the objects to set landmarks on the back. In case you have problems to define landmarks you may use an existing set of landmarks by loading the file `landmarkSet` from the directory `data/tutorials`. Once landmarks have been created, the next step is to transform the two objects into each other.

2.8.3 Registration via a Rigid Transformation

To register one object to the other connect a *LandmarkWarp* module to the *LandmarkSet* object by clicking with the right mouse button on the *LandmarkSet* icon in the object pool and selecting *Compute LandmarkWarp*.

We want to perform an alignment of the first lobe to the second. Therefore the *LandmarkWarp* module must be connected to the image data of the first lobe (use the right mouse button in the tiny rectangle of the *LandmarkWarp* icon).

The *LandmarkWarp* module is able to perform several transformations. We start with a purely rigid transformation to match the corresponding landmarks as good as possible by perfoming only rotations and translations of the first object. To do that choose

> *Method*: *Rigid*

in the *LandmarkWarp* module and make sure that *Direction* is set to

> *Direction*: $1 \rightarrow 2$.

Then press *DoIt* to start the computation. The module creates a new data object named *lobus.Warped*. To visualize the result, connect the isosurface that was initially connected to the data of the first lobe to the result, select it and press the *DoIt* button. In order to compare the result with the second lobe, adjust the viewer toggle of its *Isosurface* module to display it in the first viewer. You should see that the result of the transformation fits the second object quite well.

2.8.4 Warping Two Image Volumes

Using the rigid transformation the object will not be deformed. To perform a deformation and obtain a better fit we can use another transformation method of the *LandmarkWarp* module. Select the latter and choose

Method: *Bookstein* and press the *DoIt* button.

To visualize the result, the isosurface has to be recomputed. Having done that you can see both the deformed and the second lobe in the first viewer. To merely see the resulting deformation in the first viewer, switch off the viewer toggle of the second lobe's *Isosurface* module. Only a little deformation will be seen because the two original objects were rather similar. Using more different data sets results in larger deformations.

We hope you have had some fun with our tutorials, got to know the basic features of amira and learned to use them. For details and more information about other features see chapter 3, the Program Description.

2.9 Alignment of 2D Physical Cross-Sections

Many microscopic techniques require the sample to be physically cut into slices. Then images are taken from each cross-section separately. Usually the images will be misaligned relative to each other. Before a 3-dimensional model of the sample can be reconstructed the images have to be aligned taking into account translation and rotation. This tutorial shows how this task can be performed using the amira module *Align Slices*.

The following issues will be discussed:

1. Basic manual alignment
2. Alignment via landmarks
3. Optimizing the quality function
4. Resampling the input data
5. Other alignment options

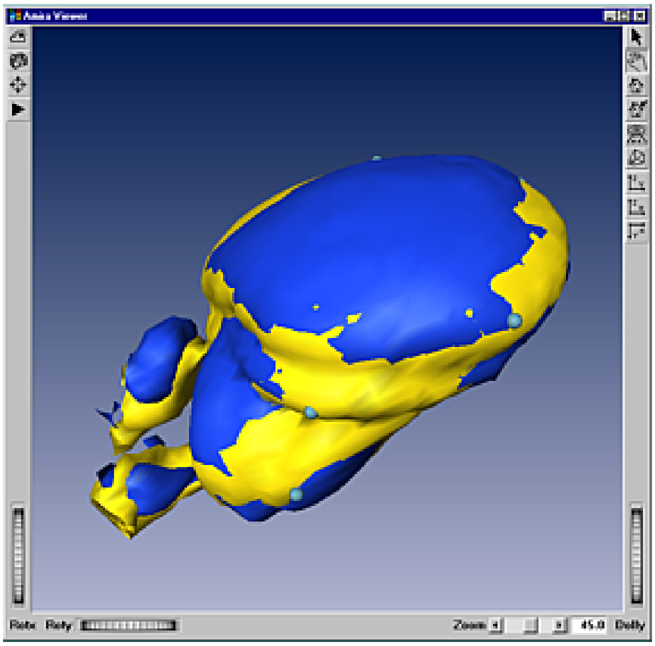


Figure 2.29: Result of landmark-based elastic warping using the Bookstein method.

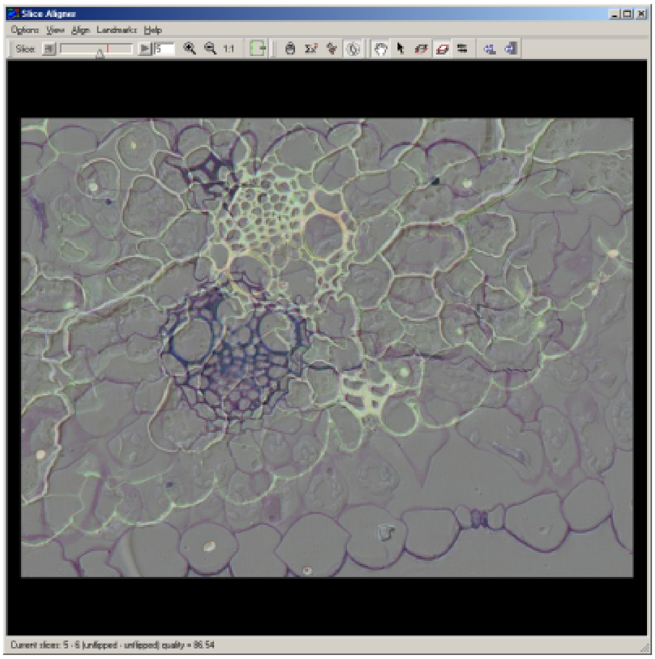
2.9.1 Basic Manual Alignment

In this tutorial we want to align 10 microscopic cross-sections of a leaf showing a stomatal pore. The images are located in the amira data directory in the subdirectory *align*. Each slice is stored as a separate JPEG image. The file *leaf.info* defines a 3D image stack consisting of the 10 individual slices. It is a simple ASCII file as described in the stacked slices file format section.

- Load the file `data/align/leaf.info` into amira.
- Create an align module by choosing *Compute AlignSlices* from the popup menu over the *leaf.info* icon.
- Press the *Edit* button of *AlignSlices*.

A new graphics window is popped up allowing you to interactively align the slices of the 3D image stack. To facilitate this task usually two consecutive slices are displayed simultaneously. One of the two slices is *editable*, i.e., it can be translated and rotated using the mouse. On default the upper slice is editable. This is indicated in the tool bar of the align window (the “upper slice” button is selected).

- Translate the upper slice by moving the mouse with the left mouse button pressed down.
- Rotate the upper slice by moving the mouse with the left mouse button and the Ctrl key pressed



```
Current slices: 5 - 6 (unflipped - unflipped) quality = 86.54
```

Figure 2.30: User-interface of the align tool.

down. Alternatively, slices can be rotated using the middle mouse button.

- Make the lower slice editable by selecting the “lower slice” tool button. Translate and rotate the lower slice.
- Hold down key number 1. While this key is hold down only the lower slice is displayed.
- Hold down key number 2. While this key is hold down only the upper slice is displayed.
- Pressing key number 1 and 2 also changes the editable slice. Note, how the slice tool buttons change their state.

Other pairs of slices can be selected using the slider in the upper left part of the align window. Note, that the number displayed in the text field at the right side of the slider always refers to the editable slice. The next or the previous pair of slices can also be selected using the space bar or using the backspace key, respectively. The cursors keys are used to translate the current slice by one pixel in each direction.

- Browse through all slices using the space bar and the backspace key. Translate and rotate some slices in an arbitrary way.
- Translate all slices at once by moving the mouse with the left mouse button and the Shift key pressed down.
- Rotate all slices at once by moving the mouse with the left mouse button and the Shift and Ctrl key pressed down.

Transforming all slices at once can be useful in order to move the region of interest into the center of the image.

2.9.2 Alignment Via Landmarks

Besides manual alignment four automatic alignment options are supported, namely alignment using a principal axes transformation, automatic optimization of a quality function, edge detection-based alignment and alignment via user-defined landmarks. The principal axes method and the edge detection method is only suitable for images showing an object which clearly separates from the background. The optimization method requires that the images are already roughly aligned. Often such a pre-alignment can be achieved using the landmarks method.

Alignment via landmarks first requires to interactively define the positions of the landmarks. This can be done in *landmark edit* mode.

- Activate *landmark edit* mode by pressing the arrow-shape tool button located between the hand-shape button and the lower slice button.

In *landmark edit* mode only one slice is displayed instead of two. Two default landmarks are defined in every slice.

- Click on one of the default landmarks. The landmark gets selected and is drawn with a red border.
- Click somewhere into the image in order to reposition the selected landmark.
- Click somewhere into the image while no landmark is selected. This causes the next landmark to be selected automatically.
- Click at the same position again in order to reposition the next landmark.

The double click method makes it very easy to define landmark positions. Of course, additional landmarks can be defined as well. Landmarks can also be deleted, but the minimum number of landmarks is two.

- Choose *Add* from the *Landmarks* menu.
- Click anywhere into the image in order to define the position of the new landmark.
- Select the yellow landmark by clicking on it.
- Choose *Remove* from the *Landmarks* menu in order to delete the selected landmark again.

Two landmarks should be visible now, a red one, and a blue one. Next, let us move theses landmarks to some reasonable positions so that we can perform an alignment.

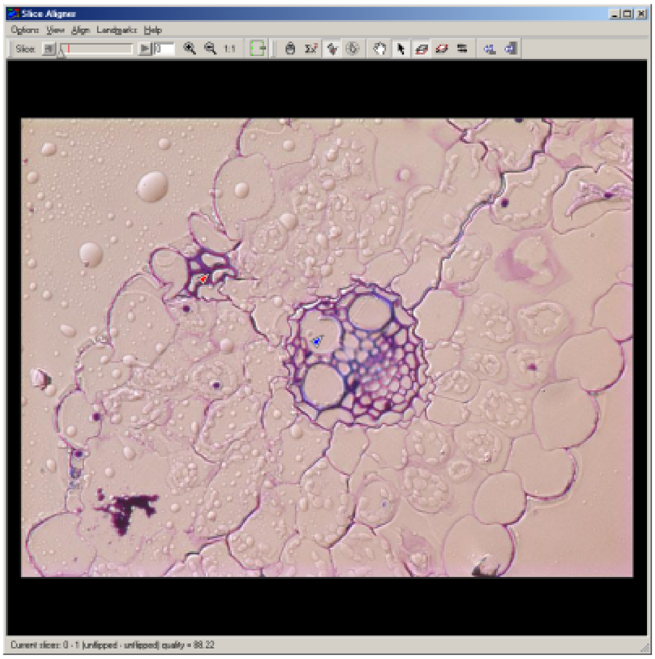
- Select slice number 0.
- Place the landmarks as shown in Figure 2.31. Make use of the double click method.
- In all other slices place the landmarks at the same positions.

Once all landmarks have been set, we can align the slices. It is possible to align only the current pair of slices, or to align all slices at once. Note that all alignment actions as well as landmark movements can be undone by pressing Ctrl-Z.

- Switch back to *transform* mode by pressing the hand-shape tool button. Two slices should be displayed again.
- Align the current pair of slices by pressing the second tool button from the right (the one with only two lines).
- Align all slices by pressing the first tool button from the right (the one with many lines).
- Move and rotate the whole object into the center of the image using the mouse with the Shift key hold down.

In most slices the alignment now should be quite good. However, looking at the pairs 3-4 and 4-5 (displayed in the lower left corner of the align window) you'll notice that there is something wrong. In fact, slice number 4 has been accidently inverted when taking the microscopic images. Fortunately, this error can be compensated in amira.

- Select slice pair 3-4 and make sure that the upper slice, i.e., slice number 4, is editable.



Current slices: 0 - 1 (unflipped - unflipped) quality = 88.22

Figure 2.31: The fi gure shows how the landmarks should be set in the tutorial.

- Invert the upper slice by pressing the invert button (third one from the right).
- Realign the current pair of slices by pressing the second button from the right).
- Select slice pair 4-5 and realign this pair of slices as well.

Alternatively, you could have aligned all slices from scratch by pressing the first button from the right.

2.9.3 Optimizing the Quality Function

Once all slices are roughly aligned we can further improve the alignment using the automatic optimization method. At the bottom of the align window the quality of the current alignment is displayed. This is a number between 0 and 100, where 100 indicates a perfect match. The quality function is computed from the squared grey value differences of the two slices. The optimization method tries to maximize the quality function. Since only local maxima are found, it is required that the slices are reasonably well aligned in advance.

- Activate the optimization mode by pressing the tool button with the sum x squared symbol. Remember the current quality measure.
- Align the current pair of slices by pressing the second button from the right. Observe, how the quality is improved.

Automatic alignment is an iterative process. It may take quite a long time depending on the resolution of the images and of the quality of the pre-alignment. You can interrupt automatic alignment at any time using the amira stop button.

- Automatically align all slices by pressing the first button from the right.

2.9.4 Resampling the Input Data

If you are satiesfied with the alignment you can resample the input data set in order to create a new aligned 3D image. This is done using the *Resample* button of the *AlignSlices* module.

- Press the *Resample* button of the *AlignSlices* module.
- Attach an *OrthoSlice* module to the resulting object *leaf.align* and verify that the slices are aligned.

Sometimes you may want to improve an alignment later on. In this case it is a bad idea to align the resampled data set a second time, since this would require a second resampling operation. Instead, you could write the transformation data into the original image object and store this object in *AmiraMesh* format. After reloading the *AmiraMesh* file you can attach a new *AlignSlices* module and continue with the stored transformations.

- Choose *Save transformation* from the *Options* menu of the align tool. This will store the transformation data in the parameter section of the input object *leaf.info*.
- Delete the *AlignSlices* module.
- Save *leaf.info* in *AmiraMesh* format.
- Reload the saved object *leaf.am*.
- Attach a new *AlignSlices* module to *leaf.am* and click the *Edit* button. Note that the original alignment is restored.

2.9.5 Using a Reference Image

In some cases you might want to correct the alignment after image segmentation has been performed. In order to avoid segmenting the newly resampled image from scratch, you can apply the same transformations to a label field using a reference image.

- Delete any existing align tool.
- Load the file `data/align/leaf-unaligned.labels` into amira.
- Attach a new *AlignSlices* module to the label field.

In the label field the guard cells of the stomatal pore are marked. Segmentation has been performed before the images were aligned. Now we want to apply the same transformation defined for the image data to the labels.

- Connect the *Reference* port of *AlignSlices* to *leaf.am* (this is done by activating the popup menu over the small rectangular area at the left side of the *leaf.am* icon). Observe how the transformations are applied to the label field.
- Export an aligned label field by pressing the *Resample* button.

The image volume used in this tutorial is an RGBA color field. However, the image segmentation editor only supports gray level images. Therefore you must convert the color field into a scalar field using *CastField* before you can invoke the image segmentation editor for the resampled labels.

2.10 Registration of 3D image datasets

In medical imaging a frequent task has become the registration of images from a subject taken with different imaging modalities, where the term *modalities* here refers to imaging techniques such as Computed Tomography (CT), Magnetic Resonance Tomography (MRT) and Positron Emission Tomography (PET). The challenge in inter-modality registration lies in the fact that e.g in CT images ’bright’ regions are not necessarily bright regions in MRT images of the same subject.

In registration typically one of the datasets is taken as the *reference*, and the other one is transformed until both datasets match. amira's Registration module provides an affine registration, i.e. it determines an optimal transformation with respect to translation, rotation, anisotrope scaling, and shearing.

Closely related to registration is the task of image fusion, i.e. the simultaneous visualization of two registered image datasets.

This tutorial shows how a registration can be performed and how to visualize the results. The following issues will be discussed:

1. Basic manual registration using the *Transform Editor*
2. Automatic registration
3. Image fusion

2.10.1 Basic Manual Registration

In this tutorial we want to register a CT and an MRT dataset of a patient, showing the pelvic region. The images are located in the amira data directory in the subdirectory *registration*.

- Load the files `data/registration/CT-data.am` and `data/registration/MRT-data.am` into amira.
- Attach an *OrthoSlice* module to each of the datasets.
- Select *Coronal* at the *Orientation* port of the *OrthoSlice* module connected to the MRT data.
- Select a camera position for the 3D viewer where you can see both the axial slices of the CT data and the coronal slices of the MRT data.
- Select slice 31 at the *Slice Number* port of the *OrthoSlice* module connected to the CT data.

Now one OrthoSlice module should show an axial slice through the hip joints. Move the coronal slice through the MRT data. You will observe that the two datasets are not correctly aligned.

- Select the green icon of the MRT dataset. Invoke the Transform Editor by pressing the transform box button (the third button from the right in the title bar) in the Working Area. The *Transform Editor* enables you to specify an affine transformation, including translation, rotation, and scaling. This transformation will be applied to the corresponding 3D dataset. You can edit the transformation interactively in the 3D viewer using different Open Inventor draggers. You can also enter transformations numerically.
- Press the *Dialog* button. A dialog window will pop up.
- Enter `-2` at the third text field at Port *Translation* of the dialog window. This means a translation of -2 cm in the z direction.
- Enter `5` at the first text field at the *Rotation* port. This means a rotation of 5 degrees. The axis of rotation is defined at the next ports, here it is the z-axis.
- Press the *Close* button of the dialog window. Leave the Transform Editor by pressing again the transform box button.

Inspect some coronal slices through the MRT dataset. Now there is a better alignment of the CT and MRT data, but it's still not perfect.

2.10.2 Automatic Registration

The Registration module provides an automatic registration via optimization of a quality function. For registration of datasets from different imaging modalities, the *Normalized Mutual Information* is the best suited quality function. In short, it favors an alignment which 'maps similar gray values to similar gray values'. A hierarchical strategy is applied, starting at a coarse resampling of the datasets, and proceeding to finer resolutions later on.

- Attach an *Registration* module to the MRT dataset by choosing *Compute/AffineRegistration* from the popup menu over the *MRT-data.am* icon.
- Connect the second input port *Reference* of module *Registration* to the CT dataset. For this click with the right mouse button on the small rectangle at the left hand side of the module's icon.
- Select toggle *Extended options*. More ports of the *Registration* module will become visible.

The first three ports of the *Registration* module define the optimization strategy. The default settings mean that an *ExtensiveDirection* optimizer is used for the coarse levels and a *QuasiNewton* optimizer for the finest two levels of the resampling hierarchy. At the *CoarsestResampling* port you can select the resampling rate for the coarsest resolution level. The default resampling rate is smaller in the z direction because the reference dataset has a finer resolution in the x and y direction (0.17 cm) than in the z direction (0.5 cm). For the default settings (8,8,3), the resampling hierarchy will consist of four levels: (8,8,3), (4,4,2), (2,2,1), and the original resolution, (1,1,1).

The *Normalized Mutual Information* is calculated from gray value histograms. The selected histogram ranges should enclose the essential information of each dataset. Normally you can choose the same range as for visualization via an *OrthoSlice* module.

- Set `-200` and `200` at the two text fields of the *Histogram range reference* port.

At the *Transformation* port you can specify the type of affine transformation. The default settings mean that only rigid body motions will be applied, i.e. translations and rotations.

Option *IgnoreFinestLevel* means that optimization is done on all but the finest level of the resampling hierarchy. This will slightly reduce the accuracy, but save a large amount of computing time.

Automatic registration may take some time depending on the resolution of the images and the quality of the pre-alignment. You can interrupt automatic registration at any time using the stop button. Interruption may take some seconds. The progress bar shows the current hierarchy level and the progress at that level.

- Start automatic registration by pressing the *Register* button at the *Action* port.

2.10.3 Image Fusion

The task of image fusion is the simultaneous visualization of two datasets. To that end amira offers for all types of slicing modules (Orthoslice and ObliqueSlice) the *Colorwash* module. Using *Colorwash*, the images from one dataset can be overlayed over that of another taking into account their orientaion in space.

- Remove the *OrthoSlice* module connected to the MRT dataset.
- Select the green icon of the MRT dataset.
- Select a Colorwash module from the popup menu over the icon of the *OrthoSlice* module connected to the CT dataset.
- Select the yellow icon of the *Colorwash* module.
- Select the *physics.icol* colormap at port *Colormap*.
- Set `70` as the upper bound for the colormap range.
- Select the icon of the *OrthoSlice* module.
- Inspect axial slices with slice numbers between 15 and 45.

You will observe a good alignment of the pelvic bone from both datasets. The soft tissue contours are not perfectly aligned because there was some soft tissue deformation between both scans. This cannot be described by a rigid transformation.

In image fusion it is sometimes necessary to observe all three orthogonal directions simultaneously. For that the StandardView module can be used for image fusion. The *StandardView* module opens a separate window with four viewers, three of them showing the three orthogonal slices of the image data and a fourth being a new instance of the 3D viewer.

- Attach a *StandardView* module to the CT dataset by choosing *Display StandardView* from the popup menu over the *CT-data.am* icon. amira's *Viewer* window will now be split into four parts showing three orthogonal slices through the CT data, and the 3D Viewer in the upper left part.
- Connect the second input port *OverlayData* of the *StandardView* module to the MRT dataset. For this, click with the right mouse button on the small rectangle at the left hand side of the module's icon.
- Select slice numbers 179, 149, and 31 at ports *Slice x*, *Slice y*, and *Slice z*, respectively. The three orthogonal slices will show the hip joints now.
- Increase the zoom-factor by clicking twice on button ¿ at the *Zoom* port.
- Select *checkerboard* at the *Overlay mode* port.
- Vary the size of the checkerboard tiles by moving the slider at the *Pattern size* port. In this way you can again check the alignment of the CT and MRT datasets.

The bone contours around the hip joints show a good match. Note that bone is represented by white (i.e high intensity) voxels in the CT data, but may occur as both white and black voxels in the MRT data. In the axial slice you can observe larger deviations of the outer body contour between the CT and

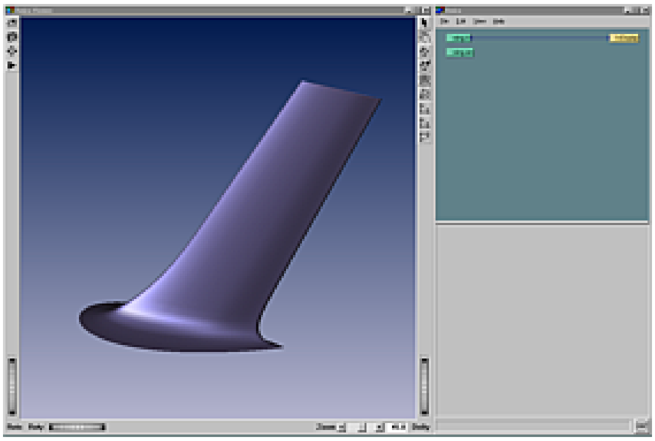


Figure 2.32: Open Inventor geometry of the airfoil.

MRT data.

2.11 Visualization of Vector Fields

This step-by-step-tutorial briefly explains some amira modules for vector field visualization. The use of these modules is explained by way of data representing the flow around an airfoil. The two methods referred to in steps 2 and 3 are independent of each other. These topics will be covered:

1. Loading the Wing and the Flow Field
2. Line Integral Convolution
3. Illuminated Stream Lines

2.11.1 Loading the Wing and the Flow Field

As in the previous tutorials, we use the file dialog to import data.

- Import the geometry of the wing by loading the file `wing.iv` from the directory `data/tutorials`.
- Attach an *IvDisplay* module to this data object.
- Load the vector field data set called `wing.am` from the directory `data/tutorials`.

The extension *.iv* indicates that the wing geometry is defined in the Open Inventor file format. The *IvDisplay* module can be used for displaying geometry in this format. The vector field itself is stored in amira's native *AmiraMesh* file format. The data represents the flow around an airfoil computed on a regular grid with curvilinear coordinates. By selecting the green data icon *wing.am*, you can find out that the number of grid nodes in x,y,z-direction is 125 x 41 x 21.

2.11.2 Line Integral Convolution

Line Integral Convolution (LIC) is a method for visualizing 2D vector fields, i.e., depicting the vector field direction for a suitably sampled subset of points in the 2D domain. The direction is represented by local streamlines, i.e. by curves whose tangent vectors coincide with those of the given vector field. Local streamlines are computed such that all image pixels are covered. The streamlines are projected onto a random noise input texture map of the same size as the vector field domain. The projection step involves summations of texture pixel intensities along streamline paths by way of a convolution integral with a filter kernel. This causes pixel intensities in the resulting image to be highly correlated along individual streamlines but statistically independent perpendicular to the latter. Thus the directional structure of the vector field becomes clearly visible.

Here we use the 3D vector field that we have already loaded and visualize two-dimensional slices:

- Connect a *PlanarLIC* module to the vector field *wing.am* and select it.
- Choose the slice number 22 in the *Translate* port.
- Set filter length to 40 and resolution to 200.
- Press the *Do It* button of the *Action* port.

Whenever the projection plane of the *PlanarLIC* module is changed or other values for filter length or resolution are taken, the LIC texture must be recalculated, i.e., the *Do It* button must be pressed again. Otherwise, a checkerboard pattern will be displayed. Experiment with different filter lengths and resolution values to see what kinds of textures can be produced.

To get an impression of the magnitude of the vectors, you can apply a colormap to the image. Some default colormaps are already loaded when amira starts up. The corresponding icons usually will be hidden. In order to use one of the colormaps, you have to select it explicitly:

- From the main window's *Edit Show* menu select *temperature.icol* or any other colormap: the corresponding icon becomes visible.
- Select *Magnitude* from port *Colorize* of the *PlanarLIC* module. A new port labeled *Colormap* appears in the working area.
- Connect the colormap port to *temperature.icol* by clicking with the right mouse button in the red rectangle and choosing *temperature.icol* from the appearing popup menu.

The two integer values of the colormap port specify the range of values to which the colormap is applied. Vector magnitudes within this range are depicted symbolically by coloring streamline pixels such that each pixel gets the unique color associated with the magnitude value by the *temperature.icol* colormap.

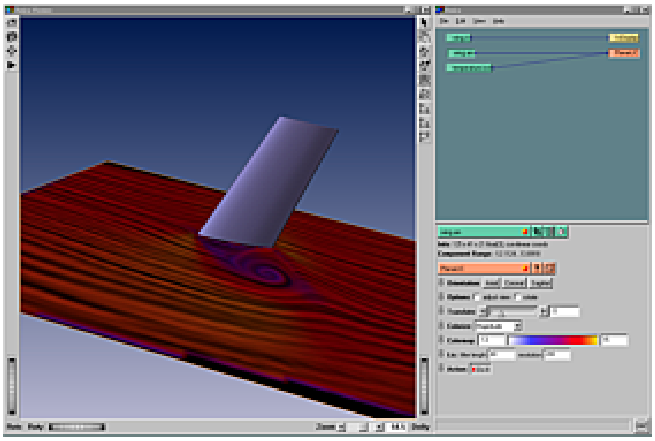


Figure 2.33: Air fbw around the wing visualized using Line Integral Convolution

- Select the *wing.am* icon and read off the magnitude range.
- Shift-select the *PlanarLIC* icon and enter the magnitude range into the *Colormap* port.

2.11.3 Illuminated Stream Lines

Illuminated Stream Lines is a technique for interactive 3D vector field visualization which makes use of large numbers of properly illuminated stream lines. A realistic shading model is employed which significantly increases realism of the resulting images and enhances spatial perception.

Now you will learn which tools are used for illuminated stream line visualization and how to use them to get a 3D impression of our airflow vector field.

- Remove the *PlanarLIC* module or disable its display.
- Connect a *DisplayISL* module to the vector field *wing.am*.
- Set *Num Lines* to 300.
- Press the *DoIt* button of the *Distribute* port.
- Click on the *TabBox* button of port *Box* and zoom out the viewer if you cannot see a box.

A *TabBox* appears in the viewer. Only stream lines flowing through this box are visible. The green ticks at the corners and edges of the box allow you to change the dimensions of the box.

- Switch the viewer into interaction mode.
- Try out what happens if you click with the left mouse button on one of the green ticks on the corners or edges of the *TabBox* and drag them around.

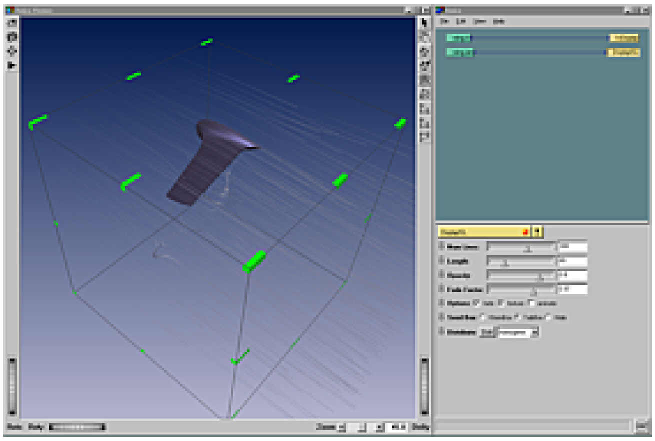


Figure 2.34: Illuminated streamlines around a wing

- To move the whole *TabBox*, click with the left mouse button in the box and move it.
- Try to get the *TabBox* into a shape and position as shown in the image.
- Press *DoIt* button again in order to recalculate the stream lines.

Now some information about Console Window commands. Most amira modules provide more control features than those that are available by the ports displayed in the Working Area. All of them are available by commands that you type in the console window. You can get a list of commands associated with a particular module currently in use just by entering its name. Now we will use such commands to form and position the TabBox exactly as in the image above.

- Type "DisplayISL" into the Console Window.
- Type "DisplayISL getBox" into the Console Window.

The first command lists all commands of the *DisplayISL* module and the second shows the scale and translation of the current TabBox. The first word of a command always has to be the name of a module as shown on its icon. Note that module commands are not recognized unless corresponding modules have been loaded into the object pool. However, you do not need to select a module for typing in its commands.

- Type "DisplayISL setBoxTranslation -0.31 0.00 0.17"
- Type "DisplayISL setBoxScale 0.11 0.04 0.14"

Now the sub-field that corresponds to the clipping of the vector field as implied by the settings of the *TabBox* should look like the one shown in the image above.

2.12 Creating animated demonstrations

In this tutorial you learn how to use the DemoMaker module for creating an animated sequence of operations within amira. In our example, we will visualize a polygon model using effects such as transparency, camera rotation, and clipping to make the visualization more meaningful and attractive.

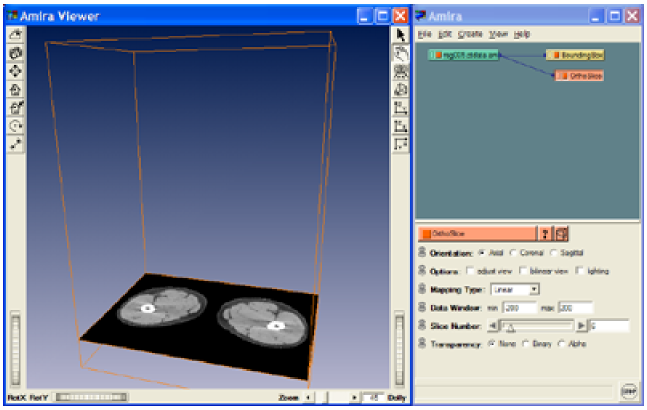
The tutorial covers the following topics:

- creating an initial network for the demo
- animating an *OrthoSlice* module
- activating additional modules during the demo
- using a camera rotation or path
- editing or removing events that are already defined
- overlaying a bone model with a transparent skin model
- using clipping to make the skin appear gradually over the bone
- advanced clipping issues
- inserting breaks and defining demo segments
- using function keys for jumping between demo segments
- defining partial loops within the demo sequence
- storing and replaying a demo sequence

Once you have learned how to define an animated demo sequence, you can further learn how to record the demonstration into a movie file in Section 2.13.

2.12.1 Creating a Network

First, we need an amira network that contains all the data and modules for the visualization and animation we want to do. In our example, we pick the medical CT scan dataset `reg005`. Start by loading `data/medical/reg005.ctdata.am` from the amira root directory. By right-clicking on the green data icon and selecting from the dataset's popup menu, attach a BoundingBox module as well as a OrthoSlice module to the data. If you use the mouse to navigate around the model in the 3D viewer, you should manage to get a result similar to this:

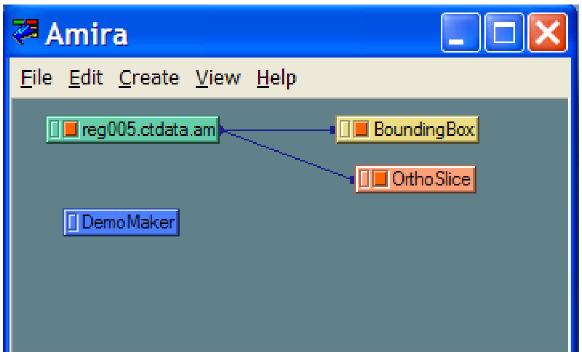

Amira Viewer
Amira
File Edit Create View Help
reg005.ctdata.am
BoundingBox
OrthoSlice
OrthoSlice
Orientation: Axial Coronal Sagittal
Options: adjust view bilinear view lighting
Mapping Type: Linear
Data Window: min max
Slice Number:
Transparency: None Binary Alpha
RotX RotY
Zoom
Dolly

(load network)

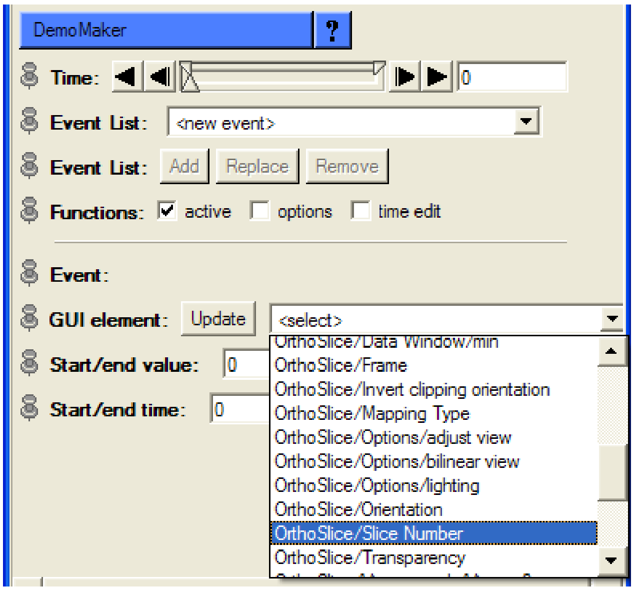
2.12.2 Animating an OrthoSlice module

Let us move the OrthoSlice plane up and down to show what the data looks like. Note that the *OrthoSlice* module has a port called *Slice Number*. If you change the value of that slider, you see the plane move in the viewer.

Now let us animate this slider using the DemoMaker module as our first exercise. From the menu bar, select *Create / Animation/Demo / DemoMaker*. A blue script object appears in the object pool:

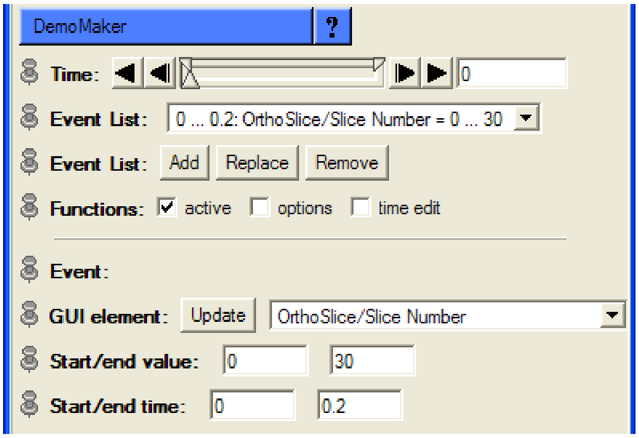


Click on the blue icon to see its user interface. Whenever you want to animate some port of the current network, you must select that port in the selection list called *GUI element*. Try to find the entry called *OrthoSlice/Slice Number*, which corresponds to the *Slice Number* port of the current *OrthoSlice* module. If you cannot find the entry, you may need to press the *Update* button to the left of the selection menu (see below for an explanation).


DemoMaker
?
Time:
0
Event List: <new event>
Event List: Add Replace Remove
Functions: active options time edit
Event:
GUI element: Update <select>
Start/end value: 0
Start/end time: 0
OrthoSlice/Data Window/min
OrthoSlice/Frame
OrthoSlice/Invert clipping orientation
OrthoSlice/Mapping Type
OrthoSlice/Options/adjust view
OrthoSlice/Options/bilinear view
OrthoSlice/Options/lighting
OrthoSlice/Orientation
OrthoSlice/Slice Number
OrthoSlice/Transparency

Once you have selected *OrthoSlice/Slice Number*, you see two more ports appear in the *DemoMaker* module: *Start/end value* as well as *Start/end time*. The start and end value specify between which two values the OrthoSlice slider will be moved. Click the mouse into the start or end value fields, hold down the `Shift` key, and drag the mouse with `Shift` held down. With this feature called the *virtual slider* you can quickly set the desired value within the allowed range. Set the start value to 0 and the end value to 30. Then, set the start time to 0 and the end time to 0.2. These time values specify times on the *Time* slider of the *DemoMaker* module (first port in the module). You have now specified an *event* starting at time 0 and ending at time 0.2, varying the OrthoSlice slice number between 0 and 30 during that time.

Now press the *Add* button in the *Event List* port to add the newly defined event to the list of events. This list of events is represented in the selection menu above the *Add* button.



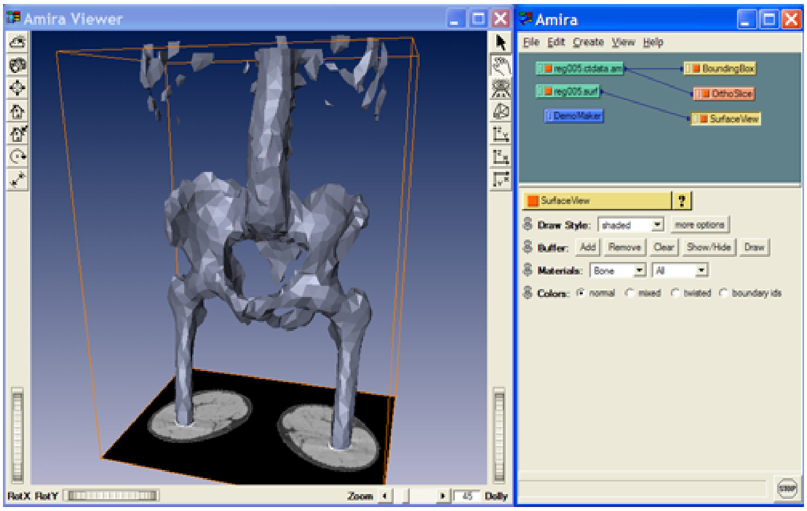
Time: 0
Event List: 0 ... 0.2: OrthoSlice/Slice Number = 0 ... 30
Functions: [✓ active] options time edit
Event:
GUI element: OrthoSlice/Slice Number
Start/end value: 0 30
Start/end time: 0 0.2

Test the result by pressing the play button of DemoMaker's time slider, represented by the triangle pointing to the right. When you press it, the time slider will start moving from the left to the right. When the time value is between 0 and 0.2, you should see that the OrthoSlice plane is moving between the specified start and end values in the viewer (load network). You can also play your demo backwards using the play button to the left of the time slider, or simply click somewhere on the time slider to jump to any point in time of the demonstration.

If the demo sequence runs too slow or too fast, you can adjust this by right-clicking anywhere on the time slider and selecting *Configure* from the popup menu. Change the *Increment* value in the dialog box that appears. A smaller increment will make the animation slower, whereas a larger increment makes it faster. If you choose too big an increment value, the animation might become "jerky".

2.12.3 Activating a module in the viewer window

Next, let us add a visualization of the bone structure in the dataset after we have moved the OrthoSlice. Load the dataset `data/medical/reg005.surf` in addition to the current network. Attach a SurfaceView module to it. Click on the yellow *SurfaceView* module to see its user interface. Press the *Clear* button in the *Buffer* port, then select *Bone* and *All* in the *Materials* port and press the *Add* button in the buffer port. This will visualize the bone surface of the model.



SurfaceView

Draw Style: shaded

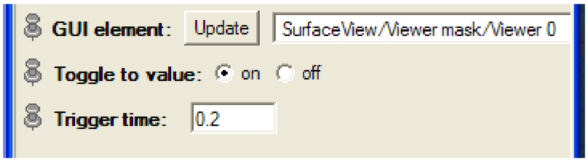
Buffer: Add Remove Clear Show/Hide Draw

Materials: Bone All

Colors: [✓ normal] mixed twisted boundary ids

If you want to switch the bone visualization on and off manually, you would use the *viewer toggle* (orange rectangle) of the *SurfaceView* module. If you want to include this action in your demo sequence, you need to do the following:

- Click on the *DemoMaker* module.
- Click on the *Update* button in the *GUI element* port. This is necessary whenever you add new modules to the object pool after you have created the *DemoMaker* module.
- Select *SurfaceView/View Mask/Viewer 0* from the *GUI element* list.
- Enter *on* in the *Toggle to value* port.
- Enter *0.2* in the *Trigger time* port.
- Press the *Add* button in the *Event List* port to add this newly defined event to the event list.



GUI element: Update SurfaceView/Viewer mask/Viewer 0

Toggle to value: [✓ on] off

Trigger time: 0.2

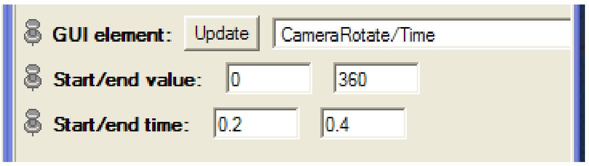
To test the newly added event, first toggle the *SurfaceView* module off using its orange viewer toggle icon. Now click to the very left of the *DemoMaker* time slider to jump to the start of the demo sequence. Click the play button. As before, the slice moves up. When it reaches the maximum value at time 0.2, the bone model is switched on (load network).

2.12.4 Using a camera rotation

To look at the 3D patient model from all sides, let us add a camera rotation to our demo sequence. Select *Create / CameraRotate* from the menu. Try the rotation by playing the time slider in the *CameraRotate* module. If you do not like the axis of rotation, reset the time slider to 0, navigate to a good starting view in the viewer window, and click on *recompute* in the *CameraRotate* module. Note that the values of the *CameraRotate* time slider range from 0 to 360.

Once you are satisfied with the camera rotation, add it to the event list:

- Click on the *DemoMaker* module.
- Click on the *Update* button.
- Select *CameraRotate / Time*
- Enter 0 and 360 as the start and end value.
- Enter 0.2 and 0.4 as the start and end time.
- Click on the *Add* button in the *Event List* port.


GUI element: Update CameraRotate/Time
Start/end value: 0 360
Start/end time: 0.2 0.4

Now play the demo to see the result. After moving the slice and switching on the bone model, the view is rotated so that the bone can be seen from all sides (load network).

2.12.5 Editing or removing an already defined event

When you look at the demo sequence so far, you may think that it would be nice to wait for a short time before rotating the bone model. This can be done by starting the rotation at a later time step. We can easily correct this in the *DemoMaker* module:

- Select the *0.2 ... 0.4: CameraRotate* event in the *EventList* port.
- You see the start/end value and start/end time of this event appear in the lower part of the *DemoMaker* module.
- Change the start/end time to 0.3 and 0.5.
- Click on the *Replace* button in the *Event List* port. This replaces the currently selected event in the list by the event as defined in the lower part of the module.

Now you have moved the camera rotation event from 0.2-0.4 to 0.3-0.5 on the time line. Check the results by playing the time slider (load network).

Please note that you can delete an event from the list by simply selecting it from the *Event List* menu and clicking the *Remove* button.

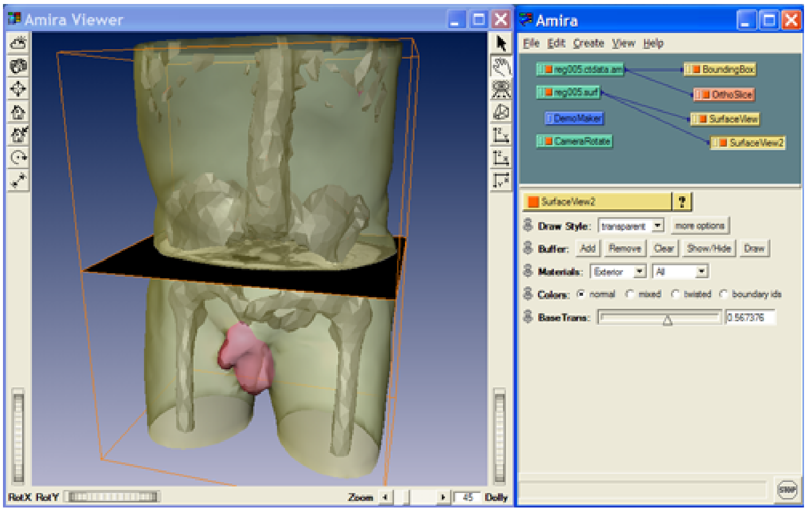
2.12.6 Overlaying the bone with skin

Now we want to show the patient's outer surface overlayed over the bone model.

- Attach a second *SurfaceView* module to the *reg005.surf* dataset. Since *Exterior* and *All* are selected as the default materials, this brings up the patient's exterior surface.
- Click on the second *SurfaceView* module. It should be called *SurfaceView2*.
- Select *transparent* from the *Draw Style* port.
- It will be helpful to show the bone underneath the exterior surface, so jump to time step 0.2 or later in the DemoMaker module.
- Adjust the grade of transparency using the *BaseTrans* slider in *SurfaceView2*.
- Smooth out the outer surface by clicking on *more options* in the *Draw Style* port and selecting *Vertex normals*.

Like we did with the bone model, we can switch on the skin model at some point in the demo sequence:

- Click on the *DemoMaker* module.
- Click on the *Update* button in the *GUI element* port.
- Select *SurfaceView2/View Mask/Viewer 0* from the *GUI element* list.
- Check *on* in the *Toggle to value* port.
- Enter *0.6* in the *Trigger time* port.
- Click on *Add* in the *Event List* port.



SurfaceView2

Draw Style: transparent

Materials: Exterior, All

Colors: [✓ normal] mixed twisted boundary ids

BaseTrans: 0.567376

Again, check out the results by playing the demo sequence.

2.12.7 Using clipping to add the skin gradually

Instead of just switching the skin on at one point, we can make it appear gradually over the bone from bottom to top. In order to do so, we use the *OrthoSlice* plane to *clip* the skin model, and then move the *OrthoSlice* plane up.

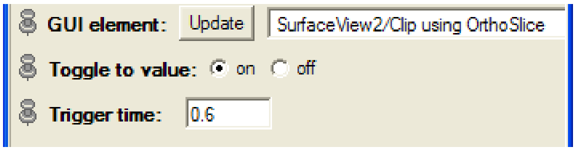
First, we need to move the *OrthoSlice* plane down again to where we want to start the clipping:

- Click on the *DemoMaker* module.
- Select *OrthoSlice/Slice Number* from the *GUI element* list.
- Enter 30 and 3 as the start/end values.
- Enter 0.3 and 0.5 as the start/end time.
- Click on *Add* in the *Event List* port.

Now, when you play the demo, the *OrthoSlice* plane will move down again during the camera rotation (load network).

Now, we will clip the skin model using the OrthoSlice plane:

- Click on the *DemoMaker* module.
- Select *SurfaceView2 / Clip using OrthoSlice* from the *GUI element* list.
- Enter *on* as toggle value and *0.6* as trigger time.
- Click on *Add* in the *Event List* port.



GUI element: Update SurfaceView2/Clip using OrthoSlice

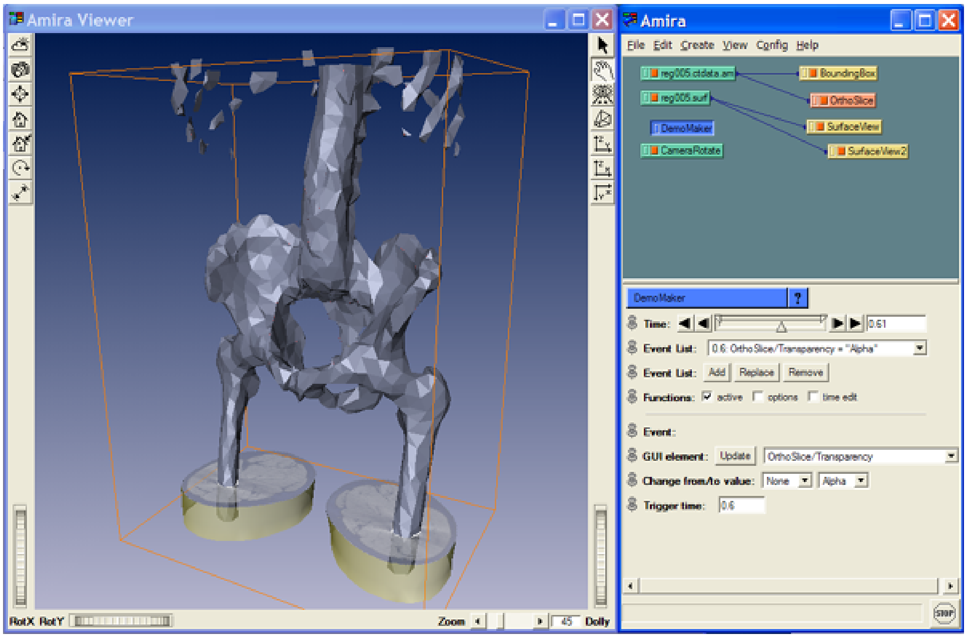
Toggle to value: [✓ on] off

Trigger time: 0.6

When you run the animation now, you will not see the skin surface. This is because it is clipped above the *OrthoSlice* plane, and only visible below that plane. To see the partial surface below the plane, we must make the *Orthoslice* display transparent:

- Click on the *DemoMaker* module.
- Select *OrthoSlice / Transparency* from the *GUI element* list.
- Select *None* and *Alpha* as the from/to values.
- Enter *0.6* as the trigger time.
- Click on *Add* in the *Event List* port.

This way we have specified that at time 0.6, the *Transparency* port of the *OrthoSlice* module will be changed from value *None* to the new value *Alpha*. When running the demo sequence, the result should look like this:



DemoMaker

Time: 0.61

Event List: 0.6: OrthoSlice/Transparency = "Alpha"

Event List: Add Replace Remove

Functions: [✓ active] options time edit

Event:

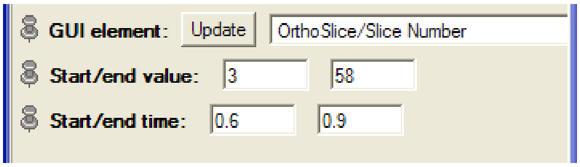
GUI element: Update OrthoSlice/Transparency

Change from/to value: None Alpha

Trigger time: 0.6

As you see, part of the skin model is showing below the transparent *OrthoSlice* plane. To show all of the skin, we simply move the plane upwards pretty much the same way we did before:

- Click on the *DemoMaker* module.
- Select *OrthoSlice / Slice Number* from the *GUI element* list.
- Enter *3* and *58* as the start/end value.
- Enter *0.6* and *0.9* as the start/end time.
- Click on *Add* in the *Event List* port.

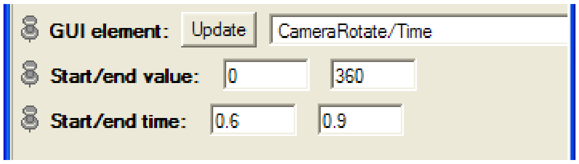


GUI element: Update OrthoSlice/Slice Number
Start/end value: 3 58
Start/end time: 0.6 0.9

Now you see the skin slowly appearing over the bone as the clipping plane moves upwards.

As a last step, you might want to rotate the view again while the skin is appearing. You can simply reuse the old camera rotation during a second time range:

- Click on the *DemoMaker* module.
- Select *0.3 ... 0.5: CameraRotate* from the *Event List* menu.
- You will see start/end value and start/end time appear in the lower part.
- Change the start/end time to 0.6 and 0.9.
- Click on *Add* in the *Event List* port. This will leave the old event untouched and add a second camera rotation event to the list.



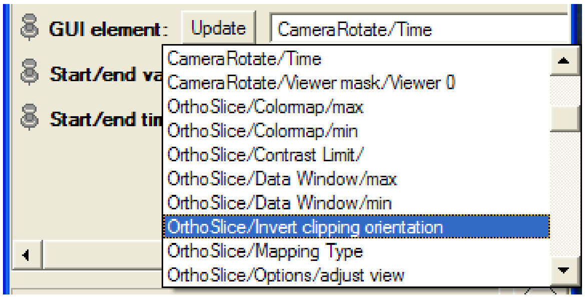
GUI element: Update CameraRotate/Time
Start/end value: 0 360
Start/end time: 0.6 0.9

You can check out the final animation by loading a saved network script.

2.12.8 More comments on clipping

Clipping can sometimes be a little bit more complicated than in our example, because clipping can be applied to a plane in two different orientations. This means that you can either clip away everything *above* the plane, or *below* the plane. Unfortunately it is not always obvious which of the two cases you are in.

However, you can simply invert the orientation of the clipping in *DemoMaker*. In our example, you would simply select *OrthoSlice / Invert clipping orientation* from the *GUI elements* port and add that event at the very beginning of your demo sequence (e.g., at some time before the clipping takes effect).


GUI element:
Update
CameraRotate/Time
Start/end va
Start/end tin
CameraRotate/Time
CameraRotate/Viewer mask/Viewer 0
OrthoSlice/Colormap/max
OrthoSlice/Colormap/min
OrthoSlice/Contrast Limit/
OrthoSlice/Data Window/max
OrthoSlice/Data Window/min
OrthoSlice/Invert clipping orientation
OrthoSlice/Mapping Type
OrthoSlice/Options/adjust view

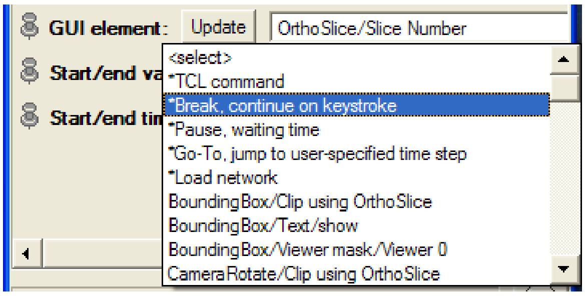
You do not need to use an *OrthoSlice* module to do clipping. As you have seen, the *OrthoSlice* might occlude parts of what you want to show. In that case, it is better to create an empty ClippingPlane module by selecting *Create / Clippping Plane* from the menu. Attach the module to the dataset you want to clip (e.g., to *reg005.surf* in our example), and then use the *ClippingPlane* for clipping just as you used the *OrthoSlice* before.

2.12.9 Breaks and Function Keys

The demo sequence that we have created in this tutorial automatically runs through the complete time range that we defined. Sometimes it might be desirable to split the sequence into several segments, so that the demo will stop at some point and can be continued whenever the user desires to do so.

To take this into account, you can insert *breaks* in the *DemoMaker* event list. Let us insert one such break right after the bone model appears:

- Click on the *DemoMaker* module.
- Select **Break, continue on keystroke* from the *GUI elements* list.
- Enter 0.21 as the trigger time.
- Click on *Add* in the *Event List* port.


GUI element:
Update
OrthoSlice/Slice Number
Start/end va
Start/end tin
<select>
*TCL command
*Break, continue on keystroke
*Pause, waiting time
*Go-To, jump to user-specified time step
*Load network
BoundingBox/Clip using OrthoSlice
BoundingBox/Text/show
BoundingBox/Viewer mask/Viewer 0
CameraRotate/Clip using OrthoSlice

This way the demo will stop at time 0.21, which is right after the time when the bone model is switched on (0.2). When you play the demo from the start, you will notice that after the bone is switched on, the demo will stop.

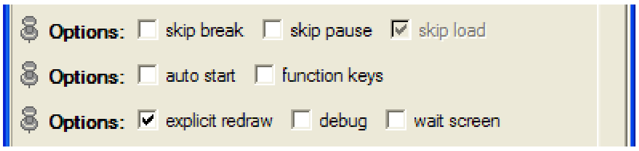
Let us insert a second break at time step 0.51, which is right before the skin is starting to show. Proceed as above, using a trigger time of 0.51 instead of 0.21 (load network).

If you run the demo from the very beginning, it will stop after the bone is displayed, and you can read a message in the console window telling you that *DemoMaker* just stopped and you may press `F4` to continue. Try this by pressing the function key `F4`. The demo continues.

Likewise, the demo will stop just before showing the skin. Again, you can continue the demo by pressing `F4`. In general, at any point while the demo is running, you can press the `F3` key to stop it manually. Pressing `F4` will continue from the point where the demo stopped.

If you have defined breaks as we did above, there are two more interesting function keys that in some sense allow you to *navigate* through the demo segments: pressing `F9` will jump back to the previous break or to the very beginning of the demo, and `F10` will jump to the next break, or to the very end of the demo. If you use `F9/F10` when the demo is stopped, it will just jump, and you need to press `F4` to start playing it from the new time step. If you press `F9/F10` while the demo is running, it will just jump to the new time step and continue running.

Please note that you can disable the breaks by checking the *skip break* toggle in the *Options* port of the *DemoMaker* module. You may even disable the definition of function keys by checking the *options* toggle in the *Functions* port, and then unchecking *function keys* in the second *Options* port. This is especially important if you want to use multiple *DemoMaker* modules, since only one of the modules can define the keys.


Options: skip break skip pause [✓ skip load]
Options: auto start function keys
Options: [✓ explicit redraw] debug wait screen

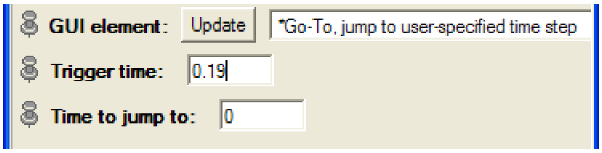
2.12.10 Loops and go-to

One more feature that might be required for certain kinds of demos is the definition of loops. If you just want the whole demo to run in a loop, you can do this easily using the built-in features of the time port slider: right-click on the slider and select *loop* or *swing*. Now if you play the time slider, it will start over from the beginning (loop mode), or play forwards, backwards, forwards ... (swing mode).

However, you may want to define some part of the demo to run in a loop, and then stop the loop and continue with the demo upon key press. You can easily do this with the *go-to* feature of *DemoMaker*:

- Click on the *DemoMaker* module.
- Select **Go-to, jump to user-specified time step* from the *GUI elements* list.
- Enter 0.19 as the *Trigger time*.

- Enter 0.0 as the *Time to jump to*.
- Click on *Add* in the *Event List* port.


GUI element: Update *Go-To, jump to user-specified time step
Trigger time: 0.19
Time to jump to: 0

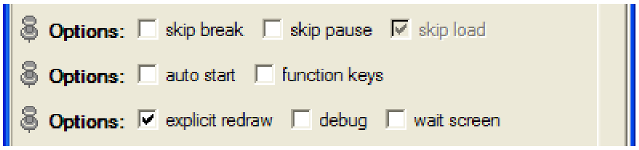
When you run the demo sequence now, it will loop in the first segment, only showing the *OrthoSlice* move up, jump down, move up again ... You can stop this by clicking on the stop button of the *DemoMaker* time slider or by pressing `F3`. To continue after the loop, you need to jump to the next segment by pressing `F10`, and then start playing again by pressing `F4`.

2.12.11 Storing and replaying the animation sequence

As you may have noticed by now, storing a demo sequence once you have defined it is quite easy: simple save the whole amira network by selecting *File / Save Network...* from the menu. The *DemoMaker* module will be saved along with the network, and so will the demo sequence you have defined.

When you load the network back into amira, the state of the network will be the same as it was when you saved it. This means that you should be careful to reset the *DemoMaker* time slider to 0 before saving the network, if you want the demo to start from the beginning.

After loading the network, you can start the demo by clicking on the play button of the *DemoMaker* module, or by pressing `F4`. If you want to run the demo automatically right after the network is loaded, you can use the *auto start* feature that you find when you check *options* in the *Functions* port:


Options: ☐ skip break ☐ skip pause ☑ skip load
Options: ☐ auto start ☐ function keys
Options: ☑ explicit redraw ☐ debug ☐ wait screen

Just check the *auto start* toggle and save the network. When you load it again, the demo will start running automatically (load network).

2.13 Creating movie fi les

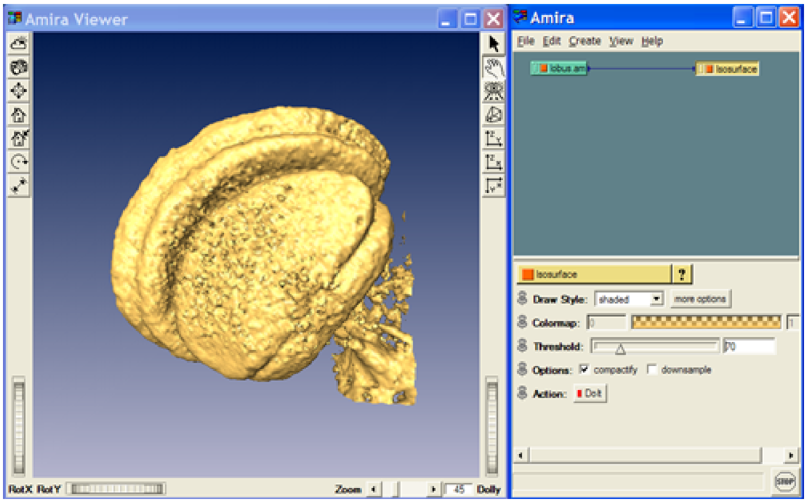
In this turorial you learn how to record a self-created animated sequence into a movie file using the MovieMaker module.

In our first example we will just use a camera path to animate the scene, whereas in our second example we will rely on the demo sequence created in Section 2.12.

2.13.1 Attaching MovieMaker to a camera path

If you have created a visualization of your data and want to create a movie showing this visualization from all sides or from certain interesting viewpoints, you can create an appropriate camera path and record a movie by following the camera along that path.

Let us create a simple example. Load the `lobus.am` dataset from the `tutorial` subdirectory and attach an *Isosurface* module to it. Choose an iso surface threshold of 70 and press the *Do It* button. The result should look similar to this:

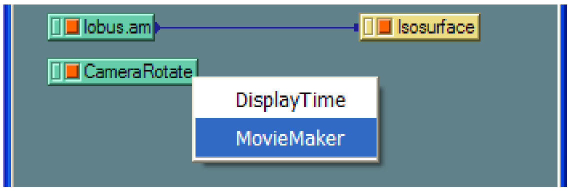


Isosurface
Draw Style: shaded · more options
Colormap: 0 · 1
Threshold: 70
Options: [✓ compactify] downsample
Action: DoIt

The easiest way to create a simple camera path is to use the CameraRotate module. Select *Create / CameraRotation* from the menu, and press the play button of the newly created module. You can watch the scene rotate in the viewer while the time slider is playing (load network).

To record an animated scene into a movie file, you need to attach a *MovieMaker* module to a module that posses a time slider port. The movie is recorded by going through the individual time steps and taking snapshots of the viewer along the way.

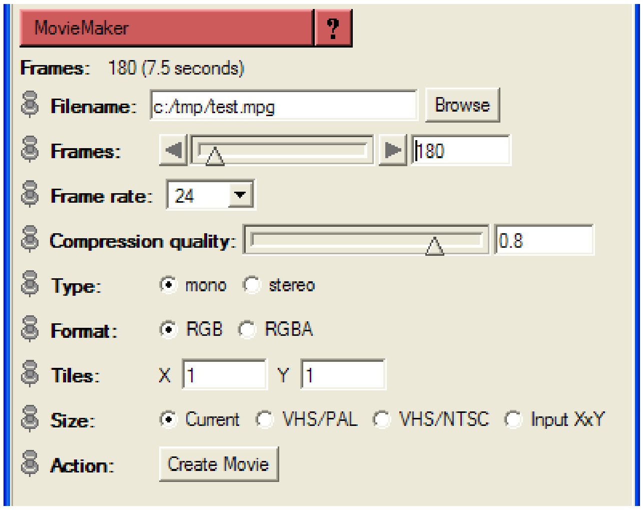
In our example, the *CameraRotate* module has a time slider, so we can attach a *MovieMaker* module to it by right-clicking on the *CameraRotate* icon in the object pool and selecting *MovieMaker* from the popup menu:



In the *MovieMaker* module, first click on the *Browse* button in the *Filename* port and enter a movie file name like `c:/tmp/test.mpg`. The `.mpg` suffix suggests that the movie file format will be MPEG, which is a widely accepted standard format for digital movies achieving a good compression ratio.

Next, adjust the parameters of the *MovieMaker* module to your liking, e.g., change the *number of frames*, the *image size*, or the *compression quality*. Please refer to the MovieMaker documentation for details.

In our example, let us choose 180 frames and leave all other parameters untouched. Since the *CameraRotate* module does a full rotation of 360 degrees, each of the 180 frames will represent a rotation of two degrees with respect to the previous frame. Press the *Create Movie* button to start recording.



Frames: 180 (7.5 seconds)
Filename: c:/tmp/test.mpg
Frames: 180
Frame rate: 24
Compression quality: 0.8
Type: [✓ mono] stereo
Format: [✓ RGB] RGBA
Tiles: X 1 Y 1
Size: [✓ Current] VHS/PAL VHS/NTSC Input XxY
Action: Create Movie

Wait for some time while the *MovieMaker* module drives the *CameraRotate* module and accumulates the snapshots. *Please note that the speed during the recording process is different than the playback speed of the movie.* Now view the resulting movie file `test.mpg` with a movie player of your choice (e.g., Windows Media Player or a similar tool). Experiment with the recording parameters until you get the desired result (e.g., control the file size and image quality by changing the *Compression quality* value, choose different image sizes to see up to which image size your computer is capable of smoothly displaying the movie, and change the number of frames to control the speed of the rotation).

2.13.2 Attaching MovieMaker to DemoMaker

Now we try to record a movie of a more complex animated scene. To this end, we load one of the networks that we have created in in Section 2.12: load network.

As you might remember, the basic idea of the DemoMaker module was that you define a set of events to be executed on a certain time line. Check this out by clicking the play button of the time slider in the *DemoMaker* module. You should see a nicely animated demonstration.

If you remember the previous section in this tutorial, you might already have an idea of how we can record this animated demonstration into a movie file. Like the *CameraRotation* module in the first example, the *DemoMaker* module is controlled via a *time slider* that we can attach to. So simply right-click on the *DemoMaker* icon in the object pool and attach a *MovieMaker* module. Like before, enter a movie file name and select the number of frames before you click on the *Create Movie* button to start recording.

Chapter 3

Program Description

This chapter contains a detailed description of amira interface components and data types. No in-depth knowledge of amira is required to understand the following sections, but it is a good idea to have a look at one of the tutorials contained in Chapter 2, particularly the very first one described in Section 2.1 (Getting Started).

3.1 Interface Components

In this section the following interface components are described:

- File Menu, Edit Menu, Create Menu, View Menu, Help Menu
- Main Window, Viewer Window, Console Window
- File Dialog, Job Dialog, Preferences Dialog, Snapshot Dialog, System Information

3.1.1 File Menu

The file menu lets you load and save data objects as well as amira network scripts. In addition, it gives you access to amira's job dialog and allows you to quit the program. In the following text, all menu entries are discussed separately.

3.1.1.1 Load

The *Load* button activates amira's file dialog and lets you import data sets stored in a file. Most file formats supported by amira will be recognized automatically via the file header or the file name extension. For each file, the file dialog will display its format. If you try to load a file for which the format couldn't be detected automatically, an additional dialog pops up asking you to select the format manually. You may also manually set the file format for any file by selecting the file, activating the file dialog's popup menu using the right mouse button, and then choosing the *Format* option.

A list of all supported file formats is contained in the reference manual. Hints on how to import your own data sets are given in Section 4.1.

If you select multiple files in the file dialog, all of them will be loaded, provided all of them are stored in the same format. 2D images stored in separate files usually will be combined into a single 3D data object. On the other hand, there are some file formats which cause multiple data objects to be created. Finally, you can also import and execute amira network scripts using the *Load* button.

3.1.1.2 Load Time Series

This button also activates the file dialog, but in contrast to the ordinary *Load* option it is assumed that all selected files represent different time steps of a single data object. When loading such a time series an instance of a time series control module is created. This module provides a time slider allowing you to adjust the current time step. Whenever a new time step is selected the corresponding data file is read, and data objects associated with a previous time step are replaced. The module also provides a cache so that the data files only need to be read once provided the cache is large enough.

3.1.1.3 Save Data

The *Save Data* button allows you to save a single modified data object again using the same filename previously chosen under *Save Data As*. The button will only be active if the data object to be saved is selected and if this data object already has been saved using *Save Data As*. A common application of the *Save* button is to store intermediate results during manual segmentation in amira's image segmentation editor.

3.1.1.4 Save Data As

This button lets you write a data object into a file. To do so you must first select the data object (click on the corresponding green data icon). Then choose *Save Data As* to activate amira's file dialog. The file dialog presents a list of all formats suitable for saving that data object. Choose the one you like and press *OK*. Note that you must specify the complete file name including the suffix. amira will not automatically add a suffix to the file name. However, it will update the suffix whenever you select a new format from the file format list. Also, amira will ask you before it overwrites an existing file.

Some file formats create multiple files for a single data object. For example, each slice of a 3D image data set might be saved as a separate raster file. In this case, the file name may contain a sequence of hashmarks. This sequence will be replaced by consecutive numbers formatted with leading zeros.

If no file format at all has been registered for a certain type of data object, the *Save as* button will be disabled. It will also be disabled if more than one data object is selected in the object pool.

3.1.1.5 Save Network

This button allows you to save the complete network of icons and connections shown in the object pool. You need to specify the name of an amira network script in the file dialog. When executed, the network script restores all data objects and modules as well as the current object transformations and the camera settings. The feature is useful for resuming work at a point where it was left in a previous amira run.

Note that usually all data objects must have been stored in a file in order to be able to save the network. If this is not the case, a dialog is popped up listing all the data objects that still need to be saved. In the dialog you can specify that all required data objects should be saved automatically in a separate subdirectory.

Instead of the option *amira script* you can also choose *amira script and data files (pack & go)* from the file dialog's format menu. In this case *all* data objects currently loaded will be saved in a separate directory. More options affecting the export of network scripts can be adjusted in the preference dialog.

3.1.1.6 Recent Files

This button can be used to load recently used files. When choosing this menu entry a submenu appears listing the five most recent files. If multiple 2D images have been loaded this is indicated with the name of the first file followed by three periods (...).

3.1.1.7 Recent Networks

This button can be used to load recently used network scripts. When choosing this menu entry a submenu appears listing the five most recent network scripts.

3.1.1.8 Jobs

This button brings up amira's job dialog which is used to control the execution of batch jobs running in the background. For example, tetrahedral grids can be generated in a batch job (see module TetraGen). However, for most users the batch queue will be of minor interest.

3.1.1.9 Quit

This button terminates amira. The current network configuration will be lost unless you explicitly save it using *Save Network*.

3.1.2 Edit Menu

The *Edit* menu provides control over the visibility of object icons and lets you delete or duplicate objects. Depending on how many icons are selected in the Object Pool, some menu options might be disabled.

3.1.2.1 Hide

The *Hide* button hides all currently selected objects. The object's icons are removed from the Object Pool but the objects themselves are retained. You get the same effect by pressing the `Ctrl-H` key. Hidden objects can be made visible again using *Show* or *Show All*.

3.1.2.2 Remove

The *Remove* button deletes all selected objects and removes the corresponding icons from the Object Pool. You can get the same effect by pressing `Ctrl-X`. If you want to reuse a data object later on, be sure to save it in a file before deleting it. If a data object has been modified but has not yet been saved to a file, it is marked by a little asterisk in the object icon. In the Perferences dialog you can choose whether a warning dialog should be printed if you try to delete unsaved data objects which cannot be recomputed by an up-stream compute module. If you delete a data object, all connected modules will be deleted as well. However, if you delete a module connected data objects (e.g., the results of a compute module) will be retained.

3.1.2.3 Duplicate

The *Duplicate* button creates copies of all selected data objects. For each copy a new data icon is put in the Object Pool. The name of a duplicated data object differs from the original one by one or more appended digits. The duplicate option is not available if you have selected icons that do not represent data objects (e.g., display or compute modules).

3.1.2.4 Rename

This button allows you to change the name of a selected object in a small dialog box which is popped up when the button is pressed. If no object is selected or if multiple objects are selected the button is disabled. Note, that no two objects in amira can have the same name. Therefore, the name entered in the dialog may be modified by appending digits to it, if necessary.

3.1.2.5 Show

The *Show* button allows you to make hidden objects visible, so that their icons are displayed in the Object Pool. Among the hidden objects there are usually some colormaps which are loaded at start-up. This option will be unavailable if there are no hidden objects.

3.1.2.6 Show All

The *Show All* button makes all currently hidden objects visible, so that their icons are displayed in the Object Pool. This option will be unavailable if there are no hidden objects.

3.1.2.7 Remove All

The *Remove All* button deletes all currently visible icons and the associated objects from the Object Pool. A pre-loaded colormap that is currently visible is also deleted, but all hidden objects are retained. If you select the option *check if data objects need to be saved* in the Preferences dialog, a warning dialog is popped up if there are data objects which have not yet been saved to a file.

3.1.2.8 Database

The *Database* button activates an extended version of the Parameter Editor, allowing you to manipulate amira's global parameter database. Among others, the parameter database contains a set of predefined materials (to be used for image segmantation and surface reconstruction) and of predefined boundary ids (to be used for surface editing and FEM pre-processing). For example, for each material and for each boundary id a default color can be defined in the database.

Modification, insertion, and removal of parameters is performed in the same way as in the ordinary parameter editor. In addition, the database dialog provides a menu bar allowing you to load, import, save, or search the global parameter database. amira's default database is stored in the file `share/materials/database.hm` located in the directory where amira was installed. With the option *Set Default Database* an arbitrary other database file can be used instead. This change is permanent, i.e., it takes effect also if amira is restarted. To switch back to the system default, use the option *Use System Default* in the *Edit* menu.

3.1.2.9 Preferences

This option opens the amira preferences dialog described in Section 3.1.11. Among others, the dialog controls the way how network scripts are exported. It also lets you choose if warning dialogs should be popped up if you try to delete data objects which have not yet been saved to file, or if you try to exit amira without having saved the current network before.

3.1.3 Create Menu

The *Create* menu lets you create modules or data objects that cannot be accessed via the popup menu of any other object. The *Create* menu provides different categories like the popup menu in the object pool. For example, you can create a procedurally defined scalar field (where you can type in some arithmetic expression) by choosing *Scalarfield* from the *Data* sub-menu. The icon of a newly created object usually will not be connected to any other object in the Object Pool. In order to establish connections later on, use the popup menu over the small rectangular connection area of the object's icon. You can also put in links to scripts in the *Create* menu. Details are defined in Section 5.5 (Configuring popup menus).

3.1.4 View Menu

The *View* menu provides control over several *Viewer* options affecting the display independent of the *Viewer* input.

3.1.4.1 Layout

The *Layout* button lets you select between one, two, or four 3D viewers. All viewers will be placed inside a common window using a default layout. If you want to create an additional viewer in a separate window, choose *Extra Viewer*. You may create even more viewers using the Tcl command `viewer <n> show`. Starting from *n=4*, viewers will be placed in separate windows.

3.1.4.2 Background

The *Background* button opens the background dialog, allowing you to switch between the different background styles *uniform*, *gradient*, and *checkerboard*. In addition, the dialog allows you to adjust the two colors used by these styles.

In order to change the background color via the command interface use the viewer commands `viewer <n> setBackgroundColor` and `viewer <n> setBackgroundColor2`. The command interface also allows you to place an arbitrary raster image into the viewer background (see Section 3.1.7, viewer commands).

3.1.4.3 Transparency

The *Transparency* button controls the way of calculating pixel values with respect to object transparencies during the rendering process.

- *Screen Door:* Transparent surfaces are approximated using a stipple pattern.
- *Add:* Additive alpha blending.
- *Add Delay:* Additive alpha blending with two rendering passes. Opaque objects come first and transparent objects come second.
- *Add Sorted:* Like *Add Delay*, but transparent objects are sorted by distances of bounding box centers from the camera and are rendered in back to front order.
- *Blend:* Multiplicative alpha blending.
- *Blend Delay:* Multiplicative alpha blending with two rendering passes. Opaque objects come first and transparent objects come second.
- *Blend Sorted:* Like *Blend Delay*, but transparent objects are sorted by distances of bounding box centers from the camera and are rendered in back to front order.

3.1.4.4 Lights

The *Lights* menu lets you activate different light settings for the 3D viewer. By default, the viewer uses a single headlight, i.e., a directional light pointing in almost the same direction as the camera is looking. The headlight can be switch on or off in each viewer via the viewer's popup menu. Alternatively, the headlight can be switched on or off for all viewers using the headlight toggle in this *Lights* menu. This standard light settings can be restored using the *Standard* button. More light settings can be defined by creating appropriate file in `$AMIRA_ROOT/share/lights`. On default, amira provides one additional light setting including colored lights (*BlueRed*).

At any time, additional lights can be created via the *Create light* option. Except for the viewer's default headlight, all lights are represented by little blue icons in the Object Pool, just like ordinary data objects or modules. In order to make all hidden light icons visible, use the *Show all icons* option. *Hide all icons* hides the icons of all light objects. For more information about lights, please refer to the Reference Section of this manual.

3.1.4.5 Fog

The *Fog* button introduces a fog effect into the displayed scene and controls how opacity increases with distance from the camera. The fog effect will only be seen on a *uniform* background. More fine tuning is provided by the *fogRange* Viewer command.

- *None:* No fog effect (default).
- *Haze:* Linear increase in opacity with distance.
- *Fog:* Exponential increase in opacity with distance.
- *Smoke:* Exponential squared increase in opacity with distance.

3.1.4.6 Axis

The *Axis* button creates an Axis module named *GlobalAxis* which immediately displays a coordinate frame in the viewer window. This button is a toggle, so clicking on it again deletes the *GlobalAxis* module and removes the coordinate frame from the viewer window. The axes will be centered at the origin of the world coordinate system. You may also create local axes by selecting the appropriate entry from a data object's popup menu.

3.1.4.7 Fading effect

The *Fading effect* toggle lets you switch on a fading effect which is applied to all kinds of scene movements. Before a new image is rendered only a certain fraction of the background will be cleared. In this way older images remain visible until they fade out after a while. Note that this mode requires single buffer rendering, and therefore, flickering may be visible in some cases.

3.1.5 Online Help

amira user's documentation is available online. You can access it via the *User's Guide* entry of the main window's *Help* menu. The user's guide contains some introductory chapters, as well as a reference part containing documentation for specific

- modules,
- data types,
- editors,
- file formats,
- and other components.

You may access the documentation of any such object via a separate index page accesible from the home page of the online help browser. amira modules also provide a question mark button in the working area. Pressing this button directly pops up the help browser for the particular module.

Going through the online documents is similar to text handling within any other hypertext browser. In fact, the documentation is stored in HTML format and can be read with a standard web browser as well. Some specially marked (colored and underlined) text items allow you to jump quickly to related or referenced topics, where blue items point to unread sections, and red items to already viewed sections. Use the *Backward* and *Forward* buttons to scroll in the document history and *Home* to move to the first page.

Searching the online documentation
The online help browser provides a very simple interface for a full text search. For example, if you are looking for information about the *surface editor*, type these two words into the text field in the upper part of the help window. Then, press the search button in order to perform the search. If you want to look for multiple words, you must prepend them with a plus sign, e.g., *surface +editor*.

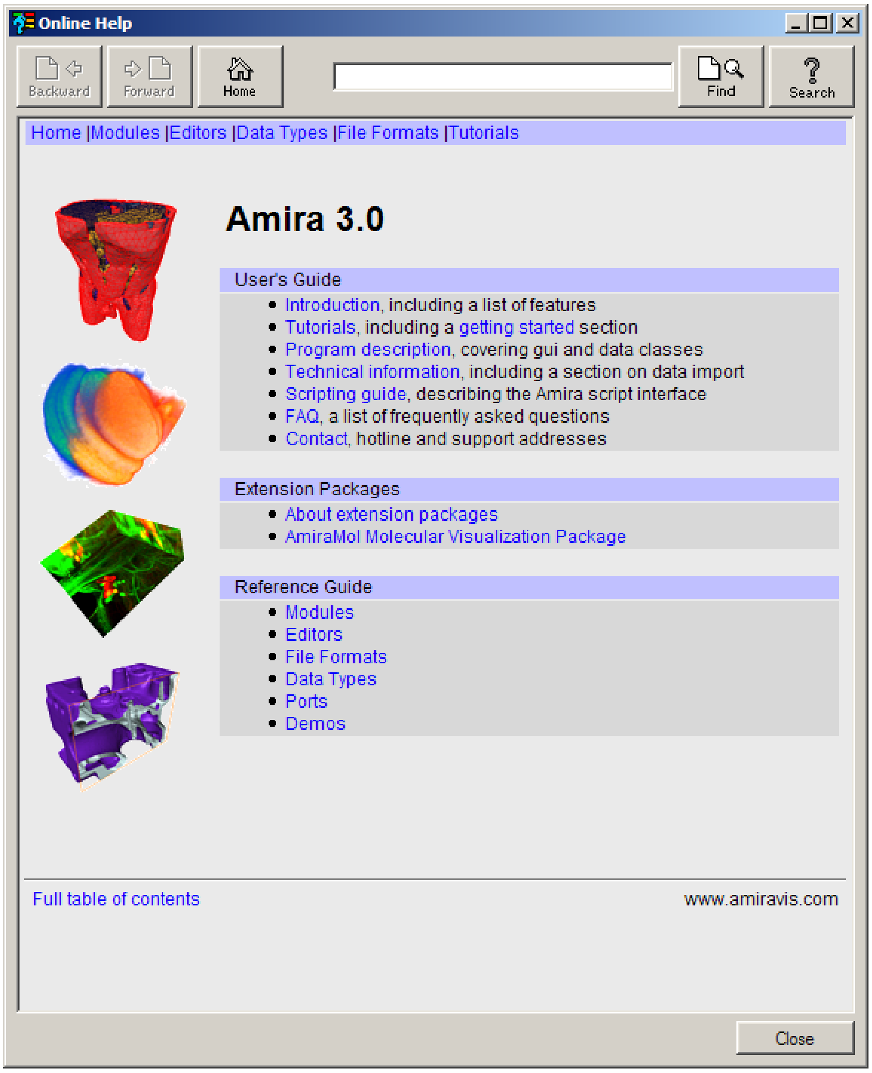
Running demo scripts
In the demo section of the on-line manual you can easily start any demonstration just by clicking on the marked text. The script will be loaded and executed immediately. You may interrupt running demo scripts by using the stop button in the lower right of the amira main window.

Commands

`help`
Makes the help dialog appear and loads the home page of the online help.

`help getFontName`
Returns the name of the font of the browser.

`help setFontName`
Sets the font of the browser.



Home |Modules |Editors |Data Types |File Formats |Tutorials

Amira 3.0

User's Guide

- Introduction, including a list of features
- Tutorials, including a getting started section
- Program description, covering gui and data classes
- Technical information, including a section on data import
- Scripting guide, describing the Amira script interface
- FAQ, a list of frequently asked questions
- Contact, hotline and support addresses

Extension Packages

- About extension packages
- AmiraMol Molecular Visualization Package

Reference Guide

- Modules
- Editors
- File Formats
- Data Types
- Ports
- Demos

Full table of contents www.amiravis.com

Figure 3.1: amira's help window.

`help getFontSize`
Returns the size of the font of the browser.

`help setFontSize`
Sets the size of the font of the browser. In order to permanently change the font size, put this command in the `.amira` file in your home directory or in an `amira.init` file in the current working directory. For details see Section 4.4.

`help load` *`file.html`*
Load the specified hypertext document in the file browser. Note that only a subset of HTML is supported.

`help reload`
Reload the current document.

3.1.6 Main Window

amira's main window consists of two components, the *Object Pool* in the upper part of the window and the *Working Area* in the lower part. The Object Pool contains icons representing data objects and modules currently in use as well as lines connecting icons indicating dependencies between objects and modules. The Working Area is the place where the user interface of selected objects is displayed. Typically, the interface consists of buttons and sliders arranged in *Ports*. They can be thought of as ports because the user can pass information to a module solely through them.

3.1.6.1 Object Pool

Once a data object has been loaded or a module has been created it will be represented by an icon in the Object Pool. Some objects, especially colormaps, may not be visible here. Such hidden objects are listed in the *Edit Show* menu of the main window. Selecting an object from this menu causes the corresponding icon to be made visible in the Object Pool.

Icon colors indicate different object types. Data objects are shown in green, computational modules in red, and visualization modules in yellow. Orange icons represent visualization modules of *slicing* type. Such modules may be used to clip the graphical output of any other module. Connections between data objects and processing modules, shown as blue lines, represent the flow of data. You may connect or disconnect objects by picking and dragging a blue line between object icons.

As you might expect, not all types of processing modules are applicable to all kinds of data objects. If you click on an icon with the right mouse button, a menu pops up that shows all types of modules that can be connected to that object. Selecting one of them will automatically create an instance of that module type and connect it to the data object. A new icon and a connecting line will appear in response. This way you can set up a more or less complex network that represents the computational steps required to carry out a specific visualization task and is indeed used to trigger them.

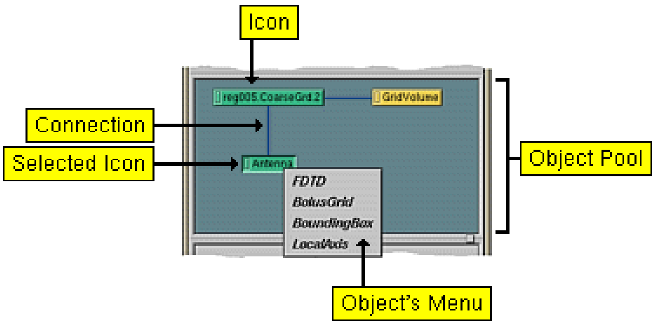


Figure 3.2: The object pool contains data objects and module icons.

If you look closer at an object's icon you will notice a tiny rectangle on its left. If you click on it with the right mouse button a menu pops up that shows all connection ports of that object. As mentioned above, for most objects the required connections are automatically established on creation. However, in order to set up optional connections you must use the connection popup menu. For example, you may attach an optional scalar field to an Isosurface module in order to let its values on an existing isosurface be encoded by colors. The colormap used for pseudo-coloring is specified by another connection port.

Once you have selected an entry from the connection popup menu, you can choose a new input object for that port. In order to do so, click on the input object's icon in the Object Pool. The blue connection line will become yellow if the connection port can be connected to the chosen object. In order to disconnect an input object click on the icon of the module the port belongs to. Data objects possess a special connection port called *Master*. This port refers to a computational module or editor the data object is attached to. It indicates that the computational module or editor controls the data object, i.e., that it may modify its contents.

Each object has an associated control panel containing buttons and sliders for setting or changing additional parameters of the object. The control panel becomes visible once the object has been selected, i.e. by clicking on its icon with the right mouse button. In order to select multiple objects you must shift-click the corresponding icons. Clicking on the icon of a selected object deselects it again. Clicking somewhere on the background of the Object Pool causes all selected objects to be deselected. An icon may be dragged around in the Object Pool by clicking on it and moving the mouse pointer while holding down the mouse button.

3.1.6.2 Working Area

Once an object has been selected, its input controls will be displayed in the Working Area below the Object Pool. Each object has a specific set of controllable parameters or options. These are described in detail for each module in the index section of the reference manual. Computational modules and visualization modules also provide a question mark button which lets you access the documentation of that module directly.

At the top of an object's control panel its name is displayed and a number of additional control buttons are provided. All objects have one or more orange viewer buttons for each 3D viewer. These buttons control whether any graphical output of an object is displayed in a particular viewer or not. For example, if you have two viewers and two isosurface modules you may want to display one isosurface in each viewer.

Display modules of *slicing* type (orange ones) provide a clip button. Clicking this button will cause the graphical output of any other module to be clipped by that slice. Clipping does not affect modules with hidden geometry or modules that are created after the clip button has been pressed.

Data objects provide a number of additional *editor* buttons. Editors are used in order to modify the contents of a data object interactively. For example, you can perform manual segmentation of 3D image data by editing label fields using the image segmentation editor. Some editors display their controls in the working area like all other objects, while others use a separate dialog window that allows you to perform object manipulations.

As already mentioned, specific input controls of an object or a module are organized in *Ports*. Each port has a pin button on its left. If a port is pinned it will still be visible even when the object is deselected. The ports are composed of various widgets that reflect an operational meaning, e.g., a value is entered by a slider, a state is set by radio buttons, a binary choice is presented as a toggle button. The control elements have a uniform layout and are divided into several basic types. A description of the basic port types is contained in the component index section of the User's Reference Manual.

3.1.7 Viewer Window

The 3D viewer plays a central role in amira. Here all geometric objects are shown in 3D space. The 3D viewer offers powerful and fast interaction techniques. It can be regarded as a virtual camera which can be moved to an arbitrary position within the 3D scene. The left mouse button is used to change the view direction by means of a virtual trackball. The middle mouse button is used for panning, while the left and the middle mouse button pressed together allow you to zoom objects.

Sometimes you need to manipulate objects directly in the 3D viewer. For example, this technique, called 3D interaction, is used by the transform editor. The editor provides special draggers that can be picked and translated or rotated in order to specify the transformation of a data object. Before you can interact with these draggers, you must switch the viewer into *interaction mode*. This is done by clicking on the arrow button in the upper right corner. If the viewer is in interaction mode, the mouse cursor will be an arrow instead of a hand symbol. You can use the `[ESC]` key in order to quickly switch between interaction mode and viewing mode. If the viewer is in interaction mode, use the `[Alt]` key to temporarily switch to viewing mode.

More than one viewer can be active at a time. Standard screen layouts with one, two, or four viewers can be selected via the view menu. Additional viewers can be created using the Tcl command `viewer <n> show`, where `<n>` is an integer number between 0 and 15. While viewers 0 to 3 will be placed

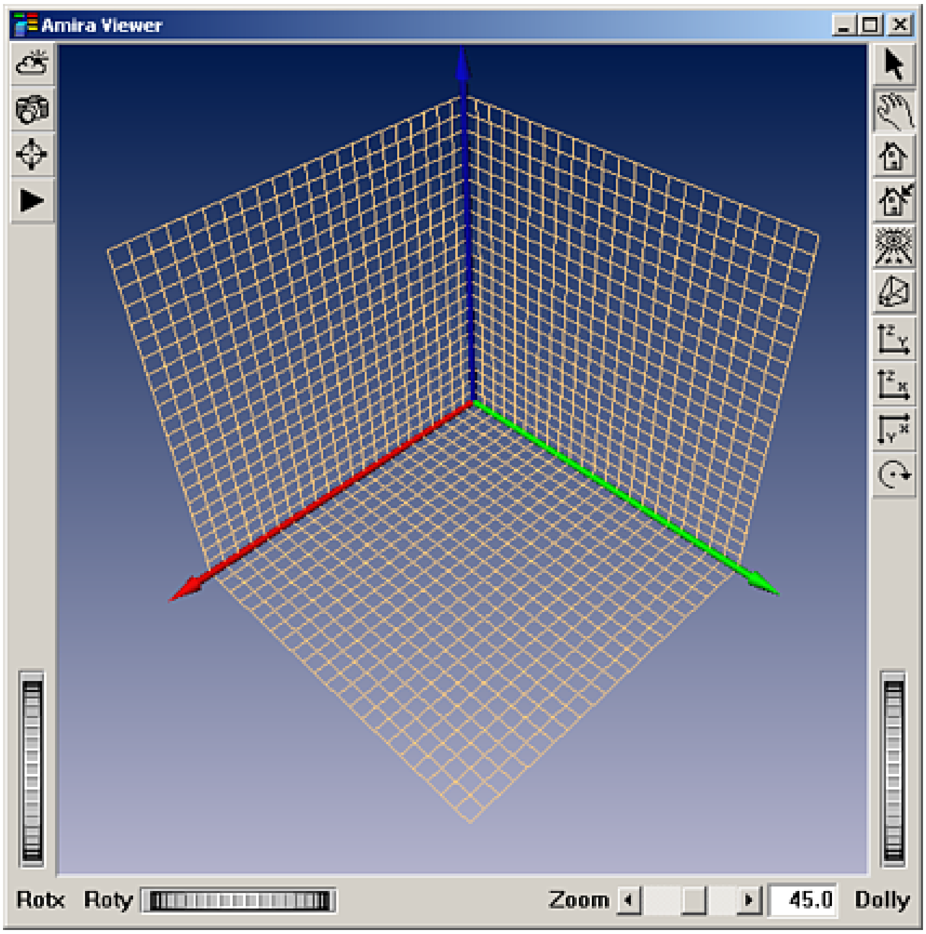
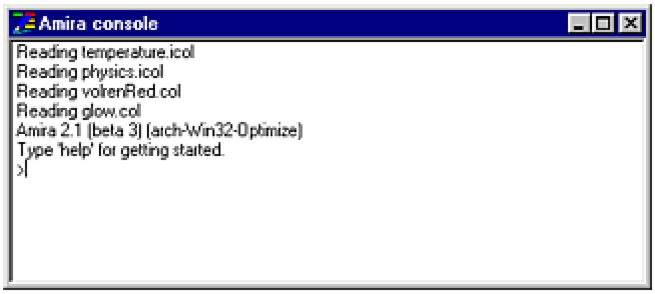


Figure 3.3: amira's viewer window provides a virtual trackball for easy navigation. The decoration frame contains several controls, allowing you for example to switch between viewing mode and interaction mode, to choose certain orienations, or to take snapshots.

in a common panel window, viewers 4 to 15 will create their own top-level window. For more specific control, the viewer provides an extensive command set, which is documented in Section 5.3.3.1.

The decoration of the viewer window provides several buttons and controls, see Figure 3.3. The precise meaning of these controls is described below.

- *Edit Background Color:* Pops up the *Background Dialog* that allows you to change the appearance of the viewer background. In *Mode* you may choose between *uniform*, *gradient*, and *checkerboard* mode. In *Color* you may specify the colors by pressing the buttons adjacent to the *color 1* and *color 2* labels in which case a Color Editor pops up. The *Swap* button exchanges the two colors used with non-uniform backgrounds and the *Reset* button restores the default.
- *Snapshot:* Takes a snapshot of the current rendering area and saves it in a file. The filename as well as the desired output format have to be entered through the snapshot dialog. Snapshots may also be taken using the viewer command `snapshot`.
- *Seek:* Pressing the seek button and then clicking on an arbitrary object in the scene causes the object to be moved into the center of the viewer window. Moreover, the camera will be oriented parallel to the normal direction at the selected point. Seeking mode may also be activated by pressing the `[S]` key in the viewer window.
- *Home:* Resets camera to the home position.
- *Set Home:* Sets the current position as the new home position.
- *Pick:* Switches the viewer into interaction mode. You can also use the `[ESC]` key to toggle between viewing mode and interaction mode.
- *View:* Switches the viewer into viewing mode. You can also use the `[ESC]` key to toggle between interaction mode and viewing mode.
- *Home:* Resets camera to the home position.
- *Set Home:* Sets the current position as the new home position.
- *Rotate:* Rotates the camera around the current view direction. By default, a clockwise rotation of one degree is performed. If the Shift-key is pressed while clicking, a 90 degree rotation is done. If the Ctrl-key is pressed, the rotation will be counterclockwise.
- *Measuring:* Pressing this button creates an instance of a Measuring module that lets you measure distances and angles on objects within the viewer.
- *View All:* Repositions the camera so that all objects become visible. The orientation of the camera will not be changed.
- *Perspective/Ortho:* Toggles between a perspective and an orthographic camera. By default, a perspective camera is used. You may want to use an orthographic camera in order to measure distances or to exactly align objects in 3D space.
- *YZ-, XZ- and XY-Views:* Adjusts the camera according to the specified viewing direction. The viewing direction is parallel to the coordinate axis perpendicular to the specified coordinate plane. Medical doctors are used to viewing series of tomographic images parallel to the XY-plane with the y-axis pointing downwards. This convention is followed by the XY-button. The opposite view direction is used if the Shift key is pressed.



```
Reading temperature.icol
Reading physics.icol
Reading volrenRed.col
Reading glow.col
Amira 2.1 (beta 3) (arch-Win32-Optimize)
Type 'help' for getting started.
>
```

Figure 3.4: amira's console window displays info messages and lets you enter Tcl commands.

In addition to theses buttons the amira viewers provide an extensive set of Tcl commands, which are listed in Section 5.3.3.1.

3.1.8 Console Window

The *console window* is a command shell allowing to access amira's advanced control features. It serves two purposes. First, it gives you some feedback on what is currently going on. Such feedback messages include warnings, error indications and notes on problems as well as information on results. Second, it provides a command line interface where amira commands can be entered.

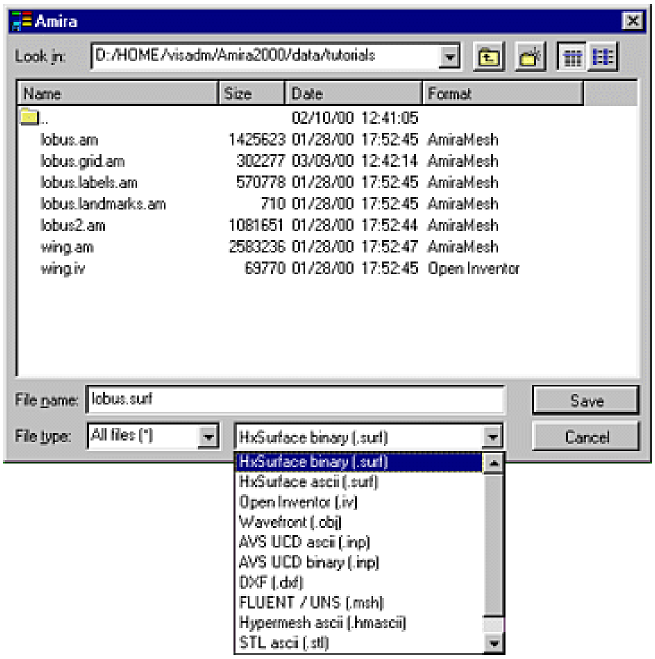
amira's console commands are based on the *Tcl* script language *(Tool Command Language)*. Examples are:

```
load C:/MyData/something.am
viewer 0 setSize 200 200
viewer 0 snapshot C:/snapshot.tif
```

The amira scripting syntax and the specific commands are described in the Chapter 5 (Scripting). To execute a single console command just type in its name and arguments and press 'Enter'. If you select an object and then press the [TAB] key on the empty command line, then the name of the object will be automatically inserted.

You can also type the beginning of a command word and type the [TAB] key to complete the word. This only works if the beginning is unique. Pressing [TAB] a second time will show the possible completions. Often, this saves a lot of typing. Commands provided by data objects and modules are documented in the reference section of the users guide. Pressing the [F1] key for such a command without any arguments pops up the help text for this command. This is also true for commands provided by the ports of an object.

Additionally the *console window* provides a command history mechanism. Use 'up arrow' and 'down arrow' to scroll up and down in the history list.



Look in: D:/HOME/visadm/Amira2000/data/tutorials

Name	Size	Date	Format
..		02/10/00 12:41:05	
lobus.am	1425623	01/28/00 17:52:45	AmiraMesh
lobus.grid.am	302277	03/09/00 12:42:14	AmiraMesh
lobus.labels.am	570778	01/28/00 17:52:45	AmiraMesh
lobus.landmarks.am	710	01/28/00 17:52:45	AmiraMesh
lobus2.am	1081651	01/28/00 17:52:44	AmiraMesh
wing.am	2583236	01/28/00 17:52:47	AmiraMesh
wing.iv	69770	01/28/00 17:52:45	Open Inventor

File name: lobus.surf

File type: All files (*) HxSurface binary (.surf)

HxSurface binary (.surf)
HxSurface ascii (.surf)
Open Inventor (.iv)
Wavefront (.obj)
AVS UCD ascii (.inp)
AVS UCD binary (.inp)
DXF (.dxf)
FLUENT / UNS (.msh)
Hypermesh ascii (.hmascii)
STL ascii (.stl)

Figure 3.5: amira's file dialog.

To execute a file containing many Tcl commands use `source <filename>` or load the script file via amira's file dialog from the file menu. amira script files are usually identified by the extension `.hx`. For advanced script examples take a look at amira's demo files located in `$AMIRA_ROOT/share/demo`.

3.1.9 File Dialog

The *File Dialog* is the user interface component for importing and exporting data into resp. from amira. It is used at several places in amira, most prominently by the *Load*, *Save Data As*, and *Save Network* items of the main window's *File* menu.

The dialog provides two modes of information, a detail mode and a multi-column mode. In the detail mode, which is active by default, some file data are shown next to each filename, namely the file size, the file's last modification time, and the file format. You may sort the file list according to each of these properties by clicking on the particular column's header bar. Subdirectories will always be displayed first. In multi-column mode only the file name is displayed. You may switch between both modes using the tool buttons in the upper right part of the dialog window.

Most file formats supported by amira will be recognized automatically, either by analyzing the file header or by looking at the file name suffix. A list of all supported file formats is contained in the reference section of this manual. You may manually set the format of a file by means of the dialog's popup menu (see below).

3.1.9.1 Changing Directories

You can change the current directory by double-clicking a subdirectory in the file list or by entering a new directory in the dialog's path list. By default, the path list contains the current directory, the directory containing the demo data sets provided with amira, as well as all directories defined by the environment variable `AMIRA_DATADIR`. In `AMIRA_DATADIR` multiple directory names have to be separated by colons `[:]` on Unix systems or by semicolons `[;]` on Windows systems. In addition, on Windows system the names of the twelve most recently visited directories are stored in the path list.

3.1.9.2 Selecting Files

To select a single file just click on it or type in its name in the file name text field.

In some cases you might want to select more than one file at once, e.g., when loading a 3D image data set as a series of single 2D images. You can do this by selecting the first file first and then shift-selecting the last file. Then all intermediate files will be selected as well. Moreover, you may ctrl-click a file in order to toggle its selection state individually.

3.1.9.3 Using the Filename Filter

The filename filter is visible when the dialog is in import mode (*Load File*). It is useful to restrict the list of filenames to a subset matched by the filter expression. The filter expression may contain the wildcard characters `?` (matches any character) and `*` (matches an arbitrary character sequence). For example, the expression `*.img` matches all filenames with the suffix `.img`.

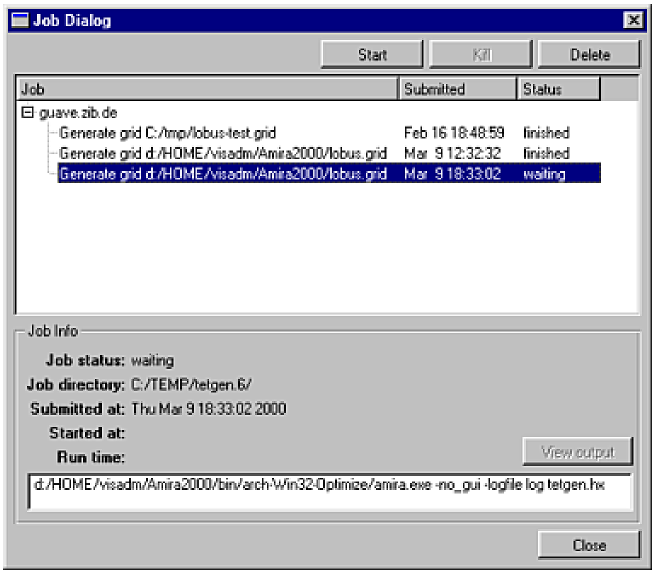
3.1.9.4 The File Dialog's Popup Menu

The file dialog provides a popup menu which may be activated by pressing the right mouse button over the file list. Among others, this menu lets you rename or delete files or directories, provided you have the permission to do that. Note that you may only delete empty directories.

Using the *Format* option of the popup menu you may manually set the format to be used when loading a file into amira. This option is useful if for some reason the wrong format has been detected automatically, or if no format at all could be detected. Note however, that any format specification set manually will be overwritten when the directory is reread the next time.

3.1.10 Job Dialog

Certain time-consuming operations in amira can be performed in batch mode. For this purpose amira provides a job queue, where jobs like generation of a tetrahedral grid can be submitted. You can inspect



Job	Submitted	Status
guave.zib.de		
Generate grid C:/tmp/lobus-test.grid	Feb 16 18:48:59	finished
Generate grid d:/HOME/visadm/Amira2000/lobus.grid	Mar 9 12:32:32	finished
Generate grid d:/HOME/visadm/Amira2000/lobus.grid	Mar 9 18:33:02	waiting

Job status: waiting
Job directory: C:/TEMP/tetgen.6/
Submitted at: Thu Mar 9 18:33:02 2000
Started at:
Run time:

```
d:/HOME/visadm/Amira2000/bin/arch-Win32-Optimize/amira.exe -no_gui -logfile log tetgen.hx
```

Figure 3.6: The job dialog lets you start, stop, examine, and delete batch jobs.

the current status of the job queue, start and delete jobs from the queue by selecting *Jobs* from amira's file menu. This will bring up the *Job Dialog*.

In the upper part of the job dialog the current list of jobs of a user is shown. For each job a short description is displayed, as well as the time when the job has been submitted and the current state of the job. A job may be waiting for execution, running, finished, or it may have been killed.

The job directory

For each job a temporary directory is created containing any required input data, scripts, state information, and log files. On Unix systems this directory is created at the location specified by the environment variable `TMPDIR`. If no such variable exists, `/tmp` is used. On Windows systems the default temporary directory is used. Typically this will be `C:/TEMP`.

Controlling the job queue

A job's state may be manipulated using the action buttons shown above the job list. In order to start the job queue select the first job waiting for execution and then press the *Start* button. Note that only one job can be executed at a time. In order to kill a running job, select it in the job list and press the *Kill* button. You may delete a job from the job queue using the *Delete* button. When deleting a job the temporary job directory will be removed as well.

Information about a job

Once you have selected a job in the job queue, more detailed information about it will be displayed in the lower part of the dialog window, notably the state of the job, the temporary job directory, the submit time, the time when the job has been started, the run time, and the name of the command to be executed. Any console output of a running job will be redirected to a log file located in the temporary job directory. Once such a log file exists and has non-zero size you may inspect it by pushing the *View output* button.

Commands

`job submit` *cmd info [tmpdir]*
Submits a new job to the job queue. `command` specifies the command to be executed. `info` specifies the info string displayed in the job dialog. `tmpdir` specifies the temporary job directory. If this argument is omitted a temporary job directory is created by amira itself. In any case, the directory will be automatically deleted when the job is removed from the job queue. Example: `job submit "clock.exe" "Test job"`

`job run`
Starts the first job in job queue pending for execution. When a job is finished, execution of the next job in the queue starts automatically, thus all jobs in the queue will be executed successiveley by `job run`.

3.1.11 Preference Dialog

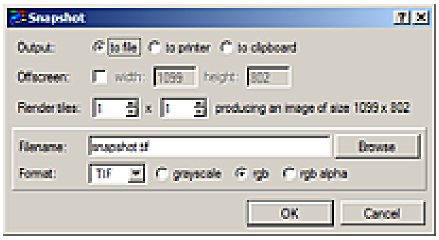
The *Preference Dialog* allows you to adjust certain global settings of amira. The preferences are stored in a permanent fashion on a per-user basis, i.e., changes take effect after restarting amira. The preference dialog is subdivided into two sections. The first one is related to the user interface, while the second one affects the way how network scripts are exported.

Draw compute indicator
If set a small red rectangle is drawn inside the icon of a module to indicate that the module is currently working. The default is on.

Auto-select new modules
If set a new module selected from the popup menu of its parent object is shown automatically in the work area. The default is on.

Deselect previously selected modules
This option can only be set if auto-selection is turned on. If set all objects are deselected before selecting the new module. Otherwise the new module will be appended at the end of the work area. The default is on.

Draw viewer toggles on icons
If set small viewer mask toggles are drawn on the icons of data objects and display modules. This allows to show or hide a module in a viewer without selecting it first. The default is on.


Snapshot

Output: [✓ to file] to printer to clipboard

Offscreen: ☐ width: 1099 height: 802

Render tiles: 1 x 1 producing an image of size 1099 x 802

Filename: snapshot.tif Browse

Format: TIF grayscale [✓ rgb] rgb alpha

OK Cancel

Figure 3.7: The snapshot dialog allows you to save or print the contents of a viewer window.

2-pass firing algorithm
If set a slightly more complex firing algorithm is used which ensures that down-stream modules connected to an up-stream object via multiple paths are only fired once if the up-stream object changes. The default is off.

Include unused data objects
If set all data objects including hidden colormaps are stored in a network scripts. When executing such a script all existing objects are removed first. If not set only visible data objects and objects which are referenced by others are stored in a network script. When executing the script hidden data objects are not removed. The default is on.

Overwrite existing files in auto-save
If set no overwrite check is performed for data objects which need to be saved automatically in order to create a network script. Otherwise a unqiue file name will be chosen. The default is on. Details about the auto-save feature are described in Section 3.1.1.5.

3.1.12 Snapshot Dialog

The *Snapshot Dialog* provides the user interface of the viewer's snapshot facility. You get the dialog by clicking on the camera icon in the left toolbar of the viewer.

- **Output:** Specifies the output device. With *to file* the grabbed image is saved to a file, with *to printer* the image is sent directly to the printer, and with *to clippboard* it is sent to the clippboard. In the *to printer* mode you first have to select and configure a printer by pushing the *Configure* button. In addition, you may enter an arbitrary text string which is printed as an annotation text below the snapshot image.
- **Offscreen:** Lets you grab images larger than the actual screen size. When this option is checked, the output dimensions can be specified in the *width:* and *height:* textfields up to a maximum of 2048 x 2048 pixels.
- **Render tiles:** Use this option to render snapshots of virtually unlimited resolution (e.g. for high quality printouts). In this mode the scene is divided into n x m tiles where n and m can be entered into the adjajent text fields. Then the camera position is set such that each tile fills the

current viewer and a snapshot is taken. Finally the tiles are internally merged to a single image and sent to the device specified in the *Output:* port.

- **Filename:** Lets you specify the filename if the *to file* option is set. The *Browse* button allows you to browse to a desired location within the filesystem.
- **Format:** The format option lets you select the file format (EPS or Raster Image) to be produced for file output. If raster image has been selected the file format will be determined from the file name suffix. The following formats are supported: TIFF (`.tif,.tiff`), SGI-RGB (`.rgb,.sgi,.bw`), JPEG (`.jpg,.jpeg`), PNM (`.pgm,.ppm`), BMP (`.bmp`), PNG (`.png`), and Encapsulated Postscript (`.eps`). In addition, this port offers three radio buttons to choose between *grayscale*, *rgb*, and *rgb alpha* type of raster images. If *rgb alpha* option is set images are produced such that the viewer background is assigned to the alpha channel. This option is not available for file formats that do not support an alpha channel.

3.1.13 System Information Dialog

The system information dialog provides diagnostics information allowing the user or the amira support team to better analyse software problems. The dialog contains a tab bar with two pages. The first page lists information about the current OpenGL graphics driver. The second page lists version information about the currently installed amira components. In the lower left part of the dialog you find a button *Save Report*. With this button all information can be written into a text file. In case of a support call you may be asked to send this text file to the hotline.

3.1.13.1 The OpenGL Tab

This page displays information about the current OpenGL graphics driver. In particular, a list of available OpenGL extensions is printed. This list allows it to check if certain rendering techniques like direct volume rendering via 3d textures are supported on a particular hardware platform or not.

3.1.13.2 The Libraries Tab

This page displays a list of all DLLs or shared libraries contained in the amira lib directory. For each library certain version informations as well as an MD5 checksum are printed. In this way it is possible to check whether a certain patch has already been installed or not. For most libraries the version information is compiled in. For other libraries this information is read from the version information files found in `share/versions` in the amira root directory.

3.2 General Concepts

This section contains some general comments on how data objects are organized and classified in amira. In particular, the following topics are discussed:

3.2.1 Class Structure

In this section we discuss the object-oriented design of amira in a little more detail. You already know that data objects, e.g., grey level image data or vector field sets, appear as separate icons in the *Object Pool*. You also know that there are certain display modules which can be used to visualize the data objects. While some modules can be connected to many different data objects, e.g., the *Bounding Box* module, others cannot, e.g., the *Ortho Slice* module. The latter can only be connected to voxel data or to scalar distributions on voxel grids. The reason is that internally both are represented as a scalar field with uniform Cartesian coordinates. Consequently, the same visualization methods can be applied to both. On the other hand, for example a volumetric tetrahedral grid model of the object of interest usually looks completely different. But since it is also a 3D data object, the same *Bounding Box* module can be connected to it.

In summary, there are amira data objects that might be conceived of different type, but with respect to mathematical structure, applicability of viewing and other processing modules, as well as programming interface design have many common properties. Obeying principles of object-oriented design, the data types of amira are organized in class hierarchies where common properties are attributed to ’higher up’ classes and inherited to ’derived’ classes, as sub-classes of a class are commonly referred to. Conceptually each object occuring in amira is an instance of a class and each of its predecessors in the hierarchy that the class belongs to. The classes and their hierarchies are defined within amira. As the user you normally deal with instances of classes only. For instance, there is a class called ”HxObject” with sub-classes ”HxData” and ”HxModule”. ”HxData” comprises the types of data associated with data objects used for modeling the objects of interest, e.g., volumetric tetrahedral grids or surfaces. ”HxModule” comprises data types that have been assigned to display and other processing modules, again in accordance with principles of object-oriented design. This is why amira’s data objects and processing modules are commonly referred to as ”objects”.

There are also classes in amira that are not derived from ”HxObject” and constitute other data types, and there are several independent class hierarchies. e.g., there is a class called ”HxPort” from which all classes supporting the operation and display of interface control elements are derived (see section Working Area and the List of Ports in the index section of the user’s guide).

A single class hierarchy is usually figured as an upside-down tree, i.e. with the root at the top. Thus the *data* class tree is the one to which the information as to which processing module is applicable to which data object is hooked. Its classes reflect the mathematical structure of the object models supported by amira. For example, scalar fields and vector fields are such structures and derived from a common ”field” class which represents a mapping $R^3 \rightarrow R^n$. Deriving a sub-class from this base class requires a value to be specified for n.

At the same time fields defined on Cartesian grids are distinguished from fields defined on tetrahedral grids, i.e., this distinction is part of the classification scheme that gives rise to branches in the *data* class subtree. In the next section of this chapter you will learn more about the *data* class hierarchy. In the second section we discuss how some data types frequently used for various visualization tasks fit into it.

Internally, all class names begin with a prefix *Hx*. However, you don't have to remember these names, unless you want to use the command shell to create objects. For example, a bounding box is usually created by choosing the *BoundingBox* item from the pop-up menu of a data object that is to be visualized, but you may also create it by typing `create HxBoundingBox` in the command window.

3.2.2 Scalar Field and Vector Fields

The most important fields in amira are three-dimensional ones. These fields are defined on a certain domain $\subseteq \mathbb{R}^3$. A field can be evaluated at any point inside its domain. If the field is defined on a discrete grid, this usually involves some kind of interpolation.

3.2.2.1 Scalar Fields

A 3D scalar field is a mapping $R^3 \rightarrow R$. The base class of all 3D scalar fields in amira is *HxScalarField3*. Various sub-classes represent different ways of defining a scalar field. There are a number of visualization methods for them, for example pseudo-coloring on cutting planes, iso-surfacing, or volume rendering. However, many visualization modules in amira rely on a special field representation. Therefore, they can only operate on sub-classes of a general scalar field. Whenever a given geometry is to be pseudo-colored, any kind of scalar field can be used (cf. Colorwash, GridVolume, Isosurface).

The class *HxTetraScalarField3* represents a field which is defined on a tetrahedral grid. On each grid vertex a scalar value, e.g., a temperature, is defined. Values associated to points inside a tetrahedron are obtained from the four vertex values by linear interpolation. This class does not provide a copy of the grid itself, instead a reference to the grid is provided. This is indicated in the Object Pool by a line which connects the grid icon and the field icon. As a consequence, a field defined on a tetrahedral grid cannot be loaded into the system if the grid itself is not already present.

The class *HxRegScalarField3* represents a field which is defined on a regular Cartesian grid. Such a grid is organized as a three-dimensional array of nodes. In the most simple case these nodes are axis-aligned and have equal spacings. The coordinates of such a uniform grid can be obtained from a simple bounding box containing the origin vector and increments for all directions. Stacked coordinates are another example. Here the spacing in z-direction between subsequent slices may be different. In any case scalar values inside a hexahedral grid cell are obtained from the eight vertex values using trilinear interpolation. While the OrthoSlice module can only be used to visualize scalar fields with uniform or stacked coordinates, other modules like ObliqueSlice or Isosurface work for all scalar fields with regular coordinates.

Yet another example of a scalar field is the class HxAnnaScalarField3. It represents an analytically defined scalar field. To create such a field, select *ScalarField* from the *Edit Create* menu of amira's main window. You have to specify a mathematical expression which is used to evaluate the field at each requested position. Up to three other fields can be connected to the object. These can be combined to a new scalar field, even if they are defined on different grids.

3.2.2.2 Vector Fields

As for scalar fields amira provides a number of vector field classes, these are derived from the base classes *HxVectorField3* and *HxComplexVectorField3*. While ordinary vector fields return a three-component vector at each position, complex vector fields return a six-component vector. Complex vector fields are used for encoding stationary electromagnetic wave pattern as required by some applications. Usually complex vector fields are visualized by projecting them into the space of reals using different phase offsets. The Vectors module even allows you to animate the phase offset. In this way a nice impression of the oscillating wave pattern is obtained.

3.2.3 Coordinates and Grids

amira currently supports two important grid types, namely grids with hexahedral structure (regular grids), and unstructured tetrahedral grids. Other types, e.g., unstructured grids with hexahedral cells or block-structured grids will be added in future releases of amira.

3.2.3.1 Regular Grids

A regular grid consists of a three-dimensional array of nodes. Each node may be addressed by an index triple *(i,j,k)*. Regular grids are further distinguished according to the kind of coordinates being used. The most simple case comprises *uniform* coordinates, where all cells are assumed to be rectangular and axis-aligned. Moreover, the grid spacing is constant along each axis. A grid with *stacked* coordinates may be imagined as a stack of uniform 2D slices. However, the distance between neighbouring slices in z-direction may vary. In case of *rectilinear* coordinates the cells are still aligned to the axes, but the grid spacing may vary from cell to cell. Finally, in case of *curvilinear* coordinates each node of the grid may have arbitrary coordinates. Grids with curvilinear coordinates are often used in fluid dynamics because they have a simple structure but still allow for accurate modeling of complex shapes like rotor blades or airfoils.

3.2.3.2 Tetrahedral Grids

The *TetraGrid* class represents a volumetric grid composed of many tetrahedrons. Such grids can generally be used to perform finite-element simulations, e.g., E-field simulations.

A considerable amount of information is maintained in a *TetraGrid*. For each vertex a 3D coordinate vector is stored. For each tetrahedron the indices of its four vertices are stored as well as a number indicating the segment the tetrahedron belongs to as obtained by a segmentation procedure. Beside this fundamental information a number of additional variables are stored in order for the grid being displayed quickly. In particular all triangles or faces are stored separately together with six face indices for each tetrahedron. In addition for each face pointers to the two tetrahedrons it belongs to are stored. This way the neighborhood information can be obtained efficiently.

When simulating E-fields using the finite-element method, the edges of a grid need to be stored explicitly, because vector or Whitney elements are used. These elements and its corresponding coefficients are defined on a per-edge basis. When a grid is selected information on the number of its vertices, edges, faces, and tetrahedrons is displayed.

3.2.4 Surface Data

amira provides a special-purpose data class for representing triangular surfaces, called HxSurface. This class is documented in more detail in the index section of the user's guide. For the moment, we only mention that the class maintains connectivity information and that it may represent manifold as well as non-manifold topologies.

The surface class provides a rich set of Tcl commands. It is a good example of an amira data class that does not simply store information, but allows the user to query and manipulate the data by means of special-purpose methods and interfaces.

3.2.5 Vertex Set

Another example of data abstraction and inheritance is the VertexSet class. Many data objects in amira are derived from this class, e.g., landmark sets, molecules, surfaces, or tetrahedral grids. All these objects provide a list of points with x-, y-, and z-coordinates. Other modules which require a list of points as input only need to access the *VertexSet* base class, but don't need to know the actual type of the data object.

One such example of a generic module operating on *VertexSet* objects is the VertexView module. This module allows you to visualize vertex positions by drawing dots or little spheres at each point.

3.2.6 Transformations

Data objects in amira can be modified using an arbitrary affine transformation. For example, this makes it possible to align two different data objects so that they roughly match each other. Internally, affine transformations are represented by a 4x4 transformation matrix. In particular, a uniform scalar field remains a uniform scalar field, even if it is rotated or sheared. Display modules like *OrthoSlice* still can exploit the simple structure of the uniform field. The possible transformation is automatically applied to any geometry shown in the 3D viewer.

In order to interactively manipulate the transformation matrix use the Transform Editor (documentation is contained in the index section of the user's guide).

Be careful when saving transformed data sets! Most file formats do not allow to store affine transformations. In this case you have to apply the current transformation to the data. This can be done using the Tcl-command `applyTransform`. In case of vertex set objects the transformation is applied to all vertices. Old coordinates are replaced by new ones, and the transformation matrix is reset to identity afterwards. After a transformation has been applied to a data set, it cannot be unset easily anymore.

If a transformation is applied to uniform fields, e.g., to 3D image data, the coordinate structure is not changed, i.e., the field remains a uniform one. Instead, the data values are resampled, i.e., the transformed field is evaluated at every vertex of the final regular grid. The bounding box of the resulting grid is modified so that it completely encloses the transformed original box.

3.2.7 Parameters

For any data object an arbitrary number of additional parameters or attributes may be defined. Parameters can be interactively added, deleted, or edited using the parameter editor. Parameters are useful for example to store certain parameters of a simulation or of an experiment. In this way the history of a data object can be followed.

There are certain parameters which are interpreted by several amira modules. The meaning of these parameters is summarized in the following list:

- `Colormap name`
  This specifies the name of the default colormap used to visualize the data. Some modules automatically search the object pool for this colormap and for example use it for pseudocoloring.
- `DataWindow minVal maxVal`
  This indicates the preferred data range used for visualizing the data. The OrthoSlice module automatically maps values below `minVal` to black and values above `maxVal` to white.
- `LoadCmd cmd`
  This parameter is usually set by import filters when a data object is read. It is used when saving the current network into a file and it allows to restore the object automatically. Internal use only.

Note that there are many file formats which do not allow to store parameters. Therefore, information might get lost when you save the data set in such a format. If in doubt, use the amira specific AmiraMesh format.

Chapter 4

Technical Information

This chapter contains technical information about amira which is not covered in the previous chapters.

- Data Import
- Command Line Options
- Environment Variables
- amira start-up script
- Frequently Asked Questions
- System Requirements
- Acknowledgements and Copyrights
- Contact and Support

4.1 Data Import

Usually, one of the first things amira users want to know is how to import their own data into the system. This section contains some advice intended to ease this task.

In the simplest case, your data is already present in a standard file format supported by amira. To import such files, simply use the *File Load* menu. A list of all supported formats can be found in the index section of the user's guide. Usually, the system recognizes the format of a file automatically by analyzing the file header or the filename suffix. If a supported format is detected, the file browser indicates the format name.

Often, *3D image volumes* are stored slice by slice using standard 2D image formats such as TIFF or JPEG. In case of medical images, slices are commonly stored in ACR-NEMA or DICOM format. If you select multiple 2D slices simultaneously in the file browser, all slices will automatically be combined into a single 3D data set. Simultaneous selection is most easily achieved by first clicking the first slice and then shift-clicking the last one.

If your data is not already present in a standard file format supported by amira you will have to write your own converter or export filter. For many data objects such as 3D images, regular fields, or tetrahedral grids amira's native *AmiraMesh* format is most appropriate. Using this format you can even represent point sets or line segments for which there is hardly any other standard format. The AmiraMesh documentation explains the file syntax in detail and contains examples of how to encode different data objects. One important amira data type, triangular non-manifold surfaces, cannot be represented in a *AmiraMesh* file but has its own file format called HxSurface format.

Finally, in case of images or regular fields with uniform coordinates you may also read binary raw data. Note that for raw data the dimensions and the bounding box of the data volume must be entered manually in a dialog box which pops up after you have selected the file in the file browser.

4.2 Command Line Options

This section describes the command line options understood by amira. In general, on Unix systems amira is started via the `start` script located in the subdirectory `bin`. Usually, this script will be linked to `/usr/local/bin/amira` or something similar. Alternatively, the user may define an alias `amira` pointing to `bin/start`.

On Windows systems amira is usually started via the start menu or via a desktop icon. Nevertheless, the amira executable may also be invoked directly by calling `bin/arch-Win32-Optimize/amira.exe`. In this case, the same command line options as on a Unix system are understood.

The syntax of amira is as follows:

```
amira [options] [files ...]
```

Data files specified in the command line will be loaded automatically. In addition to data files, script files can also be specified. These scripts will be executed when the program starts.

The following options are supported:

- `-help`
  Prints a short summary of command line options.
- `-version`
  Prints the version string of amira.
- `-no_stencils`
  Tells amira not to ask for a stencil buffer in its 3D graphics windows. This option can be set to exploit hardware acceleration on some low-end PC graphics boards.
- `-no_overlays`
  Tells amira not to use overlay planes in its 3D graphics windows. Use this option if you experience problems when redirecting amira on a remote display.

- `-no_gui`
  Starts up amira without opening any windows. This option is useful for executing a script in batch mode.
- `-logfile filename`
  Causes any messages printed in the console window also to be written into the specified log file. Useful especially in conjunction with the `-no_gui` option.
- `-depth_size number`
  This option is only supported on Linux systems. It specifies the preferred depth of the depth buffer. The default on Linux systems is 16 bits.
- `-style={windows | motif | cde}` This option sets the display style of amira's Qt user interface.
- `-debug` This options applies to the developer version only. It causes local packages to be executed in debug version. By default, optimized code will be used.
- `-cmd command [-host hostname] [-port port]`
  Send Tcl command to a running amira application. Optionally the host name and the port number can be specified. You must type `app -listen` in the console window of amira before commands can be received.

4.3 Environment Variables

In order to execute amira no special environment settings are required. On Unix systems some environment variables like the shared library path or the amira root directory are set automatically by the amira start script. Other environment variables may be set by the user in order to control certain features. These variables are listed below. On Unix systems environment variables can be set using the shell commands `setenv` (csh or tcsh) or `export` (sh, bash, or ksh). On Windows environment variables can be defined in the file `autoexec.bat` (Windows 98/ME) or in the system properties dialog (Windows 2000/XP).

- `AMIRA_DATADIR`
  A list of data directory names separated by semicolons `[;]` on Windows systems and colons `[:]` on Unix systems. The first directory will be used as the default directory of the file dialog. Other directories are quickly accessible via the file dialog's path list.
- `AMIRA_TEXMEM`
  Specifies the amount of texture memory in megabytes. If this variable is not set some heuristics are applied to determine the amount of texture memory available on a system. However, these heuristics may not always yield a correct value. In such cases the performance of the *Voltex* module might be improved using this variable.
- `AMIRA_MULTISAMPLE`
  On high-end graphics systems like SGI Onyx, a multi-sample visual is used by default. In this way, efficient scene anti-aliasing is achieved. If you want to disable this feature, set the

environment variable `AMIRA_MULTISAMPLE` to 0. Note that on other systems, especially on PCs, anti-aliasing cannot be controlled by the application but has to be activated directly in the graphics driver.

- `AMIRA_NO_OVERLAYS`
  If this variable is set, amira will not use overlay planes in its 3D graphics windows. The same effect can be obtained by means of the `-no_overlays` command line option. Turn off overlays if you experience problems with redirecting amira on a remote display, or if your X server does not support overlay visuals.
- `AMIRA_LOCAL`
  Specifies the location of the local amira directory containing user-defined modules. IO routines or modules defined in this directory replace the ones defined in the main amira directory. This environment variable overwrites the local amira directory set in the development wizard (see amira programmer's guide for details).
- `AMIRA_SMALLFONT`
  Unix systems only. If this variable is set a small font will be used in all ports being displayed in the Working Area even if the screen resolution is 1280x1024 or bigger. By default, the small font will be used only in case of smaller resolutions.
- `AMIRA_XSHM`
  Unix systems only. Set this variable to 0 if you want to suppress the use of the X shared memory extension in amira's image segmentation editor.
- `AMIRA_SPACEMOUSE`
  This variable has to be set in order to support a spaceball or spacemouse device (see *www.spacemouse.com*). With the spacemouse you can navigate in the 3D viewer window. Two modes are supported, a rotate mode and a fly mode. You can switch between the two modes by pressing the spacemouse buttons 1 or 2.
- `AMIRA_STEREO_ON_DEFAULT`
  If this variable is set the 3D viewer will be opened in OpenGL raw stereo mode by default. In this way some screen flicker can be avoided which otherwise occurs when switching from mono to stereo mode. Currently the variable is supported on Unix systems only.
- `TMPDIR`
  This variables specifies in which directory temporary data should be stored. If not set, such data will be created under `/tmp`. Among others, this variable is interpreted by amira's job queue.

4.4 User-defined start-up script

amira may be customized in certain ways by providing a user-defined start-up script. The default start-up script, called `Amira.init`, is located in the subdirectory `share/resources` of the amira installation directory. This script is read each time the program is started. Among other things, the start-up script is responsible for registering file formats, modules, and editors and for loading the default colormaps.

If a file called `Amira.init` is found in the current working directory, this file is read instead of the default start-up script. If no such file is found, on Unix systems it is checked if there exists a start-up script called `.Amira` in the user's home directory. Below an example of a user-defined start-up script is shown:

```
# Execute the default start-up script
source $AMIRA_ROOT/share/resources/Amira.init 0

# Set up a uniform black background
viewer 0 setBackgroundMode 0
viewer 0 setBackgroundColor black

# Choose non-default font size for the help browser
help setFontSize 12

# Restore camera setting by hitting F2 key
proc onKeyF2 { } {
    viewer setCameraOrientation 1 0 0 3.14159
    viewer setCameraPosition 0 0 -2.50585
    viewer setCameraFocalDistance 2.50585
}
```

In this example, first the system's default start-up script is executed. This ensures that all amira objects are registered properly. Then some special settings are made. Finally, a hot-key procedure is defined for the function key F2. You can define such a procedure for any other function key as well. In addition, procedures like `onKeyShiftF2` or `onKeyCtrlF2` can be defined. These procedures are executed when a function key is pressed with the Shift or the Ctrl modifier key being pressed down.

4.5 Frequently Asked Questions

Questions

General

1. What is amira?
2. What is the latest version of amira?
3. How can I try amira? Are there demo or evaluation keys?

Installation, hardware and platform related questions

4. What are the supported platforms for amira?
5. Compatibility between Windows and Unix version ?
6. Can I display amira on a remote screen?

7. Do I need to have root or administrator privileges in order to install the product?
8. What are the software and hardware requirements?
9. What is the minimum configuration required for my platform?
10. What is the recommended hardware for my purpose?
11. What is the resource consumption of amira, for memory, disk, cpu, graphics?
12. Does amira make use of multiple processors ?
13. Are there any limits to the size of textures that graphics boards can use?
14. Does amira support the VolumePro volume rendering hardware?

Resources, examples, documentation

15. Where should I start to learn how to use amira?
16. How long do I need to learn how to use amira?
17. Can I get training courses?
18. What is the relevant documentation available?
19. How do I see what command line options amira accepts?
20. How can I check the version of my amira package?
21. Are there any examples or demos?
22. Is there a specific newsgroups?
23. Is there a mailing list?
24. Is there a web site?
25. Where can I find (free) modules for amira? Is there a public repository for 3rd party contributions?
26. How can I learn Tcl?

Technology

27. What graphics libraries are used by amira?
28. What is Open Inventor?
29. What is OpenGL?
30. What is Tcl?
31. What is Qt?
32. Is amira data-flow oriented?
33. How do modules communicate?
34. What is the firing order of modules?

Data input/output, printing

35. What are the supported data formats (input and output)?
36. How can I use amira to import/export image formats other than the amira image format?
37. How can I define the pixel size for my 3D image volume ?
38. What are the data/mesh/UCD cell types supported by amira?

71. Can I embed executable or shell scripts as modules?
72. How can I connect amira visualization to my computation code?
73. Can I develop with Open Inventor or 3D-MasterSuite with an amira End User or Developer License?
74. Can I get the source code for an amira module?
75. What is the compatibility with 3D-MasterSuite?
76. How can I (re)use 3D-MasterSuite or Open Inventor code with amira?
77. What is the difference between the developer and the end-user version?
78. Can I execute custom modules, created with amiraDev, with an ordinary amira version ?
79. What is the runtime policy for your own modules (technically)?

Answers

General

1. What is amira?

   amira is a professional general-purpose visualization and 3D reconstruction software. Visualization means that you can display various data sets, notably 3D image data, vector fields, and finite element data. 3D reconstruction means that you can create polygonal surface models as well as tetrahedral grids from 3D image data. amira is used for visualization and data analysis in microscopy, biology, medicine, engineering, geo-sciences, material science, bio-chemistry and many other fields.

2. What is the latest version of amira?

   The latest version is amira 3.1.

3. How can I test amira? Are there demo keys available?

   For evaluation purposes a fully functional version of amira can be downloaded from `www.amiravis.com` after electronic registration. A temporary license key will be send to you via e-mail. If you need a longer evaluation period, or want to purchase a permanent license, please contact us `http://www.amiravis.com/contact.html`

Installation, hardware and platform related questions

4. What are the supported platforms for amira?

   amira 3.1 runs on Microsoft Windows 98SE/ME/2000/XP, HP-UX 11.00, SGI Irix 6.5.x, Sun Solaris 8, and on Linux (RedHat 8.0 or compatible). Details are described in the user's guide in section System Requirements.

5. Compatibility between Windows and Unix version ?

   The Windows version and the Unix version provide de-facto the same functionality. Data files can be exchanged between Windows and Unix without limitations. Minor differences between the versions are due to differences of the underlying hardware. For example, direct volume rendering via 3D textures requires a suitable graphics card. Support for the VolPro 500/1000 cards currently is only available for Windows.

6. Can I display amira on a remote screen?

   In general, it is not recommended that you use an X11 remote display for demanding interactive 3D graphics applications like amira.

   However, in principle you can redirect the output of the Unix version to a remote display. For IRIX, HP-UX, and SunOS, the remote X server needs to support the GLX extension. Call `xdpyinfo` to find out whether your computer has that extension installed. Then simply set the `DISPLAY` variable and start amira.

7. Do I need to have root or administrator privileges in order to install amira?

   No. On Windows systems you can run the setup tool without having administrator privileges. On Unix systems simply extract the provided tar file. In order to install amira for all users of the system, Administrator privileges may be needed.

   However, on Sun and HP-UX it is recommended that you set the default visual of the X server to 24-bit true color. This may require root privileges. In addition, on some HP-UX systems it is recommended that you increase certain kernel parameters like process data size or stack limit. This requires root privileges as well.

8. What are the software and hardware requirements?

   Software and hardware requirements are described in the user's guide in section System Requirements.

9. What is the minimum configuration required for my platform?

   You need a graphics board with at least 24 bits of color per pixel (16 bits for Linux). At least 64 MB of main memory are required, 512 MB or more are recommended.

10. What is the recommended hardware for my purpose?

    In principle, all features are available even on low-end machines as well. However, for most applications it is highly recommended to have a sufficiently large amount of main memory (512 MB or more) and to have a graphics card which supports both texturing and geometry processing (transformation and lighting) in hardware.

11. What is the resource consumption of amira, for memory, disk, CPU, graphics?

    amira needs roughly about 60 MB of disk space. Memory, CPU and graphics performance (of course) depend on the kind of data you are going to visualize. CPU speed is less critical than graphics performance. Enough memory should be available in order to completely store the data to be visualized.

12. Does amira make use of multiple processors ?

    The bottleneck for most visualization modules in amira is the performance of the graphics board, which is in general not increased by using multiple processors. Some computational modules, like the amira deconvolution extension use multiple processors for acceleration. The amiraVR edition uses multiple processors for rendering on multi-graphics-pipe systems. If you use amiraDev for custom module development you can use parallelized code in your own modules as well.

13. Are there any limits to the size of textures that graphics boards can use?

Yes. The exact value depends on the graphics board. A typical limit will be 2048x2048 pixels per texture. On some architectures there is also a limit to the total amount of texture memory available. This is of special importance for texture-based volume rendering. Details are given in the documentation for the Voltex module.

14. Does amira support the VolumePro volume rendering hardware?

    Yes, amira supports the VolumePro 500 and the VolumePro 1000 on Windows 2000 and Windows XP. If you need VolumePro support for other platforms, please contact us.

Resources, examples, documentation

15. Where should I start to learn how to use amira?

    Probably, the best starting point are the tutorials in the user's guide. They provide a step-by-step learning-by-doing introduction.

16. How long does it take to learn how to use amira?

    amira is easy to use. After going through one of the tutorials for 15 minutes you will have an idea of the basic functionality. Usually, this is sufficient in order to be able to do first visualizations of your own data. Of course, becoming an amira wizard and becoming familiar with all the features available in the system will take substantially more time.

17. Can I get training courses?

    Yes, training courses and consulting services are available. For more information refer to the web sites `www.amiravis.com` or `www.tgs.com`.

18. What is the relevant documentation? Where can I find help?

    The primary source for documentation is the amira user's guide. This guide, as well as additional information, is provided on the amira web site `www.amiravis.com` and also on `www.tgs.com`. If you have specific questions you will find contact information on these web sites.

19. How do I see what command line options amira accepts?

    Command line option are documented in the user's guide in Section 4.2. In addition, starting amira with `-help` gives you a short summary of options.

20. How can I check the version of my amira package?

    To get the version of amira type `app -version` in amira's console window. In order to see when amira was compiled use `app -built`. Please indicate the version string or compilation date whenever your report bugs or problems.

21. Are there any examples or demos?

    Yes. The amira distribution contains tutorials and example with demo data. The tutorials are contained in Chapter 2 of the user's guide. Demos are listed in the reference section of the user's guide.

22. Is there a specific newsgroups?

    There is no amira specific newsgroup yet.

23. Is there a mailing list?

No.

24. Is there a web site?

    For technical information refer to `www.amiravis.com`. For sales and marketing contact see `http://www.amiravis.com/contact.html` or `www.tgs.com`.

25. Where can I find (free) modules for amira? Is there a public repository for 3rd party contributions?

    In the future a public repository for 3rd party contributions will be set up at `www.amiravis.com`.

26. What is Tcl and how can I learn Tcl?

    Tcl is the scripting language used by amira. You do not need to know Tcl for normal use of amira, however it enables you to extend the functionality by writing custom scripts. There are many good Tcl books. For example, you can try *Tcl and the Tk Toolkit* by *John K. Ousterhout*, the creator of Tcl. Like many others this book also covers the Tk GUI toolkit. Note that Tk is not used in amira. There are many Tcl online tutorials in the Internet. Simply type “TCL tutorial” into a search engine like `www.google.com` to find some.

Technology

27. What graphics libraries are used by amira?

    amira is based on the Open Inventor graphics toolkit. Furthermore, amira contains a number of custom Inventor nodes which implement special visualization techniques. These nodes apply direct OpenGL rendering.

28. What is Open Inventor?

    Open Inventor is a C++ library allowing you to describe and render 3D scenes. Open Inventor is built on top of OpenGL. This guarantees portability and hardware-accelerated performance across a wide range of platforms.

29. What is OpenGL?

    OpenGL is a library for rendering 3D graphics. OpenGL is the industry standard for professional 3D graphics. It is supported by all professional graphics hardware and by an increasing number of consumer graphics cards.

30. What is Tcl?

    Tcl is a popular scripting language. Tcl has a simple syntax, so you can learn Tcl in one afternoon. amira has a built-in Tcl interpreter. This way amira is script-able.

31. What is Qt?

    Qt is a multi-platform GUI software toolkit developed by Troll Tech (`www.troll.no`). An application written with Qt can be compiled on Unix/X11 as well as on Windows. While the user interface of amira 2.0 was based on Motif, all releases of amira 2.1 and later (including the Windows version) are based on Qt. For the end-user this guarantees that the set of features and the user interface will be compatible across all platforms.

32. Is amira data-flow oriented?

No, amira is not data-flow oriented. amira is object oriented. Data objects are persistent in memory and represented in the user interface. Data are accessed by the modules using the C++ interfaces of the data classes.

33. How do modules communicate?

    Modules are loaded into a common process space at runtime, by using shared libraries. This way they can communicate like C++ objects in a normal C++ program. There is no overhead for module communication.

34. What is the firing order of modules?

    Most modules are fired in downstream order. If you create a new module from the popup menu of an existing one the new module will be downstream. amira networks are typically much less complex than in data-flow-oriented visualization systems. Therefore the firing order is usually not of concern for the end-user.

Data input/output, printing

35. What are the supported data formats (input and output)?

    A list of supported file formats is contained in the index section of the user's guide.

36. How can I use amira to import/export image formats other than the amira image format?

    amira supports several standard image formats such as TIFF, JPEG, SGI-RGB, ACR-NEMA, or DICOM. When amira reads data it can usually determine the file format automatically. You simply select the file in the file browser and click OK. When amira writes data, the file browser presents an option menu containing all file formats which can be used to export that data. Use this menu to select a non-default format.

37. How can I define the pixel size for my 3D image volume ?

    There is a difference between the number of pixels in a 3D image volume (e.g., 512x512x200) and its physical bounding box (e.g., 30cm x 30cm x 20cm). Often voxels are even not equally sized in all directions. Many 2D image formats do not contain this extra information. When reading images, you can supply this information in amira's image input dialog. You can also change this information later by selecting the data set (green icon) and choosing the Image Crop Editor button.

38. What are the data/mesh/UCD cell types supported by amira?

    The list of supported data types includes

    - 2D and 3D grayscale images (8, 16, and 32 bits)
    - 32 bit RGBA-color images
    - segmentation results based on labeled voxels
    - unstructured tetrahedral meshes
    - unstructured hexahedral meshes
    - scalar and vector fields defined on uniform, stacked, rectilinear, curvilinear, tetrahedral, or hexahedral grids
    - manifold and non-manifold surfaces

- colormaps including transparency values
- Open Inventor scene graphs

39. How can I read my data (with some specific file format)?

    amira supports a number of standard file formats. Therefore it is likely that you can find a converter if your file format is not supported. For image data, amira provides a powerful Raw-Data interface, which can handle most simple binary file formats with some additional manual work.

    In order to implement custom I/O methods, the extensible version of amira called amiraDev is required.

40. How can I get access to my database?

    Data I/O is handled via files. The developer version, of course, allows the user to add any database interface he/she wants.

41. How can I reuse my work with amira? Can I compose modules?

    You can save networks, and you can save data objects that have been created or modified. In order to build new modules, you must use the developer version, or write script objects in Tcl.

42. How can I print with amira? How can I take a snapshot of the viewer window?

    In the viewer, on the left hand side, there is an icon showing a photo camera. This allows you to write snapshots of the 3D scene to a file or to a printer. If no printers show up in the list then probably they are not properly installed. In the latter case you can still print to a PostScript file on disk and print that file from a different computer.

    You can also use the command line interface to make snapshots. This is useful for generating animations via a Tcl script. The syntax is `viewer <n> snapshot <filename>`, where `<n>` denotes the viewer window to be captured. The format of the output file is determined automatically from the file name suffix.

43. What image formats are supported for snapshots?

    Snapshots can be stored in TIFF, JPEG, SGI-RGB, PNM, BMP, PNG, or EPS format. The file type is determined automatically from the file name suffix.

44. How can I create printed reports including amira images?

    You may use any desktop publishing or word processing system of your choice. Probably all of them allow you to import either TIFF or JPEG or EPS files.

45. How can I publish amira images or animations on the Web?

    Make snapshots and save them as JPEG images, or create animation sequences as described below.

Visualization

46. Can I display axes?

    Yes. Use the menu entry *Axis* in the view menu. This will display global axes located at the origin of the world coordinate system. You may also attach local axes to any data object by selecting *Display LocalAxis* from the object's popup menu.

47. Can I do image processing with amira?

    Basic image processing functionality is provided although amira is not a dedicated image-processing program. For example, the *Image Filters* editor supports smoothing, sharpening, as well as certain morphological operations.

48. Does the surface reconstruction support non-manifold topologies?

    Yes. In contrast to many other products, non-manifold topologies are handled in a consistent way by amira.

49. Is it possible to start amira up with NO display in order to do batch processing of data or to generate pictures and plots without displaying anything on the console?

    You can start amira with the `-no_gui` command line option in order to execute scripts in batch mode.

50. How does amira behave with large data sets?

    amira is an interactive visualization system. Therefore, data sets must be loaded into main memory in order to be processed. In some projects very large dynamic data sets (up to several 100 GB) have been visualized with amira. In this case special reader modules have been used which only read subsets of the data at once.

51. How can I change the background color of the viewer?

    There are three different background modes, namely *uniform*, *gradient*, and *checkerboard*. These modes can be set for all viewers via the *View Background* menu of the main window or via the command `viewer <n> setBackgroundMode <mode>` for a particular viewer. The primary background color can be adjusted via the camera icon in the upper left corner of the viewer window or via the command `viewer <n> setBackgroundColor <color>`. The secondary color used in gradient and checkerboard mode can be adjusted via the command `viewer <n> setBackgroundColor2 <color>`.

    You can also place an arbitrary raster image in the background using the command `viewer <n> setBackgroundImage <filename>`. Any image file in TIFF, SGI-RGB, JPEG, PNM, BMP, or PNG format can be read. However, note that the image is not shown if its size is greater than that of the viewer window.

52. Can I display a colormap in the viewer window as legend?

    First make the colormap icon visible in the Working Area. This can be done by selecting the *Show* or *Show All* item of *Edit* in the menu bar of the Working Area. Then click on the green colormap icon with the right mouse button and select Show Colormap.

53. How can I adjust color and transparency of individual parts of a surface?

    A surface object may consist of multiple patches referring to different materials. The color of each material can be adjusted using the Tcl command `setColor` described in Surface. Likewise, for each material a specific transparency value may be set using the command `setTransparency`. In this way certain parts of a surface may be highlighted. Note that you must choose draw style *transparent* in order to enable transparencies. Also note that color mode *mixed* is most appropriate for transparent surfaces in terms of performance and meaning.

54. How can I create an iso-surface with fewer polygons than the iso-surface module extracts?

First of all, the iso-surface module provides a special option called compactify which produces about 40 percent fewer triangles than standard method. Moreover, very large data sets may be downsampled on-the-fly during isosurface generation.

If you need more flexibility, you can create a separate surface object by selecting *create surface* from the *more options* menu of the iso-surface module. You may then use the simplification editor in order to remove as many triangles from the surface as you want. You can display the resulting simplified surface using the SurfaceView module.

55. How do I visualize data with holes in it?

    There are several choices. You can apply a slicing module such as OrthoSlice or ObliqueSlice. Alternatively, you can clip away parts of a 3D geometry using an arbitrary slicing module. Slicing modules are indicated by an orange icon. Such modules provide a little push button that must be pressed in order to activate clipping. An empty clipping module can be created via the Edit Create menu of the main window.

    If you want to visualize surfaces or finite-element grids you can also use the selection box feature of the corresponding viewing modules. Most of these modules such as *SurfaceView* or *GridVolume* support a buffer concept which allows you to select which parts of the object should be displayed. Even the *Isosurface* module has such a buffer. For this module it can be enabled using the amira command `Isosurface showBox`.

56. How can I read a series of single image files such that I get a 3D stack?

    Select all image files in the file browser at once. This can be done by clicking the first file and then shift-clicking the last one. Individual files can be selected and deselected by ctrl-clicking. After pressing the Ok button all images will be combined in a single 3D data stack. Note that the images should be of the same size.

57. How can I quickly switch between two different data sets?

    Click with the left mouse on the blue line connecting one of the data icons and the visualization module icon in the Working Area and - holding the left mouse button down - move the line to the icon of the other data object.

58. How can I compare two data sets?

    One solution is to display each data set in a different viewer. You can activate up to four viewers via the *View Layout* menu. If two viewers are visible attach a display module to each data set. You can control in which of the viewers the output of a module is displayed by selecting the module and setting or unsetting the orange viewer toggles. If you are using two *OrthoSlice* modules, make sure that the same slice is displayed in both viewers.

    In order to get the same camera settings in both viewers use the Tcl command `viewer 0 setSlaveViewer 1`. Whenever you navigate in viewer 0 the camera of viewer 1 will be adjusted as well.

    An alternative method to compare two different data sets is to compute and visualize the difference of both. To subtract two fields from each other use the Arithmetic module. Connect the module to both data sets by activating the popup menu over the small rectangle of the module's icon. Then enter an expression like A-B in order to compute the difference. You can visualize the result of the *Arithmetic* module by any of the ordinary display modules.

Specific features

59. Does amira support Stereo viewing?

    Yes. You will need special shutter glasses, e.g., Stereographics Crystal Eyes. Stereo viewing is successfully being used on SGI and HP-UX systems. The Windows version is also be stereo enabled. You can use red/blue stereo as well as shutter stereo, if your hardware supports stereo for OpenGL applications.

60. Does amira support VR devices, such as a 3D mouse, head-mounted displays or CAVE systems?

    Yes. Depending on your exact requirements you will need the amiraVR edition.

61. Can I use anti-aliasing?

    Yes. On SGI Infinite Reality systems amira will automatically use a multi-sample visual. On other systems you can switch on anti-aliasing in the graphics driver. If your system does not support hardware anti-aliasing, you may use the command `viewer 0 antiAlias 3` to enable 3-pass jittered rendering. To reset, type `viewer 0 antiAlias 1`.

Developing applications with amira

62. Is it possible to extend amira?

    Yes, a special version of amira called amiraDev allows you to write your own modules, data classes, editors, and I/O methods.

63. Is amira an application builder?

    No, it isn't.

64. Can I define my own user interface for my specific application?

    You cannot customize the user interface of existing modules. With the end user version you can write scripts with a specific set of ports (amira GUI elements). With the developer version you can write modules with any user interface you like. For non-standard components this might require a Qt developer license which is not part of the amira developer version. If you want to build an application with a completely customized look-and-feel, you will have to implement your own user interface to wrap and hide the existing amira components.

65. Can I record user interaction in "macros" ?

    No. But you can save the current network.

66. Can I automate operations with amira?

    Yes, you can use Tcl scripts and script objects. Script objects allow you to specify parameters for your scripts using pre-defined GUI elements such as buttons, option menus, or sliders.

67. Is there an upgrade from End User edition to Developer Edition?

    Yes, the end-user license can be upgraded to amiraDev.

68. Can I write data input, processing and visualization modules with Tcl?

    The Tcl interface is intended to access special features of modules, to automate routine tasks, or to solve certain problems by combining existing modules and components. Writing new visualization or data processing modules in Tcl is difficult and is not recommended. Writing data I/O methods in Tcl can make sense in some situations.

69. Is it possible to script any interaction with amira? Are all amira features available through Tcl scripts ?

    Any interaction with modules is fully scriptable. There are features in interactive editors which are not scriptable. These are mainly interactions with the 3D viewer.

70. What programming languages can I use: C++, C, FORTRAN ...?

    amira is written in C++. Implementing a new module with amiraDev version requires you to derive from an existing C++ class. Inside this class, of course, you can call routines written in other languages such as C or FORTRAN.

71. Can I embed executable or shell scripts as modules?

    You can use Tcl scripts and script objects. From within these scripts you can call external programs using the `system` command. Data exchange with these programs typically will be via files.

72. How can I connect amira visualization to my computation code?

    You can write simulation results (e.g., time steps) to files and than tell a running amira to read them. To do that use the -cmd option of amira, i.e. call `amira -cmd somecmd` where `somecmd` typically will be a Tcl procedure.

    If you have the amiraDev version, you can either embed your simulation code in an amira module (possibly as a separate thread), or you can write a module which communicates with your simulation via sockets or shared memory.

73. Can I develop with Open Inventor or DataViz with an amira Developer License?

    If you have the amiraDev version, you may use Open Inventor in your own amira modules, but you can't compile standalone Open Inventor applications. This would require a separate Open Inventor SDK license.

74. Can I get the source code for an amira module?

    The amiraDev version contains source code for demo modules which you may use as a template for your own modules. In general, the source code of amira modules will not be released.

75. What is the compatibility with TGS DataViz ?

    amira doesn't use DataViz components. However, you can use such components in your own modules without limitations.

76. Can I (re)use Open Inventor or DataViz code with amira?

    Yes.

77. What is the difference between the amiraDev version and the end-user version?

    In addition to the end-user version amiraDev contains all files (like header files of amira base modules and a makefile environment) needed to compile specific extensions. It also contains a “wizard” to create skeletons of new modules and readers.

78. Can I execute custom modules, created with amiraDev, with an ordinary amira version ? Yes, you can. Details are given in the programmer's guide.

79. What is the runtime policy for my own modules (technically)?

    You may distribute your own modules without limitations. In order to use them, other users will have to purchase an amira end-user version.

4.6 System Requirements

amira 3.1 runs on Microsoft Windows 98/ME/NT4/2000/XP, HP-UX 11.00, SGI Irix 6.5.x, Sun Solaris 8 and 9, Linux (RedHat 8.0), and Linux IA64 (RedHat AW 2.1).

amira relies on fast hardware-accelerated OpenGL 3D graphics. We strongly recommend hardware texture mapping, since many visualization tools in amira rely on it. Hardware texture mapping is available, on all decent PC 3D graphics boards. On Unix systems it is available for example, on SGI O2, Octane and Onyx systems, on HP workstations with fx/4, fx/6 or fx/10 graphics, or on Sun Creator 3D, Elite, Expert 3D or newer graphics boards. For details on hardware acceleration, see below.

Apart from 3D graphics hardware probably, the most important system parameter is main memory. You should have at least 128 MB, preferably 512 MB or more. amira can also be started with only 64 MB but working with large data sets definitely requires more main memory. Keep in mind that a single 3D image data set can easily occupy 60 MB of memory (240 slices with 512 x 512 pixels of 1 byte).

The speed of the processor of course is also an important parameter. However it is less critical than the graphics system and the main memory size. For the PC versions, we recommend at least a 500 MHz PIII processor.

4.6.1 On System Stability

amira is a very demanding application that extensively uses high-end features. Experience shows that such applications tend to reveal instabilities in system hardware, hardware drivers, and the operating system. A common problem is insufficient main memory. We recommend you configure enough swap memory in addition to physical memory. The total amount of virtual memory should be at least 512MB. 1GB would be even better.

Especially on PC platforms, OpenGL drivers today are often not as robust as desired. Also, system crashes due to bad memory chips or unstable power-supply are not rare. If you experience problems or instabilities with amira on your Windows platform, we recommend that you follow these steps:

1. Click on all the demo scripts in the Online User's Guide. If the system crashes, turn off hardware acceleration (choose the *extended* button from the Windows display settings dialog) and try again. If this eliminates the problem, there is a bug in your OpenGL driver. Try to get a new driver from the web site of the manufacturer of your graphics board.
2. Try using a different color depth in the Windows display settings dialog. Try 16, 24, or 32 bit.
3. Load the `lobus.am` data set and visualize it with a Voltex module. Turn on the spin rotation (turn it with the mouse in the viewer and release the mouse button while moving the mouse, so

that the object continues moving). Let it run over night (turn off the screen saver). If the system has crashed or frozen the next morning, you probably have a hardware problem.

If this does not help, or if a reproducible error occurs on different computers, then it might be a bug in the amira software itself. Please report such bugs so that they can be eliminated in the next release or a patch can be prepared.

4.6.2 Microsoft Windows

amira runs on Intel or AMD-based systems with Microsoft Windows 98, ME, Windows NT (NT4, Service Pack 4 or later), on Windows 2000, and on Windows XP. We recommend Windows 2000 or Windows XP.

Graphics Hardware: You should use a graphics board with OpenGL support and texture mapping capabilities. Both is the case for almost all newer 3D boards.

4.6.3 Silicon Graphics

Graphics Hardware: To get optimal graphics performance, the machine should support *texture mapping in hardware*. Currently this is the case for all O2 systems, and for Octane systems with High Impact, Maximum Impact (not Solid Impact) and Odyssey graphics. amira provides a number of modules which make use of texture mapping, e.g., slicing, pseudo-coloring, or volume rendering. On machines without hardware texture mapping, these modules either run much slower or may not work at all. The advantage of the Octane is a higher speed in polygon rendering. For a complex model with an isosurface of 100,000 triangles, the frame rate is 10 per second for an Octane, compared to 3 per second for an O2. The MXE and MXI Octanes have larger texture memory than the SSE and SSI. Thus the MXE and MXI enable a direct volume rendering using 3D textures, which is not possible on an SE, SI or O2.

Software: The current version of amira requires IRIX 6.5.x or higher. We recommend you install the newest version of the operating system (see `http://www.sgi.com/support`).

4.6.4 HP-UX

amira performs pretty well on HP workstations equipped with Visualize fx/4, fx/6+, and fx/10 graphics cards under HP-UX 11.00. Probably it will run on other machines as well provided the OpenGL runtime environment has been installed. In any case we recommend to install the texture acceleration option for your graphics system (hardware texture mapping), especially if you intend to work with large 3D image data sets.

Important: By default some HP workstations are configured with a data size limit of 64 MB for each process. In order to load reasonable data sets, you should increase this value to 1 GB and the stack size to 128 MB. Do this by modifying the values in sam/Kernel Configuration `maxdsiz=0x80000000, ssiz=0x8000000, tssiz=0x8000000`.

4.6.5 SunOS

amira runs on Sun workstations with Solaris 8 or Solaris 9.

amira is successfully being used on systems with Creator 3D, Elite 3D, Expert 3D, and Zulu graphics boards. It runs on a simple Creator graphics boards as well. However, since no hardware texturing is available, performance is limited.

4.6.6 Linux

The Linux version of amira 3.1 has been developed and tested on RedHat 8.0. On other Linux distributions this version might not run because certain required system libraries are missing or because different versions of these libraries are installed. For such cases we provide a ”patch” containing the original RedHat 8.0 libs. Usually with this patch amira runs of other Linux systems as well. For more information please refer to resources section on `http://www.amiravis.com`.

There is also a version for Linux IA64 systems (Itanium 64-bit architecture). This version has been compiled and tested on the RedHat Advanced Workstation 2.1 distribution.

amira 3.1 works with the current 3D graphics drivers from nVidia and ATI under XFree86 4. It has also been successfully tested with other X-servers like the *3D Accelerated-X* servers from XI graphics.

Note: After a standard installation of RedHat 8.0, hardware acceleration is not necessarily activated, although X-Windows and amira may work fine. To enable OpenGL hardware acceleration specific drivers may have to be installed, like the nVidia drivers from `http://www.nvidia.com`. This can increase rendering performance by an order of magnitude. Sometimes it is necessary to disable the stencil buffers (by starting amira with the option -no_stencils) to get acceleration.

In any case amira requires an X server resolution of at least 1024x768 and at least 15 bit of color depth. We recommend 1280x1024 with 24 bit color depth.

Please note that if software rendering is used, rendering performance may drop significantly, especially for visualization techniques like volume rendering.

4.7 Acknowledgments and Copyrights

We thank the Department of Genetics at the University of W¨urzburg, the Hermann F¨ottinger Institut of the Technical University of Berlin and the Rudolf-Virchow Klinikum Berlin for providing demo data sets.

The amira software project was started at the scientific visualization group at Konrad-Zuse-Zentrum Berlin (ZIB). Since 1999 amira is being jointly developed further by ZIB and by Indeed - Visual Concepts GmbH, a spin-off company from ZIB. amira is based on the latest release of the Open Inventor toolkit from TGS Template Graphics Software, Inc. TGS is the worldwide distributor of amira.

amira uses the following non-commercial libraries:

- The Tcl library developed by John Ousterhout, subject to following license terms:

- The libtiff library developed by Sam Leffler, subject to the following license terms:

- The zlib library developed by Jean-Loup Gailly and Mark Adler.
- The JPEG input and output routines are based in part on the work of the Independent JPEG Group.

amira is a registered trademark of ZIB, Berlin. OpenGL, Open Inventor, Silicon Graphics, and IRIX are registered trademarks of Silicon Graphics Inc. Sun and Solaris are registered trademarks of Sun Microsystems, Inc.

4.8 Contact and Support

For purchasing an amira license and for hot-line support, contact:

TGS Inc.
5330 Carroll Canyon Road., Suite 201, San Diego, CA 92121-3758
Phone: +1-858-457-5359 Fax: +1-858-452-2547
Email: info@tgs.de Support: support@tgs.com

TGS Europe, P.A. Kennedy,
I-BP 227, Avenue Henri Becquerel, 33708 Merignac Cedex, France
Phone: +33-5-5613-37-77 Fax: +33-5-5613-02-10
Email: info@europe.tgs.de Support: support@europe.tgs.com

For special requests, such as development of special purpose modules, customization, consulting and training, contact:

Indeed - Visual Concepts GmbH
Ihnestr. 23, 14195 Berlin, Germany
Phone: +49-30-84185-221 Fax: +49-30-82701-747
Email: indeed@zib.de URL: http://www.indeed3d.com

For general questions about amira and for research collaborations contact:

Konrad-Zuse-Zentrum (ZIB)
Dept. Scientific Visualization, Takustrasse 7, 14195 Berlin, Germany
Phone: +49-30-84185-171 Fax: +49-30-84185-107
Email: amira@zib.de URL: http://www.amiravis.com

When sending a bug report, try to specify the problem in detail. If possible, describe how the problem can be reproduced. Please indicate which version of amira you are using. You can get this information by typing `app -version` in the console window or by starting amira with the command line option `-version`.

Chapter 5

Scripting

5.1 Introduction

This chapter is intended for advanced amira users only. If you do not know what scripting is, it is very likely that you will not need the features described in this chapter.

Beside the interactive control via the graphical user interface most of the amira functionality can also be accessed using specific commands. This allows you to automate certain processes and to create scripts for managing routine tasks or for presenting demos. amira's scripting commands are based on Tcl, the *Tool Command Language*. This means you can write command scripts using Tcl with amira specific extensions.

amira commands can be typed into the amira console window, as described in Section 3.1.8. Commands typed directly into the console window will be executed immediately. Alternatively, commands can be written into a text file, which can then be executed as a whole.

This chapter is organized as follows:

Section 5.2 (Introduction to Tcl) gives a short introduction into the Tcl scripting language. This section is not very amira specific.

Section 5.3 (amira Script Interface) explains amira specific commands and concepts related to scripting. This includes a reference of global commands.

Section 5.4 (amira Script Files) explains the different ways of writing and executing script files including references to script object, resource files and function-key bound Tcl procedures.

Section 5.5 (Configuring Popup Menus) describes how the popup menu of an object can be configured using script commands, and how new entries causing a script to be executed can be created.

Data Type: Script Object describes how to use Tcl scripts for defining custom modules that have their own graphical user interface and can be used like the built-in amira objects.

5.2 Introduction to Tcl

This chapter gives a brief introduction to the Tcl scripting language. If you are familiar with Tcl you can skip this section. However please notice that instead of the `puts` command use `echo` for output to the amira console.

This chapter is not intended to cover all details of the language. For a complete documentation or reference manual of the Tcl language refer to a text book like *Tcl and the Tk Toolkit* by *John K. Ousterhout*, the creator of Tcl. Like many other books about Tcl this also covers the Tk GUI toolkit. Note that Tk is not used in amira.

Alternatively you can easily find Tcl documentation and reference manuals on the internet e.g. at `www.scriptics.com` or looking up keywords like `Tcl tutorial` or `Tcl documentation` with a search engine like `www.google.com`.

When you type Tcl commands into the amira console they will be executed as soon as the return key is pressed. Use the completion and history functions provided by the amira console, as described in Section 3.1.8 (console window).

5.2.1 Tcl Lists, Commands, Comments

First, please note that Tcl is case sensitive: `set` and `Set` are not the same.

A Tcl command is a space-separated list of words. The first word represents the command name, all further words are treated as arguments to that command. As an example try the amira-specific command `echo` which will print all its arguments to the amira console. Try typing

```
echo Hello World
```

This will output the string *Hello World*. Note that Tcl commands can be separated by a semi-colon (;) or a newline character. If you want to execute two successive `echo` commands, you can do it this way:

```
echo Hello World ; echo Hello World2
```

or like this:

```
echo Hello World
echo Hello World2
```

Instead of a command, you can also place a comment in Tcl code. A comment starts with a hash character (#) and is ended by the next line break:

```
# this is a comment
echo Hello World
```

5.2.2 Tcl Variables

In Tcl variables can be used. A variable represents a certain state or value. Using Tcl code, the value of the placeholder can be queried, defined, and modified. To define a variable use the command

```
set name value
```

e.g.

```
set i 1
set myVar foobar
```

Note that in Tcl internally all variables are of string type. Since the set command requires exactly one argument as the variable value you have to quote values that contain spaces:

```
set Output "Hello World"
```

or

```
set Output {Hello World}
```

In order to substitute the value of a variable with name *varname*, a $ sign has to be put in front of that name. The expression $varname will be replaced by the value of the variable. After the above definitions,

```
echo $Output
```

would print

```
Hello World
```

in the console window, and

```
echo "$i.) $Output"
```

would yield the output *1.) Hello World*. Note that variable substituion is performed for strings quoted in `"`, while it is not done for strings enclosed in braces {}. Even newline-characters are allowed in a { } enclosed string. Note however that it is not possible to type in multi-line commands into the amira console.

5.2.3 Tcl Command Substitution

To do mathematical computations in Tcl you can use the command `expr` which will evaluate its arguments and return the value of the expression. Examples are:

```
expr 5 / ( 7 + 3)
expr $i + 1
```

In order to use the result of a command like `expr` for further commands an important Tcl mechanism has to be used: command substitution, denoted by brackets []. Any list enclosed in brackets [] will be executed as a separate command first, and the [...] construct will be replaced with the *result* of the command. This is similar to the '...' construct in Unix command shells. For example, in order to increase the value of the variable i by one you can use:

```
set i [expr $i + 1]
```

Of course, command expressions can be arbitrarily nested. The order of execution is always from the innermost bracket pair to the outermost one:

```
echo [expr 5 * [expr 7 + [expr 3+3]]]
```

5.2.4 Tcl Control Structures

Further important language elements are `if-else` constructs, `for`- and `while`- loops. These constructs typically are multi-line constructs and can therefore not conveniently be typed into the amira console. If you want to try the examples shown below, write them into a file like `C:\test.txt` by using a text editor of your choice, and execute the file by typing

```
source C:\test.txt
```

We start with the `if-then` mechanism. It is used to execute some code *conditionally,* only if a certain *expression* evaluates to "true (meaning a value different from 0):

```
set a 7
set b 8
if {$a < $b} {
    echo "$a is smaller than $b"
} elseif {$a == $b} {
    echo "$a equals $b"
} else {
    echo "$a is greater than $b"
}
```

The `elseif` and `else` parts are optional. Multiple `elseif` parts can be used, but only a single `if` and `else` part.

Another important construct is the conditional loop. Like the `if` command, it is based on checking a conditional expression. In contrast to `if`, the conditional code is executed multiple times, as long as the expression evaluates to true:

```
for {set i 1} {$i < 100} {set i [expr $i*2]} {
    echo $i
}
```

In fact this code is identical to:

```
set i 1
while {$i < $100} {
    echo $i
    set $i [expr $i * 2]
}
```

both loops would produce the output *1, 2, 4, 8, 16, 32, 64*.

If you want to execute a loop for all elements of a list, there is another very convenient command for that:

```
foreach x {1 2 4 8 16 32 64} {
  echo $x
}
```

This will generate the same output as the previous example. Note that the expression enclosed in braces is a space-separated list of words.

User-Defined Tcl Procedures

A new function or procedure is defined in Tcl using the `proc` command. Proc takes two arguments: a list of argument names, and the Tcl code to be executed. Once a procedure is defined, it can be used just like any other Tcl command:

```
proc computeAverageA {a b} {
    return [expr ($a+$b)/2.0]
}
proc computeAverageB {a b c} {
    return [expr ($a+$b+$c)/3.0]
}
```

```
echo "average of 2 and 3: [computeAverageA 2 3]"
echo "average of 2,3,4: [computeAverageB 2 3 4]"
```

As you can see in the example, the argument list defines the names for local variables that can be used in the body of the procedure (e.g. `$a`). The `return` command is used to define the result of the procedure. This result is the value that is used in the command bracket substitution `[]`.

If you want to define a procedure with a flexible number of arguments, you must use the special argument name `args`. If the argument list contains just this word, the newly defined command will accept an arbitrary number of arguments, and these arguments are passed as a *list* called `args`:

```
proc computeAverage args {
    set result 0
    foreach x $args {
        set result [expr $result + $x]
    }
    return [expr $result / [llength $args]]
}
```

In this example, the `llength` command returns the number of elements contained in the `args` list.

Note that the variable `result` defined in the procedure has *local* scope, meaning that it will not be known outside the body of the procedure. Also, the value of globally defined variables is not known within a procedure, unless that global variable is declared using the keyword `global`:

```
set x 3
proc printX {} {
    global x
    echo "the value of x is $x"
}
```

There is much more to be said about procedures, e.g. concerning argument passing, evaluation of commands in the context outside of the procedure, and so on. Please refer to a Tcl reference book for these advanced topics.

List and String Manipulation

Finally, at the end of this brief Tcl introduction, we come back to the concept of lists. Basically everything in Tcl is constructed using lists, so it is very important to know the most important list manipulation commands as well as to understand some subtle details.

Here is an example of how to take an input list of numbers and construct an output list in which each element is twice as big as the corresponding element in the input list:

```
set input [list 1 2 3 4 5]
set output [list]
foreach element $input {
  lappend output [expr $element * 2]
}
```

You can think of lists as simple strings in which the list elements are separated by spaces. This means that you can achieve the same result as in the previous example without using the list commands:

```
set input "1 2 3 4 5"
set output ""
foreach element $input {
  append output "[expr $element * 2] "
}
```

The *append* command is similar to *lappend,* but it just adds a string at the end of an existing string. List manipulation becomes much more involved when you start nesting lists. Nested lists are represented using nested pairs of braces, e.g.

```
set input {1 2 {3 4 5 {6 7} 8 } 9}
foreach x $input {
  echo $x
}
```

The result of this command will be

```
1
2
3 4 5 {6 7} 8
9
```

Please note that Tcl will automatically *quote* strings that are not single words when constructing a list. Here is an example:

```
set i [list 1 2 3]
lappend i "4 5 6"
echo $i
```

will yield the output

```
1 2 3 {4 5 6}
```

You can use the *lindex* command to access a single element of a list. *lindex* takes two arguments: the list and the index number of the desired element, starting with 0:

```
set i [list a b c d e]
echo [lindex $i 2]
```

will yield the result c.

5.3 amira Script Interface

Although the Tcl language is not intrinsically object oriented, the amira script interface is. There is one command for each object in the amira object pool. In addition there are several global commands associated with global objects in amira such as the viewer or the amira .

A command associated with an object in the object pool (e.g., an OrthoSlice module or an Isosurface module) only exists while the object exists. These commands are identical to the name of the object as displayed in the object pool. Typically the script interface of a specific object contains many different functions. The general syntax for an amira object-related command is

```
<object-name> <command-word> <optional-arguments> ...
```

For example, if an object called GlobalAxis exists (choose View/Axis from the amira menu) then you can use commands like

```
GlobalAxis deselect
GlobalAxis select
GlobalAxis setIconPosition 100 100
```

Remember to use the completion and history functions provided by the amira console, as described in Section 3.1.8 (console window) to save typing.

If you have already used amira you have noticed that the parameters and the behavior of an amira module are controlled via its *ports*. The ports provide a user interface to change their values when the module is selected. All ports can also be controlled via the command interface. The general syntax for that is

```
<object-name> <port-name> <port-command> <optional-arguments> ...
```

For example for the GlobalAxis you can type

```
GlobalAxis options setValue 1 1
GlobalAxis thickness setValue 1.5
GlobalAxis fire
```

When you type in these commands you will notice that the values in the user interface change immediately. However the module's compute method is not called until explicitly firing the module using the `fire` command. This allows you to first set values for multiple ports without a recomputation after each command. However, note that some modules automatically reset some of their ports for example when a new input object is connected. In such cases you may need to call `fire` after setting the value of every single port.

Usually the name of a port is identical to the text label displayed in the graphical user interface, except that white spaces are removed and command names start with a lower case letter. To find out the names of all ports of a specific module use the command

```
<object-name> allPorts
```

Almost all ports provide a `setValue` and a `getValue` command. The number of parameters and the syntax of course depend on the ports.

Commands of the type `<object-name> <port-name> setValue ...` make up more than 90% of a typical amira script. However, besides the port commands many amira objects provide additional specific commands. The command interface for a particular module is described in the User's Reference Guide. You can quickly find the corresponding page by clicking the ?-button in the work area when the module has been selected.

As a quick-help entering an object's name without further options will display all commands available for that object. Note that this will also show undocumented, unreleased, and experimental commands. In order to get more information about a particular module or port command you can type it into console window without any arguments and then press the F1 key. This opens the help browser with a command description.

amira objects are part of a class hierarchy. Similar to the C++ programming interface, also script commands are inherited by derived classes from its base classes. This means that a particular object like the axis object will beside its own specific commands also provide all the commands available in its base classes. Links to the base class commands are given in a modules documentation.

5.3.1 Predefined Variables

There exist some variables in amira Tcl, which are predefined and have a special meaning. These are

- `AMIRA_ROOT:` amira installation directory.
- `AMIRA_LOCAL:` Personal amira development directory (amiraDev only).
- `SCRIPTFILE:` Tcl script file currently executed.
- `SCRIPTDIR:` Directory in which currently executed script resides.
- `hideNewModules:` If set to 1, icons of newly created modules will initially be hidden.

5.3.2 Object commands

The basic command interface of amira modules and data objects is described in the data type chapter of the reference part of the usersguide in the Object section. The basic syntax of object commands is

```
<object> <command> <arguments> ...
```

where `<object>` refers to the name of the object and `<command>` denotes the command to be executed. Each module or data object may define its own set of commands in addition to the commands defined by its base classes. The commands described in the Object section are provided by all modules and data objects.

In the following section Global commands are described.

5.3.3 Global commands

This section lists amira specific global Tcl commands. Some of these commands are associated with certain global objects in amira, such as the console window, the main window, or the viewer window. Other commands are such as `load` or `echo` are not. These commands are described in one common subsection. In summary, the following command sections are provided:

- viewer command options (`viewer`)
- main window command options (`theMain`)
- console command options (`theMsg`)
- common commands for top-level windows
- progress bar command options (`theWorkArea`)
- application command options (`app`)
- other global commands

5.3.3.1 Viewer command options

Commands to a viewer can be entered in the console window. The syntax is

```
viewer [<number>] command,
```

where `<number>` specifies the viewer being addressed. The value 0 refers to the main viewer and may be omitted for convenience.

Commands

`viewer [<number>] snapshot [-offscreen [<width> <height>]] <filename>`
This command takes a snapshot of the current scene and saves it under the specific filename.

The image format will be automatically determined by the extension of the file name. The list of available formats includes: TIFF (`.tif,.tiff`), SGI-RGB (`.rgb,.sgi,.bw`), JPEG (`.jpg,.jpeg`), PNM (`.pgm,.ppm`), BMP (`.bmp`), PNG (`.png`), and Encapsulated PostScript (`.eps`). If the viewer number is not given, the snapshot is taken from all viewers, if you have selected the 2 or 4 viewer layout from the View menu.

If the `-offscreen` option is specified, offscreen rendering with a maximum size of 2048x2048 is used. In this case the viewer number is required even if viewer 0 is addressed. If the width and height is not specified explicitly, the size of the image is the current size of the viewer.

Caution: If you have more than one transparent object visible in the viewer and you want to use offscreen rendering set the transparency mode to *Blend Delayed* and check to see if all objects are rendered properly prior to taking a snapshot.

`viewer [<number>] setPosition <x> <y>`
Sets the position of the viewer window relative to the upper left corner of the screen. If more than one viewer is shown in the same window the position of the toplevel window is set.

`viewer [<number>] getPosition`
Returns the position of the viewer window. If more than one viewer is shown in the same window the position of the toplevel window is returned.

`viewer [<number>] setSize <width> <height>`
Sets the size of the viewer window. Width and height specifiy the size of the actual graphics area. The window size might be a little bit larger because of the viewer decoration and the window frame.

`viewer [<number>] getSize`
Returns the size of the viewer window without decoration and window frame.

`viewer [<number>] setCamera <camera-string>`
Restores all camera settings. The camera string should be the output of a `getCamera` command.

`viewer [<number>] getCamera`
This command returns the current camera settings, i.e., position, orientation, focal distance, type, and height angle (for perspective cameras) or height (for orthographic cameras). The values are returned as amira commands, which can be executed in order to restore the camera settings. The complete command string may also be passed to `setCamera` at once.

`viewer [<number>] setCameraPosition <x> <y> <z>`
Defines the position of the camera in world coordinates.

`viewer [<number>] setCameraPosition <x> <y> <z>`
Returns the position of the camera in world coordinates.

`viewer [<number>] setCameraOrientation <x> <y> <z> <a>`
Defines the orientation of the camera. By default, the camera looks in negative z-direction with the

y-axis pointing upwards. Any other orientation may be specified as a rotation relative to the default direction. The rotation is specified by a rotation axis *x y z* followed by a rotation angle *a* (in radians).

`viewer [<number>] getCameraOrientation`

Returns the current orientation of the camera in the same format used by `setCameraOrientation`.

`viewer [<number>] setCameraFocalDistance <value>`

Defines the camera's focal distance. The focal distance is used to compute the center around which the scene is rotated in interactive viewing mode.

`viewer [<number>] getCameraFocalDistance`

Returns the current focal distance of the camera.

`viewer [<number>] setCameraHeightAngle <degrees>`

Sets the height angle of a perspective camera in degrees. The smaller the angle the bigger the field of view. The command has no effect if the current camera is an orthographic one.

`viewer [<number>] getCameraHeightAngle`

Returns the height angle of a perspective camera.

`viewer [<number>] setCameraHeight <height>`

Sets the height of the view volume of an orthographic camera. The command has no effect if the camera is an perspective one.

`viewer [<number>] getCameraHeight`

Returns the height of an orthographic camera.

`viewer [<number>] setCameraType <perspective|orthographic>`

Sets the camera type.

`viewer [<number>] getCameraType`

Returns the camera type.

`viewer [<number>] setTransparencyType <type>`

This command defines the strategy used for rendering transparent objects. The argument *type* may be a number between 0 and 6, corresponding to the entries *Screen Door*, *Add*, *Add Delay*, *Add Sorted*, *Blend*, *Blend Delay*, and *Blend Sorted* as described for the View menu.

Most accurate results are obtained using mode 6, which is the default. However, some objects may not be recognized correctly as being transparent. In this case you may switch them off and on again in order to force them to be rendered last. Also, if lines are to be rendered on a transparent background problems may occur. In this case, you may use transparency mode 4 and ensure the correct rendering order manually.

`viewer [<number>] getTransparencyType`

This command returns the current transparency type as a number in the range 0...6. The meaning of this number is the same as in `setTransparencyType`.

`viewer [<number>] setBackgroundColor <r> <g> <b>`
This command sets the color of the background to a specific value. The color may be specified either as a triple of integer RGB values in the range 0...255, as a triple of rational RGB values in the range 0.0...1.0, or simply as plain text, e.g., *white*, where the list of allowed color names is defined in `/usr/lib/X11/rgb.txt`.

`viewer [<number>] getBackgroundColor`
Returns the primary background color as an RGB triple with values between 0 and 1.

`viewer [<number>] setBackgroundColor2 <r> <g> <b>`
Sets the secondary background color which is used by non-uniform background modes.

`viewer [<number>] getBackgroundColor2`
Returns the secondary background color as an RGB triple with values between 0 and 1.

`viewer setBackgroundMode <mode>`
Allows you to specify different background patterns. If mode is set to 0 a uniform background will be displayed. Mode 1 denotes a gradient background. Mode 2 causes a checkerboard pattern to be displayed. This might help to understand the shape of transparent objects. Finally, mode 3 draws an image previously defined with `setBackgroundImage` on the background.

`viewer getBackgroundMode`
Returns the current background mode.

`viewer setBackgroundImage <imagefile> [<imagefile2>] [-stereo]`
This command allows you to place an arbitrary raster image into the center of the viewer's background. The image must not be larger than the viewer window itself. Otherwise it will be clipped. The format of the image file will be detected automatically by looking at the file name extension. All formats mention for the `snapshot` command are supported except of Encapsulated Postscript. If a second image file is specified, this file will be used as the right eye image in case of active stereo rendering. If the options `-stereo` is specified and only one image file is given, it it assumed that this file contains a left eye view and a right eye view composited side by side. These views then will be separated automatically.

`viewer getBackgroundImage`
This command returns the file name of the last background image file defined with `setBackgroundImage`. If a pair of stereo images was specified, two file names are returned. If the option `-stereo` was used in `setBackgroundImage`, this option will be returned too.

`viewer [<number>] setAutoRedraw <state>`
If *state* is 0, the auto redraw mode is switched off. In this case the image displayed in the viewer window will not be updated, unless a redraw command is sent. If *state* is 1, the auto redraw mode is switched on again. In a script it might be useful to disable the auto redraw mode temporarily.

`viewer [<number>] isAutoRedraw`
Returns true is auto redraw mode is on.

`viewer [<number>] redraw`

This command forces the current scene to be redrawn. An explicit *redraw* is only necessary if the auto redraw mode has been disabled.

`viewer [<number>] rotate <degrees> [x|y|z|m|u|v]`

Rotates the camera around an axis. The axis to be taken is specified by the second argument. The following choices are available:

x: the x-axis (1,0,0)
y: the y-axis (0,1,0) z: the z-axis (0,0,1) m: the most vertical axis of x, y, or z u: the viewer's up direction v: the view direction

The last option does the same as the rotate button of the user interface. In most cases the *m* option is most adequate. For backward-compatibility the defualt is *u*.

`viewer [<number>] setDecoration <state>`

The decoration is an extended window frame that serves as a user-interface. It contains buttons and thumb wheels for adjusting the view or switching between interaction and viewing mode. The `decoration` command can be used to show or hide the decoration area. Hiding the decoration is useful when multiple viewers are open and the size of a single viewing window is rather small.

`viewer [<number>] saveScene <filename>`

Saves all of the geometry displayed in a viewer in Open Inventor 3D graphics format. Warning: Since many amira modules use custom Inventor nodes, the scene usually can not be displayed correctly in external programs like *ivview*.

`viewer [<number>] viewAll`

Resets the camera so that the whole scene becomes visible. This method is called automatically for the first object being displayed in a viewer.

`viewer [<number>] show`

This command opens the specified viewer and ensures that the viewer window is displayed on top of all other windows on the screen.

`viewer [<number>] hide`

This command closes the specified viewer.

`viewer [<number>] fogRange <min> <max>`

Sets a range of attenuation for the fog affect that can be introduced into a viewer scene by the View menu. The default range is $[0, 1]$. Values within this range correspond to distances of scene points from the camera, such that points nearest to the camera have value zero and those farthest away have value one. Restricting the range of attenuation means that attenuation will start at points where the specified minimum is attained and reach its maximum at points where the specified maximum is attained. Maximum attenuation by fog is equivalent to invisibility, thus all points beyond that maximum will appear as background.

`viewer [<number>] setVideoFormat pal|ntsc`
Sets the size of the viewer window according to PAL 601 or NTSC 601 resolution, i.e., 720x576 pixels or 720x486 pixels. The current setting of the decoration is taken into account.

`viewer [<number>] setVideoFrame <state>`
If *state* is 1, a frame is displayed in the overlay plane of the viewer. This frame depicts the area where images recorded to video are safely shown on video players. Setting *state* to 0 switches the frame off. Note: Objects displayed in the overlay planes are not saved to file with the `snapshot` command (see above).

5.3.3.2 Main window command options

The command `theMain` allows you to access and control the amira main window. Besides the specific command options listed below also all sub-commands listed in Section 5.3.3.4 (Common commands for top-level windows) can be used.

Commands

`theMain snapshot` *filename*
Creates and saves a snapshot image of the main window. The format of the image file is determined from the file name extension. Any standard image file format supported by amira can be used, e.g., `.jpg`, `.tif`, `.png`, or `.bmp`.

`theMain setViewerTogglesOnIcons {0|1}`
Enables or disables the display of the orange viewer toggles on object icons in the amira object pool.

`theMain ignoreShow [0|1]`
Enables or disables the special purpose *no show flag*. If this flag is set, subsequent `mainWindow show` commands are ignored. This can be useful to run standard amira scripts in a amiraVR environment. Calling the command without an argument just returns the current value of the flag.

5.3.3.3 Console command options

The command `theMsg` allows you to access and control the amira console window. Besides the specific command options listed below also all sub-commands listed in Section 5.3.3.4 (Common commands for top-level windows) can be used.

Commands

`theMsg error <message> [<btn0-text>] [<btn1-text>] [<btn2-text>]`
Pops up an error dialog with the specified message. The dialog can be configured with up to three

different buttons. The command blocks until the user presses a button. The id of the pressed button is returned.

`theMsg warning <message> [<btn0-text>] [<btn1-text>] [<btn2-text>]`

Pops up a warning dialog with the specified message. The dialog can be configured with up to three different buttons. The command blocks until the user presses a button. The id of the pressed button is returned.

`theMsg question <message> [<btn0-text>] [<btn1-text>] [<btn2-text>]`

Pops up a question dialog with the specified message. The dialog can be configured with up to three different buttons. The command blocks until the user presses a button. The id of the pressed button is returned.

`theMsg overwrite <filename>`

Pops up a dialog asking the user if it is ok to overwrite the specified file. If the user clicks *Ok*, 1 is returned, otherwise 0.

5.3.3.4 Common commands for top-level windows

These commands are available for all amira objects which open a separate top-level window. In particular, these are the amira main window (`theMain`), the console window (`theMsg`), and the viewer window (`viewer 0`). For example, you can set or get the position of these windows using the corresponding global command followed by `setPosition` or `getPosition`.

Commands

`getFrameGeometry`

Returns the position and size of the window including the window frame. In total four numbers are returned. The first two numbers indicate the position of the upper left corner of the window frame relative to the upper left corner of the desktop. The last two numbers indicate the window size in pixels.

`getGeometry`

Returns the position and size of the window without the window frame. In total four numbers are returned. The first two numbers indicate the position of the upper left corner of the window relative to the upper left corner of the desktop. The last two numbers indicate the window size in pixels.

`getPosition`

Returns the position of the upper left corner of the window including the window frame. This is the same as the first two numbers returned by `getFrameGeometry`.

`getRelativeGeometry`

Returns the position and size of the window including the window frame in relative coordinates.

The size of the desktop is (1,1). The position and size of the window is specified by fractional numbers between 0 and 1.

`getSize`
Returns the size of the window without the window frame. This is the same as the last two numbers returned by `getGeoemtry`.

`hide`
Hides the window.

`setCaption <text>`
Sets the window title displayed in the window frame.

`setFrameGeometry <x y width height>`
Sets the position and size of the window including the window frame. Four numbers need to be specified, the x- and y-positions, the window width and the window height.

`setGeometry <x y width height>`
Sets the position and size of the window without the window frame. Four numbers need to be specified, the x- and y-positions, the window width and the window height.

`setPosition <x y>`
Sets the position of the upper left corner of the window frame.

`setRelativeGeometry <x y width height>`
Sets the position and size of the window including the window frame in relative coordinates. The size of the desktop is (1,1). The position and size of the window is specified by fractional numbers between 0 and 1.

`setSize <width height>`
Sets the size of the window without the window frame.

`show`
Makes the window visible in normal state. Also raises the window.

`showMinimized`
Makes the window visible in iconified state.

`showMinimized`
Makes the window visible in maximized state.

5.3.3.5 Progress bar command options

The command `theWorkArea` allows you to access the progress bar located in the lower part of the amira main window. You can print messages or check if the stop button was pressed.

Commands

`theWorkArea setProgressInfo <text>`

Sets an info text to be displayed in the progress bar. The text can be used to describe the status during some computation.

`theWorkArea setProgressValue <value>`

Sets the value of the progress bar. The argument must be a floating point number between 0 and 1. For example, a value of 0.8 indicates that 80% of the current task has been done.

`theWorkArea startWorking [<message>]`

Activates the stop button. After calling this command the amira stop button becomes active. In your script you can check if the stop button was hit by calling `theWorkArea wasInterrupted`. When the stop button is active you can't interact with any other widget unless you call `theWorkArea stopWorking` in your script. Therefore you must not enter this command directly in the console window, but you should only use it in a script file or in a Tcl procedure.

`theWorkArea stopWorking`

Deactivates the stop button. Call this command when the compute task started with `theWorkArea startWorking` is done or if the user pressed the stop button. This command also restores the progress info text which was shown before calling `startWorking`.

`theWorkArea wasInterrupted`

Checks if the user pressed the stop button. You should only use this command between `theWorkArea startWorking` and `theWorkArea stopWorking`. If there are multiple nested compute tasks and the user presses the stop button, all subsequent calls to `wasInterrupted` return true until the first level is reached.

5.3.3.6 Application command options

The `app` command provides several options not related to a particular object or component in amira, but related to amira itself.

Commands

`app version`

Returns the current amira version.

`app uname`

Returns simplified name of operating system.

`app arch`

Returns amira architecture string, e.g., arch-Win32-Optimize, arch-IRIX64-Optimize.

`app hostid`

Returns host id needed to create an amira license key.

`app listen [port]`
Opens a socket to which Tcl commands can be sent. The TCP/IP port can be specified optionally. WARNING: This can create security holes. Do not use this unless behind a firewall and if you know what you are doing.

`app close`
Closes the amira Tcl port.

`app port`
Get port number of amira Tcl port. Returns -1 if socket has not been opened.

`app send <command> [<host> [<port>]]`
Sends a Tcl command to a listening amira. If no host or port are specified, the amira instance will send the command to itself.

`app opengl`
Retrieve information about the used OpenGL driver including version number and supported extensions. This is useful information to send to the hot-line if reporting rendering problems.

5.3.3.7 Other global commands

Commands

`addTimeout` *msec procedure* `[`*arg*`]`
Schedules a Tcl procedure for being called after *msec* milliseconds. If *arg* is specified it will be passed to the procedure. The specified procedure will be called only once. If necessary, you can schedule it again in the time-out procedure. Example: `addTimeout 10000 echo {10 seconds are over.}`

`all [-selected | -visible | -hidden] [`*type*`]`
Returns a list of all amira objects currently in the object pool. If *type* is specified only object with that C++ class type (or derived objects) are returned. Search can be limited to selected, visible, or hidden objects, respectively. Example: `all -hidden HxColormap`.

`aminfo [-a` *outfile*`|-b` *outfile*`]` *AmiraMesh-File*
If used with only a file name as argument this command will open the file which has to be in AmiraMesh format and print header information. If used with the -a or -b option,

`clear`
Clears console window.

`create` *class name* `[`*instance name*`]`
Creates an instance of an amira object like a module or data object. Returns the instance name. Note that data objects are normally not created this way but by loading them from a file. Example: `create HxOrthoSlice MySlice`.

`dso` *options*
Controls loading of dynamic libraries. The following options are provided:

- `addPath path ...` Adds a path to the list of directories to be searched when loading a dynamic library.
- `verbose` {`0|1`} Switches on and off debug information related to dynamic shared object loading.
- `open <package>` Trys to load the specified dynamic library. It is enough to specify the package name, e.g., `hxfield`. This name will be automatically converted into the platform dependent name, e.g., `libhxfield.so` on Linux or `hxfield.dll` on Windows.

`echo` *args*
Prints its arguments to the amira console. Use this rather than the native Tcl command `puts` which prints to stdout.

`help` *arguments*
Without arguments this opens the amira help browser.

`httpd [port]`
Start a built-in httpd server. The http server will deliver any document requested. If a requested document ends with `.hx` amira will instead of delivering it execute the file as a Tcl script. This can be used to control amira from a web browser. WARNING: This command can create security holes. Do not use this unless behind a firewall and if you know what you are doing.

`limit` {`datasize | stacksize | coredumpsize`} *size*
Change process limits. Available on Unix platforms only. Use “unlimited” as size for no limit. The size has to be specified in bytes. Alternatively you can use for example 1000k for 1000 kilobytes or 1m for one megabyte.

`load [fileformat]` *files*
Load data from one or more files. Optionally a file format can be specified to override amira’s automatic file format recognition. The file format is specified by the same label which is displayed in the file format combo box in the amira file dialog.

`mem`
Prints out some memory statistics.

`quit`
Immediately quits amira.

`remove` *objectname*
Removes the specified amira object.

`removeTimeout` *procedure* `[`*arg*`]`
Unschedules a Tcl procedure previously scheduled with addTimeout.

`rename` *objectname newname*
Changes instance name of an object. Identical to *objectname* `setLabel` *newname*. Returns the new name, which is normally identical to *newname*, unless the name did already exist.

`sleep` *sec*
Wait for *sec* seconds. amira will not process events in that time.

`source` *filename*
Loads and executes Tcl commands from the specified file. If the script file contains the extension `.hx` the `load` command may be used as well.

`system` *command*
Execute an external program. Do not use this unless you know what you are doing.

5.4 amira Script File

It is worth noticing that an amira network is simply a Tcl script that will regenerate the current amira state. Therefore it is often an efficient way to interactively create an amira network, save it with “Save Network”, and use this as a starting point for scripting.

The simplest way to execute Tcl commands in amira is to type them into the amira console window. This however is not practical for multi-line constructs, like loops or procedures. In this case, it is recommended to write the Tcl code into a file and execute the file with the command `source` *filename*. You can also use the `source` command inside a file in order to include the contents of a file into another file.

Alternatively one can also use the command `load` *filename* or the *Load* menu entry from the *File* menu and the file browser. Then however, in order to let amira recognize the file format, then either the file name has to end with `.hx`, or the file contents must start with the header line

```
# Amira-Script-File
```

There are some Tcl files that are loaded automatically when amira starts. At startup, the program looks for a file called `.Amira` in the current directory or in the home directory (see Section 4.4 (Start-up script) for details). If no such user-defined start-up script is found the default initialization script `Amira.init` is loaded from the directory `AMIRA_LOCAL/share/resources` or `AMIRA_ROOT/share/resources`. This script then reads in all files ending with `.rc` from the `share/resources` subdirectory. The `.rc` files are needed to register modules and data types. Therefore one can customize the startup behavior of amira by simply adding a new `.rc` file to that directory or by modifying the `Amira.init` file.

Another way of executing Tcl code is to define procedures that are associated with function keys. If predefined procedures with the names `onKeyF2, onKeyF3, ..., onKeyShiftF2, ...,` `onKeyCtrlF2, ..., onKeyCtrlShiftF2, ...` exist, these procedures will be automatically called when the respective key is hit in the amira main window, console window, or viewer window. Note that F1 is reserved for help. To define these procedures write them into a file and `source` it or write them into `Amira.init` or in one of the `.rc` files. An example is

```
  proc onKeyF2 { } {
      echo "Key F2 was hit"
      viewer 0 viewAll
  }
```

Finally Tcl scripts can also be represented in the GUI and be combined with a user interface. In amira this is called a script object. There is a separate documentation for that.

5.5 Configuring Popup Menus

In amira all modules which can be attached to a data object are listed in the object's popup menu which is activated by clicking on the object's icon with the right mouse button. For some applications it makes sense to customize new modules using Tcl commands after they have been created. Sometimes it also makes sense to add new entries to an object's popup menu, causing a particular script to be executed. This sections describes how to achieve these goals by modifying amira resource files or creating new ones.

amira resource files are located in the directory `$AMIRA_ROOT/share/resources`, where `$AMIRA_ROOT` denotes the directory where amira has been installed. Resource files are just ordinary script files, although they are identified by the suffix `.rc`. When amira is started all resource files in the resources directory are read. In a resource file modules, editors, and IO routines are registered using special Tcl commands. Registering a module means to specify its name as is should appear in the popup menu, the type of objects it can be attached to, the name of the shared library or DLL the module is defined in, and so on. For example, the *LabelVoxel* module is registered by the following command in the file `hxlattice.rc`:

```
module -name "LabelVoxel" \
    -primary "HxUniformScalarField3 HxStackedScalarField3" \
    -check { ![$PRIMARY hasInterface HxLabelLattice3] } \
    -category "Labelling Compute" \
    -class "HxLabelVoxel" \
    -package "hxlattice"
```

The different options of this command have the following meaning:

- The option `-name` specifies the name or label of the module, as it will be printed in the popup menu.

- The option `-primary` says that this module can be attached to data objects of type `HxUniformScalarField3` or `HxStackedScalarField3`. This means that *LabelVoxel* will be included in the popup menu of such objects only.
- With `-check` an additional Tcl expression is specified which is evaluated at run-time just before the menu is popped up. If the expression fails the module is removed from the menu. In case of the *LabelVoxel* module it is checked if the input object provides a `HxLabelLattice3` interface, i.e., if the input itself is a label field. Although a label field can be regarded as a 3D image, it makes no sense to perform a threshold segmentation on it. Therefore *LabelVoxel* is only provided for raw 3D images, but not for label fields.
- The option `-category` says that *LabelVoxel* should appear in the submenus *Compute* and *Labelling* of the main popup menu. If a module should appear not in a submenu but in the popup menu itself, the category `Main` must be used.
- The option `-class` specifies the internal class name of the module. The internal class name of an object can be retrieved using the command `getTypeId`. It is this class name which has to be used for the `-primary` option described above, not the object's label defined by `-name`.
- Finally, the option `-package` specifies in which package (shared library or DLL) the module is defined in.

Besides these standard options additional Tcl commands to be executed after the module has been created can be specified using the additional option `-proc`. For example, imagine you are working in a medical project where you have to identify stereotactic markers in CT images of the head. Then it might be a good idea to add a customized version of the *LabelVoxel* module to the popup menu, which already defines appropriate material names and thresholds. This could be done by adding the following command either in a new resource file in `$AMIRA_ROOT/share/resources` or directly in `hxlattice.rc`:

```
module -name "Stereotaxy" \
    -primary "HxUniformScalarField3 HxStackedScalarField3" \
    -check { ![$PRIMARY hasInterface HxLabelLattice3] } \
    -category "Labelling" \
    -class "HxLabelVoxel" \
    -package "hxlattice" \
    -proc { $this regions setValue "Exterior Bone Markers";
            $this fire;
            $this boundary01 setValue 150;
            $this boundary12 setvalue 300 }
```

The variable `$this` used in the Tcl code above refers to the newly created module, i.e., to the *LabelVoxel* module. Note, that the commands are executed *before* the module is connected to the source object for which the popup menu was invoked. Some modules do some special initialization when they are connected to a new input object. These initializations may overwrite values set using Tcl commands defined by a custom `-proc` option. In such a case you can explicitly connect the module to the input object via the command sequence

```
    $this data connect $PRIMARY;
    $this fire;
```

Here the Tcl variable `$PRIMARY` refers to the input object. The same variable is also used in Tcl expressions defined by a `-check` option, as described above.

Besides creating custom popup menu entries based on existing modules it is also possible to define completely new entries which no nothing but executing Tcl commands. For example, we could add a new submenu *Edit* to the popup menu of every amira object and put in the *Hide*, *Remove*, and *Duplicate* commands here which are normally contained in the *Edit* menu of the amira main window. This can be achieved in the following way:

```
module -name "Remove" \
    -primary "HxObject" \
    -proc { remove $source } \
    -category "Edit"

module -name "Hide" \
    -primary "HxObject" \
    -proc { $source hideIcon } \
    -category "Edit"

module -name "Duplicate" \
    -primary "HxData" \
    -proc { $source duplicate } \
    -category "Edit"
```

Of course, it is also possible to execute an ordinary amira script or even an amira script object with a `-proc` command.

Part II

amira Reference Manual

Chapter 6

Alphabetic Index of Modules

6.1 Access LargeDiskData

This kind of module is attached to a LargeDiskData object. provides an interface to select a subblock of the large volumen which is stored on disk only.

You may select the volume either by typing in numbers to the ports or by using the blue dragger in the viewer.

Additionally, it is possible to request a lower resolution of the data. Dependent on the file format other resolutions are stored on disk and might be accessed rather fast or they are calculated on the fly which might take some time.

After selecting a volume you might hit the *(re)load* button. An amira field object is created and filled with the data from disk. You can now use all standard amira visualization techniques on this subblock.

Connections

Data `[required]`

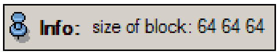
Connection to the LargeDiskData object.

ROI `[optional]`

If a SelectRoi is connected it will define the subvolume to be loaded.

Ports

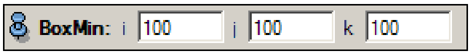
Info



Info: size of block: 64 64 64

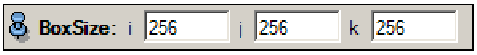
Indicates the size of the block to be loaded.

BoxMin


BoxMin: i 100 j 100 k 100

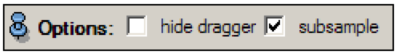
The position of the lower corner of the block. Printed as a lattice index.

BoxSize


BoxSize: i 256 j 256 k 256

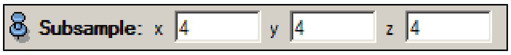
The size of the block to be loaded. The size is in lattice indices in the base resolution. The numbers will not change if you select subsampling but the real size of the block to be loaded will change. This is indicated in the *Info* port.

Options


Options: ☐ hide dragger ☑ subsample

Hides the dragger and/or enables subsampling.

Subsample


Subsample: x 4 y 4 z 4

Type in the number of voxels to be averaged in each dimension.

DoIt

DoIt: (re)load

Load the block. If you snap the button you might change the selected volumen and an immediate reload is started. This might be useful to easily explore a large volume.

6.2 AlignPrincipalAxes

This modules computes a rigid transformation for a triangulated surface, such that its principal axes are aligned to a reference coordinate system. This can either be the standard basis in 3D or the principal axes of a reference triangulated surface.

Connections

Model `[required]`

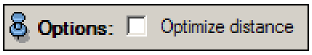
Model surface, that will be transformed.

Reference `[optional]`

Reference surface, whose principal axes will serve as a reference coordinate system.

Ports

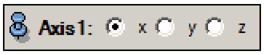
Options



Options: ☐ Optimize distance

This port is only visible, when a reference surface is connected. For each possible solution (see explanation below at Port Action), the root mean square distance is computed, when the option 'Optimize distance' is chosen. The final transformation will be the one yielding the minimum value.

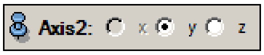
Axis1



Axis1: [✓ x] y z

If no reference surface is connected, the principal axes with the largest moment of inertia will be aligned to the axis specified at this port.

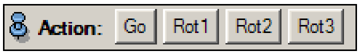
Axis2



Axis2: x [✓ y] z

If no reference surface is connected, the principal axes with the second largest moment of inertia will be aligned to the axis specified at this port.

Action



The 'Go' button starts the alignment. Since the principal axes are only computed up to their orientation the model surface can subsequently be rotated 180 degree around each of its principal axes by pressing one of the 'Rot' buttons. The number corresponds the first, second or third largest moment respectively.

Commands

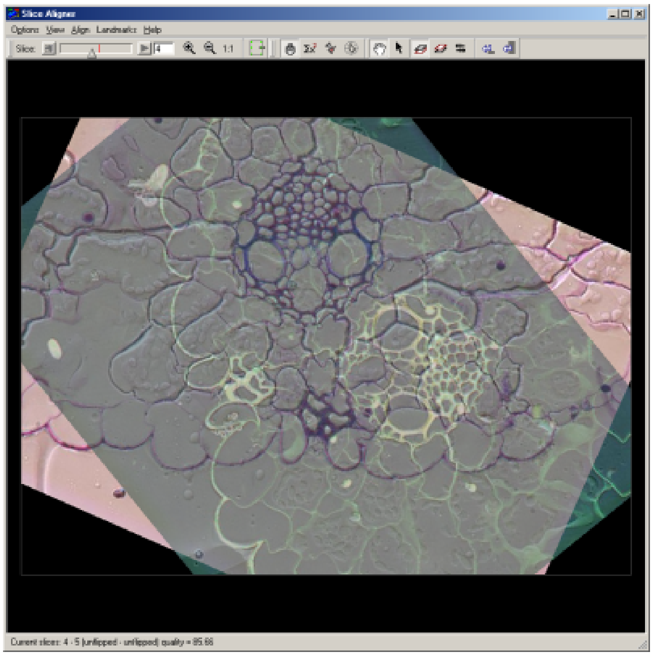
`getRefSystem`

Prints the center of gravity, the axes of inertia, and the moments of inertia in the console.

6.3 AlignSlices

This module allows you to interactively align 2D slices of a 3D image stack. Alignment is performed in a separate graphics window that is activated by pressing the *Edit* button of the module's Action port. In addition to this documentation, there is a separate tutorial about slice alignment contained in the amira User's Guide.

The align window displays by default two consecutive slices using different draw styles that can be selected from the View menu. The two displayed slices can be chosen using the slice slider on the tool



Current slices: 4 - 5 [unflipped - unflipped] quality = 85.66

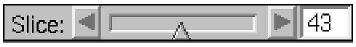
Figure 6.1: User interface of the align tool.

bar. Only one of these two slices is editable at a given time. The slice that can be edited is selectable and it can be either the lower or the upper one. By dragging the editable slice using the left mouse button, the slice will be translated. The slice can be rotated around its center by dragging it keeping the middle mouse button pressed. At any time the numbers of the current slice pair as well as the quality of the current alignment is displayed in the status bar at the bottom of the align window.

The documentation of the align tool is organized into the following sections (click on the links to open a section):

- Tool bar - describes the buttons of the tool bar.
- Menu bar - describes all entries of the menu bar.
- Image viewer - describes how to align images interactively.
- Key bindings - provides a list of all hot keys.

6.3.1 Tool Bar

- **Slice slider:** 

  allows the user to change the slices that are displayed and can be edited. The displayed number is the number of the editable slice. If this number is *n* and the editable slice is the upper one, slices *n* and *n-1* are shown. If the editable slice is the lower one, the displayed slices are slices *n* and *n+1*.

- **Zoom in button:**
  Increases the size of the image.
- **Zoom out button:**
  Decreases the size of the image.
- **Zoom label:**
  Shows the current zoom factor. E.g., a zoom of 2:1 means that 2 pixels on the screen correspond to one pixel of the original data set (magnification), while 1:4 would mean, that four pixels of the data set correspond to one pixel on the screen.
- **Resize Button:**
  Opens the Resize Dialog which coordinates a resize of the output image-stack after the alignment procedure.
- **Gravity centers button:**
  Sets the gravity centers alignment as the current alignment algorithm.
- **Least squares button:**
  Sets the least squares alignment as the current alignment algorithm.
- **Landmarks button:**
  Sets the landmarks alignment as the current alignment algorithm. When this is the active align mode, the landmarks will be displayed for the current slices. If one of the other two algorithms is selected, the landmarks are not shown. The selected algorithm is used when the Align current slices or Align all slices button is pressed or the corresponding menu items are chosen.
- **Edge detection button:**
  Sets the edgedetection alignment as the current alignment algorithm.
- **Edit slice mode:**
  Sets the active edit mode to be editing slices. This means that the editable slice can be translated/rotated.
- **Edit landmarks mode:**
  Sets the active edit mode to be editing landmarks. When this is the active edit mode, the landmarks that have been already defined can be moved/removed and new landmarks can be defined (see Image Viewer section). This mode can be activated only only when Landmarks button is ON (i.e. the active align mode is landmarks alignment).
- **Lower slice:**
  This button forces the lower slice of the two displayed slices to become editable.

- **Upper slice:**
  This button forces the upper slice of the two displayed slices to be editable (default). The editable slice can be translated and rotated and landmarks can be defined for it.
- **Mirror button:**
  A mirroring transformation is applied to the editable slice.
- **Align current slices:**
  The two currently displayed slices are automatically aligned using the selected algorithm.
- **Align all slices:**
  All slices of the given 3D data are automatically aligned using the selected algorithm.

6.3.2 Resize options

The *Resize options* dialog allows the user to give the output images of the image stack a different size than the input images. This is especially useful if the slices are rotated or moved during the alignment procedure and some areas of single slices are outside the borders of the image stack. Beside this, a background color for the new areas in the image can be determined. The following image shows the dialog box.

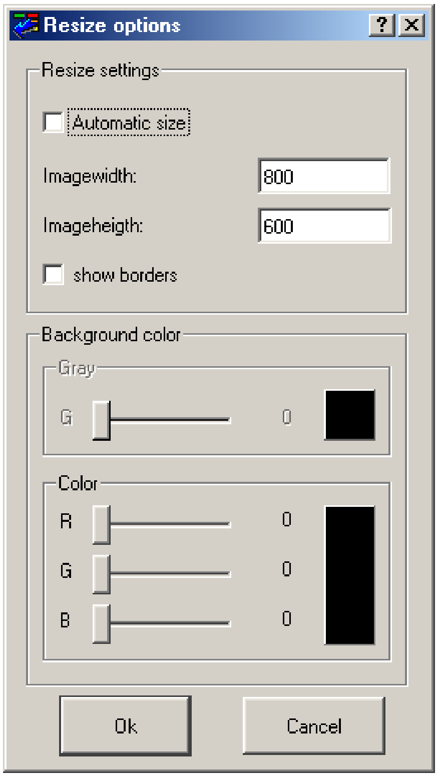
The following parameters can be set using this dialog:

- **Automatic size:** If this feature is activated, a minimal image size is calculated which contains all pixels of all slices. The dimensions and the position of the new bounding box are set that all slices fit in this new bounding box.
- **Imagewidth:** Sets the width of the output image.
- **Imageheight** Sets the height of the output image.
- **Show borders** Shows the new image borders of the output image in the *Slice Aligner*.
- **Background color** The user can set the background color for the output image. The color can be set as a gray value or as an RGB color, depending on the type of the input image. The current background color is depicted in the right box.

6.3.3 Menu Bar

6.3.3.1 Options

- **Undo:** This menu provides an undo feature that undoes the last operation. Successive invocation of *Undo* is possible, allowing to undo several operations. Each operation made manually or automatically (translation, rotation, automatic alignment, landmarks editing) can be undone. The maximum number of operations that can be undone is 100.
- **Redo:** This menu provides a redo feature that redoes the last undone operation. Successive invocation of *Redo* is possible after several undo operations.



Resize settings

Automatic size: ☐

Imagewidth: 800

Imageheigth: 600

show borders: ☐

Background color

Gray

G 0

Color

R 0

G 0

B 0

Figure 6.2: The *Resize options* window.

- **Reset all:** Set the translations on X and Y and the rotation to 0 for all slices.
- **Reset:** Set the translations on X and Y and the rotation to 0 for the current slice (the slice that is currently editable).
- **Transform all:** This is a toggle option. If the option is ON, the transformations made for the editable slice are also made for all slices above when the editable slice is the upper one. That means that the existing alignment of all other slices not imvolved in the current transformation will be preserved.
- **Fix reference:** This option decides if a certain slice is used as a reference slice during the entire alignment procedure. The reference slice is marked by a red tag in the *Slice slider*. The other slices are to aligned according to this slice.
- **Read transformation:** Reads the *AlignTransform* and *AlignPoints* parameters from the input data and sets the transformation values accordingly. This is useful to return to a previous saved alignment. This can be seen as another kind of reset or undo.
- **Save transformation:** Save the actual translations, rotations and landmarks as parameters in the input data.
- **Close:** Close the align tool.

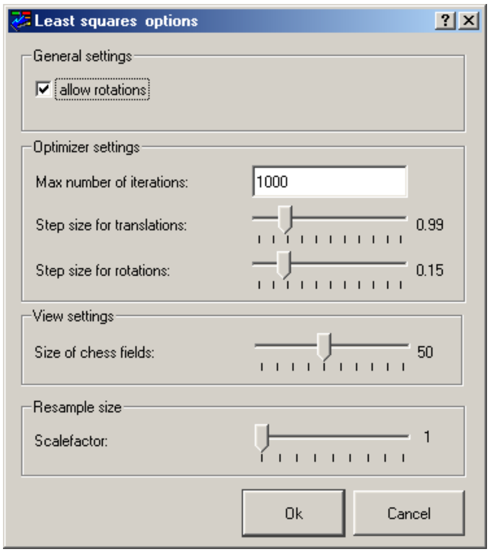
6.3.3.2 View

- **Zoom In:** Increases the magnification factor of the image.
- **Zoom Out:** Decreases the magnification factor of the image.
- **Red green:** If this option is set, the two images are displayed in magenta and green, respectively. If both images match perfectly a gray image is obtained. Color images are first converted to grayscale and then transformed to magenta and green.
- **Checkerboard:** If this option is set, one image is displayed in the white parts of a checkerboard pattern while the other image is displayed in the black parts. The size of the parts can be adjusted in the *Options* dialog (see below).
- **Average:** If this option is set the, two images are averaged or blended in the viewer window.
- **Invert:** If this option is set, the lower image is inverted. The inverted image is then blended with the upper image. If both images match perfectly a constant gray image is obtained. This option is the default.

6.3.3.3 Align

This menu offers several alignment algorithms that can be selected in order to obtain the best alignment. The performance of each algorithm is dependent on the data that has to be aligned. There are four alignment algorithms that can be chosen:

- **Gravity centers** - align gravity centers and principal axes
- **Gray values** - least squares algorithm based on gray values


Least squares options

General settings

☑ allow rotations

Optimizer settings

Max number of iterations: 1000

Step size for translations: 0.99

Step size for rotations: 0.15

View settings

Size of chess fields: 50

Resample size

Scalefactor: 1

Figure 6.3: The *Least squares options* window.

- **Landmarks** - align user-defined sets of landmarks
- **Edge detection** - align the outer bounds of the objects

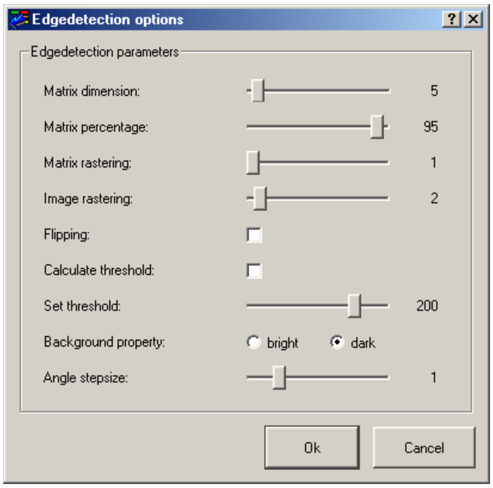
The *Options* button opens a dialog window allowing you to define certain settings of the automatic least squares algorithm and the edge detection algorithm. The button is only enabled when either the gray values algorithm or the edge detection algorithm is selected.

The following parameters for the gray values algorithm can be set using this dialog:

- **allow rotations** - if this option is checked, rotations are considered during the alignment process. By default this option is checked. Translations are always considered.
- **max number of iterations** - this is only intended to prevent an infinite loop.
- **step size for translations** - the step size used to search for better positions in X and Y. A small step size, though in general may lead to more accurate results, also slows down alignment.
- **step size for rotations** - the step size used for rotations.
- **Size of checkerboard** - adjusts the size of the pattern if view mode is set to *checkerboard*.
- **Scale factor** - during the alignment procedure, the image can be downsampled for the first iterationsteps so that a better result can be reached in a shorter time.

If the *Edge detection* algorithm is selected, the dialog window has the following appearance.

The parameters for the *Edge detection* algorithm can be set using this dialog:


Edgedetection options

Edgedetection parameters

Matrix dimension:	5
Matrix percentage:	95
Matrix rastering:	1
Image rastering:	2
Flipping:	☐
Calculate threshold:	☐
Set threshold:	200
Background property:	bright [✓ dark]
Angle stepsize:	1

Figure 6.4: The *Edgedetection options* window.

- **Matrix dimension** This is the dimension of the matrix which scans the image and decides whether a pixel belongs to the object or the background.
- **Matrix percentage** This percentage defines the amount of pixels in the surrounding of a pixel which have to belong to the object to indicate that this pixel belongs likewise to the object. The considered surrounding is defined by the matrix dimension.
- **Matrix rastering** This indicates if the surrounding should be rasterized. If so, the alignment is done in a shorter time.
- **Image rastering** This decides if the image should be downsampled during the alignment procedure. This speeds up the alignment.
- **Flipping** This option indicates if a possible flipping of the images should be considered.
- **Calculate a threshold** This activates an automatic calculation of the threshold.
- **Set threshold** This allows the threshold to be set manually.
- **Background property** This must be set to describe the background of the image. According to this setting, the threshold is interpreted as an upper or a lower boundary of gray values which separate the object from the background.
- **Angular stepsize** This sets the size of the search algorithm's angular steps (in degrees). A higher value speeds up alignment at the cost of accuracy.

More information about the align methods and the appropriate settings for the methods can be found in the alignment section.

6.3.4 Alignment Methods

The four different alignment methods are based on different concepts and thus the results may differ significantly. However, not only the method decides on the result but the right settings for the method. This page shall help to use the best alignment method and to use the right settings. Therefore, the algorithms behind the methods are shortly described. This requires a little background knowledge about mathematics.

- **Gravity centers:** This method calculates a center of gravity of the gray values and the orientation by means of the covariances of the image.
- **Gray values:** This method compares the single gray values of two images. The more pixels of two compared slices have the same gray values, the better the alignment will be valued for this method. That means that this method tries to move one slice and calculates the difference of the gray values of both images. If the quality is getting higher after the movement, this method will keep on changing the positions of to slices to each other until a maximum quality according to the above mentioned feature will be reached.
- **Landmarks:** This method is the one that requires most user interaction. The user can define some points, so called landmarks, which will be aligned to each other during the process. The method is mostly one of the best because the user decides about on the alignment. The problem of this method is that the user has to set all of these landmarks so that this method is very time-intensive.
- **Edge detection:** This method is an outline-based method. It works in two steps. In the first step the method tries to identify the object and clears the surrounding of any noise. If the object is separated from the background, the outlines of the objects in two successive slices will be compared and aligned with each other.
-

Two methods allow to define some settings. These settings often decide on the quality of an alignment procedure. Sometimes, unintentional artifacts appear which result from wrong settings.

- **Gray values:** This is the first method that allows the user to set some parameters. As described above, the alignment compares the gray values of two successive slices. The possible settings are described in the Least-squares options window. The scale factor defines if an image is re-scaled before the entire process. This means that the image will be resampled in a *scale factor*-times coarser resolution. This speeds up the first steps of the process to find a first maximum. If this factor is too large, the editable slice could *flee* out of the canvas. This is a normal behavior since only points which overlap each other are considered for the calculation of the alignment quality, i.e., if one image moves out of the canvas the quality will be calculated as one hundred percent.
- **Edge detection:** This method has several settings which determin quality and speed of the alignment. As mentioned above, the alignment is divided into two phases. In the first phase, the object must be separated from the background, and in the second phase the rotation of two objects is calculated according to their outlines. The separation of image and background is

achieved by using a matrix which decides according to a surrounding if one pixel belongs to the object or the background. The size of the surrounding region is set by the first parameter. The higher this value is, the slower the alignment procedure will be. If the background noise is rather small, the matrix size can be set small in order to speed up the alignment procedure. If an image has an high resolution, the image rastering can be set to value greater than one. This speeds up the alignment procedure considerably. If the object has dis-symmetrical outlines, the angle-step-size can be set on a higher value. This also speeds up the alignment procedure. If the outlines are symmetrical, it is strongly recommended to set the step size to a small value.

6.3.4.1 Landmarks

The buttons of this menu are only active if Edit landmarks mode is selected. The buttons have the following meaning:

- **Add:** set the shape of the mouse cursor to a black landmark and clicking on the slice a new landmark is created.
- **Remove:** delete the selected landmark. This menu item is active only when a landmark is selected and the number of landmarks is greater than 3.

6.3.4.2 Help

The help menu opens the help viewer with the documentation of the align tool.

6.3.5 Image Viewer

The *Image Viewer* represents the main part of the align tool. It allows to align the slices manually, using the mouse and/or the keyboard. While two slices are displayed simultaneously, only one slice can be edited at a time.

A slice can be translated over the other by keeping the left mouse button pressed and dragging the slice. Also, the slice can be moved using the cursor keys (see Key Bindings section). The rotation can be obtained by dragging the slice while keeping the middle mouse button pressed. All rotations are made around the middle of the slice.

If the Transform all option is ON (Options menu), the actual alignment will be preserved for all pairs of two consecutive slices (except the slices actually edited).

The *Image Viewer* alows to edit the landmarks. In order to edit the landmarks, the Edit landmarks mode must be set.

A landmark can be **selected** by clicking inside the triangle that represents the landmark. Once selected, a landmark can be moved, removed or deselected. The landmarks can be also selected consecutively, by clicking anywhere on the slice (not on a landmark). After a so selected landmark has been moved, the next landmark in the list can be selected in the same way.

In order to **move** a selected landmark, just click on the new position.

A selected landmark can be **removed** by choosing the *Remove* item of the Landmarks menu or pressing the **Del** key. A landmark can be removed only if the number of landmarks defined for each slice is greater than 2. The corresponding landmark will be removed from each slice so that the number of landmarks is the same for each slice.

You can create a **new** landmark using the *Landmark/Add* menu or pressing the **Ins** key and clicking on the desired position. A new landmark will be created for each slice.The creation of a new landmark can be canceled by pressing **Esc**.

6.3.6 Key Bindings

- **Changing the *slice number***
  Space - Go to next slice
  Backspace - Go to previous slice
  Home - Go to first slice
  End - Go to last slice

- **Translating the editable slice**
  CursorUp - Translation one pixel up
  CursorDown - Translation one pixel down
  CursorLeft - Translation one pixel left
  CursorRight - Translation one pixel right
  Shift+CursorUp - Translation five pixels up
  Shift+CursorDown - Translation five pixels down
  Shift+CursorLeft - Translation five pixels left
  Shift+CursorRight - Translation five pixels right

- **Rotating the editable slice**
  Ctrl+CursorUp - 0.10 degree counter clockwise rotation
  Ctrl+CursorDown - 0.10 degree clockwise rotation
  Ctrl+Shift+CursorUp - 1.0 degree counter clockwise rotation
  Ctrl+Shift+CursorDown - 1.0 degree clockwise rotation

- **Changing the editable slice**
  1 - Set the editable slice to be the lower slice. While you hold the key down, only the lower slice will be visible.
  2 - Set the editable slice to be the upper slice. While you hold the key down, only the upper slice will be visible.

- **Changing the alignment algorithm**
  C - set the gravity centers alignment as current alignment algorithm.

G - set the least squares alignment as current alignment algorithm.
L - set the landmarks alignment as current alignment algorithm.
E - set the edge detection alignment as current alignment algorithm.

- **Editing landmarks**
  Insert Set the shape of the mouse cursor to a black landmark. Clicking on the slice a new landmark is to be created.
  Delete Delete the selected landmark. This takes effect only when the number of landmarks is greater than 3.
  Escape Pressing Escape if a landmark is selected causes this landmark to be deselected. Pressing Escape after the Insert key was pressed, causes the insert operation to be aborted.
- **Changing the edit mode**
  Escape - pressing Escape allows you to switch between editing slices mode and editing landmarks mode. however this doesn't happen if the current edit mode is editing landmarks and a landmark is selected. In this case the landmark will be deselected. Pressing Escape once more causes editing slices to become the new edit mode.

Connections

Data `[required]`

3D field for which the slices will be aligned. Currently, uniform Cartesian grids of type *HxUniformScalarField3*, *HxUniformStackedScalarField3*, and *HxUniformColorField3* are supported.

Reference `[optional]`

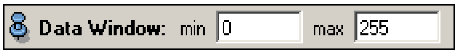
3D field from which the alignment informations can be read and transferred to the field that must be aligned (port **Data**). The reference field must have the same number of slices and the same dimensions as the data field.

Mask `[optional]`

An additional *HxLabelLattice3* that can be connected in order to be used as mask during the alignment process.

Ports

Data window


Data Window: min 0 max 255

This port allows the user to restrict the range of visible data values. Values below the lower bound are mapped to black, while values above the upper bound are mapped to white. Only the values between the two bounds are used by the gray value-based alignment algorithm. This port is only active if a gray value image is used as input data.

Resample method

This port allows the user to decide which resample method will be used for the calculation of an output image. The calculation takes place when the *apply* or the *realign* button from the Action port is pressed. Two resample methods are possible:

- **Preview:** A very quick but not very precise method. It should only be used as solution preview.
- **High Quality:** A much slower but very accurate interpolation method.

Action

Press the *Edit* button to open the align tool. Press the *Resample* button to create a new field according to the transformations made in the align tool. The new field will have per default the dimensions of the input image. The dimensions can be altered by the resize dialog. Press the *Resample to Result* button to overwrite an attached result. If a labelfield is attached to the existing result, the labelfield will also be transformed so that former labels will also fit after the align procedure.

Commands

`performance`
Computes the number of frames per second in the *Image Viewer*.

`setSliceNumber n`
Sets the current editable slice to be *n*.

`setEditableSlice lower|upper`
Of the two currently displayed slices, sets the editable slice to be the lower/upper one.

`translate tx ty`
Translates the currently editable slice by *tx* on the X axis, respectively by *ty* on the Y axis.

`rotate phi`
Rotates the currently editable slice by *phi*. Angle *phi* is considered to be in degrees and the rotation is made counterclockwise.

`setAlignMode 0|1|2|3`
Changes the current alignment mode. The meaning of the parameter is 0 = principal axes alignment, 1 = least squares alignment, 2 = landmarks alignment and 3 = edge detection alignment.

`setViewMode 0|1|2|3`
Changes the display mode. The view modes are encoded as follows: 0 = magenta/green view, 1 = checkerboard view, 2 = average view, 3 = invert view.

`align`

Aligns the current slices pair.

`alignAll`

Aligns all slices.

`setLandmark i k x y`

Sets the landmark with index *k* of slice *i* to be the point *(x,y)*.

`edit`

Shows the tool window (the same as the *Edit* button of the the Action port).

6.4 AlignSurfaces

This module computes a rigid transformation (6 degrees of freedom) to align two triangulated surfaces. Several different alignment strategies are implemented:

- Minimization of the root mean square distance between the points of the model surface to corresponding points on the reference surface (often referred to as Procrustes method):
  the corresponding points will be the closest points on the reference surface in the Euclidean measure. The iteration of this process is called the iterative closest point algorithm (ICP). If fixed correspondences have been pre-computed by some other method, these can be used alternatively.
- Alignment of the centers of mass: all nodes of the triangulations are assigned the same mass.
- Alignment of the principal axes of the inertia tensor: since the principal axes are defined modulo orientation (see also AlignPrincipalAxes), the final solution is the one with the minimum root mean square distance.

Connections

Data `[required]`

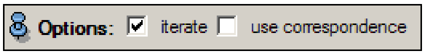
Surface to be transformed onto the reference.

Reference surface `[required]`

Reference surface to be aligned to.

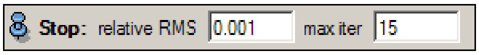
Ports

Options


Options: [✓] iterate [ ] use correspondence

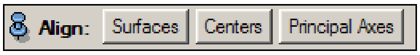
Offers the possibility to iterate the process of finding the closest points on the reference surface (ICP) and to use pre-determined correspondences, if existent.

Stop



Stopping criteria for the ICP algorithm: either a given value for the root mean square distance relative to its value in the first iteration, or a maximum number of iterations.

Align



Starts one of the three different alignment methods, as described above.

6.5 Animate

The Animate module can be connected to any other object to animate its visibility or to animate the value of one of the following ports: PortFloatSlider, PortIntSlider, PortFloatTextN, or PortIntTextN. The animate module provides a time port which can be used to set the value of one of these ports of the other object. The time value is not used directly to set the other port's value but is modified by a user-defined expression. For example, if the time is running from 0 to 100 and you want to animate a slice slider from 0 to 50, you have to use the expression *t/2*. More complicated expressions involving functions such as *sin* or *cos* can be used as well. For a description of supported functions please refer to the Arithmetic module.

Connections

Object `[required]`

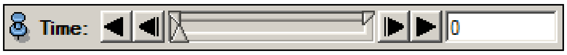
Connection to the object containing the port to be animated.

Time `[optional]`

Optional connection to a time object. If the port is connected to a time object the time of that object is used. In this way multiple modules with a time port can be synchronized.

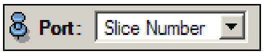
Ports

Time



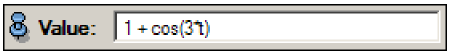
This slider allows you to set or animate the current time value. The time range and the increment used for animation can be adjusted in the configure dialog which can be popped up using the right mouse button.

Port


Port: Slice Number

Option menu listing all ports of the connected object which can be animated. The selected port actually will be animated. The special entry *visible* corresponds to the viewer mask rather than an actual port.

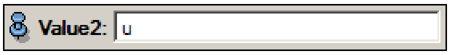
Value


Value: 1 + cos(3*t)

Use this text field to specify the expression used to compute the actual value of the port to be animated. The expression can include two different variables, namely *t* denoting the current time value, and *u* denoting the previous value of the port.

For the *visible* entry , the expression is interpreted as a boolean expression, where 0 means invisible, while a number different from 0 means visible. You can use expressions like `t>20` or `( (t>5) && (t<15)) || (t>100)`.

Value2


Value2: u

If the port to be animated has multiple text fields one expression port will be shown for each text field. If you don't want to change the value of a particular text field, use the expression *u*, which is the previous value of the port.

6.6 Annotation

This module displays a 2D text in the 3D viewer. Position and color of the text can be specified by the user. This is useful for annotating snapshots. In order to create the module choose *Annotation* from the main window's *Edit Create* menu.

For color legends refer to the module Display Colormap.

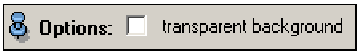
Ports

Color

Color: ☐

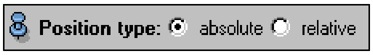
Text color. The color can be changed using the color editor, which is popped up when the color button is clicked.

Options


Options: ☐ transparent background

If this toggle is off the annotation text is drawn inside a filled rectangle. The background color of this rectangle can be set via the Tcl command `setBackgroundColor`.

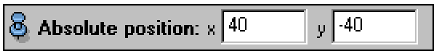
Position type


Position type: [✓ absolute] relative

Radio box defining whether the text position should be specified in *absolute* coordinates (screen pixels) or in *relative* coordinates ranging from (0,0) in the lower left corner to (1,1) in the upper right corner.

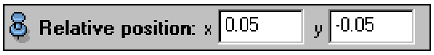
In both cases when the *x* coordinate is negative the right side of the text is placed relative to the right side of the viewer. Likewise, if the *y* coordinate is negative the top side of the text is placed relative to the top side of the viewer.

Absolute position


Absolute position: x 40 y -40

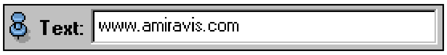
Text position in screen pixels.

Relative position


Relative position: x 0.05 y -0.05

Text position in relative coordinates.

Text


Text: www.amiravis.com

The text to be displayed.

Commands

`getFontSize`
Returns the current font size.

`setFontSize <points>`
Changes the font size. The default font size is 14 points.

`setBackgroundColor <color>`
Sets the color of the background rectangle which is drawn when the *transparent background* toggle is off.

6.7 AnonymizeImageStack

In the *Parameters* section of an image stack created from *DICOM* data the patient's name may occur at several locations:

- in the parameter bundle DICOM-¿All, Parameter 'G0010-0010'
- in the parameter bundles DICOM-¿Slice..., Parameters 'Filename' and 'Name'

Module *AnonymizeImageStack* creates a copy of the image stack and removes all occurences of the patient's name from its parameter section.

Connections

Data
This port should be connected to the image stack to be anonymized.

Ports

Action

The *DoIt* button initiates creation of the anonymized image stack.

6.8 Apply Transform

Every data object in amira can carry a transformation as described in Transformations. But only the visualization of a field is changed not the representation in memory. That means if you scale a field with the Transform Editor dimensions and voxelsize will be retained though the size seems to be different on the screen. If you rotate a field the directions of the sampling planes will be still parallel to the local axis but no longer parallel to the global axis. The module *ApplyTransform* tries to create a new field in a way that it is displayed like the source field attached to the *Data* connection but without transformation.

You can e.g., align one field to another and apply the transformation. Afterwards you can directly work with the two fields without having to take into account any transformation.

There are two different ways to use the module:

- You can sample onto a given reference lattice.
- You can give a plane, e.g., an ObliqueSlice. The result is sampled in planes parallel to the given plane.

Applying a transformation

Proceed as follows: Transform the field with the Transform Editor, the Registration module or any other way that creates a transformation. If you're satisfied with the transformation, attach an Apply-Transform module to the field. Now you have to choose some options.

The *Interpolation* method:

- *Nearest Neighbor* chooses for every new voxel the value of the voxel in the source field nearest to it.
- *Standard* linearily interpolates between the sourounding voxels.
- *Lanczos* is the slowest but most accurate method that tries to approximate a low pass filter that is in accordance with the sampling theorem. If you have time and want to get the best result, use this one.

The *Mode*:

- *cropped*: The new field has the same dimensions and size as the source field. It is adjusted to contain as much as possible of the source field.
- *extended*: The new field's size is adjusted to contain all of the original field. To do this, the voxelsize or the dimensions have to be changed. If you select this mode the Preserve port gets visible.

The *Preserve* port:

- *Voxel Size*: Adjust the result field's dimensions in a way that it has the same voxelsize as the source field. **Warning**: This may lead to huge dimensions.
- *Dimensions*: The result field will have the same dimensions as the source field.

After choosing the options hit *DoIt* and proceed.

Sampling onto a reference lattice

Connect the source field to *Data*. Connect the reference field to *Reference*. Choose an *Interpolation* method. Hit *DoIt* and here we go.

Sampling parallel to a given plane

Connect the source field to *Data*. Connect a plane (e.g., ObliqueSlice) to *Reference*. Choose an *Interpolation* method. Choose whether the result has the same dimensions (*cropped*) or the result is *extended* to contain all of the source field. Hit *DoIt* and the result is created. The result will get a transformation attached to in a way that the sampling planes are displayed parallel to the reference plane. Use the Transform Editor if you want to get rid of the transformation.

Connections

Data `[required]`

The source field. The module operates on any regular 3D field with uniform coordinates and an arbitrary number of data components per node.

Reference `[optional]`

A regular field with uniform coordinates providing a reference lattice or a plane defining a sampling direction.

Ports

Interpolation

Interpolation: Standard

The interpolation method that will be used.

Mode

Mode: [✓ cropped] extended

The result can be *cropped* to have the same properties as the source or it will be *extended* to contain all of the source field.

Preserve

Preserve: [✓ Voxel Size] Dimensions

If the result is extended should it keep *Voxel Size* or the *Dimensions* of the source.

Action

Proceed.

6.9 ArbitraryCut

This module defines a plane which can be positioned and oriented arbitrarily inside the bounding box of a data object connected to port *Data*. Mostly, you will work with derived modules displaying some kind of geometry inside the plane defined by this base class. Examples of derived modules are ObliqueSlice or PlanarLIC.

However, this base class also serves as a background plane on which so-called *overlay modules* display their geometry. Examples of overlay modules are the Isolines module, the Vectors module, or the Intersect module. In fact, if you select an overlay module from a data object's popup menu an instance of *ArbitaryCut* called *EmptyPlane* will be created automatically.

You may also create an instance of *ArbitaryCut* by selecting *Clipping Plane* from the global *Edit Create* menu. The module does not display useful geometric output by itself, but it can be used to clip the output of any other module in amira. In order to perform clipping you first have to connect port *Data* to the data object being investigated. Then you can simply click on the module's *clip button* located in the orange header block.

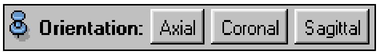
Connections

Data `[required]`

Connection to the data object from which the bounding box is taken.

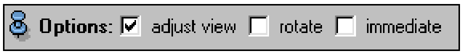
Ports

Orientation



By clicking one of the three push buttons the plane's orientation is reset to the xy-, xz-, or yz-plane, respectively. The plane will be translated into the center of the bounding box.

Options

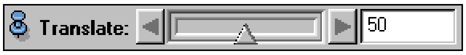


This port provides three toggle buttons. If toggle *adjust view* is active then the camera of the 3D viewer will be reset whenever one of the orientation buttons is clicked.

If the *rotate* toggle is active then a virtual trackball is displayed. By picking and dragging the trackball you may change the orientation of the plane. Remember that the viewer must be in interaction mode in order to do so. The ESC-key inside the viewer window toggles between navigation mode and interaction mode. The trackball of the last active ArbitraryCut can also be turned on and off by pressing the TAB-key inside the viewer window.

Finally, the *immediate* toggle determines whether derived modules and downstream modules receive an update signal while the plane is being translated or rotated or not. This toggle might not always be visible.

Translate



This port lets you translate the plane along its normal direction.

Commands

`frame {0|1}`
This command lets you turn on or off the orange frame indicating the intersection of the plane with the bounding box of the data object connected to the data port.

`setFrameColor <color>`
Lets you change the color of the plane's frame.

`setFrameWidth <width>`
Lets you change the width of the plane's frame in pixels.

`getPlane`
Returns 9 numbers specifying the current plane settings. The numbers comprise the x, y, and z-coordinates of the plane's origin, the u-vector, and the v-vector.

`setPlane <origin uVec vVec>`
Adjust position and orientation of the plane. The command expects 9 number in the same format as returned by the `getPlane` command.

6.10 Arithmetic

The *Arithmetic* module performs calculations on up to three input data objects according to a user-defined arithmetic expression. The result is stored in a new data object called *Result*. The calculations are triggered by a *DoIt* button. The arithmetic expression is evaluated either on the grid of the first data object (in case of regular, tetrahedral, or hexahedral grids) or on a regular 3D uniform grid for which the number of points can be set by the *Resolution* port.

The module is able to process input fields with up to 4 different channels. Scalar input fields are referenced by the variables A, B, and C. The values of multi-channel input fields are referenced for example by Ax, Ay, Az (if input A is a vector field) or Br, Bg, Bb, Ba (if input B is an RGBA color field).

In any case the expressions are evaluated per point, i.e., the result for a point (X,Y,Z) depends on input values at the same point only. If the resulting object is based on a regular grid, grid indices I, J, or K may also appear in the arithmetic expression. This means that for each grid point its associated I, J, or K index value will be substituted in the arithmetic expression on evaluation. This is useful in connection with comparison operators which produce a result of either zero or one. Computations may be confined to a specific sub-grid this way.

An expression consists of variables and mathematical and logical operators. The syntax is basically the same as for C expressions. The following variables are defined:

- `A:` The values of a scalar field at input A.
- `B:` The values of a scalar field at input B.
- `C:` The values of a scalar field at input C.

- `Ar:` The real part of a complex scalar field at input A.
- `Ai:` The imaginary part of a complex scalar field at input A.
- `Ax:` The x component of a vector field at input A.
- `Ay:` The y component of a vector field at input A.
- `Az:` The z component of a vector field at input A.
- `Ar:` The red component of a color field at input A.
- `Ag:` The green component of a color field at input A.

- `Ab:` The blue component of a color field at input A.
- `Aa:` The alpha component of a color field at input A.

- The same variables as above for fields at input B or C, but with `A` replaced by `B` or `C`.

- `I:` First index of a point (i,j,k) in a regular grid, or index of a point in an unstructured grid.
- `J:` Second index of a point (i,j,k) in a regular grid, undefined for unstructured grids.
- `K:` Third index of a point (i,j,k) in a regular grid, undefined for unstructured grids.

- `X:` The x-coordinate of the current point.
- `Y:` The y-coordinate of the current point.
- `Z:` The z-coordinate of the current point.
- `R:` The radius $r = \sqrt{X^2 + Y^2 + Z^2}$.

This is a list of available mathematical and logical operators:

- `+ - / *` The basic mathematical operators.
- `!` Unary negation.
- `-` Unary minus.
- `%` The modulo operator.
- `> < <= >= != ==` The comparison operators *greater, less, less or equal, greater or equal, not equal*, and *equal*. If the comparison is true the result is 1, otherwise it is 0.
- `&& || ^` The logical operators *and, or,* and *xor*. An non-zero operand is interpreted as true, while a zero operand is false. The result is either 1 (true) or 0 (false).

There are also some builtin functions:

- `pow(x,a)` Power function. Note that there is no power operator (the ˆoperator exists but means logical *xor*).
- `sin(x) cos(x) tan(x)` Trigonometric functions.
- `asin(x) acos(x) atan(x)` Inverse trigonometric functions.
- `sqrt(x)` Square root.
- `ln(x) exp(x)` Logarithm and exponent.
- `rand()` Pseudo-random variable uniformly distributed between 0 and 1.
- `gauss()` Pseudo-random variable with Gaussian distribution (mean value 0, standard deviation 1).

For better understanding some examples of how to use the *Arithmetic* module follow:

Expression: `A`
The result is a copy of input object A. If A is defined on a tetrahedral or hexahedral grid and the result

type is set to *regular* together with an appropriate resolution, the expression leads to a conversion from an unstructured grid to a structured regular grid. The same trick can also be used to resample a regular field with stacked coordinates onto a uniform grid.

Expression: `A-Bl`
The result is the difference between input objects A and B. This expression is sometimes useful in order to compare to different data sets.

Expression: `255*(A>127)`
Simple thresholding: For every value of A the result is set to 255 if the value is greater than 127. Otherwise the result is 0.

Expression: `A*(B>0)`
Simple masking operation: If B is zero, the result is also set to zero. Otherwise, the result is set to A. For example, if A is a 3D image and B is a corresponding label field, the exterior parts of the object (where B is zero) are masked out by this expression.

Connections

InputA `[required]`

The input may either be a 3D data field with an arbitrary number of channels or a tetrahedral or hexahedral grid. The primitive data type of the result will be the same as this input, e.g., if input A contains bytes the result will also contain bytes.

InputB `[optional]`

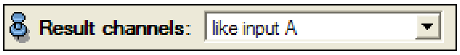
Second input, can be any 3D data field.

InputC `[optional]`

Third input, can be any 3D data field.

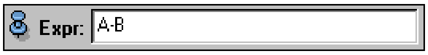
Ports

Result channels


Result channels: like input A

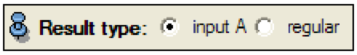
This port determines the number of channels of the result. On default, the result has the same number of channels as input A. Alternatively, you can specify that the result should have 2 channels (complex scalar field), 3 channels (vector field), or 4 channels (RGBA color field).

Expr


Expr: A-B

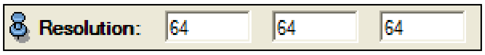
This port specifies the mathematical expression used to compute the result. If the result has more than one channel, more expression ports will be shown.

Result type


Result type: [✓ input A] regular

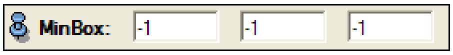
With this radio box the grid type of the result can be set either to either the same as input A or to a regular grid with uniform coordinates.

Resolution


Resolution: 64 64 64

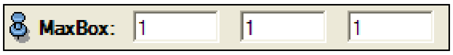
If port *Result Type* is set to *regular* the resolution of the regular field to be generated is set to the values given here.

MinBox


MinBox: -1 -1 -1

Port to set the minimum x-, y-, z-coordinates of a bounding box around the output data object, if one or more input data objects are connected the bounding box of the first input data object is also taken as output bounding box.

MaxBox


MaxBox: 1 1 1

Port to set maximum x-, y-, z-coordinates of the bounding box around the output data object.

Action

Pushing this button triggers the computation.

6.11 Axis

The *Axis* module displays a coordinate frame. If the module is attached to an input data object, the coordinate frame is adapted to the bounding box of the input object. Otherwise, global axes centered at the origin of the world coordinate system are displayed. For convenience the *View* menu of the main window contains a toggle button labelled *GlobalAxis*. Activating this toggle automatically creates an *Axis* module in the object pool.

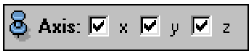
Connections

Data `[optional]`

Data object the axes should be adapted to.

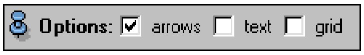
Ports

Axis



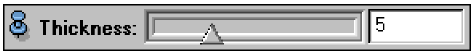
This port lets you select for which direction axes should be drawn.

Options



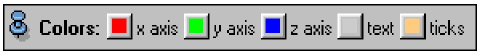
This port provides three toggles: If *arrows* is set arrows are drawn at the top of each axis. If *text* is set labeled ticks are drawn indicating the coordinate values. If *grid* is set a coordinate raster is drawn between every pair of visible axes.

Thickness



Lets you adjust the thickness of the axes.

Colors



Provides buttons for changing the colors of different parts of the geometry. On default, the x-, y-, and z-axes are drawn in red, green, and blue, respectively.

Commands

`getFontName`
Returns the name of the text font. Default is *Helvetica*.

`setFontName <name>`
Lets you change the text font.

`getFontSize`
Returns the size of the text font. Default is 10 points.

`setFontSize <size>`
Lets you change the size of the text font.

`getBoundingBox`
Returns the bounding box of the volume enclosed by the axes.

`setBoundingBox <xmin> <xmax> <ymin> <ymax> <zmin> <zmax>`
Lets you change the bounding box of the axis module. Once the bounding box has been set automatically, the box will not be updated automatically anymore, unless the `setBoundingBox` command is issued again with no arguments.

6.12 BoundaryConditions

The *Boundary Conditions* module visualizes boundary conditions defined in a tetrahedral grid for a *finite elements* simulation.

Visible faces are stored in an internal buffer similar to the *GridVolume* module. Likewise, the selection domain can be restricted interactively by adjusting a selection box. Ctrl-clicking on a face makes it invisible.

In Amira there are predefined boundary condition ids with special meanings. The meaning of an id depends on the type of simulation to be performed on the grid (E-field or temperature simulation).

- predefined boundary condition ids for E-field simulation
  id 1: outer boundary ('open' boundary conditions)
  id 11-59: antenna triangles (metallic boundary conditions)
  id 61-99: triangles of antenna junctions
- predefined boundary condition ids for temperature simulation
  id 1: predefined temperature 37°C (Dirichlet condition)
  id 2: insulating surface (Neumann condition)
  id 3: boundary to water bolus (Cauchy condition)
  id 5: boundary to air (Cauchy condition)

Connections

Data `[required]`

The tetrahedral grid to be visualized.

Ports

Bound.cond.id

This port provides a menu where all boundary condition ids occuring in the input grid are listed. Faces belonging to the selected id are displayed using a red (highlighted) wireframe in the viewer. You may crop the faces by turning the viewer into interactive mode and move the green handles of the selection box. Only the faces inside the bounding box can be added to the buffer.

Buffer

The Buffer buttons give you some control of the internal face buffer of the BoundaryConditions module. Only the faces present in the buffer are displayed according to the current drawing style. The *Add* button adds the highlighted faces to the buffer. The *Remove* button removes highlighted faces from the buffer. The *Clear* button removes all faces from the buffer. The *Hide* button deselects all faces but does not change the buffer.

Draw Style

This option menu lets you select between different drawing styles:

- **Outlined:** The faces are shown together with their edges.
- **Shaded:** The faces are shown.
- **Lines:** The edges of the faces are rendered as a wireframe.
- **Flat:** The edges of the faces are rendered as a wireframe without lighting.
- **Points:** Only the vertices of the faces are rendered as points.

6.13 BoundingBox

The *Bounding Box* module can be connected to any spatial data object in order to visualize its bounding box. The bounding box of a data object encloses its geometrical domain. More precisely, it is obtained by determining the minimum and maximum coordinates in x-, y-, and z-direction.

Connections

Data `[required]`

Any spatial data object.

Ports

Text

Shows or hides the coordinates of the lower left front and upper right back corner of the box.

6.14 CastField

This is a computational module that allows you to change the primitive data type of a regular 3D scalar field or of an RGBA color field. A new scalar field will be created having the same dimensions and the same coordinates as the incoming one, but the data values will be shifted, scaled, and casted to a new data type.

As an additional feature, *CastField* is also able to convert a uniform scalar field of bytes into a label field, which is commonly used to store segmentation results in amira. For each value or label occuring in the incoming field a corresponding material will be created. Material colors may be defined via an optional colormap.

Connections

Data `[required]`

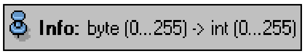
The scalar field or color field to be converted.

Colormap `[optional]`

Optional colormap specifying material colors if a uniform scalar field of bytes is to be converted into a label field. Only visible, if *LabelField* has been selected for output.

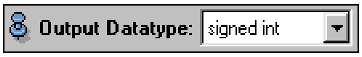
Ports

Info


Info: byte (0...255) -> int (0...255)

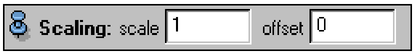
Shows how the input data will be mapped during conversion. If no clipping occurs the input range covers all values between the minimum and maximum data value of the incoming scalar field. This range will be mapped as indicated. If clipping occurs, the minimum or maximum value of the output range or both will be equal to the smallest respectively biggest value which can be represented by the selected output data type. In this case the input range shows which values correspond to these limits. Data values below the lower limit or above the upper limit will be clamped.

Output Datatype


Output Datatype: signed int

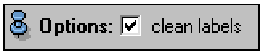
Lets you select the primitive data type of the output field. The item *LabelField* is special. If this is selected the incoming scalar field is converted into a label field.

Scaling


Scaling: scale 1 offset 0

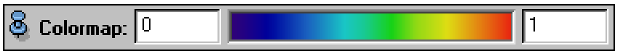
Defines a linear transformation which is applied before the data values are clamped and casted to the output datatype. The transformation is performed as follows: `output = SCALE*(input+OFFSET)`
If you want to convert data with a range `inmin ... inmax` into a range `outmin ... outmax`, `SCALE` and `OFFSET` are determined like this: `SCALE = (outmax - outmin) / (inmax - inmin)` and `OFFSET = outmin / SCALE - inmin`

Options


Options: ☑ clean labels

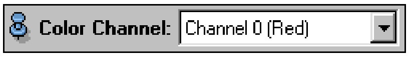
This port is only shown if you want to convert the incoming scalar field into a label field. If option *clean labels* is set then the materials found in the input data set will be relabeled so that the first material is 0, the second is 1, and so on. If *clean labels* is not set then the resulting label field will contain exactly 256 materials and no check is performed if a material actually can be found.

Colormap



Port to select a colormap.

Color Channel


Color Channel: Channel 0 (Red)

This option menu will only be shown if an RGBA color field is to be converted. It allows you to specify which channel of the color field should be regarded. In addition to the four RGBA channels also *Gray* and *Alpha*Gray* can be selected. The gray channel is computed on-the-fly from the RGB values of the color field according to the NTSC formula, i.e. `I=.3*R+.59*G+.11*B`.

Action

Start data conversion.

6.15 ChannelWorks

This module allows to convert between scalar and vector fields. You can e.g., combine three scalar fields into one vector field, or extract one of the six channels of a complex valued vector field to obtain a scalar field.

Connections

Input1 `[required]`

Connect to some regular field (e.g., a uniform vector field, or a uniform scalar field or 3D image volume).

Input2 `[optional]`

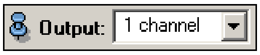
Optional second input, which accepts the same data types as the first input.

Input3 `[optional]`

Optional third input, which accepts the same data types as the first input.

Ports

Output



Select the desired output type: scalar field (1 channel), complex valued scalar field (2 channels), vector field (3 channels), color field (4 channels) or complex vector field (6 channels).

Channel 1

Select which of the input channels is used as first output channel.

Channel 2

Select which of the input channels is used as second output channel. This port is only present if the output has more than one channel.

Channel 3

Select which of the input channels is used as third output channel. This port is only present if the output has more than two channels.

Channel 4

Select which of the input channels is used as fourth output channel. This port is only present if the output has more than three channels.

Channel 5

Select which of the input channels is used as fifth output channel. This port is only present if the output has six channels.

Channel 6

Select which of the input channels is used as sixth output channel. This port is only present if the output has six channels.

Action

Start computation.

6.16 ClippingPlane

The clipping plane module is an instance of the Arbitrary Cut module (for a description see there). It can be created by selecting *Clipping Plane* from the main window's *Edit Create* menu.

6.17 ClusterDiff

This module displays displacement vectors between corresponding points in two different data objects of type Cluster. The displacement vectors may be colored according to their lenghts or according to any data variable defined in one of the input clusters. Like in the ClusterView module subsets of points may be selected by drawing a contour in the viewer window or by specifying an arithmetic filter expression.

Connections

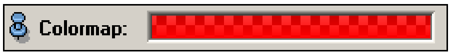
Data
The first point cluster.

Data2
The second point cluster. Initially, this port will not be connected to any data set. In order to use the module you have to estabilish a connection manually by activating the popup menu over the tiny rectangle of the module's icon.

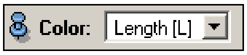
Colormap
The colormap used for pseudo-coloring. The alpha channel of the colormap will be correctly taken into account unless the option *opaque* has been selected. However, note that many default colormaps are completly opaque. In this case, use the colormap editor to make them transparent.

Ports

Colormap

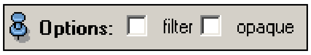

Colormap:

Color


Color: Length [L]

An option menu containing all data variables which can be used for pseudo-coloring. The data variables' symbols are displayed in brackets behind the names.

Options



The following two options can be set:

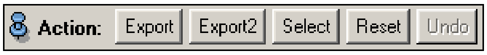
Opaque influences the way how the displacement vectors are drawn. On default, anti-aliased semi-transparent lines are rendered. If the opaque option is set transparencies will be ignored. Opaque rendering is faster but gives less nice results. In addition, you can't supress certain lines my making them more transparent.

Filter causes a text field to be shown, which can be used to define an arithmetic expression for selecting a subset of points.

Filter

This port allows you define an arithmetic filter expression. The filter expression is evaluated for every pair of points. A displacement vector will only be drawn if for both points a non-zero result is obtained. The filter expression may contain any arithemic and logical operator defined in the C programming language. All symbols listed in the color menu may be used in the filter expression. For example, in order select all vectors shorter than 0.2 and with an energy larger than -4 use $(L < 0.2)$ && $(E > -4)$.

Action



The button labeled *Export A* creates a new point cluster containing all selected points of the first input cluster together with the corresponding data variables. Similarly, the button labeled *Export B* creates a new point cluster containing all selected points of the second input cluster.

After pressing the *Select* button you may draw a contour in the viewer window in order to deselect certain points. On default, all points inside the contour are deselected. If you keep the Ctrl-key pressed down when starting to draw the contour all points outside the contour are deselected. Using the Alt-key allows you to define straight-line segments.

The *Reset* button causes all points matching the filter expression to be shown again.

The *Undo* button undoes the effect of the last interactive contour selection operation.

Commands

```
setLineWidth <width>
```

Specifies the width of the displacement vectors. On default, the vectors are one pixel wide.

```
setLineSmooth {0|1}
```
Enables or disables line smoothing. On default, smoothing is on unless the *opaque* option has been selected.

6.18 ClusterGrep

This module identifies common points in two different data objects of type HxCluster and copies them into new cluster objects. Using this module you may for example extract the same set of points from multiple clusters representing different time steps in a dynamic simulation.

Connections

A
The first point cluster.

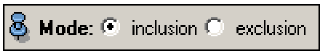
B
The second point cluster. Initially, this port will not be connected to any data set. In order to use the module you have to estabilish a connection manually by activating the popup menu over the tiny rectangle of the module's icon.

Ports

Points

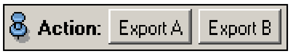
This port displays the number of points in the two input clusters. If both input ports are connected also the number of common points is displayed.

Mode


Mode: [✓ inclusion] exclusion

This port provides two radiobox buttons allowing you to specify which points are copied into the output clusters. If *inclusion* is selected only common points are copied. If *exclusion* is selected only points not present in the other cluster are copied.

Action



Button *Export A* copies points from the first input cluster together with associated data values into a new cluster. Button *Export B* copies points from the second input cluster together with associated data values into a new cluster.

6.19 ClusterSample

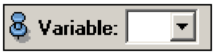
This module converts a set of points with associated data values into a uniform scalar field. The conversion is performed by determining for each node of the uniform grid the nearest point of the point set. The data value of that point will be taken at the grid node. The input data object must be of type Cluster.

Connections

Data
Point cluster to be processed.

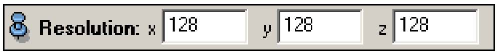
Ports

Variable



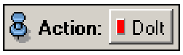
Option menu determining which variable should be converted into an uniform scalar field.

Resolution



This port provides three text field specifying the resolution of the resulting scalar field in x-, y-, and z-direction. The default resolution is chosen so that a sufficiently high sampling rate is obtained provided the input points are distributed homogeneously.

Action



Starts the computation.

6.20 ClusterView

This module visualizes data objects of type Cluster. The vertices of the cluster may be rendered using semi-transparent points, or using textured plates. Both, points and plates may be colored according to any data variable defined in the cluster. Vertices of the cluster may be selected or deselected by drawing a contour in the viewer window or by specifying an arithmetic filter expression. Finally, it is possible to display bonds between neighbouring points. This is useful for example if the points represent atoms in a crystal lattice.

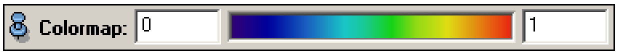
In plates mode (see below) individual points may be deselected by shift-clicking them. Clicking them without the shift-key pressed causes their id to be printed in the console window.

Connections

Data

The point cluster to be visualized.

Colormap



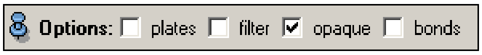
The colormap used for pseudo-coloring. The alpha channel of the colormap will be correctly taken into account unless the option *opaque* has been selected. However, note that many default colormaps are completly opaque. In this case, use the colormap editor to make them transparent.

Ports

Color

An option menu containing all data variables which can be used for pseudo-coloring. The data variables' symbols are displayed in brackets behind the names.

Options



Options: ☐ plates ☐ filter ☑ opaque ☐ bonds

The following four options can be set:

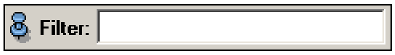
Plates activates the textured plates mode. If this mode is active each selected vertex is represented by a little sphere. For large numbers of points a conisderably amount of time may be required each time new colors are set. On default, the plates mode is deactivated. Instead, simple points are rendered. The point size is constant irresectively of the distance of a point from the camera.

Filter causes a text field to be shown, which can be used to define an arithmetic expression for selecting a subset of points.

Opaque influences the way how point primitives are drawn. On default, anti-aliased semi-transparent points are rendered. If the opaque option is set transparencies will be ignored. Opaque rendering is faster but gives less nice results. In addition, you can't supress certain points my making them more transparent.

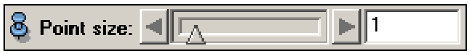
Bonds enables the display of bonds between neighbouring points. If necessary, connectivity information is computed automatically. This may take some time, especially for big data sets.

Filter



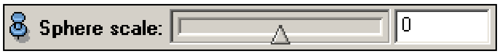
This port allows you define an arithmetic filter expression. The filter expression is evaluated for every point. Only points for which a non-zero result is obtained are drawn. The filter expression may contain any arithemic and logical operator defined in the C programming language. All symbols listed in the color menu may be used in the filter expression. In addition, the symbols *x*, *y*, and *z* are defined. These symbol indicate the coordinates of a 3D point. For example, to select all points with positive x-coordinates and with an energy

Point size



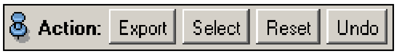
Specifies the point size in pixels. Only visible if point mode is activated, i.e., if no spheres are shown.

Sphere scale



Allows you to adjust the size of spheres shown if option *plates* has been selected. The default sphere radius is computed from the bounding box of the data set.

Action



The *Export* button creates a new point cluster containing all selected points of the input cluster together with the corresponding data variables.

After pressing the *Select* button you may draw a contour in the viewer window in order to deselect certain points. On default, all points inside the contour are deselected. If you keep the Ctrl-key pressed down when starting to draw the contour all points outside the contour are deselected. Using the Alt-key allows you to define straight-line segments. In order to be visible a point must pass the filter test (see above) and the contour selection test. Both options are independent of each other, i.e., if you change the filter the current contour selection remains valid.

The *Reset* button resets contour selection mode. All points passing the filter test become visible.

The *Undo* button undoes the effect of the last interactive contour selection operation.

Commands

`setBondColor <color>`

Specifies the color used for drawing bonds between neighbouring points. The color may be specified by either an RGB triple in range 0...1 or by a common X11 color name, e.g., *red* or *blue*.

`setBondWidth <width>`

This command allows you to change the width of the lines representing the bonds between neighbouring points. On default, lines are drawn one pixel wide.

6.21 Color Combine

This module combines up to three source fields into an RGBA color field. The resulting field has the same dimensions as the field connected to *source1*. Therefore it is imperative that there is a *source1*. *Color Combine* connects to color fields, scalar fields and scalar fields with a colormap. The relative scale of the input fields is determined by their bounding boxes. The module supports three different ways of merging the input data. ”Average” is the simple geometrical average of the inputs. ”Add & Clamp” adds the values, making sure they do not exceed 255. ”Alpha Weighted” weights the different inputs according to their alpha values, i.e. inputs with a large alpha value become more prominent in the resulting field. If all connected source fields are scalar byte fields of the same dimensions the module offers two more options: ”colormaps”, which does exactly the same as the procedures above, yet implemented in a much more efficient way, and ”RGB planes”, which interprets source1, source2 and source3 as the R, G and B values of the resultfield, respectively.

Connections

Source1 `[required]`
A scalar or color field. This one determines the size of the result field.

Source2 `[optional]`
Another scalar or color field.

Source3 `[optional]`
And yet another one.

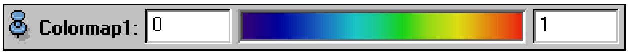
Colormap1 `[optional]`
The colormap for source field 1.

Colormap2 `[optional]`
The colormap for source field 2.

Colormap3 `[optional]`
The colormap for source field 3.

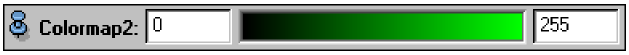
Ports

Colormap1


Colormap1: 0 1

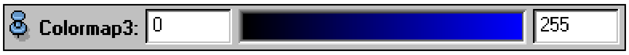
Colormap input port for source field1.

Colormap2


Colormap2: 0 255

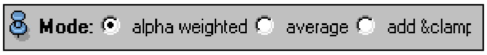
Colormap input port for source field2.

Colormap3



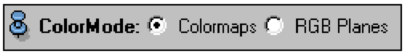
Colormap input port for source field3.

Mode


Mode: [✓ alpha weighted] average add &clamp

This option menu lets you choose the combining algorithm. The choices are: geometrical average, simple addition and a addition weighted in relation to the corresponding alpha values. This port is only visible, if Colormode is set to ”colormaps”.

ColorMode


ColorMode: [✓ Colormaps] RGB Planes

This port is only visible if all source fields are of equal dimensions and if they are all byte fields. It means that faster algorithms are now in use and offers the special choice ”RGB planes”, which makes source1, source2 and source3 the R, G, B compoment of the result, respectively.

Action

Pushing this button triggers the field computation.

6.22 Colorwash

The *Colorwash* module helps you to visualize two scalar fields in combination, e.g., medical CT data and a dose distribution. The module is attached to an OrthoSlice module visualizing the first field, e.g., medical CT data. The image of the *OrthoSlice* is modulated so that it also encodes the second field, e.g., a dose distribution. The standard modulation technique is to multiply color into an underlying grey scale image. This explains the name of the module.

In order to use the module first select the scalar field to be colorwashed, e.g., a dose distribution. Then choose *Colorwash* from the pop up menu of an existing *OrthoSlice* modules. The new *Colorwash* module automatically connects to the selected scalar field. Alternatively, of course you can also connect the *Data* port of the module by hand.

Connections

Data `[required]`

The 3D scalar field to be colorwashed.

Module `[required]`

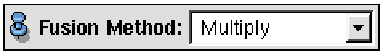
Connection to the underlying OrthoSlice module.

Colormap `[required]`

Connection to a Colormap.

Ports

Fusion Method


Fusion Method: Multiply

Specifies how the underlying *OrthoSlice* image is modulated. In any case the scalar field connected to the *Data* port is first mapped to a color image using the colormap connected to the *Colormap* port. This color image is then combined with the *OrthoSlice* background. The following modes are supported:

Multiply: The colors (scaled to the range 0...1) are multiplied. A colormap containing bright colors or white should be used, such as amira's *temperature.icol*. Click here for an example.

Add: The colors (scaled to the range 0...1) are added and clamped. Both the background image and the colormap should not be too bright in order to avoid overflows.

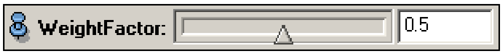
WeightedSum: The background image and the color image are blended using a weight factor which can be adjusted in a separate port (see below). If the weight factor is 0.5 the two images are just averaged.

MagicLens: A checkerboard pattern is generated, displaying the two images in the different squares. The square size can be adjusted in a separate port (see below).

Overlay: In this mode a data range can be specified. Inside this range the background image is replaced by the color image.

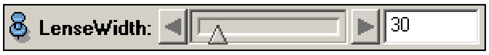
AlphaBlending: The background image and the color image are blended depending on the alpha values stored in the colormap. If the colormap contains no semi-transparent values at all, opacity is computed from the luminance of the color values.

WeightFactor


WeightFactor: 0.5

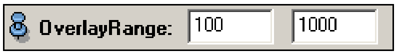
Interpolation factor used in *WeightedSum* mode.

LenseWidth


LenseWidth: 30

Width of the checkerboard tiles in *MagicLense* mode.

OverlayRange


OverlayRange: 100 1000

Range of data values where the overlay image replaces the background in *Overlay* mode.

Commands

`setNearestNeighbor`
Enables nearest neighbor interpolation for regular scalar fields. On default regular fields are interpolated trilinearly, except for label fields, which are evaluated using nearest neighbor interpolation in any case.

6.23 CombineLandmarks

This module merges an arbitrary number of vertex set objects into a single landmark set.

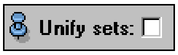
Connections

Data `[required]`
Accepts a vertex set object. As soon as this port gets connected, an additional source port will be created. This allows to merge a practically unlimited number of data objects.

Source2 `[optional]`
See above.

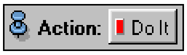
Ports

Unify sets



If this option is chosen, the resulting landmark set will contain only one set of landmarks, i.e., all vertices will be added to the same set. Otherwise, the vertices of the different input objects will be added to different sets. In order to ensure that all sets have the same number of markers the input object with the maximum number of vertices is determined and all other sets are filled with (0,0,0) if necessary.

Action



Starts computation.

6.24 CompareLatticeData

This module takes to regular fields with the same dimensions and the same number of data variables per voxel as input and computes the average difference per voxel between both. In addition, a new field with the point-wise difference of both inputs can be generated.

Connections

InputA `[required]`

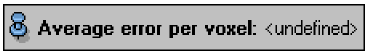
A 3D regular field. Any coordinate type, primitive data type, or number of data values per voxel are accepted.

InputB `[required]`

A 3D regular field with the same dimensions and the same number of data values per voxel as input A.

Ports

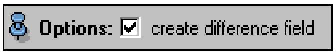
Average error per voxel



Average error per voxel: <undefined>

Displays the difference between the two input data sets.

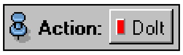
Options



Options: ☑ create difference field

If this toggle is set a new field with the point-wise difference between both input data sets is created.

Action



Action: DoIt

Starts computation.

6.25 ComponentField

This class is a special data object which helps you to analyze uniform complex scalar fields. The class itself is a uniform real scalar field. However, it can be attached to a complex scalar field like an ordinary display module. The component field provides an option menu allowing you to select which real quantity should be computed from the complex input values. These real quantities may be visualized using standard modules like OrthoSlice, Isosurface, or Voltex.

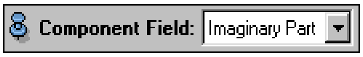
Connections

ComplexField `[required]`

Complex scalar field for which a real quantity should be computed.

Ports

Component Field


Component Field: Imaginary Part

Specifies the real quantity to be computed. Possible choices are *Real Part*, *Imaginary Part*, *Magnitude*, and *Phase*. The phase is defined in radians ranging from $-\pi$ to π.

6.26 ComputeContours

This module compute contours for a 3D label set using 2D cutting planes orthogonal to the selected axis (x,y or z). There is one cutting plane per each slice of the set; the plane has the same coordinate as the slice. If the labeled set contains subvoxel accuracy informations (weights) the contours can be computed using this information too (this is an option and by default is on). It is possible to compute only the contours that separate one certain material; this can be done by selecting a material from a menu. The default settings compute contours between all the materials.

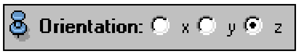
Connections

Data `[required]`

The label set to extract contours from.

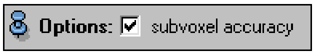
Ports

Orientation


Orientation: x y [✓ z]

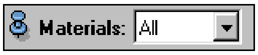
Controls the direction used for contours computing. The cutting planes will be orthogonal to this axis. For example, if z axis is selected, all the contours will be contained in xy planes.

Options


Options: [✓ subvoxel accuracy]

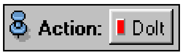
There is only one check button which controls the use of subvoxel accuracy information, if present.

Materials


Materials: All

Selects a material; only contours separating this material from others will be computed. By default, all the materials and implicitly all the contours are considered.

Action


Action: DoIt

Triggers the computation.

6.27 ConePlot

ConePlot is a display module that can be attached to a vector field. It visualizes scalar and complex vector fields by colored objects (cones) pointing in the direction of the local flow. The cones can be animated to generate a sequence of cones ”walking” through the vector field.

The module features SelectRoi, Colorfields, and arbitrary shapes to be displayed instead of cones.

The module can also generate a density based on the cone positions over time. This can be used with for volume rendering.

```
ConePlot setGhostDims 30 30 30
ConePlot generateGhost
```

The module uses a Gaussian kernel with a variance of a tenth of the maximum bounding box size of the volume. This options is computationally very expensive.

The module also exports the path of the cones as a LineSet. To generate the LineSet, call:

```
ConePlot saveAsLineSet
```

in the amira console window. The LineSet is computed with an data entry per LineSet point which represent the magnitude of the vector at this position.

ConePlot uses the OpenInventor render caching. All animation steps are computed beforehand and then played back by making the appropriate parts of the scene visible. This technique requires little CPU computations but larger amounts of memory and a fast graphics card. Notice that the first pass through the animation may be slow but subsequently render passes speed up.

Connections

Data `[required]`

The 3D vector field to be visualized. It can be either 3 or 6 floats per voxel. This module works therefore with arbitrary grid topologies by sampling them uniformly.

Colormap `[optional]`

Port to connect a colormap object. The color is computed from the magnitude of the vectors.

Animate `[optional]`

Attach a Time object to control the Animate slider. Please make sure that the range of the time slider is set to integers and that 0 is within that range.

ROI `[optional]`

Connect a region of interest to the module and the cones will only be generated inside this region. This helps in exploring complicated vector fields.

Colorfield `[optional]`

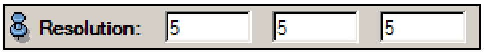
Instead of the vector field applied, the module will use a second scalar field to color the cones.

Shape data `[optional]`

Supply a custom shape to be replicated instead of cones. Basically any geometry displayed in Amira can be used. Be careful not to use very complex objects because the number of triangles used in this case can be a limiting factor for the speed of the animation.

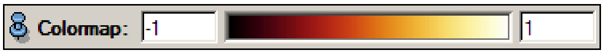
Ports

Resolution



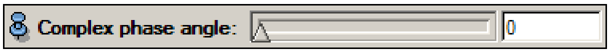
Provides three text inputs defining the resolution of the regular array of cones in the local x- y- and z-direction. The larger these values are, the more cones are displayed.

Colormap



Port to select a colormap. The color is computed from the magnitude of the vectors.

Complex phase angle

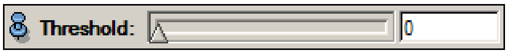


This port will only be visible if a complex-valued vector field is connected to the module. It provides a phase slider controlling which part of the complex 3D vectors is visualized by the arrows.

$$y(t) \quad = \quad \cos(\texttt{phase})\texttt{Re}(x(t)) + \sin(\texttt{phase})\texttt{Imag}(x(t)) \tag{6.1}$$

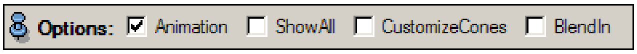
A value of 0 degree corresponds to the real part, while a value of 180 degrees corresponds to the imaginary part. The display can be animated with respect to the phase by the *cycle* button. This way polarization properties of the field can be revealed or wave phenomena become visible.

Threshold



Any cone is removed from display which would be placed on a position with a vector length smaller than this threshold. Using this option you can ”thin-out” parts of the volume, e.g., regions which would contain cones that never move.

Options



This port provides the following toggle options.

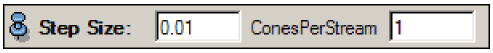
Animation: If this option is set, then two additional ports are displayed. The Time port allows you to set a time frame (from $-m > 0$ to n frames). The *Step size* port allows you to set the time between successive time steps. Currently we use an Euler integration. The smaller the step size, the more accurate the flow field is sampled.

Show All: Shows all cones in the current animation. When switching on this mode, there is no need for animations because the animation is done by switching cones on and off.

Customize Cones: You can change the height and the bottom radius of all cones.

BlendIn: If this option is set, the size of the cones is increased or decreased at the start or end point of the animation. This behavior generates nicer graphics for continuous animations especially in the case of ConesPerStream > 1.

Step Size


Step Size: 0.01 ConesPerStream 1

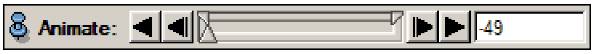
The current implementation uses Euler integration and this defines the step size calculated by

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t \cdot \mathbf{v}(\mathbf{x}(t), t),$$

where **x**(t) is the position and **v**(t) the velocity at time t. Please be careful to not work on stiff flow fields using this method. In general, small values result in higher accuracy and smaller path length of the cones.

If you set the value of *ConesPerStream* to > 1, more than one cone is displayed at a time. Together with the loop mode of the time slider and the *BlendIn* option set, this creates continuous animations.

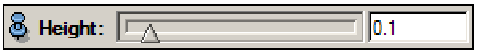
Animate


Animate: -49

This time slider allows you to start and stop animations. If you right-click in the associated text window you can switch to loop mode which generates an infinitly long animation.

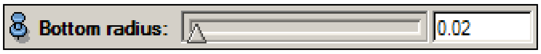
The range of the slider must be set to integer numbers, For example: $0 \ldots 50$ is interpreted as 50 time steps for the forward integration, or $-30 \ldots 30$ is interpreted as 30 steps in both forward and backward directions. The modules makes sure that the increment is always 1.

Height


Height: 0.1

For cones this changes their height. If another shape is connected to the module, the port controls the global scaling of the shape.

Bottom radius



For cones this changes the radius of their base circle.

DoIt

Recomputes the animation or the single plot.

6.28 ConnectedComponents

This module searches connected regions in a 3D image volume. The regions are detected based on thresholding, i.e., a region is a set of adjacent voxels with intensity values lying inside a user-defined range. The module is useful for counting cells on a dark background.

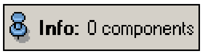
Connections

Data `[required]`

Image dataset to be analyzed. Only byte fields are supported.

Ports

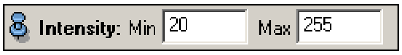
Info



Info: 0 components

After hitting *DoIt*, this port will display the number of detected regions, the volume of the smallest and the largest region, and the average volume.

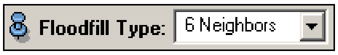
Intensity



Intensity: Min 20 Max 255

Voxels with values outside this intensity range are considered to be part of the background.

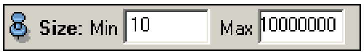
Floodfill Type



Floodfill Type: 6 Neighbors

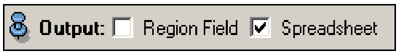
Defines the connectivity type. In case of *6 neighbors* voxels with a common face are considered to be connected. In case of *18 neighbors* voxels with at least one common edge are considered to be connected. Finally, in case of *26 neighbors* voxels with at least one common vertex are considered to be connected.

Size



Minimal and maximal size in voxels of the regions. Regions smaller or larger than this range are considered part of the background. If *Max* is zero no upper size limit will be implied.

Output



If *Region field* is checked, a scalar field of type byte will be created. Voxels which were considered to belong to a region are set to some non-zero value, while background voxels are set to zero. Different connected regions will be assigned different values, so that the regions can be visualized using color coding. However, since the output type is byte the region labels are not unique if more than 255 components are found. The regions can be visualized in 3D by using e.g., an Isosurface module with threshold *0.5*. Alternatively, the result can be casted to a label field using CastField and then the SurfaceGen module can be applied.

If *Spreadsheet* is checked, a spreadsheet object will be created, containing a list with the size and position of all detected regions.

Action

Start computation.

6.29 ContourView

This module displays contours of a surface. Contours are typically computed automatically and they are defined to be the boundaries of patches. If a surface contains no contours, this module will not display anything. You can automatically compute contours by using the `recompute` command of the Surface.

This module is mostly useful for developers.

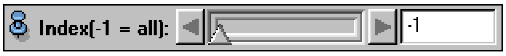
Connections

Data `[required]`

The underlying surface data.

Ports

Index)

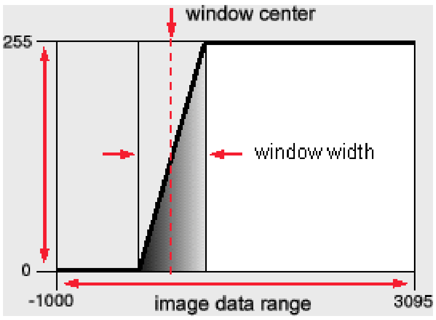


Choose index of contour to display. If -1 is choosen, all contours will be shown.

6.30 ContrastControl

The *ContrastControl* module (*Contrast* in short) is an extension to the OrthoSlice and ObliqueSlice modules. It is particularly useful for the grey or color value analysis of scalar fields like medical image data, as well as a preliminary stage for image segmentation. It allows a fast and intuitive adjustment of the transfer function, mapping scalar values stored with the image data to those values used for visualization.

With the help of *ContrastControl* the linear mapping of the data values to a subset of scalar values or indices to color values, defined by a Colormap, can be modified. The term *windowing* is often used within this context. The *window* defines a possibly narrowed view to the data's scalar values, for enhancing the contrast of certain structures within the image data. The width of the *window* defines the range of data values between a lower and an upper threshold which shall be mapped to a range of values specified by the height of the *window*. In terms of gray value mapping a subset of thousands of possible data values can be mapped to i.e. 256 gray values, linearly distributed on a ramp, varying from black (0) to white (255). Black values are represented by the lower border of the *window* and white values by the upper border. The center of the *window* might be located at any value of the image data, thus the entire window can be shifted to various points for data evaluation.



The *window*'s center and width can be modified with the two sliders in the working area of the *Contrast* module. These values can either be adjusted by shifting the sliders to the left or right, varying the values by a certain percentage of the image data range, or they can be specified numerically via the appropriate entry fields.

For an overview of the current mapping, a graphical representation of the mapping function can be displayed within the viewer window. In order to do so the toggle button in the working area with the *Window Show* label must be activated. The horizontal extent of the *window graph* specifies the image data range from the lowest to the highest value. The linear ramp indicates the mapping function where the area below or above the ramp represents the currently chosen mapping *window* .

Experienced users can directly use the mouse for a fast and simultaneous modification of the *window*'s center *and* width. The contrast can be adjusted via the left mouse button while the shift button is pressed. Moving the mouse to the left or right moves the center of the *window* accordingly. Up and down movements increase or decrease the width of the *window*. A vertical line (that means lower and upper threshold are identical respectively the *window* width equals zero) indicates a binary mapping of the image data values. All values below the *window*'s center will be mapped to black and all other values will be mapped to white. Moving the mouse further down will invert the mapping, thus exchanging black and white values.

Connections

Module `[required]`

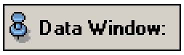
Connection to an OrthoSlice or ObliqueSlice module.

Data `[required]`

Connection to the scalar field is automatically established via the OrthoSlice module.

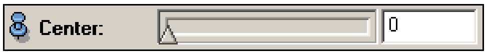
Ports

Data Window


Data Window:

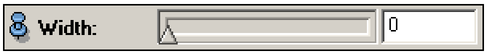
This port displays information on the image data range. In verbose mode the current data window will be displayed, too.

Center


Center: 0

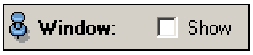
The window center can be positioned between the minimum and maximum value of the image data range. The increment value for slider movement is 1 percent of the image data range. More accurate values can be specified via the numeric input field.

Width


Width: 0

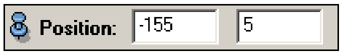
The window width can be adjusted between 0 and the total image data range. The increment value for slider movement is 1 percent of the image data range. More accurate values can be specified via the numeric input field.

Window


Window: Show

This port enables or disables the display of a graphical representation of the linear transfer function within the viewer. If it is enabled an additional port appears, which allows the specification of the graph's position. The graph will be updated synchronously with the contrast adjustment.

Position


Position: -155 5

This port is only visible when *Window Show* is enabled. The input values are the X- and Y position of the graph within the viewer. The lower left corner of the viewer has the coordinates (0, 0). Negative coordinates are relative to the right respectively upper border of the viewer. Both values can also be modified by moving the mouse pointer into the appropriate entry field, pressing the shift key and moving the mouse.

Commands

`verboseMode {0|1}`
Setting verboseMode to a value not equal to zero leads to a permanent refresh of the current data window settings via the mapping port. Due to the possibilities of changing the window settings quickly this leads to flickering results and furthermore reduces the interactive adjustment speed. A value of 0 is the default.

`showWindowGraph`
This is just a command line interface for the window port, bringing the window graph up front.

`hideWindowGraph`
This is the counterpart to showWindowGraph, removing the window graph.

`info`
Prints out a short info to the *ContrastControl* module.

`setLineColor <r> <g> <b>`
Using setLineColor you can change the visual appearance of the window graph. This is useful when the currently chosen background color interferes with the line color of the graph. The RGB triple specifies how much a certain color component contributes to the resulting color. These values are clipped to 0 and 1.

`setMouseSensitivity [<value>]`
Direct contrast adjustment with the mouse (Shift - Left Mouse Button) in interactive mode can be amplified or damped using this command. The default value is 0.5 times 1 percent of the total image data range. The value is clipped to 0.1 and 1

`setPosition <x> <y>`
The position of the window graph can be modified with setPosition. Negative values are relative to the right and upper border of the viewer. If no window graph appears with showWindowGraph try setPosition 0 0 which should set the display to the lower left corner.

`setScaleFactor <factor>`
The size of the window graph can be modified using setScaleFactor. This might be useful if the window graph is disturbing the visual output of the viewer.

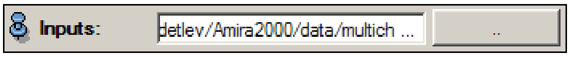
6.31 ConvertToLargeDiskData

This modules allows to convert Raw Data or stacks of image data as described in the TIFF format to LargeDiskData.

You can create a *ConvertToLargeDiskData* module with the *Create* menu. Use the submenu *Data*. You have to select the input files with the *Inputs* port, select the output with the *Output* port. The *CreateDiskData* button starts the conversion process. You are asked parameters as described in TIFF format or Raw Data during the conversion process. After a successful conversion a LargeDiskData data object appears in the object pool.

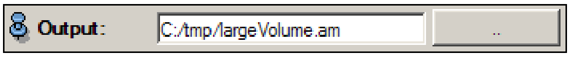
Ports

Inputs


Inputs: detlev/Amira2000/data/multich ...

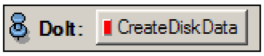
The list of input files.

Output


Output: C:/tmp/largeVolume.am

One output file.

DoIt



Starts conversion.

6.32 CorrectZDrop

This module lets you fix artifacts in 3D microscopic images caused by light absorption in other slices. If such artifacts are present, the average intensity in lower slices seems to be decreased. This so-called *z-drop* or *intensity attenuation* can be corrected automatically by fitting an exponential curve to the average intensities in each slices, or manually by providing a user-defined formula.

Connections

Data `[required]`

The image data exhibiting a z-drop artifact. Scalar field with uniform or stacked coordinates as well as multi-channel fields are supported.

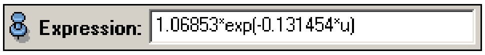
Ports

Mode

Mode: [✓ automatic] manual

Lets you select between automatic mode and manual mode.

Expression


Expression: 1.06853*exp(-0.131454*u)

This port is only available if manual mode has been selected. It provides a text field where you can enter a formula specifying a factor used to multiply the intensity values in each slice. Within the formula the variable *u* specifies the slices. *u* will take the value 0 for the first slice and 1 for the last slice. For the other slices it takes intermediate values depending on the actual slice location (this makes support of stacked coordinates easy). In automatice mode the following formula `a*exp(b*u)` will be used, where `a` and `b` are fitted automatically. If you first perform an automatic z-drop correction and then switch to manual mode, the fitted expontial will be displayed in the port's text field.

Action

Starts z-drop correction.

6.33 CorrelationPlot

This module computes a 2D correlation histogram of two uniform scalar fields (e.g. image stacks). It detects regions of correlated intensity in two images of the same object. The result of correlation analysis can be applied for an image segmentation because regions of high correlation are likely to represent a unique material or tissue type. The module can be connected to a *MultiChannelField* or to two separate images of the same dimensions. The correlation histogram is computed as follows:

- The data values of each input image are subdivided into a user defined number of intervals.
- Element `(i,j)` of the 2D histogram contains the number of all voxels where the data values fall into interval `i` for the first image and interval `j` for the second image.

Typically, *CorrelationPlot* is used for co-localization analysis in fluorescence microscopy, where the spatial distribution of two fluorochromes has been recorded in two different channels. There, the module finds the regions where the fluorescence signal is high in both images.

A further application of *CorrelationPlot* is the segmentation of multi-modal medical images such as CT and MRT. In this case, the images must be registered, i.e., the object position must be identical in both images. Use module Registration for achieving this alignment, followed by module Resample for resampling the model dataset to the lattice of the reference dataset. For the correlation analysis of medical image data from different modalities, *CorrelationPlot* lets you define arbitrary intensity ranges.

Besides the 2D histogram, the module computes basic statistics for the image pair such as the number of co-localized voxel pairs, the Peak-Signal-to-Noise-Ratio, and the correlation coefficient.

A graphical representation of the correlation histogram is shown in an extra 2D plot window using a user defined color map. The objective of a correlation analysis is to find and select regions of high correlation, which occur as local maxima in the 2D plot. Such regions can be selected either by manually drawing in the 2D plot window or by numerically entering subrange limits.

The selected regions can be used for a segmentation of the image datasets. For this, *CorrelationPlot* optionally generates a *LabelField*. The number of labels is given by the number of selected regions in the 2D plot. For each selected region, all voxels of the image datasets with corresponding data values are assigned to the respective label.

Connections

Source1 `[required]`

First input, must be a scalar field with regular coordinates or a *MultiChannelField* field.

Source2 `[optional]`

Second input, must be a scalar field with regular coordinates or a *MultiChannelField* field.

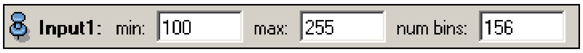
The module takes the first two fields that it can find in the input objects connected to the two source ports. If, for instance, a *MultiChannelField* field with three channels is connected to source1 and a scalar field to source2, the module takes the lattices from channel1 and channel2 of the *MultiChannelField* as input lattices, ignoring the third channel and the scalar field connected to source2.

Colormap `[optional]`

Optional colormap used to map the values of the correlation histogram. The range values of the colormap are ignored by this module.

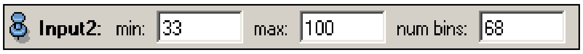
Ports

Input1


Input1: min: 100 max: 255 num bins: 156

The first two text fields define the data range of *Input1* used for computing the correlation histogram. Typically, the number of background voxels (low intensity or 0) that co-localize in two images overwhelms those of the structures of interest. Therefore, *min* and *max* can be set to exclude certain intensity ranges from the analysis. The text field labeled *num bins:* adjusts the number of bins and thus the coarseness of the histogram. In the case of input fields with byte or short as primitive data type the number of bins is internally adjusted such that each bin has the same integer width.

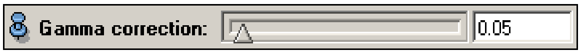
Input2



Input2: min: 33 max: 100 num bins: 68

Defines data range as well as the number of bins for the 2nd input. See description above for details.

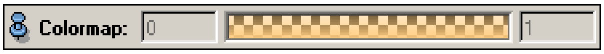
Gamma correction



Gamma correction: 0.05

This port defines the value of an exponent *gamma* used to specify the mapping of the histogram entries to color values. First, the histogram entries are normalized to their maximal value. Then, before the color is looked up, an element *x* of the correlation histogram is replaced by x^{gamma}. Choosing a gamma value less than 1 emphasizes small values. This is useful for similar reasons as described for port *Input1*.

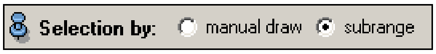
Colormap



Colormap: 0 1

Before plotting them, the histogram entries are normalized to values between 0 and 1. The colormap connected to this port is used for the histogram look-up. The range of the colormap will be ignored. If no colormap is connected, a grey map is used.

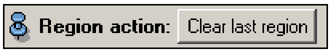
Selection by



Selection by: manual draw [✓ subrange]

Two methods are supported for selecting regions of the histogram. In *manual draw* mode one can use one of the draw tools from the tool bar at the upper border of the plot window. In the second mode labeled *subrange*, one can define a range of values for each of the two input fields. This selects a rectangular subregion of the histogram. In *subrange* mode, two additional numbers are computed as described below.

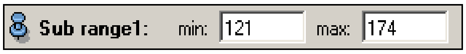
Region action



Region action: Clear last region

Press this button to remove the last selected region. This port is only shown in manual drawing mode.

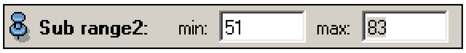
Sub range1



Sub range1: min: 121 max: 174

This port is only available in *subrange* mode. Here the boundaries of the subrange for Input1 are defined. To change the subrange with a visual feedback in the plot window, use the mouse wheel to single step or hold down the shift key while moving the left pressed mouse for making bigger steps.

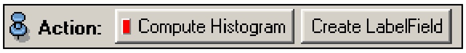
Sub range2



Sub range2: min: 51 max: 83

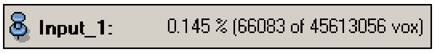
Like the above port for Input2.

Action



The *Compute Histogram* button recomputes the correlation histogram and pops up the plot window. The *Create LabelField* button computes a *LabelField* using the current selection.

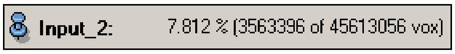
Input 1



Input_1: 0.145 % (66083 of 45613056 vox)

Prints relative (%) and absolute (n of n_tot) numbers of voxels specified in the *min* and *max* fields of port *Input1*.

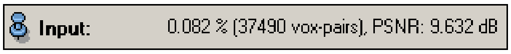
Input 2



Input_2: 7.812 % (3563396 of 45613056 vox)

Prints relative (%) and absolute (n of n_tot) numbers of voxels specified in the *min* and *max* fields of port *Input2*.

Input



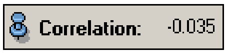
Input: 0.082 % (37490 vox-pairs), PSNR: 9.632 dB

Prints the fraction (relative to total number of voxel pairs) and absolute number of voxel pairs within specified interval. Only these pairs are considered for the histogram generation. The second value is the Peak-Signal-to-Noise-Ratio [dB], which is calculated according to

$$\mathrm{PSNR} = 10 * \log_{10}(\frac{1}{\mathrm{MSE}}), \tag{6.2}$$

with MSE being the (normalized) mean square error.

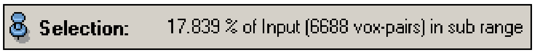
Correlation



Correlation: -0.035

Prints the correlation coefficient of the voxel pairs considered for histogram generation.

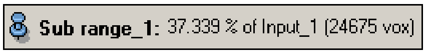
Selection



Selection: 17.839 % of Input (6688 vox-pairs) in sub range

Prints the relative (%, relative to Input) and absolute numbers of selected voxel pairs.

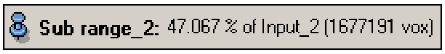
Sub range 1



Sub range_1: 37.339 % of Input_1 (24675 vox)

This port is available only in *subrange* mode. It prints the number of voxel pairs inside the set specified by the first subrange. The final selection is the intersection of the voxel pair sets defined by the first and the second subrange.

Sub range 2


Sub range_2: 47.067 % of Input_2 (1677191 vox)

This port is available only in *subrange* mode. It prints the number of voxel pairs inside the set specified by the second subrange.

6.34 Curl

The *Curl* module computes the curl of a vector field consisting of floats defined on a uniform grid. The output is another uniform vector field.

$$\mathrm{curl} V = \left(\frac{\partial V_z}{\partial y} - \frac{\partial V_y}{\partial z}, \frac{\partial V_x}{\partial z} - \frac{\partial V_z}{\partial x}, \frac{\partial V_y}{\partial x} - \frac{\partial V_x}{\partial y} \right)$$

Connections

Data `[required]`

Vector field defined on a uniform grid (*UniformVectorField3*). The vector components must be floats.

Ports

Compute

Pushing this button triggers the computation.

6.35 Cutting Plane

The *Cutting Plane* module can be connected to any display module derived from ViewBase. The module provides a special cutting plane of finite size which can be rotated, translated, and scaled interactively. The module then computes the intersections of all triangles shown by the *ViewBase* module with the cutting plane. The resulting line segments are stored in a LineSet object. Optionally, the module can be used without any input. In this case the intersection of any surface-like geometry shown in the main viewer is computed.

Note: Surfaces and tetrahedral grids can be more easily intersected with a plane using the Intersect module. This module also provides a Tcl command which can be used to export a *LineSet* object.

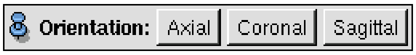
Connections

Data [optional]

A display module derived from ViewBase, e.g., Isosurface or SurfaceView. If no input is specified the entire scene will be cutted with the plane.

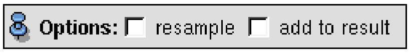
Ports

Orientation



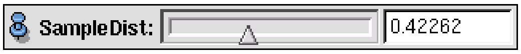
These three buttons reset the orientation of the cutting plane to the xy, xz, or yz plane, respectively.

Options



If *resample* is checked, the resulting lines are resampled so that line segments of equal length are obtained. The length of the resampled line segments can be adjusted using a separate slider (see below). If *add to result* is checked the resulting lines will be added to an existing result object. Otherwise, for each cutting operation a new result will be created.

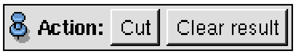
SampleDist



SampleDist: 0.42262

This port is only visible if the *resample* option is checked (see above). It specifies the preferred length of the resampled line segments.

Action



The *Cut* button actually computes the intersecting lines. The *Clear result* button removes all line segments from the current result.

6.36 CylinderSlice

This module displays the values of any scalar field on a cylinder. The cylinder surface is mapped on a planar slice and shown in an extra viewer. The cylinder is specified by the dragger position. The height is computed by the length of the diagonal of the bounding box of the scalar field. The module provides the same ports as ObliqueSlice. Additionally, it allows to save the cylindrical slice to an image. The image format is selected by the file name extension.

Connections

Data `[required]`

The scalar field.

Module `[required]`

The OrthoSlice or ObliqueSlice that specifies the dragger plane.

6.37 DataProbe

The three data probing modules *PointProbe, LineProbe, SplineProbe* are used to inspect scalar or vector data fields. They have many features in common so they are described in one section. The probes are taken at a point (PointProbe) or along a line (LineProbe, SplineProbe) which may be arbitrarily placed. The controlpoints of the data probing modules determine the locations where the samples are to be taken. PointProbe has one controlpoint which is the samplepoint, LineProbe has two controlpoints which are the endpoints of a line which is subdivided into a number of segments given by the Samples port or by the Distance port, and SplineProbe has at least three controlpoints that define a spline that goes through the endpoints and approximates the points in between. The spline curve is subdivided into a number of segments given by the Samples port or by the Distance port by which the sample points are obtained.

To place the controlpoints within the bounding box of the given geometry you can either type in the coordinates in the port Points (see below) or you can shift the points interactively with the mouse. The latter can be done by turning the 3D viewer into interactive mode and picking and moving the crosshair dragger of the current point. The plot immediately changes if the immediate mode toggle is set. Usually the sampled values are plotted against the length of the probe line or as a bar in case of *PointProbe* in an extra plot window. If you use more than one data probing module of the same type all plotted curves will be shown in one plot window.

PointProbe displays the value at the sample point and the material if material values are set. LineProbe and SplineProbe display the length of the line resp. spline.

A module of type ProbeToLineSet can be connected to a LineProbe or a SplineProbe module in order to save the probe line and the sampled values.

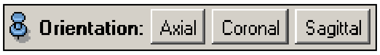
Connections

Data `[required]`

Can be connected to arbitrary 3D fields. The LineProbe module can also be attached to 3D input objects other than fields such as surfaces or Open Inventor geometry. In this case the LineProbe doesn't sample any data but simply measures distances.

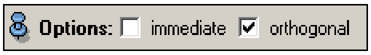
Ports

Orientation



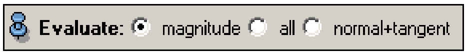
This port which is used in *LineProbe* and *SplineProbe* provides three buttons to specify the probe line orientation. Axial probe lines are perpendicular to the x-y-plane, frontal probe lines are perpendicular to the x-z-plane, and sagittal probe lines are perpendicular to the y-z-plane. If one of these buttons is pressed the start and end coordinates of the probe line are set to the minima resp. maxima of the corresponding axis and to the center of the two axes perpendicular to the probe line.

Options


Options: ☐ immediate ☑ orthogonal

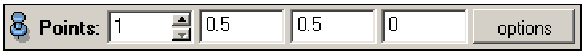
The *immediate* toggle determines whether the samples are taken while the controlpoints are being moved or when the motion is finished. If the *orthogonal* toggle is on all points are moved in sync, when the coordinates of one point are changed with the dragger. This port is not shown for module *PointProbe*.

Evaluate


Evaluate: [✓ magnitude] all normal+tangent

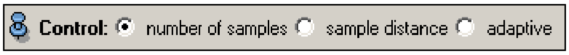
If the probe is connected to a vector field, these radiobuttons are shown. If the *magnitude* button is set the magnitude of the vectors is shown in the plot window. With the *normal+tangent Comp.* button set you get the normal and tangential components as two curves. Setting the *all* button shows all components of the vector field as separate curves.

Points


Points: 1 0.5 0.5 0

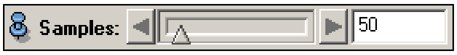
Here you can see resp. type in the coordinates of all controlpoints the data probing module makes use of. The options menu lets you toggle whether a dragger or a sphere is shown for the controlpoints. You can also append, insert or remove controlpoints if this module is a *SplineProbe* module.

Control


Control: [✓ number of samples] sample distance adaptive

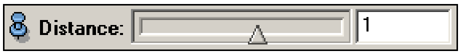
With this radio buttons you can choose which of the following two ports are taken to compute the sample points. This port is not shown for module *PointProbe*. If you have chosen the *adaptive* button, the module tries to keep the number of samples which can be seen in the plot window as close as possible to the number given with the appropriate slider. I.e. the more you zoom into the plot window the more exact are the curves in the plot window.

Samples


Samples: 50

This slider allows you to choose the numbers of samples along the probe line. This port is not shown for module *PointProbe*.

Distance



This slider allows you to choose the distance between two consecutive sample points along the probeline.

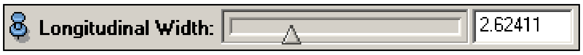
Options

The *average* option averages the probe values by taking samples on a disk perpendicular to the sampling points and smoothes the sampled values along the sampling line(s). This button is anly available for a LineProbe resp. SplineProbe module.

Radius

The radius of the sampling disk. This slider is only shown if the above average option is chosen.

Longitudinal Width



The width determines how many sampling values are used for smoothing. This slider is only shown if the above average option is chosen.

Plot

If the *Show* button is pressed a plot window appears where the sampled values are plotted against the length of the probe line. *Note:* There will be only one plot window regardless of how many Line Probe modules there are in your setup. Every line probe is represented in that plot window by a curve bearing the name of the corresponding module.

Commands

`getInterpol`

Returns the currently used interpolation method.

`setInterpol {none|linear|spline}`

Sets the interpolation method. `none` means no interpolation at all and the sample values are taken at the controlpoints only.

`getOrder`

Returns the order of the spline probe.

`setOrder <value>`
Sets the order of the spline probe.

`getPoint [<index>]`
Returns the coordinate of the requested controlpoint. `Index` defaults to 0.

`setPoint [<index>] <x> <y> <z>`
Sets the coordinates of controlpoint `index`. `Index` defaults to 0.

`getSamplePoints`
Returns the coordinates of all points where samples are taken.

`getSampledValues`
Returns all sampled values.

`setImmediate {0|1}`
Switches the immediate mode on or off, i.e. data is shown while the probe line resp. point is being moved.

`setOrtho {0|1}`
Switches the orthogonal mode on or off, i.e. all controlpoints of a probe line are moved in sync or not.

`getNumPoints`
Returns the number of control points of a probe line. In case of a point probe 1 is returned.

`getLength`
Returns the length of the probe line. 0 in case of a point probe.

`appendPoint <x> <y> <z>`
Appends a controlpoint with the given coordinates.

`insertPoint <index> <x> <y> <z>`
Inserts a controlpoint with the given coordinates as the `index`th point.

`removePoint <index>`
Removes the given controlpoint.

6.38 Delaunay2D

This module takes a set of 3D vertices and produces a Delaunay triangulated surface with 2D topology. In order to do so the vertices are projected either into the xy-, xz-, or yz-plane or alternatively onto a cylinder parallel to the x-, y-, or z-axis. The cylinder is automatically positioned so that it matches the center of the vertices.

The incoming data object must be derived from a vertex set, cf. Section 3.2.5. Note, that the Delaunay algorithm may take some time, especially for large data sets. If the input is not of planar topology, like e.g., a sphere, the result will probably not be useful. If multiple 3D vertices project to the same 2D vertex, the result is undefined. In the latter case, it might be help to issue a `jitterPoints 0.001 0.001 0.001` command on the command line.

If the input data set is of type Cluster and if this cluster has data associated to it than these data values are converted into one or more surface scalar fields.

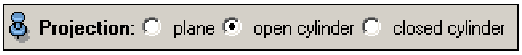
Connections

Data `[required]`

The vertex set to be triangulated.

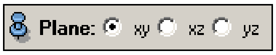
Ports

Projection


Projection: plane [✓ open cylinder] closed cylinder

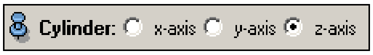
Lets you select whether the 3D points should be projected into a plane or onto a cylinder. If *closed cylinder* is selected than a closed cylindrical surface will be created.

Plane


Plane: [✓ xy] xz yz

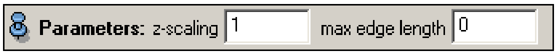
This port lets you select on which plane the 3D vertices should be projected on in case of a planar projection.

Cylinder


Cylinder: x-axis y-axis [✓ z-axis]

This port lets you select the orientation of the cylinder in case of a cylindrical projection.

Parameters


Parameters: z-scaling 1 max edge length 0

The first number defines an optional internal scaling applied to the 3D vertices. The direction in which the scaling is applied depends on the selected projection type. The option is useful if the 3D points are distributed non-isotropically, i.e., if the average feature size in different in the three coordinates. *Note, that an external scaling defined with the transformation editor is currently not taken into account by this module.*

The second number lets you specify a maximum edge length in 3D space. Delaunay triangles with edges longer than this limit are discarded. When computing the 3D edge length the scaling factor defined by the first parameter is taken into account. When a new input is connected and no other input was connected before the max edge length field is initialized with half of the average edge length of the bounding box of the data set. A maximum edge length of 0 indicates that no triangles should be discarded.

Action

Starts computation.

6.39 DemoMaker

Using the *DemoMaker* module, you can create an animated sequence of operations, e.g. for automatically running demonstrations or for advanced movie recording. Select *Create / Animation/Demos / DemoMaker* from the menu to add a DemoMaker module to the object pool.

Defining and storing a demo sequence

The demo sequence is a set of *actions* over a time range as respresented by the module's Time port. A wide range of such actions can be defined, e.g. switching a toggle value on or off, varying the numerical value of some other module's port over time, and defining breaks to interrupt the automatic demo sequence.

Actions are defined by first selecting an entry from the GUI elements port. This port lists all GUI elements of all modules that currently exist in the object pool, except the ports of the active *DemoMaker* module itself.

Once you have selected the right element (or a different action type like a pause, break, go-to, or an arbitrary TCL command) from the GUI elements menu, additional ports will show up below the *GUI elements* port, depending on the type of the selected element (e.g. numeric, toggle, button, ...). Using these ports, you can define the time (or time range) of the action on the *DemoMaker* Time line as well as the value to which the selected GUI element will be set at that time (see example below).

When action time and values have been defined, press the *Add* button in the Event List button line. This adds the desired action to the DemoMaker's list of events as represented by the *Event List* menu. Multiple sequential, overlapping, or parallel actions may be contained in one demo sequence.

Once a part of the desired action sequence is defined, simply store it by choosing *File / Save Network...* from the menu. *DemoMaker* in its current state will be saved along with the rest of the object pool. When saving the network, make sure the Time slider is in the desired starting position.

Playing a demo sequence

For playing the demo sequence as defined, simply press the Time slider's play button (triangle pointing to the right) or press the corresponding function key `F4`. The sequence will run from the current time step to the end of the demo, or to the next break as defined by the user (see Defining a break below). The play button will change into a stop button, and pressing that button or pressing the `F3` function key immediately stops playing. If the demo stops at a user-defined break, continue by again clicking the play button or pressing `F4`.

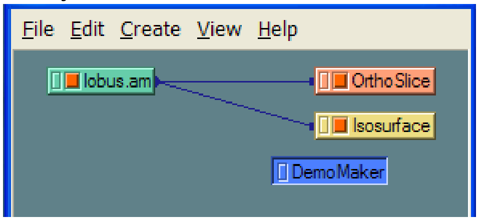
If the demo is stopped at the beginning or at some user-defined break, you can press the function key `F10` to jump to the point of the next break, or to the end of the demo. Similarly, pressing `F9` jumps to the previous break or to the very beginning of the action sequence. Once you have jumped to the right step of the demo, simply press play or `F4` again to play it from there.

If the action sequence is running too fast or too slow, you can adjust its speed using the Time port's *Configure* option (right-click onto the time port, see time port documentation). Note that the playback speed can only be adjusted for the complete time range at once. If you want only to change the speed of only part of the sequence, you must do so by adjusting the time range of the corresponding action entries.

Usage Example

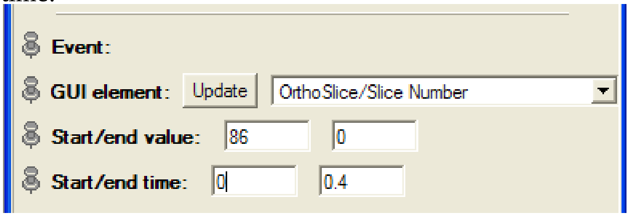
Here is a small example to demonstrate the way *DemoMaker* works. Load `lobus.am` from the `tutorials` subdirectory of the amira installation directory. Connect an *OrthoSlice* module and an *Isosurface* module. Create a *DemoMaker* by selecting it from the *Create / Animation/Demo* menu.

Now your network should look similar to this:



All available user interface ports of the modules in the object pool should now appear in the *GUI element* port of the *DemoMaker* module. If not, press the *Update* button.

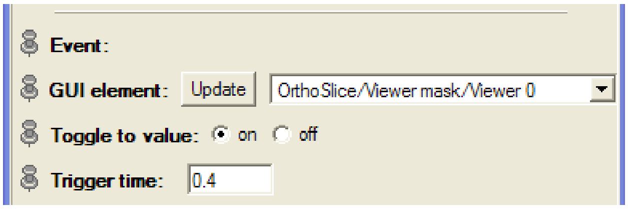
Now select *OrthoSlice/Slice Number* from the *GUI elements* menu. Fields for specifying start/end values and start/end time appear. Enter `86` and `0` as start and end values and `0` and `0.4` as start/end time:



Event:
GUI element: Update OrthoSlice/Slice Number
Start/end value: 86 0
Start/end time: 0 0.4

Now press the *Add* button to add this action event to the event list. Press the time slider's play button to see the demo sequence just specified. You should see the *OrthoSlice* plane moving during the time range from 0 to 0.4, but nothing happening in the time range from 0.4 to 1.0.

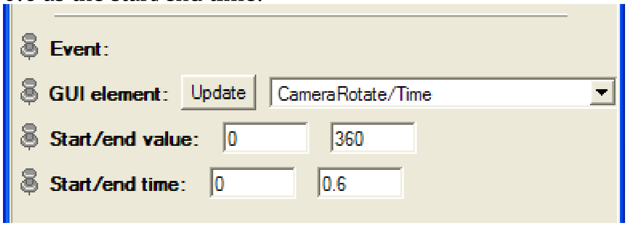
Now select a threshold of 70 in the *Isosurface* module and press *DoIt* to make the surface appear in the viewer. Now go back to *DemoMaker*, select *OrthoSlice/Viewer mask/Viewer 0* from the *GUI elements* menu, select *on* as the toggle value and `0.4` as the trigger time:



Event:
GUI element: Update OrthoSlice/Viewer mask/Viewer 0
Toggle to value: [✓ on] off
Trigger time: 0.4

Now press *Add* again to add this action to the list. Jump back to `0` by clicking on the time slider. You should see that when jumping back (from later than 0.4 to earlier than 0.4) the *Isosurface* is switched off. Then play the demo from the beginning, and you see that the surface is switched on right after the OrthoSlice is moved.

As another excercise you can create a camera path (select *Create/CameraPath* or *Create/CameraRotate* from the menu and edit the camera path to your liking. Then press *Update* in *DemoMaker* and select *CameraPath/Time* or *CameraRotate/Time* as GUI element. Now the start/end value refers to the value of the camera time slider. Enter the minimum and maximum values of that time slider. Enter 0.0 and 0.6 as the start/end time:



Event:
GUI element: Update CameraRotate/Time
Start/end value: 0 360
Start/end time: 0 0.6

Now you see that in parallel to the actions defined before, the camera path is applied to the scene. Note that you can execute only parts of a camera path, or concatenate multiple paths this way. However, multiple camera paths cannot be used in parallel.

Connections

Data `[not used]`

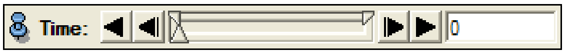
The *data* port is a default script object port, but is not used by *DemoMaker*.

Time `[optional]`

The *Time* port is a quick way for synchronizing other modules with DemoMaker's *Time* port, e.g. for using the MovieMaker module or camera path modules like CameraRotate or CameraPath. Alternatively, you can use camera path objects with *DemoMaker* by selecting these objects' time sliders from the GUI elment port and making it part of the *DemoMaker* event list.

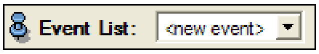
Ports

Time



The time slider defines the time range in which the demo sequence is played, and by clicking on the slider, one can jump to an arbitrary point of the demo. Clicking one of the play buttons (triangles pointing to the left/right for backwards/forwards) will start playing the demo. Right-clicking on the port brings up a dialog box allowing you to change the range of the slider (min/max value) as well as the increment determining the playback speed (smaller increment means slower playback). For more information, see documentation of the time port.

Event List (Selection)

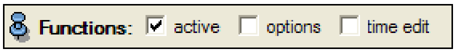


This menu contains the list of currently defined demo actions that make up the demo sequence. The buttons below this menu always refer to the currently selected menu entry.

Event List (Buttons)

Manipulate the event list menu. *Add* adds the event defined in the lower part of the *DemoMaker* module to the list of demo action events. When selecting one of the action entries from the *Event List* menu, the *Remove* button deletes this entry from the list, and the *Replace* button replaces the selected entry by the event as defined in the lower part of the *DemoMaker* module.

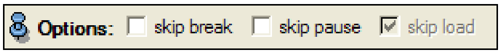
Functions



Activate/deactivate this *DemoMaker* instance, and show optional parts of the *DemoMaker* user interface, such as some option ports and some ports for editing the time line.

- *active:* activate/deactivate this module. When deactivated, the module will not define any function keys and not manipulate objects in the object pool. Deactivating is especially important when using multiple alternative *DemoMaker* objects.
- *options:* display option ports (see *Options (1)* - *Options (3)* below).
- *time edit:* display ports for editing the time line (see *Move from* - *Time edit* below).

Options (1)

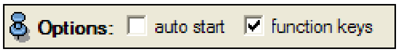


This port is only shown when the *options* toggle of the *Functions* port is enabled.

- *skip break:* do not stop at user-defined breaks when playing the demo sequence.

- *skip pause:* do not execute user-defined pause events (no waiting time).

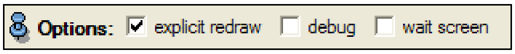
Options (2)



This port is only shown when the *options* toggle of the *Functions* port is enabled.

- *auto start:* lets the demo start automatically when the network containing the *DemoMaker* module is loaded.
- *function keys:* when toggled off, no function keys (F3/F4/F9/F10) will be defined when the network containing the *DemoMaker* module is loaded. This is especially important when combining multiple *DemoMaker* modules, since only one module can define the function keys.

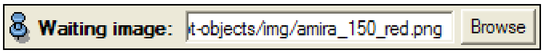
Options (3)



This port is only shown when the *options* toggle of the *functions* port is enabled.

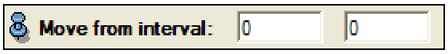
- *explicit redraw:* when toggled on, for each time step the active viewers are explicitly called to perform a redraw. If toggled off, amira's *auto redraw* feature is used. If in some cases the desired action sequence is not properly displayed in the viewer, try toggling *explicit redraw* on.
- *debug:* when toggled on, the actual TCL commands that are executed during the demo sequence are output to the amira console. This will significantly slow down the graphical display, but can be used for understanding what exactly happens in the demo sequence.
- *wait screen:* enable a waiting image to be displayed during jumps on the time line. This is only useful if jumping takes considerable amount of time, e.g. for very long event sequences. If enabled, the *Waiting image* port will appear below this option for specifying the waiting screen image file.

Waiting image



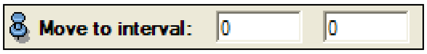
If the *wait screen* option is enabled, this port is used to specify the file name of the waiting screen image. The image will be displayed during jumps on the time slider.

Move from interval



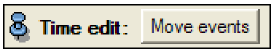
This port is only shown when the *time edit* toggle of the *Functions* port is enabled. Specify the time interval to be relocated on the time line (see Moving events on the time line below).

Move to interval



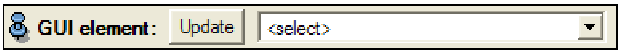
This port is only shown when the *time edit* toggle of the *Functions* port is enabled. Specify the target time interval for relocation on the time line (see Moving events on the time line below).

Time edit



This port is only shown when the *Time edit* toggle of the *Functions* port is enabled. *Move events:* move all events within the *Move from* interval to the *Move to* interval on the time line (see Moving events on the time line below).

GUI element



The selection menu contains all GUI elements in the current object pool that can be manipulated using *DemoMaker*. If the object pool is modified (e.g. by loading a new dataset or connecting a new module), press *Update* to update the contents of the selection menu.

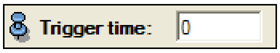
Additional Ports

Additional ports will appear in the *DemoMaker* module, depending on the type of *GUI element* the user has chosen. These ports are listed in the following section, grouped by GUI element types.

Defining a break

Selecting **Break, continue on keypress* from the GUI element menu lets you insert a break in the demo sequence. When playing, the demo will stop at that point. See playing a demo sequence above.

Trigger time

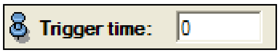


Defines the point in time (on the time slider) at which the break is inserted.

Defining a pause

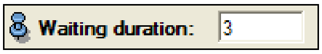
Selecting **Pause, waiting time* from the GUI element menu lets you insert a pause, where the demo will stop and wait for a user-defined amount of time. Please note that such a pause will not be executed if *DemoMaker* is driven by a *MovieMaker* object. In such a case, you must insert a pause by adjusting the time slider range and adapting the time range of the action entries.

Trigger time



Defines the starting point of the pause on the time slider.

Waiting duration

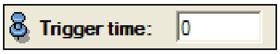

Waiting duration: 3

Defines the waiting time in seconds. Please note that this time is taken in addition to the time defined by the time slider range.

Defining a go-to

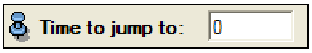
Selecting **Go-to, jump to user-specified time step* from the GUI element menu lets you jump from the one point in the demo sequence to another point. For example, you can simply define an endless loop by jumping back to some previous time step. You can end the loop by pressing the stop button or `F3` (see playing a demo sequence above).

Trigger time


Trigger time: 0

This defines the point in time (on the time slider) at which the go-to is inserted.

Time to jump to

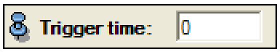

Time to jump to: 0

This defines the point in time which the go-to will jump to. For example, to create a loop between time 0.2 and 0.4, insert a go-to with *Trigger time* 0.4 and *Time to jump to* 0.2.

Defining a toggle

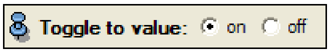
Toggle actions can toggle certain GUI elements between the two states *on* and *off*. Examples for toggle values are the different options in a *ToggleList port*¡/a¿, or the orange *viewer mask* toggle(s) present in every data object or display module (to switch the display of the module in the viewer on and off).

Trigger time


Trigger time: 0

This defines the point in time (on the time slider) at which the value is toggled.

Toggle to value

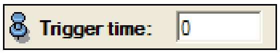

Toggle to value: [✓ on] off

This defines the value (on or off) to which the selected GUI element is set at the defined trigger time, if the demo sequence is played forwards. When playing or jumping backwards, the inverse value is used.

Defining a button press

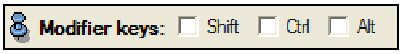
Button actions emulate the pressing of a button or executing certain events with no parameters. Examples for buttons elements are the ButtonList port, or inverting the orientation of a clipping plane.

Trigger time


Trigger time: 0

This defines the point in time (on the time slider) at which the button is pressed or other action is taken.

Modifier keys

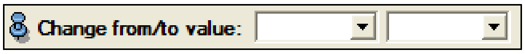

Modifier keys: Shift Ctrl Alt

This defines whether *DemoMaker* should emulate that `Shift`, `Ctrl`, or `Alt` is held down at the time when the button is pressed. The meaning of these modifier keys depends on the module defining the button to be pressed.

Defining a select action

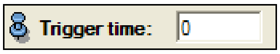
Select actions are used to set a port's value to one of a set of choices as offered by a selection menu or a radio box.

Change from/to value


Change from/to value:

This defines from which old value to which new value the port will be switched at the specified trigger time. When playing or jumping backwards, the two values will used inversely.

Trigger time

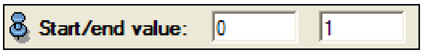

Trigger time: 0

This defines the point in time (on the time slider) at which the port is set to a new selection value.

Defining a numeric action

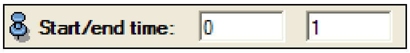
Numeric actions are used to vary a port's numerical value from a start value to an end value over time. Examples for numeric ports are numerical sliders or numerical text fields.

Start/end value


Start/end value: 0 1

The two fields of this port define from which start value to which end value the value of the GUI element will be varied. The values of this port are constrained to the range of the corresponding GUI element.

Start/end time

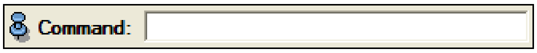

Start/end time: 0 1

This defines the time range (on the time slider) during which the port value will be varied from the start value to the end value specified above. When playing or jumping backwards, the value will be varied from the end value to the start value.

Defining a TCL command

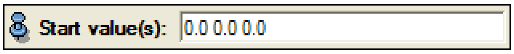
When selecting **Tcl command* from the GUI element menu, you can insert arbitrary TCL commands into the demo sequence. Furthermore, like for numeric events, the command can be parameterized by one or more numeric values that will be varied between user-defined start and end values.

TCL command


Command:

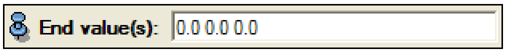
Text field for typing the desired TCL command. Within the command, special placeholders `%0%`, `%1%`, `%2%`, ... can be used that will be replaced by numeric values. The start and end values for these placeholders are defined in the ports described below.

Start value(s)


Start value(s): 0.0 0.0 0.0

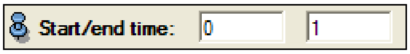
If the numeric placeholder `%0%` is used in the *Command* field, specify the start value for this placeholder. If multiple placeholders are used (e.g. `%0%` and `%1%`), specify space-separated start values for all of the employed placeholders.

End value(s)


End value(s): 0.0 0.0 0.0

If the numeric placeholder `%0%` is used in the *Command* field, specify the end value for this placeholder. If multiple placeholders are used (e.g. `%0%` and `%1%`), specify space-separated end values for all of the employed placeholders.

Start/end time

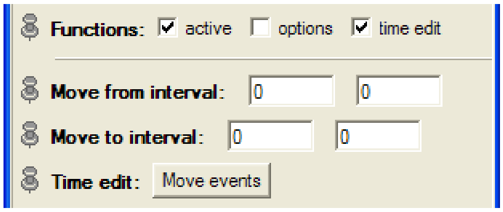

Start/end time: 0 1

This defines the time range (on the time slider) during which the specified TCL command will be executed, with the placeholders replaced by values between the specified start and end values.

Moving events on the time line

After you have defined some events on the time line, you may want to move a part of this event sequence to a different time, e.g. for inserting more events at a certain point, or for stretching or compacting part of the demo sequence.

This can be easily done if you switch on the *Time edit* toggle of the *Functions* port:



Functions: [✓ active] options [✓ time edit]

Move from interval: 0 0

Move to interval: 0 0

Time edit: Move events

Simply enter the time interval that you want to relocate in the *Move from* port, enter the destination time interval in the *Move to* port, and press the *Move events* button. If the new time interval is outside of the current min/max time, the time slider configuration will be updated accordingly.

Commands

The following commands are methods of the *DemoMaker* script object. They can be called externally by first specifying the name of the script object (e.g. DemoMaker), followed by a space and the name of the method. Example: `DemoMaker play` starts playing the demo sequence.

`play`
start playing the time slider at its current time step. This function is called when pressing `F4`.

`stop`
stop playing the time slider. This function is called when pressing `F3`. Also calles the *stop callback* (see below).

`jumpNext`
jump to the next user-defined break, or to the end of the demo. This function is called when pressing `F10`.

`jumpPrev`
jump to the previous user-defined break, or to the start of the demo. This function is called when pressing `F9`.

`setEndCallback <cmd>`
sets an internal callback function to the specified TCL code `<cmd>`. This code will be executed only once when DemoMaker reaches the end of the time slider. Call with an empty string to disable the callback.

DemoMaker tips and limitations

Although the *DemoMaker* module is quite powerful, some things should be noticed, and there are some known limitations that are listed here for convenience. If you find further important limitations or bugs, please report them to TGS as a bug or feature request.

- The **Load network* action type for loading a different network file is currently not implemented.

- For *button action*, ”snapping” a button to automatic update mode is not supported yet.
- GUI elements that are part of a Generic port are not properly represented in the *GUI elements* menu of the *DemoMaker* module.
- If a *DemoMaker* module is renamed (e.g. by selecting *Edit / Rename...* from the menu), the function keys `F3/F4/F9/F10` will no longer work. However, simply disabling and re-enabling the *function keys* option in *DemoMaker* helps.
- If you rename or remove modules after defining events with *DemoMaker*, then events involving these modules will generate errors. Please properly arrange your network before using DemoMaker.

Further links
script object documentation for script programming.

6.40 Digital Image Filters

Encloses the ImageEditor in a compute module.

6.41 Displace

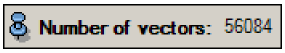
This module takes a displacement vector field defined on a tetrahedral or hexahedral grid or on a surface and creates a mesh with translated vertices. For example, if a surface vector field is used as input a new surface with modified vertex coordinates is computed. The displacement vector field must be defined on the vertices of the mesh. Other encodings are not supported. The new position of a vertex is computed by adding the scaled displacement vector defined at that vertex to the original position. The scaling factor can be adjusted globally for all vertices using the scale port.

Connections

Data `[required]`
The input vector field.

Ports

Number of vectors


Number of vectors: 56084

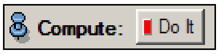
Shows the number of displacement vectors of the input field.

Scale

Scale: 1

Scaling factor applied to the displacement vectors ranging from 0 (no displacement) to 1 (full displacement).

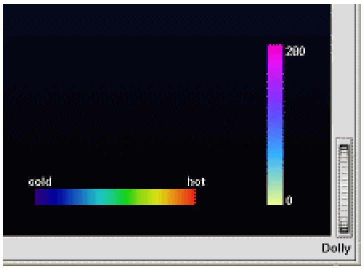
Compute


Compute: Do It

Triggers computation.

6.42 DisplayColormap

This module allows you to position a colormap-icon onto the 3D viewer. This is useful e.g., to produce snapshots, that shall contain an explanation of what specific colors mean. Note, that although colormaps are ordinary data objects in amira, they often are hidden by default. Use the Edit/Show menu of the main window to bring their icons up.



Connections

Data `[required]`

The colormap to be displayed.

Ports

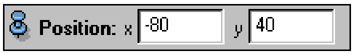
Options

Options: ☐ custom text ☑ vertical ☑ rel. size ☑ transp. bg

This port provides the following four toggles:

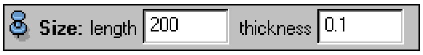
- **custom text:** Enable or disable custom text (see below).
- **vertical:** Vertical orientation instead of horizontal.
- **relative size:** Change size of colormap when viewer size changes.
- **transparent background:** If off render background rectangle.

Position


Position: x -80 y 40

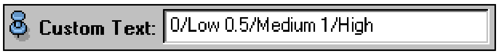
Position of colormap in viewer relative to lower left corner. If one of the numbers is negative, it is interpreted relative to upper right corner.

Size


Size: length 200 thickness 0.1

Length and width of the colormap in pixels. If the option *relative size* has been selected the length and width are interpreted relative to a window size of 1000 times 800. If the actual viewer window is smaller than the displayed colormap will be smaller too and vice versa.

Custom Text


Custom Text: 0/Low 0.5/Medium 1/High

This is only available if *custom text* option is selected. You may specify a space separeted list of values, that shall be displayed. In addition you may specify a text, which is displayed instead of the number, by using the '/' character. The example in the image above has been generated using this text: `-8/cold 50/hot`, which displays "cold" at value -8 and "hot" at the value 50 (the colormap in this case ranges from -8 to 50). If one of the labels should contain blanks you have to enclose the text in double quotes, e.g., `100/"very hot"`.

Commands

`getFontSize`
Returns the current font size.

`setFontSize <points>`
Changes the font size. The default font size is 14 points.

`setBGColor <color>`
Sets the color of the background rectangle which is drawn when the transparent background toggle is off.

`setColor <color>`
Sets the color of the text.

6.43 DisplayISL

This module visualizes a 3D vector field using so-called illuminated field lines. This technique was first presented on the *Visualization 96* conference. More details are described in *M. Zöckler, D. Stalling, H.-C. Hege, Interactive Visualization of 3D-Vector Fields using Illuminated Streamlines, Proc. IEEE Visualization '96, Oct./Nov. 1996, San Francisco, pp. 107-113, 1996.*

The module computes a large number of field lines, by integrating the vector field starting from random seed points. The lines are displayed using a special illumination technique, which gives a much better spatial understanding of the fields structure than ordinary constant-colored lines. In order to execute a script demonstrating this module click here.

The functionality of *DisplayISL* can be extended by means of the SeedSurface module.

Connections

Data `[required]`

The vector field to be visualized. Since a procedural data interface is used all types of vector fields are supported (e.g., fields on regular or curvilinear grids, as well as procedurally defined fields).

ColorField `[optional]`

An optional scalar field which, if present, will be used for pseudo-coloring. Often it is useful to create a scalar field from the vector field using the Magnitude module.

AlphaField `[optional]`

An optional scalar field which can be used to control the lines' opacity locally.

Distribution `[optional]`

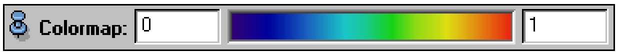
An optional scalar field which can be used to control the distribution of seed points.

Colormap `[optional]`

The colormap used to encode the color field. Will only be visible if a color field is connected to this module.

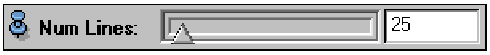
Ports

Colormap


Colormap: 0 1

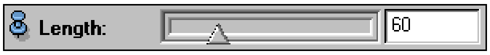
Optional colormap.

Num Lines


Num Lines: 25

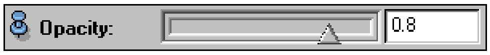
Number of field lines to be displayed. The technique typically gives the best results with a rather large number of lines (e.g., 100-500). Note however that on systems with slow 3D graphics without texture hardware, a large number of lines can slow down rendering speed significantly.

Length


Length: 60

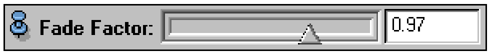
The length of the field lines, or more precisely, the number of atomic line segments, in forward respectively backward direction. The lines may stop earlier if a singularity (i.e. zero magnitude) is encountered or if the field's domain is left. On default lines are traced the same distance in forward and backward direction. You may use the command `setBalance` to change this behavior.

Opacity


Opacity: 0.8

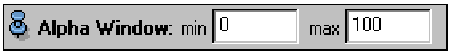
Base opacity factor of the lines. A value of 1 produces completely opaque lines, while 0 results in fully transparent lines.

Fade Factor


Fade Factor: 0.97

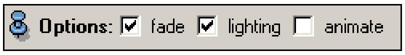
This port controls how fast opacity decreases along field lines if *fade mode* is enabled. The first atomic line segment will be assigned the base opacity set in port *Opacity*. The opacity of each successive segment will be multiplied by the value of this port. This is useful in order to encode the directional sign of a vector field. The vector field points in the direction of increasing transparency.

Alpha Window


Alpha Window: min 0 max 100

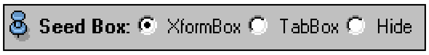
This port will only be visible if *fade mode* is disabled and if an alpha field is connected to the module. In this case the alpha field's values are mapped to opacity according to the range specified by this port. At locations where the alpha field is equal or smaller than *min* field lines will be completely transparent. Likewise, at locations where the alpha field is equal or bigger than *max* field lines will be completely opaque.

Options


Options: ✓ fade ✓ lighting ☐ animate

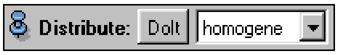
Fade: Enables fading mode as described above.
Lighting: Controls illumination of lines. If off, constant colored lines are drawn (flat shading).
Animate: Activates particle-like animation. The animation speed may be controlled via the command `setAnimationSpeed`.

Seed Box


Seed Box: [● XformBox] ○ TabBox ○ Hide

Clicking on *XformBox* or *TabBox* brings up a 3D dragger in the 3D Viewer. This allows you to restrict the region of interest, i.e. the region in which seed points are placed, by interactively transforming the dragger (Remember to switch the viewer into interaction mode by hitting ESC). After changing the box you have to hit *DoIt* to trigger recalculation.

Distribute


Distribute: DoIt homogene

On the one hand, this port provides a *DoIt* button which is used to initiate distribution of seeds and recomputation of field lines. Once the incoming vector field has changed or you have modified the number of field lines or the line's length you have to press *DoIt* in order to update the display.

On the other hand, the port also provides an option menu specifying the way how seed points are distributed inside the seed box. On default a *homogeneous* distribution will be used. Alternatively, seed points may be distributed according to the vector field's magnitude or according to the value of the distribution field, if such a field is connected to the module. The *equalize* option provides a mixture of homogeneous and proportional seed point distribution.

6.44 DisplayTime

This module displays the value of a time object in the 3D viewer using a textual or iconic representation. This is useful for for animations which shall contain an illustration of time. Four major styles are available: plain text, a round clock, a horizontal time bar, or a vertical time bar.

This module should be connected to a Time object. In this case the current time is visualized. Besides that it is possible to visualize an arbitrary value which can be set using the commands of the invisible port value.

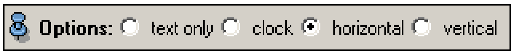
Connections

Time `[optional]`

The time object to be displayed.

Ports

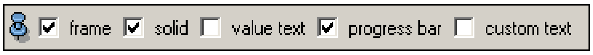
Options



Options: text only clock [✓ horizontal] vertical

This port chooses between the four major draw styles.

Options2

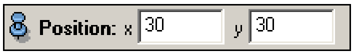


[✓ frame] [✓ solid] value text [✓ progress bar] custom text

This port adjusts additional settings influencing the drawing. The last three options are disabled for the *text only* draw style.

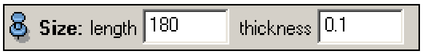
- **frame** Draw a frame around the time icon.
- **solid** Fill the icon's background.
- **value text** Show value together with time icon.
- **filled value** Fill space between zero and value.
- **custom text** Whether or not custom text should be displayed (see below).

Position



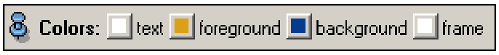
Position of the time icon with respect to the lower left corner of the screen. If negative values are entered the icon is positioned relative to the right and/or top of the screen.

Size



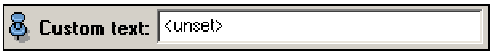
Specifies the length of the horizontal or vertical time bar (in pixels) and relative thickness of the bars. If a clock is drawn *length* refers to the diagonal, while *tickness* is ignored. For text only display both values are ignored.

Colors



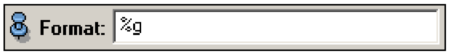
The colors of the various parts of the icon can be set here.

Custom Text



This is only available if *custom text* option is selected. You may specify a space separated list of values, that shall be displayed. Alternatively you may specify a text, which is displayed instead of the number, by using the '/' character. The syntax is the same as for the Display Colormap module.

Format



The format for displaying the time value (printf syntax of the C programming language).

Value

This port is always hidden. It can be used to specify a time value when no time object is connected (via the Tcl interface).

Commands

`setFontSize <points>`

Changes the font size. The default font size is 28 points for text only displays and 14 points otherwise.

6.45 DistanceMap

This module computes a 3D distance field of a 3D object. Each voxel will be assigned a value depending on the distance to the nearest object boundary. The boundary voxels of the object get a value of zero whereas the assigned value gets larger the more the distance increases.

To use this module it must be connected to a uniform label field where each voxel with a nonzero value is assumed to belong to the object.

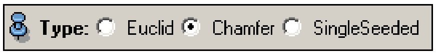
Connections

Data `[required]`

Labelfield wherefrom the distance map is computed.

Ports

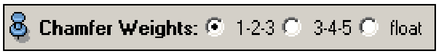
Type



Type: Euclid [✓ Chamfer] SingleSeeded

You may either choose a true euclidian distance metric or an approximation based on a 3x3x3 chamfer metric. The latter is much faster to compute an accurate enough for most applications. Single Seeded computes a distance map which expresses the distance from a single seed point rather than from the boundary.

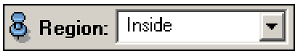
Chamfer Weights



Chamfer Weights: [✓ 1-2-3] 3-4-5 float

This port is only available in chamfer mode. Different chamfer metrics are available. The 1-2-3 metric is equivalent to only consider a 6-neighborhood when propagating the distance value, whereas the 3-4-5 considers a 26-neighborhood and is a better approximation of the euclidian distance metric. Float also corresponds to a 26-neighborhood but the resulting field will have float data type instead of short int.

Region

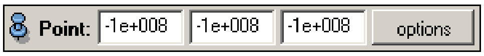


Region: Inside

This port is not available in Single Seeded mode. Choose in which region the distance field will be computed:

- **Inside:** Inside the object (outside will be set to zero).
- **Outside:** Outside the object (inside will be set to zero).
- **Both (unsigned):** Inside and outside the object. The positive distance is computed regardless to the position being inside or outside the object.
- **Both (signed):** The distance value will be negative at a position inside the object and positive outside the object.

Point



Point: -1e+008 -1e+008 -1e+008

This port is only available in Single Seeded mode. Specifies the seed point for the distance map in world coordinates. You can use a dragger to adjust.

Action

Triggers the computation.

6.46 Divergence

The *Divergence* module computes the divergence of a vector field consisting of floats defined on a uniform grid. The output is a uniform scalar field.

$$\mathrm{div} V = \frac{\partial V_x}{\partial x} + \frac{\partial V_y}{\partial y} + \frac{\partial V_z}{\partial z}$$

Connections

Data `[required]`

Vector field defined on a uniform grid (*UniformVectorField3*). The vector components must be floats.

Ports

Compute

Pushing this button triggers the computation.

6.47 DoseVolume (Tetrahedra)

The *Dose Volume* module creates a value-volume-histogram for a scalar field defined on a tetrahedral grid, e.g. a temperature-volume-histogram for a scalar field that represents a temperature distribution. For this purpose it has to be attached to a GridVolume module and a selection of tetrahedra specifying parts of the grid or the complete grid must be made there. A histogram is constructed for the selected tetrahedra only. This is done in the following way:

The range of scalar values is divided into subranges given by the number of sample points. With respect to each subrange one or more sub-volumes of the tetrahedral grid are determined where the scalar field values fall into that range. Then the volumetric sizes of the sub-volumes are computed and these are depicted as differential histogram function values over the total range of scalar field values or used to calculate cumulative histogram values.

If the selected tetrahedra are members of different regions, histograms are calculated separately for each region, as well as for all selected tetrahedra (called 'total Volume').

The histograms can be shown in several ways:

- **differential:** shows a differential form of the histogram.
- **absolute** and **relative**: show an integral form of the histogram, i.e., for each value the partial volume is shown where *at least* that value is assumed. For the integral form either the **absolute** volume [ccm] or the **relative** volume, i.e., the percentage of the total volume of the corresponding region, can be shown. With the latter type of representation you can easily find out which value is at least assumed in 90 percent of a selected region.

Connections

Data `[required]`

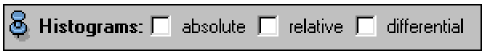
A scalar field defined on a tetrahedral grid.

PortGridVol `[required]`

A GridVolume module that selects the tetrahedra for which the histogram is calculated.

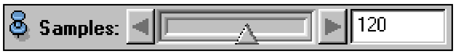
Ports

Histograms


Histograms: ☐ absolute ☐ relative ☐ differential

If you select one of these toggles, a plot window appears showing the corresponding histogram for all tetrahedra selected by the *GridVolume* module. If no tetrahedra have been selected, the plot window will not be shown. For the three different modes of representation see above.

Samples


Samples: 120

This slider lets you select the number of samples for the histogram.

6.48 DuplicateNodes

This module takes a labeled tetrahedral grid as input and produces a new grid with all nodes on *interior boundaries* being duplicated. Interior boundaries can be visualized using module GridBoundary. They consist of all triangles incident on two tetrahedra with different labels. Duplicated nodes are useful to represent discontinuities within a vector field, e.g., an electric field on a tetrahedral patient model. On an interior boundary such a field may have multiple values - one for each incident material subvolume.

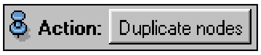
Connections

Data `[required]`

A tetrahedral grid without duplicated nodes.

Ports

Action



Button *duplicate nodes* causes a new output grid with duplicated nodes to be computed.

6.49 FieldCut

The *Field Cut* module is a tool for visualizing cross sections of a 3-dimensional scalar field defined on a tetrahedral grid. In general such grid volumes are made up of several materials. A (2-dimensional) cutting plane has to be defined for this purpose and may be shifted through the (3-dimensional) tetrahedral grid in orthogonal direction, an *Orientation* port is provided for setting the orientation of the cutting plane perpendicular to one of the main axes and a *Translate* port for specifying an orthogonal translation. *FieldCut* renders a slice with the values of the scalar field mapped to colors of a connected colormap. This is called *pseudo coloring*. The cutting plane is usually clipped to a rectangle. With the 3D viewer turned into interactive mode you can pick one of its edges and shift it through the grid volume. To define arbitrary cutting planes you have to set the *rotate* toggle which makes a rotation handle appear in the center of the cutting plane. Having turned the viewer into interactive mode you can pick the handle with the left mouse button and rotate the plane as you please.

Connections

Data `[required]`

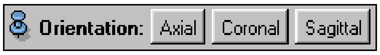
The field of type TetrahedralGrid.

Colormap `[optional]`

An optional colormap used for pseudocoloring the values of the scalar field on the cutting plane.

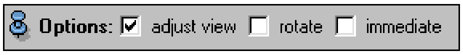
Ports

Orientation



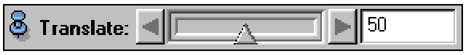
This port provides three buttons to reset the slice orientation. Axial slices are perpendicular to the z-axis, frontal slices are perpendicular to the y-axis, and sagittal slices are perpendicular to the x-axis. See also PortButtonList.

Options


Options: [✓ adjust view] rotate immediate

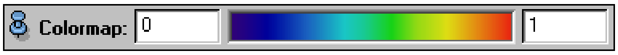
If the *adjust view* toggle is set, the camera of the main viewer is reset each time a new slice orientation is selected. With the *rotate* toggle you can switch the rotate handle for the cutting plane on and off. If the *immediate* toggle is set the slice is updated every time you drag it with the mouse in the 3D viewer. Otherwise only the bounding box of the cutting plane is moved and the update takes place when you release the mouse button.

Translate


Translate: 50

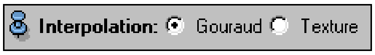
This slider allows you to select different slices. The slices may also be picked and dragged directly in the 3D viewer.

Colormap


Colormap: 0 1

See also PortColormap.

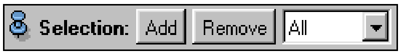
Interpolation


Interpolation: [✓ Gouraud] Texture

In pseudo-color mode you can switch between two rendering methods by the radio buttons *Gouraud* and *Texture*. See also PortRadioBox.

- **Gouraud** shaded faces have their colors assigned at their vertices. The colors in between are linearly interpolated. Therefore misleading colors occasionly may occur or colors may appear to be washed out along the edges of the faces.
- **Texture** mapping means an exact mapping of values to colors in this special case, but with the drawback of increased computing time if you have no hardware supported for texture mapping on your machine.

Selection


Selection: Add Remove All

This port maintains a list of materials assigned for cutting. With the selection menu one can select a single material. The *Add* button adds a previously selected material to the list and the *Remove* button removes the current selected material.

6.50 GetCurvature

This module computes curvature information for a discrete triangular surface of type Surface. Either the maximum principal curvature value, the reciprocal curvature value, or the direction of maximum principal curvature can be computed. The algorithm works by approximating the surface locally by a quadric form. The eigenvalues and eigenvectors of the quadric form correspond to the principal curvature values and to the directions of principal curvature. Note, that the algorithm does not produce meaningful results near locations where the input surface is not topologically flat, i.e., where it has non-manifold structure.

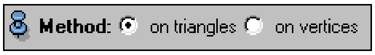
Connections

Data `[required]`

The surface for which curvature information should be computed.

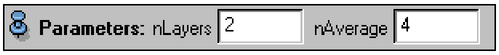
Ports

Method


Method: [✓ on triangles] on vertices

Radio box allowing the user to select between two different computational algorithms. Choice *on triangles* produces a surface field with curvature values or curvature vectors being defined on the surface's triangles. Alternatively, by selecting *on vertices* a surface field with data being defined on the vertices can be generated.

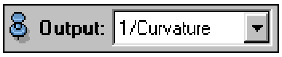
Parameters


Parameters: nLayers 2 nAverage 4

The first input, denoted *nLayers*, determines which triangles are considered to be neighbors of a given triangle and which points are considered to be neighbors of a given point. If the value of this input is 1, then only triangles sharing a common edge with a given triangle are considered to be neighbors of this triangle and only points directly connected to a given point are considered to be neighbors of this point. For larger values of *nLayers* successively larger neighborhoods are taken into account.

The second input, denoted *nAverage*, determines how many times the initial curvature values computed for a triangle or for a point are being averaged with the curvature values of direct (first-order) neighbor triangles or points. The larger the value of *nAverage* the smoother the curvature data being obtained. Note, that averaging only applies to the scalar curvature values, not to the directional curvature vectors which are computed when port output is set to *max direction*.

Output


Output: 1/Curvature

This menu controls the output of the curvature module. If *curvature* is selected then a surface scalar field is generated containing the maximal principal curvature of a triangle or of a point. If *1/curvature* is selected then the curvature values are inverted. In this case the output values have the dimension of a length, indicating the radius of a sphere locally fitting the surface.

The *mean curvature* and *1/mean curvature* are similar to the first two options. Here, however, the mean value of the two principal curvature values is computed. This quantity will be negative in strictly concave regions and positive in strictly convex regions. It can be zero in regions where a positive and negative principal curvature cancel each other.

The *Gauss curvature* is the product of the two principal curvatures. It is negative in surface areas with hyperbolic geometry (convex-concave, like near saddle points) and positive in areas with elliptic geometry (strictly convex or strictly concave).

If *max direction* is selected then a surface vector field is computed indicating the direction of maximum principal curvature. The length of a directional vector is equal to the corrsponding curvature value.

Action

Press this button to start computation.

6.51 Gradient

The *Gradient* module computes the gradient of a scalar field consisting of floats defined on a uniform grid. The output is a uniform vector field. The direction of the gradient vector depends on the setting of port *Output*. If *Force* is selected the negative gradient vector is computed.

$$\mathrm{grad}F = \pm\left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z}\right)$$

Connections

Data `[required]`

Scalar field defined on a uniform grid (*UniformScalarField3*). The scalar values must be floats.

Ports

Output

If *Force* is selected the negative gradient vector is computed.

6.52 GridBoundary

The *Grid Boundary* module is a tool for visualizing individual faces of a tetrahedral grid. The faces may be colored according to an arbitrary scalar field. As the name implies, the module extracts boundaries between tetrahedra of different material type. The particular materials to be shown can be selected manually. In some cases two materials may have no common faces and nothing will be seen. However, selecting *All* as the first material parameter and any other material as the second will show the surface of the second material. Visible faces are stored in an internal buffer similar to the GridVolume module. Likewise, the selection domain can be restricted interactively by adjusting a selection box. Ctrl-clicking on a face makes it invisible.

If the ColorField port is connected to a scalar field, an additional colormap port becomes visible and the faces are drawn in pseudo-color mode. By default the colormap port is not connected to any colormap. Therefore the grid appears in a constant color. You can click with the right mouse button over the colormap to get a popup menu of all available colormaps. When a colormap has been connected to the module, the data values at the vertices of the selected triangles are mapped to their associated colors.

Connections

Data `[required]`

The tetrahedral grid to be visualized.

ColorField `[optional]`

Scalar field used for pseudo-coloring the selected triangles. Pseudo-coloring also requires that a colormap has been connected to the module.

Colormap `[optional]`

A colormap is used to map the data values of the optional scalar field.

ROI `[optional]`

Optional connection to an object providing a region-of-interset, e.g., SelectRoi. Only triangles inside this region will be visualized.

Ports

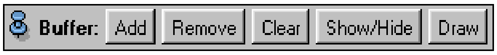
Draw Style

This port is inherited from the ViewBase class and therefore the description will be found there. In contrast to other modules derived from the *ViewBase* class this module does not provide the possibility to consider vertex and direct normals. The triangles will always be drawn flat.

Colormap

This port becomes visible only if a scalar field has been connected to the *ColorField* port. For further details see section Colormap.

Buffer



The Buffer buttons give you some control of the internal face buffer of the GridBoundary module. Only the faces present in the buffer are displayed according to the current drawing style. The *Add* button adds the highlighted faces to the buffer. The *Remove* button removes highlighted faces from the buffer. The *Clear* button removes all faces from the buffer. The *Hide* button deselects all faces but does not change the buffer. See also PortButtonList.

Materials

This port provides two menus where all different materials of the input grid are listed. Faces which belong to the boundary between the selected materials are displayed using a red (highlighted) wireframe in the viewer. You may crop the faces by turning the viewer into interactive mode and move the green handles of the selection box. Only the faces inside the bounding box can be added to the buffer, also see PortButtonList.

Color Mode

Here you may choose among several color modes. For details see the module SurfaceView.

6.53 GridCut

The *GridCut* module is a tool for visualizing a cross section of a tetrahedral grid consisting of different materials. Within the cross section the different materials are indicated by their respective colors. The module is derived from ArbitraryCut and thus provides the same methods for manipulating the position and orientation of the cross section as this base class. An similar module SurfaceCut exists for displaying filled cross-sections through a surface separating different regions in space.

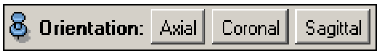
Connections

Data `[required]`

The tetrahedral grid to be visualized.

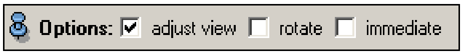
Ports

Orientation



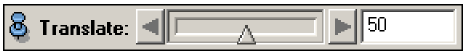
This port provides three buttons to specify the slice orientation. *Axial* slices are perpendicular to the z-axis, *coronal* slices are perpendicular to the y-axis, and *sagittal* slices are perpendicular to the x-axis.

Options



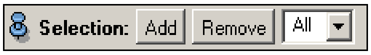
If the *adjust view* toggle is set, the camera of the main viewer is reset each time a new slice orientation is selected. With the *rotate* toggle you can switch on the rotate handle for the cutting plane and off again. If the *immediate* toggle is set the slice is updated every time you drag it with the mouse in the 3D viewer. Otherwise only the bounding box of the cutting plane is moved and the update takes place when you release the mouse button.

Translate



This slider allows you to select different slices. The slices may also be picked and dragged directly in the 3D viewer.

Selection



This port maintains a list of materials to be displayed within the cross section. With the selection menu one can select a single material. The *Add* button adds the currently selected material to the list so the that is becomes visible and the *Remove* button removes the material so that is becomes invisible.

Commands

Inherits all commands of ArbitraryCut.

`selectMaterial <id1> [<id2> ...]`

Selects the materials with the specified ids so that intersections of these materials with the cutting plane will be shown. You need to call `fire` before changes take effect.

`unselectMaterial <id1> [<id2> ...]`

Unselects the materials with the specified ids so that intersections of these materials with the cutting plane will not be shown. You need to call `fire` before changes take effect.

6.54 GridView

This module allows you to visualize the grid structure of a regular 3D scalar or vector field with stacked, rectilinear, or curvilinear coordinates. The nodes of the underlying grid can be addressed using an index triple (i,j,k). The *GridView* module extracts slices of constant i, j, or k index out of the grid and displays them as a wireframe model. In principle, the module may also be connected to a field with uniform coordinates. However, it will not be listed in the popup menu of such fields.

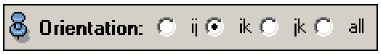
Connections

Data `[required]`

The regular field to be investigated.

Ports

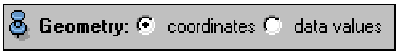
Orientation



Orientation: ij [✓ ik] jk all

Specifies whether slices in the ij, ik, or jk plane should be displayed. For stacked as well as rectilinear coordinates the indices i, j, and k refer to the x, y, and z direction, respectively.

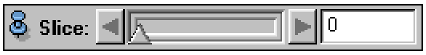
Geometry



Geometry: [✓ coordinates] data values

This port is only visible if the module is connected to a real-valued 3D vector field. In this case the grid may be built using the *data values* instead of the node's *coordinates*. This is a useful option if the vector field contains displacement vectors.

Slice



Slice: 0

Allows you to select the constant node index, e.g., the k index if orientation is set to ij.

6.55 GridVolume

The *GridVolume* module is a powerful tool for visualizing labeled tetrahedral volume grids, e.g., tetrahedral patient models. Views on various sub-grids can be specified, e.g., a view on a sub-grid corresponding to one of the tissue compartments of a patient model. The module maintains an internal buffer which contains all currently visible tetrahedra. Tetrahedra belonging to a particular compartment can be selected and added to the buffer. Selected tetrahedra are displayed using a red wireframe. Having been added to the buffer they are displayed in their associated colors. The selection can be restricted by means of an adjustable selection box. In addition, individual tetrahedra can be removed from the buffer by Ctrl-clicking on one of their faces, but notice that the viewer has to be turned into interactive mode for this. Similarly, a simple click on a face adds the tetrahedra in front of it to the buffer.

Connections

Data `[required]`

The labelled tetrahedral grid to be visualized.

ColorField `[optional]`

Arbitrary scalar field which is mapped onto the grid volume using pseudo-coloring.

Colormap `[optional]`

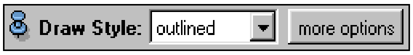
The colormap is used for pseudo-coloring the grid volume.

ROI `[optional]`

Optional connection to an object providing a region-of-interset, e.g., SelectRoi. Only triangles inside this region will be visualized.

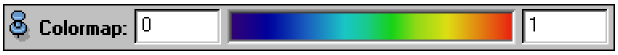
Ports

Draw Style



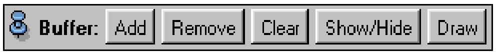
This port is inherited from the ViewBase class and therefore the description will be found there. In contrast to other modules derived from the *ViewBase* class this module does not provide the possibility to consider vertex and direct normals. The triangles will always be drawn flat.

Colormap



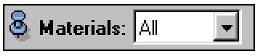
This port becomes visible only if a scalar field has been connected to the *ColorField* port. For further details see section Colormap.

Buffer



This port lets you *add* and *remove* highlighted triangles (being displayed in red wireframe) to an internal buffer. For a further description and for the functionality of each of the port buttons see ViewBase.

Materials



This port provides a menu where all materials of the input grid are listed. If you choose a material, all tetrahedra of that material are selected and displayed using a red wireframe model. You may crop the selection domain by turning the viewer into interactive mode and moving the green handles of the bounding box to a position of your choice. Only the tetrahedra inside the bounding box can be added to the buffer.

Color mode

Color mode: [✓ normal] mixed twisted

Here you may choose among several color modes. For details see the module SurfaceView.

Commands

`getSelectedTetra`

Displays a run-length encoded list of all currently selected tetrahedra. The list consists of pairs of integer numbers. The first number is the index of a selected tetrahedron. The second value denotes the number of subsequent selected tetrahedra.

`setSelectedTetra <list>`

Adds some tetrahedra to the internal buffer so that they become visible. The argument is a run-length encoded list like the one displayed by getSelectedTetra.

6.56 Grouping

This module allows you to define arbitrary groups or surface triangles and/or elements of tetrahedral or hexahedral grids. The groups can be used to easily display parts of more complex objects. The grouping module can be connected to one or more ordinary display modules like SurfaceView, GridVolume, or HexaView (in fact any module derived from ViewBase will work). In order to define a group first select the parts of the object you are interested in using the selection mechanisms provided by the input modules. For example, in case of a *SurfaceView* module different parts can be selected via the materials menu, via the selection box (buffer show/hide), via 2D lasso selection (buffer draw), or by selecting or deselecting individual triangles with the mouse. Once the parts of the object you want to be in a group are visible, select the *New group* button. Later, this particular display state can be restored by choosing the group from the *Groups* combo box again.

Internally groups are defined in a bitfield (one bit for each selectable element). The bitfields are stored in the parameter section of the corresponding data objects. If the objects are saved in the AmiraMesh or HxSurface file format after the groups have been defined the group definitions are saved too. In addition, the group definitions will also be included in network files. Note, that group definitions may become invalid if the data objects are modified, e.g., if the number of triangles of a surface is reduced using the simplification editor.

Connections

Data `[required]`

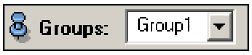
Connection to a display module like SurfaceView, GridVolume, or HexaView.

Module2 `[optional]`

Connection to an additional display module connected to some other data object than the first one. If multiple inputs are connected to the grouping module elements of all input objects can be grouped together.

Ports

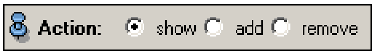
Groups



Groups: Group1

This combo box lists all groups currently being defined. Selecting a group from the box causes the elements contained in that group to be shown, to be added to the view, or to be removed from the view, depending on the value of the *Action* port (see below).

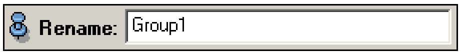
Action



Action: [✓ show] add remove

This radio box defines what happens when a new group is chosen in the *Groups* menu. If *show* is selected, the elements of the new group become visible and all other elements become invisible. If *add* is selected the elements of the new group are added to the view. If *remove* is selected the elements of the new group are removed from the view. If *nop* is selected nothing happens. This last option allows you to change the *Groups* port without changing the view.

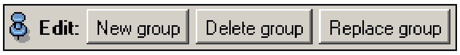
Rename



Rename: Group1

This text port allows to rename the current group. On default new groups are called *Group1*, *Group2*, etc.

Edit



This port provides three button for creating a new group containing all elements currently being visible (selected, not highlighted), for deleting the current group without changing the selection, and for replacing the contents of a group by the elements currently being visible.

6.57 HeightField

The *HeightField* module offers another method for the visualization of scalar data fields defined on regular grids, such as stacks of tomographic images. The data are visualized by extracting an arbitrary axial, frontal or sagittal slice out of the volume. This slice is represented as a surface for which the heights of the vertices are mapped to the data, using a tunable scale parameter.

Connections

Data `[required]`

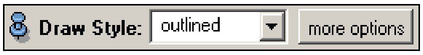
The scalar or color field to be visualized. If an RGBA color field is connected the vertices of the height field are colored as in the color field. The displacement is computed from the fields grayvalue intensity which is computed using the formula *0.3*R+0.59*G+0.11*B*.

Colormap `[optional]`

The colormap used to map data values to colors.

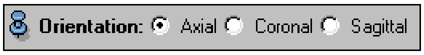
Ports

DrawStyle


Draw Style: outlined, more options

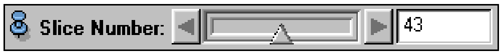
This port is inherited from ViewBase. For a description see there.

Orientation


Orientation: [✓ Axial] Coronal Sagittal

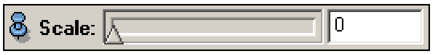
This port provides three buttons for resetting the slice orientation. *Axial* slices are perpendicular to the z-axis, *coronal* slices are perpendicular to the y-axis, and *sagittal* slices are perpendicular to the x-axis.

Slice Number


Slice Number: 43

This slider allows you to select different slices.

Scale


Scale: 0

This slider allows you to select the desired scale for the data-heights mapping. This should be in the range [-1,1].

6.58 HexToTet

The *HexToTet* module converts a hexahedral grid to a equivalent tetrahedral grid. The new tetrahedral grid has the same points as the hexahedral grid to be converted. Each hexahedron is converted to six tetrahedra which have the same material ID as the 'parent' hexahedron.

The module also converts the data objects connected to the hexahedral grid.

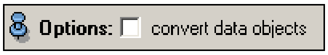
Connections

Data `[required]`

Unstructured hexahedral grid to be converted.

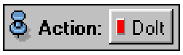
Ports

Options



Toggles wether the data objects connected to the hexahedral grid are also converted or not.

Action



Press the *DoIt* button to start the conversion.

6.59 HexaView

The HexaView module displays an unstructured hexahedral grid, or parts of the grid. Optionally an independent scalar field can be mapped onto the grid as pseudo-colors. The module behaves almost identical to the GridVolume module for tetrahedral grids.

Connections

Data `[required]`

Unstructured hexahedral grid to be visualized.

ColorField `[optional]`

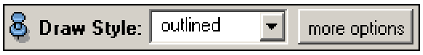
An arbitrary scalar field used for pseudo-coloring.

Colormap `[optional]`

The colormap used for pseudo-coloring, only used when a color field is connected.

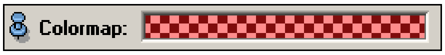
Ports

Draw Style



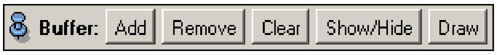
Please refer to the GridVolume documentation.

Colormap



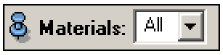
Please refer to the GridVolume documentation.

Buffer



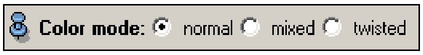
Please refer to the GridVolume documentation.

Materials


Materials: All

Please refer to the GridVolume documentation.

Color mode


Color mode: [✓ normal] mixed twisted

Please refer to the GridVolume documentation.

6.60 Histogram

This module computes the histogram of the data values of a scalar field and plots it in a separate plot window. In addition, the mean value and the standard deviation are printed.

Connections

Data `[required]`

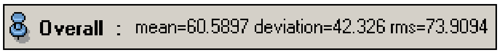
The scalar field to be investigated. Regular fields, fields defined on tetrahedral or hexahedral grids, and fields defined on surfaces are supported.

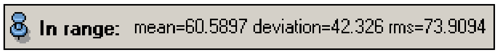
Labels `[optional]`

A uniform label field which can be used to restrict the histogram to a certain material.

Ports

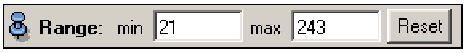
Info


Overall : mean=60.5897 deviation=42.326 rms=73.9094


In range: mean=60.5897 deviation=42.326 rms=73.9094

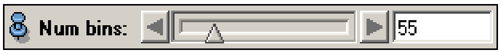
Prints the mean value and the standard deviation of all input data values once the histogram has been computed. Both info ports are only shown when the *DoIt* button has been pushed, i.e. the histogram has been computed.

Range


Range: min 21 max 243 Reset

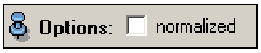
Defines the data range over which the histogram is to be computed. The *Reset* button adjusts the data range so that it completely covers the values of the input object.

NumBins


Num bins: 55

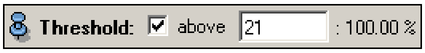
Defines the number of bins (intervals) the data range is devided into. The histogram counts the number of data values falling in each bin. In case of integer input values, i.e., bytes, shorts, or ints, the number of bins is internally adjusted so that all bins have the same integer width. This is required in order to avoid aliasing effects.

Options


Options: ☐ normalized

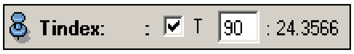
If the *normalize* toggle is set the histogram is scaled so that its area equals one. Otherwise the total number of counts is shown.

Threshold


Threshold: ☑ above 21 : 100.00 %

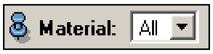
Defines a threshold value and displays the percentage of all data values which lie above the given threshold. The percentage corresponds to the given range. The threshold value is shown as a yellow vertical markerline in the plot window. This markerline can be moved within the plot window by pressing the left mouse button over the markerline and move it to the new position. The new position is then taken as the new threshold value. The computation and display of the threshold value can be toggled on or off. This port is only shown, when the histogram has been computed.

Tindex


Tindex: : ☑ T 90 : 24.3566

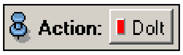
The tindex value defines a reversed threshold value, i.e., a tindex of 90 returns the value where 90% of the data values lie above that value. It is denoted by the reddish markerline in the plot window. The computation and display of the tindex value can be toggled on or off. This port is only shown, when the histogram has been computed.

Material


Material: All

If a label field is connected to the port *Labels* and the input data field is a regular scalar field this port provides the opportunity to compute the histogram for one material only. Otherwise it is hidden.

Action



The *DoIt* button actually computes the histogram and pops up the plot window containing it.

6.61 IlluminatedLines

This module displays line segments of a line set object taking into account ambient, diffuse, and specular illumination. The illumination effect is implemented internally using the same texture mapping technique which is also applied in module DisplayISL.

Connections

Data `[required]`

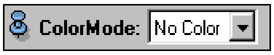
The line set to be visualized.

Colormap `[optional]`

The colormap used to encode the addional data of the line set. If no colormap is connected to this port it is used to set the default diffuse color of the illuminated lines.

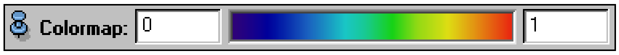
Ports

ColorMode



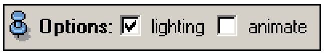
If the line set object contains additional data values per vertex this menu allows you to select one such variable which will be used to lookup vertex colors. If *No Color* has been selected the lines will be displayed in uniform default color. This color may be changed using the default (Constant) color setting of the colormap above.

Colormap



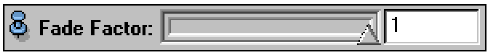
Optional colormap.

Options



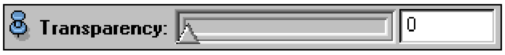
Lighting: Controls illumination of lines. If off, constant colored lines are drawn (flat shading).
Animate: Activates particle-like animation. The animation speed may be controlled via the command `setAnimationSpeed`.

Fade Factor



Values smaller than 1 have the effect that line segments become more and more transparent in backward direction.

Transparency



Base transparency of the line segments.

6.62 InterpolLabels

The computational module *InterpolLabels* interpolates between the slices of a labelfield parallel to the z-axis. The input must be a uniform label field.

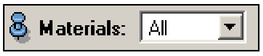
Connections

Data `[required]`

Input data object of type *Uniform label field*.

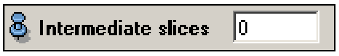
Ports

Material



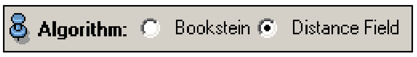
Choose if you want to interpolate all the labels or just a particular one. Note that 'Exterior' cannot be selected.

Intermediate slices



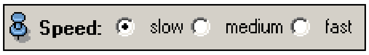
Choose how many intermediate slices should be interpolated. The default is computed such that a voxel is as homogeneous as possible.

Algorithm



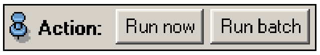
Two methods are implemented: The computation of an implicit function that minimizes a thin-plate-spline functional is used to interpolate between slices (Bookstein).
The other method computes the 2D distance field of each slice and interpolates between them.
The Bookstein method takes much longer than the distance field method. It is recommended to run this computation as a batch job (see below).

Speed



The slow options tries to interpolate every material by the Bookstein method. If this fails, the distance field method is used. The medium option tries to use the Bookstein method only if the dimension of the problem is small enough. The fast option uses only the distance field method.

Action



Pushing these buttons triggers the computation. You can chose between an online computation or a batched computation. The last is advisable for the computation of the Bookstein method. You can also just compute the contours for debugging.

6.63 Interpolate

This module takes two or more data objects as input, e.g. two surfaces or two tetrahedral grids, and computes an output object by linearily interpolating the vertex positions. This can be used to create a smooth transition between the two objects. In addition, it is possible to interpolate data fields attached to the surfaces or grids as well.

Note, that in any case the two input data sets must have the same number of points in order to produce meaningful results. For example, the second input could actually be a copy of the first one with an applied transformation.

Usually, the transition is specified by a parameter u varying between 0 and n-1, where n is the number of input objects. However, it is also possible to associate a physical time with each input. For this, each input must define an entry *Time* in its parameter list which specifies the actual physical time of this input. The time value of any input must be bigger than the time value of the preceeding input. If these conditions are met the toggle *physical time* described below becomes active. A *Time* parameter can be defined interactively using the parameter editor.

Connections

Data `[required]`

First input object to be interpolated. Must either be a surface, a tetrahedral grid, a hexahedral grid, a field defined on one of these objects, or a lattice object like a 3D image.

Input2 `[required]`

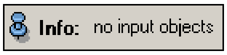
Second input object to be interpolated. This object must be of the same type as the first one.

Input3 `[optional]`

Optional third input object. Once a third input object is connected automatically additional input ports will be created. In this way it is possible to connect an arbitrary number of inputs.

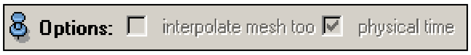
Ports

Info


Info: no input objects

This info port reports the number of input objects, and - if available - the current physical time or the current fractional time step depending on whether the physical time toggle is activated or not.

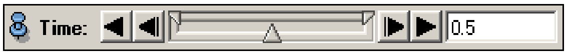
Options


Options: interpolate mesh too [✓ physical time]

If the input is a surface, a tetrahedral grid or a hexahedral grid and if there are additional fields connected to these objects the first toggle allows you to interpolate the data fields as well. If the input itself is such a data field the first toggle allows you to interpolate the mesh too. In this case the toggle's label will be changed appropriately.

The second toggle allows you to activate physical time mode, provided the input objects have a *Time* entry in their parameter list. In this case the time slider (see below) denotes physical time, whereas otherwise it denotes a fractional time step.

Time


Time: 0.5

Interpolation parameter. For t=0 the output will be identical to the first input. For t=1 output will be second input. If physical time mode is active, this port spcifies the physical time for which an output object should be computed.

6.64 InterpolateLabels

This module adds intermediate slices to a label field, thereby interpolating the labels. This is useful e.g., to avoid stair-case effects in data sets with a large slice distance compared to the slice resolution.

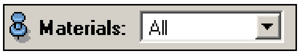
Connections

Data `[required]`

Connects to a label field.

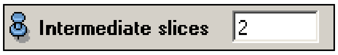
Ports

Materials


Materials: All

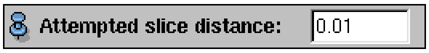
You can interpolate for all or for one material only. Multiple selected materials can be successively added to the result.

Intermediate slices


Intermediate slices 2

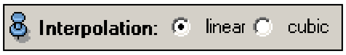
For a uniform label field, this port specifies the number of intermediate slices. e.g., if your input data set had 5 slices and you would specify 2 here, your result would have 13 slices.

Attempted slice distance


Attempted slice distance: 0.01

For a stacked label field, this port specifies the attempted slice distance for the interpolated field. Depending on the actual slice distance in the stacked field, an appropriate number of intermediate slices is inserted.

Interpolation


Interpolation: [✓ linear] cubic

Cubic interpolation will in general generate smoother shapes.

Action

Action: DoIt

Click this button to trigger computation.

6.65 Intersect

The *Intersect* module shows the intersection of a surface or of the boundaries of a tetrahedral grid and a cutting plane. For each triangle intersecting the plane a line segment is drawn. The plane description is taken from a slicing module such as OrthoSlice or ObliqueSlice connected to port Module (any other orange amira module can be used as well). *Intersect* can be chosen from the popup menu of an existing slicing module. It can also be chosen directly from the popup menu of a surface or of a tetrahedral grid. In this case an empty slicing module is created and *Intersect* is automatically attached to it.

The line segments drawn by this module can be exported into a line set data object via the Tcl-command *createLineSet* (see below).

Connections

Data `[required]`

Surface or tetrahedral grid to be intersected.

Module `[required]`

Slicing module defining the cutting plane.

Ports

Line Width

Defines the width of the intersection lines in pixels.

Line Color

This port defines the color of the intersection lines.

Selection

This port allows you to restrict the number of triangles being intersected with the cutting plane. On default all triangles of a surface or all boundary triangles of a tetrahedral grid are considered. With the *Clear* button the list of possible triangles can be resetted. Afterwards triangles adjacent to a particular material can be added again by choosing that material from the port's option menu.

Lines

This port is only available if the module is connected to a tetrahedral grid. It specifies whether only intersections with boundary triangles will be shown or intersections with all triangles of the grid.

Commands

`setColor <r> <g> <b>`

Sets the color of the intersection lines.

`createLineSet`

This command exports the set of currently visible line segments into a *LineSet* data object. The *LineSet* object will be added to the object pool. It then can be written into a file or can be processed further in some other way. Individual line segments will be sorted in such a way that polylines consisting of as many vertices as possible are obtained.

6.66 Isolines

This module computes isolines for an arbitary 3D scalar field on a 2D cutting plane. The plane itself is defined by another module which must be connected to port *Module*. Isolines are computed using two different algorithms depending on the type of the incoming scalar field. If the scalar field is defined on a tetrahedral grid then all tetrahedra intersecting the plane are determined first. The straight isoline segments are generated for these tetrahedra according to the requested threshold values. If the incoming scalar field is not a tetrahedral one then the field is sampled on a regular 2D raster first. Each rectangular cell of the raster is then checked for isolines. Note, that this approach may lead to artifacts if the sampling density is too low.

To execute a demo script illustrating the isoline module click here.

Connections

Data `[required]`

Scalar field for which isolines are to be computed.

Module `[optional]`

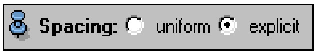
Module defining cutting plane used for isoline computation.

Colormap `[optional]`

Colormap used for pseudo-coloring. If no colormap is connected all isolines have equal color.

Ports

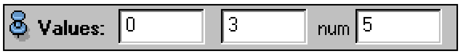
Spacing



Spacing: uniform [✓ explicit]

Control how isoline thresholds are being defined. In *uniform* a certain number of isovalues are distributed equidistantly within a user-defined interval. In *explicit* mode the threshold for each isoline has to be set manually.

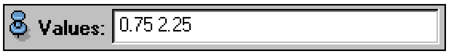
Values



Values: 0 3 num 5

In *uniform* mode this port contains three text fields allowing the user to define the lower and upper bound of the isoline interval as well as the number of isolines being computed in this interval.

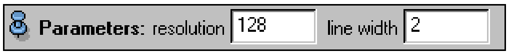
Values



Values: 0.75 2.25

In *explicit* mode an arbitrary number of blank-separated threshold values can be defined in this text field. For each threshold a corresponding isoline is displayed.

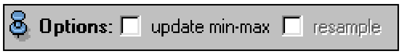
Parameters



Parameters: resolution 128 line width 2

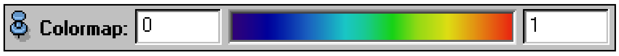
The *resolution* value determines the resolution of the sampling raster used for computing isolines. This parameter is ignored if the incoming scalar field is defined on a tetrahedral grid and the *resample* option is off. The second input of this port determines the width of the isolines in pixels.

Options


Options: ☐ update min-max ☐ resample

Two toggle buttons are provided. If *update min-max* is set then the isoline thresholds are automatically reset to some default values whenever the data range of the incoming scalar field changes. The toggle called *resample* allows you to compute isolines using the resampling approach even if the incoming scalar field is defined on a tetrahedral grid.

Colormap


Colormap: 0 1

Port to select a colormap.

6.67 Isolines (Surface)

This *Isolines* module computes isolines for a scalar field defined on a surface made of triangles. Inside the triangles the scalar field is linearly interpolated if the scalar field contains values for every node. If the field contains values for every surface triangle only, the isolines will follow the triangle edges if the values of the neighboring triangles are appropriate.

Connections

Data `[required]`

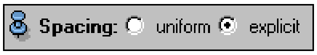
A scalar field defined on a Surface can be connected to this port.

Colormap `[optional]`

A colormap is used to map the values represented by the isolines to a corresponding color. If no colormap is connected, a constant color is used. See also Colormap.

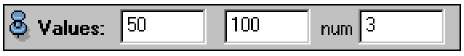
Ports

Spacing


Spacing: ○ uniform [✓ explicit]

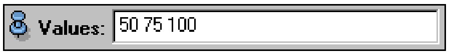
Controls how isoline thresholds are being defined. In *uniform* mode a certain number of isovalues are distributed equidistantly within a user-defined interval. In *explicit* mode the threshold for each isoline can be set manually.

Values


Values: 50 100 num 3

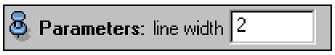
In *uniform* mode this port contains three text fields allowing the user to define the lower and upper bound of the isoline interval as well as the number of isolines being computed in this interval.

Values


Values: 50 75 100

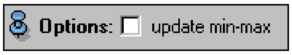
In *explicit* mode an arbitrary number of blank-separated threshold values can be defined in this text field. For each threshold a corresponding isoline is displayed.

Parameters


Parameters: line width 2

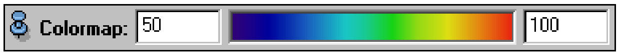
The input of this port determines the width of the isolines in pixels.

Options


Options: ☐ update min-max

If *update min-max* is set then the isoline thresholds are automatically reset to some default values whenever the data range of the incoming scalar field changes.

Colormap


Colormap: 50 100

Port to select a colormap.

6.68 Isosurface (Hexahedra)

This module computes an isosurface within a three-dimensional scalar field defined on an unstructured hexahedral grid.

A second independent scalar field may be connected to the module. This field determines how the isosurface is colored. If no color field is connected to the module the isosurface has constant color.

Connections

Data `[required]`

The scalar field defined on an unstructured hexahedral grid. The *enconding* must be `PER-VERTEX`.

ColorField `[optional]`

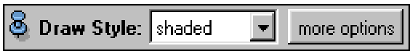
Arbitrary scalar field which is mapped onto the isosurface using pseudo-coloring.

Colormap `[optional]`

The colormap is used for pseudo-coloring the isosurface.

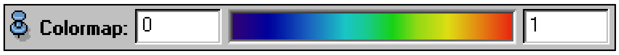
Ports

Draw Style


Draw Style: shaded

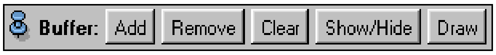
The draw style port is inherited from class *ViewBase*. For a description of this port see there.

Colormap


Colormap: 0 … 1

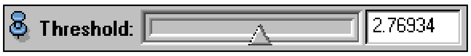
In case a colormap is connected to the isosurface module, this colormap will be shown here. If no colormap is connected to the module the port's default color is used. To change this color, click into the color bar with the left mouse button. This will bring up the color selection dialog. To connect the port to a colormap, use the popup menu under the right mouse button. See also Colormap.

Buffer



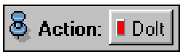
This port must be enabled explicitly with the command `Isosurface showBuffer`. For a detailed description see the ViewBase documentation.

Threshold


Threshold: 2.76934

Determines the value used for isosurface computation. The slider is automatically adjusted to cover the whole range of data values.

Action



Press the *DoIt* button of this port to start the computation of the isosurface.

6.69 Isosurface (Regular)

This module computes an isosurface within a three-dimensional scalar field with regular cartesian coordinates.

Very high-resolution datasets can be downsampled to reduce the number of polygons being produced. In addition, an internal polygon reduction method is provided that merges certain triangles of the original triangulation. This way, the number of triangles can be reduced up to 50%. A second independent scalar field may be connected to the module. This field determines how the isosurface is colored. If no color field is connected to the module the isosurface has constant color.

Connections

Data `[required]`

The scalar field defined on a regular 3-dimensional grid.

ColorField `[optional]`

Arbitrary scalar field which is mapped onto the isosurface using pseudo-coloring.

Colormap `[optional]`

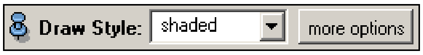
The colormap is used for pseudo-coloring the isosurface.

PointProbe `[optional]`

If this port is connected to a *PointProbe* module as isovalue the value at a certain point within the scalar field will be chosen. For details see PointProbe.

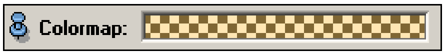
Ports

Draw Style



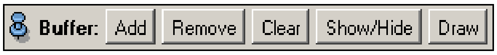
The draw style port is inherited from class *ViewBase*. For a description of this port see there.

Colormap



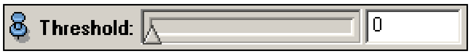
In case a colormap is connected to the isosurface module, this colormap will be shown here. If no colormap is connected to the module the port's default color is used. To change this color, click into the color bar with the left mouse button. This will bring up the color selection dialog. To connect the port to a colormap, use the popup menu under the right mouse button. See also Colormap.

Buffer



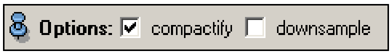
This port must be enabled explicitly with the command `Isosurface showBuffer`. For a detailed description see the ViewBase documentation.

Threshold



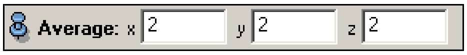
Determines the value used for isosurface computation. The slider is automatically adjusted to cover the whole range of data values. For CT data, use a value of -200 to extract the skin surface or a value of about 150 to extract the bone structures.

Options



If the *compactify* toggle is set, up to 50% less triangles will be produced, but it results in a slightly shifted surface. This port further allows you to enable *downsampling* (see below).

Sample Factor



This port only appears if *downsampling* is enabled. It allows you to specify a sample factor for each of the three main axes. The factors determine how many of the original grid points in each direction will be merged into one.

Action

Press the *DoIt* button of this port to start the computation of the isosurface.

6.70 Isosurface (Tetrahedra)

This module computes an isosurface for a scalar field defined on a three-dimensional tetrahedral grid. The triangulation inside a tetrahedron exactly matches a linearly interpolated field.

Connections

Data `[required]`

The scalar field defined on a tetrahedral grid.

ColorField `[optional]`

See port ColorField in section *Isosurface (Hexahedra)*.

Colormap `[optional]`

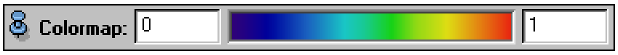
See port Colormap in section *Isosurface (Hexahedra)*.

Ports

Draw Style

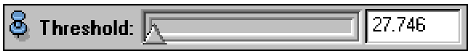
The draw style port is inherited from class *ViewBase*. For a description of this port see there.

Colormap



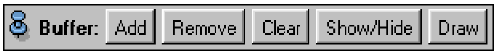
See port Colormap in section *Isosurface (Hexahedra)*.

Threshold



Determines the value used for isosurface computation. The slider is automatically adjusted to cover the whole range of data values. If the threshold value is changed the computation of the new isosurface is immediately invoked.

Buffer



This port must be enabled explicitly with the command `Isosurface showBuffer`. For a detailed description see the ViewBase documentation.

6.71 IvDisplay

The *IvDisplay* module renders an object containing *Inventor* geometry data.

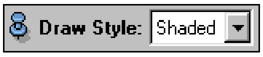
Connections

Data `[required]`

Connect a data object of type *IvData* to this port.

Ports

Draw Style



This port provides a menu to select on of three draw styles to render the *Inventor* geometry.

6.72 IvToSurface

The *IvToSurface* computational module parses the connected *Inventor Scene Graph* (*Inventor* geometry data) and converts it into the surface format. This is done when you push the *DoIt* button. A green icon named *GeometrySurface* representing the generated surface should appear.

If no data object is connected, all geometry currently displayed in the first viewer is converted.

Duplicated points are removed during the conversion. Duplicated points are points with a distance less than the diameter of the bounding box times the given *Relative Tolerance* factor.

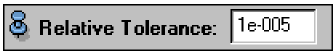
Connections

Data `[required]`

Connect a data object of type *IvData* to this port.

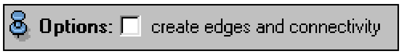
Ports

Relative Tolerance


Relative Tolerance: 1e-005

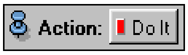
Factor to determine duplicated points.

Options


Options: create edges and connectivity

If this toggle is set the triangle connectivity of the Surface data format is filled.

Action


Action: Do It

The surface is generated if the *DoIt* button is pressed.

6.73 LabelVoxel

The *LabelVoxel* module provides a simple threshold segmentation algorithm applicable to CT or MR image data. The method is also suitable for binary segmentation of other grey level images. Up to five different regions separated by four different thresholds can be extracted. For CT images the four regions *Exterior*, *Fat*, *Muscle*, and *Bone* are predefined. However, the name of these regions as well as the corresponding thresholds may be reset by the user.

In order to find suitable thresholds an image histogram showing the (absolute) number of occurrences of voxel values and the current partitioning of the grey value range into the segments is provided. In this histogram several peaks can be identified more or less clearly, e.g., the ones for fat and muscle. The corresponding threshold or segment boundary should be set at the minima between successive peaks. The histogram window pops up by clicking on the *Histo* action button, adjustments made by moving the value sliders are shown (almost) immediately. You may change the histogram layout (scales, plots of lines and curves, colors) using the *Object Editor* which pops up when you select *Edit Objects* in the *Edit* menu of the histogram window. For more details please refer to the Plot Tool description.

Segmentation starts when you click on the *DoIt* action button. Before doing so several options may be set. These options are described in detail below. The segmentation results are stored as a LabelField data object. You may use the Image Segmentation Editor to further process this object.

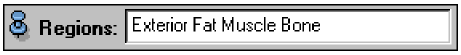
Connections

Data `[required]`

Image data to be segmented.

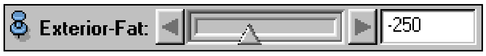
Ports

Regions


Regions: Exterior Fat Muscle Bone

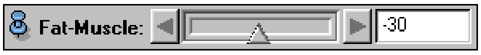
This port lets you specify the names of the different regions to be segmented. Between two and five regions may be specified. The individual region names must be separated by blanks.

Exterior-Fat


Exterior-Fat: -250

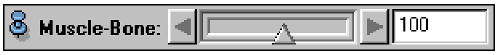
Lets you set the threshold separating the first and second region.

Fat-Muscle


Fat-Muscle: -30

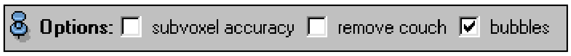
Lets you set the threshold separating the second and third region.

Muscle-Bone


Muscle-Bone: 100

Lets you set the threshold separating the third and fourth region.

Options

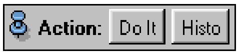

Options: ☐ subvoxel accuracy ☐ remove couch ☑ bubbles

Toggle *subvoxel accuracy* causes certain weights to be computed, indicating the degree of confidence of the assignment of a voxel to a particular region. This information is used by the surface reconstruction algorithm to create smooth boundary surfaces, c.f. the SurfaceGen module. If no weights are present quite blocky surfaces occur. Note, that weights can also be defined using the smoothing filter of the Image Editor.

The second option *remove couch* may be used for medical CT images if parts of the couch the patient is lying on are falsely classified as muscle or bone. In particular, the biggest connected component of voxels not assigned to the first region, i.e. *Exterior*, will be detected. Then all voxels not contained in this component will be assigned to *Exterior*.

Finally, if option *bubbles* is set an algorithm similar to *remove couch* is applied in order to detect bubbles or lung tissue inside the patient. Because of their low intensity values otherwise these regions would be assigned to *Exterior*.

Action



The *DoIt* button triggers the segmentation process. The *Histo* button makes a window pop up showing a histogram for the input image data.

6.74 LandmarkSurfaceWarp

This module deforms a vertex set object using a number of pairs of corresponding points, which are represented by a LandmarkSet. The warping methods and the meaning of the ports are the same as in the LandmarkWarp module. Please refer to its documentation for details.

6.75 LandmarkView

This module displays a landmark set as small spheres, or in case of medical markers shows a specific geometry.

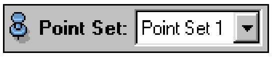
Connections

Data `[required]`

The landmark set to be displayed.

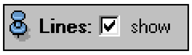
Ports

Point Set


Point Set: Point Set 1

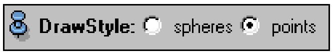
You may select whether only the first, only the second, or all point sets are shown.

Lines


Lines: [✓ show]

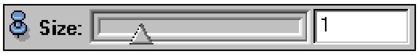
Display lines connecting corresponding points in the first and second set. This option is only present if the input contains two or more point sets.

DrawStyle


DrawStyle: spheres [✓ points]

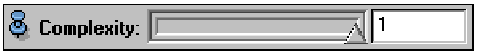
Landmarks can be displayed as spheres/geometry or as points. If the input contains a large number of points (several hundred, or thousands), point style can significantly improve display performance.

Size


Size: 1

Size of displayed spheres or points.

Complexity


Complexity: 1

The larger this value is, the nicer the spheres look, but the lower the rendering speed is.

6.76 LandmarkWarp

This module deforms a 3D uniform scalar field (e.g., image data) using a number of pairs of corresponding points, which are represented by a LandmarkSet.

Three different transformation modes are offered: *Rigid* transforms the input image by applying a global translation and rotation. *Bookstein* uses so called *thin plate splines* proposed by Bookstein. *Flow* uses scattered data interpolation. *Bookstein* mode guarantees that all landmarks will be transformed exactly to their corresponding points. This is not the case for the two others. In all three modes nearest neighbor interpolation is used for resampling.

Connections

Data `[required]`

The LandmarkSet, which defines corresponding points. It must contain at least 2 sets of landmarks.

ImageData `[optional]`

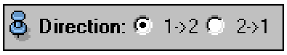
The data set (3D uniform scalar field), which is to be transformed.

Master `[optional]`

If this port is connected to a uniform scalar field, the output field will have the same bounding box and resolution as the master.

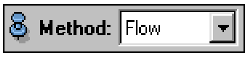
Ports

Direction


Direction: [✓ 1->2] 2->1

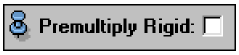
Select whether points in the first set are to be moved towards their corresponding points in the second set, or the other way round.

Method


Method: Flow

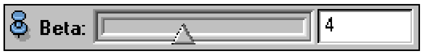
See description above.

Premultiply Rigid


Premultiply Rigid: ☐

Perform a *Rigid* transformation followed by the nonrigid *Flow* transformation.

Beta


Beta: 4

Only available in Flow mode: The larger this value is chosen, the more local and the less smooth

the resulting transformation becomes. On the other hand for beta=0, the transformation is constant. In detail the basis function used in the flow method is
f(p1,p2) = exp(- d(p1,p2) * beta).

Norm

Norm: L2

The norm to measure the distance in the *flow* algorithm. Either the L1 norm $d(p1, p2) = |x1-x2| + |y1-y2| + |y1-y2|$ is used or the L2 norm $d(p1, p2) = sqrt(|x1-x2|^2 + |y1-y2|^2 + |y1-y2|^2)$.

Action

Triggers the computation.

6.77 LegoSurfaceGen

This module reconstructs a surface from a label field like SurfaceGen, but the surface exactly matches the voxel boundaries. Useful for special purpose applications and for development.

Connections

Data `[required]`

Connection to input label field.

Ports

Options

Options: ✓ add border

Same behavior as the corresponding option in the SurfaceGen module.

Action

Trigger computation.

6.78 LineProbe

See Section Data Probing for details.

6.79 LineSetProbe

This module samples an arbitrary 3D field at the vertices of a LineSet. Since line sets can consist of many lines every line of a line set is represented as at least one curve in the plot window according to the dimension of the inspected field resp. to the state of the *Evaluate port* (see below).

The sampled values can also be saved into a copy of the input line set.

Connections

Data `[required]`

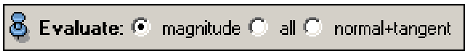
Can be connected to arbitrary 3D fields.

LineSet `[required]`

The connected line set contains the probe line(s) where the samples are taken.

Ports

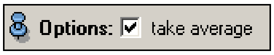
Evaluate



Evaluate: [✓ magnitude] all normal+tangent

If the probe is connected to a vector field, these radiobuttons are shown. If the *magnitude* button is set the magnitude of the vectors is shown in the plot window. With the *normal+tangent Comp.* button set you get the normal and tangential components as two curves. Setting the *all* button shows all components of the vector field as separate curves.

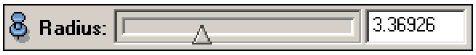
Options



Options: [✓ take average]

The *average* option averages the probe values by taking samples on a disk perpendicular to the sampling points and smoothes the sampled values along the sampling line(s).

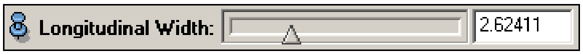
Radius



Radius: 3.36926

The radius of the sampling disk. This slider is only shown if the above average option is chosen.

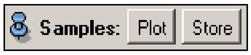
Longitudinal Width



Longitudinal Width: 2.62411

The width determines how many sampling values are used for smoothing. This slider is only shown if the above average option is chosen.

Samples



If the *Show* button is pressed a plot window appears where the sampled values are plotted against the length of the probe line(s). *Note:* There will be only one plot window regardless of how many LineSet Probe modules there are in your setup. Every lineset probe is represented in that plot window by at least one curve bearing the name of the corresponding module.

Pressing the *Store* button stores the sampled data in a copy of the input lineset.

6.80 LineSetView

This module visualizes data objects of type LineSet. Individual lines may be displayed in wireframe mode. Alternatively, simple shapes like triangles or squares may be extruded along the lines. In this way true three-dimensional tubes are obtained. Additional features of *LineSetView* are pseudo-coloring and the display of little spheres at each line vertex.

Connections

Data `[required]`

The line set to be displayed.

ROI `[optional]`

Optional connection to an object providing a region-of-interset. Only line segments inside this region will be visualized.

Colormap `[optional]`

Colormap used for pseudo-coloring.

Ports

Shape

Shape: Lines

Determines how the lines will be displayed. If *Lines* is selected a simple wireframe model will be created. The other menu entries denote 2D objects which will be extruded along the lines in order to obtain three-dimensional tubes.

ScaleMode

ScaleMode: Constant

If the line set object contains additional data values per vertex this menu allows you to select one such variable which will be used to scale the diameter of the three-dimensional tubes. If no additional data values are present only *Constant* scaling will be available.

ScaleFactor

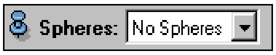
ScaleFactor: 0.1

Additional factor used to adjust the diameter of three-dimensional tubes. This factor has no effect if shape has been set to *Lines*.

ColorMode

If the line set object contains additional data values per vertex this menu allows you to select one such variable which will be used to lookup vertex colors. If *No Color* has been selected the lines or tubes will be displayed in uniform default color. This color may be changed using the command `setLineColor`.

Spheres



On default *No Spheres* is selected. Changing this selection causes a sphere to be displayed at each line vertex. If the line set object contains additional data values per vertex there will be one entry for each data variable. Selecting such an entry causes the sphere radii to be scaled according to the selected variable.

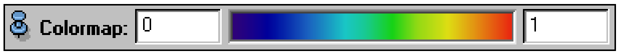
SphereScale

Additional factor used to adjust the size of the spheres. This factor has no effect if *No Spheres* is selected.

SphereColor

Like port *Color Mode*, but affects the sphere colors instead of the line colors.

Colormap



Optional colormap.

Commands

`setLineColor <color>`

Lets you adjust the default line color which is used if color mode is set to *No Color*. The color may be specified as an RGB color triple or as a X11 color name.

`setSphereColor <color>`

Lets you adjust the default sphere color which is used if sphere color mode is set to *No Color*. The color may be specified as an RGB color triple or as a X11 color name.

6.81 LineStreaks

This module takes a surface and randomly distributes a number of short line segments on it. The line segments are computed as field lines of a 3D vector field projected onto the surface or as field lines of a vector field directly defined on that surface. The latter can for example be produced by the GetCurvature module in *Max Direction* mode. Instead of being visualized directly the resulting line streaks will be stored in a LineSet data object.

Connections

Data `[required]`

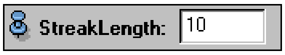
The surface where the streaks will be put on.

VectorField `[required]`

The vector field from which the streaks will be computed. May be either a surface vector field or a 3D vector field.

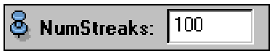
Ports

StreakLength


StreakLength: 10

Relative length of the field line segments. The length is scaled by the length of the diagonal of the surface's bounding box.

NumStreaks


NumStreaks: 100

Number of line streaks to be computed. The number of streaks per surface area will be approximately constant.

Action

Starts computation.

Commands

`setResolution <res>`

Allows you to adjust the resolution of the line segments. The higher the value the smaller the point spacing. The default value is 256, resulting in a point distance of 1/256 of the length of the surface's bounding box.

6.82 MagAndPhase

The *MagAndPhase* module connects to a UniformComplexScalarField. For each point in space it computes the phase and the squared magnitude and creates a UniformColorField from this information. The squared magnitude is used to determine the alpha value (i.e. the opacity) of a voxel. The phase determines the color using an arbitrary colormap. This module can be used to visualize quantum mechanical wave functions.

Connections

Data `[required]`

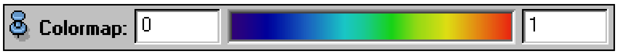
Complex scalar field defined on a regular grid.

Colormap `[optional]`

A colormap that is used to visualize the phase. In order to avoid that phase values are being clipped the range of the colormap should extend from $-\pi$ to $+\pi$.

Ports

Colormap



Phase colormap input port.

Action

Pushing this button triggers the computation.

6.83 Magnitude

The *Magnitude* module computes the magnitude of the vectors of a vector field, i.e.,

$$|V| = \sqrt{V_x^2 + V_y^2 + V_z^2}$$

These vectors might be located on regular or on irregular grids. Regular or complex vector fields are typically defined by uniform, stacked, rectilinear, or curvilinear coordinates. Vectors located on vertices of triangular surface meshes or tetrahedral grids are defined by irregular coordinates.

The result of *Magnitude* is a field of scalars, located at positions according to the underlying grid of the input data.

Connections

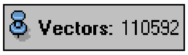
Data `[required]`

Field or complex vector field defined on a regular grid (*Lattice3*).

Vector field on an irregular grid, either specified by nodes of a surface, a tetrahedral or a hexahedral grid (*SurfaceField, TetraData, HexaData*).

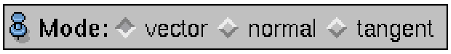
Ports

Number of Vectors



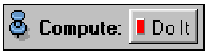
Shows number of input vectors and the vector dimension for computing the magnitude field.

Mode



Choose whether to compute the magnitude of the vector, its normal or tangential component relative to the primary surface normals. This port is only shown if a surface vectorfield has been connected.

Compute



Pushing this button triggers the computation.

6.84 Measuring

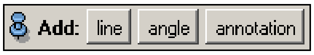
This module gives access to some three dimensional measuring tools. The idea is to click directly into the rendered scene and draw. The tools only work on the displayed objects.

You can access the tools via the menu *View - Measuring* or via the measuring tool button on the left side of the viewer - Click on it and draw a line into the scene (click and drag). If you click onto the button and keep the button pressed a popup menu will appear that let you change the tool (line, angle, annotation). If you want to change the position of a tool click onto a relevant point and drag it to a new location.

Every tool has some attributes (color, width, locked, visible, render in front, text) that are accessible by the *Tools* port of the *Measuring* object which was created in the object pool.

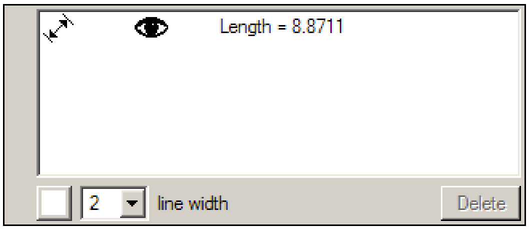
Ports

Add



You can add a line, an angle or an annotation.

Tools



All the tools are displayed as lines in the toolbox. The columns have the following meaning (from left to right):

- *locked*: If you click in the first column a key appears. The tool is locked and can't be modified in the viewer. Click again to unlock the tool. Click and drag over more rows to lock/unlock more than one tool.
- *visible*: If you click in the second column the eye disappears and the tool gets invisible in the viewer. Click again to toggle the visibility.
- *render in front*: Click in the third column. An icon appears which indicates that the tool is always rendered in front of the scene and no longer hidden by objects that are nearer to the camera. It also is always pickable.
- *type*: The fourth column indicates the type of the tool.
- *text*: The fifth column is an annotation to distinguish the tools. Double click to change the text. Annotation tools display the text in the viewer.

If you select a tool it gets highlighted by drawing red squares at the relevant points in the viewer. You can select more than one tool at the same time. With the color button and the width dropdown menu you can change the selected tools.

6.85 Merge

This module works on any 3-dimensional field on rectilinear coordinates and merges the input data by interpolation.

Merging

Proceed as follows: Attach a *Merge* module to an input field. For additional inputs, right-click on the small rectangle on the *Merge* module, select an input item and drag a connection to a data icon. Input fields can have an arbitrary transformation in which case the result field will have an axis aligned that spans the bounding boxes of the input fields. The voxel size of the result field is determined by the first input. At locations within the resulting volume where none of the input fields are defined voxels are initialized with zeros.

For image data, you have the choice among various *interpolation* methods:

- *Nearest Neighbor* chooses for every new voxel the average value of the input voxels nearest to it.
- *Standard* linearily interpolates between the surrounding voxels.
- *Lanczos* is the slowest but most accurate method and approximates a low pass filter. If you have time and want to get the best result, use this one.

For label fields, this port is hidden and always *Nearest Neighbor* is used.

Choose an additional option and press DoIt.

Merging in progress

While merging, the progress bar at the bottom of the work area indicates the percentage of merging that is done. You may cancel the merging any time by pressing the stop button on the right of the progress bar.

Connections

Data `[required]`

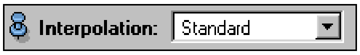
The first data field to be merged.

Lattice* `[optional]`

Additional input data.

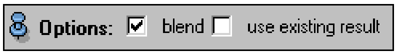
Ports

Interpolation


Interpolation: Standard

The interpolation method that will be used.

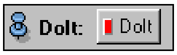
Options


Options: ☑ blend ☐ use existing result

Option *blend* controlls the merging process. If this toggle is disabled, a voxel value of the first data field is used if such exists at a given location. Otherwise, a result value will be calculated from the additional input data. If this toggle is enabled, result values are always interpolated.

Use the second option *use existing result* to overwrite an existing result. If *use existing result* is not set, a new result object is generated.

DoIt



Proceed.

6.86 MovieMaker

This module can be used to create an MPEG 1 movie or an animation consisting of a series of 2D images in TIFF, JPEG, or PNG format. The module requires a time object as input. When creating the movie the module iteratively sets the time value, fires the network so that other objects can adjust themselves, and takes a snapshot of the viewer window. The snapshots are appended on-the-fly to the specified MPEG file. In order to create an initial time object that a movie maker module can be attached to, choose *Time* from the main window's *Create* menu.

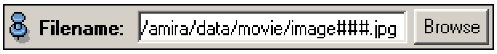
Connections

Data `[required]`

Connection to a time object. It is not required that the connected time is defined in seconds or in any particular unit. Instead the number of frames and the frame rate are specified (see below). For example, if 100 frames are to be recorded, the connected time is moved from its minimum values to its maximum values in 100 steps.

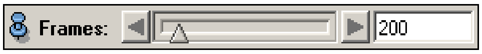
Ports

Filename



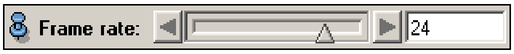
The name of the resulting movie or image file(s). In order to write a series of TIFF, JPEG, or PNG files, specify a filename with the suffix *.tif*, *.jpg*, or *.png*. To create an MPEG movie specify a file with the extension *.mpg* . On Microsoft Windows it is also possible to specify *.avi* for creating AVI movies. The filename should contain a series of hash marks *####*. The hash marks will be replaced by the actual frame number. You can also choose the output format via the file dialog which pops up when the *Browse* button is pressed. If no filename is specified, no movie can be created.

Frames



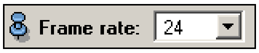
The number of frames to be recorded. This value has nothing to do with the range of the connected time object. If the input time is already defined in seconds and if you don't want any scaling, then the number of frames should be set to the number of seconds of the time object times the desired frame rate.

Frame rate


Frame rate: 24

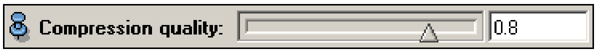
Specifies the desired frame rate. It depends on the particular movie player if the specified frame rate is actually achieved when playing the Movie.

Frame rate


Frame rate: 24

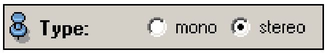
Some movie file formats are only capable of storing movies in specific frame rates. In that case this port is shown with all possible rates of the requested format.

Compression quality


Compression quality: 0.8

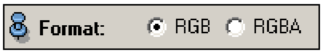
This port is shown if a lossy output file format was chosen. It allows you the adjustment of the degree of compression. Small values indicate high compression and low quality, high values indicate low compression and high quality.

Type


Type: mono [✓ stereo]

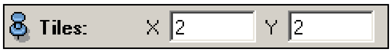
The moviemaker is able to create monoscopic and stereoscopic movies. Setup the movie type here.

Format


Format: [✓ RGB] RGBA

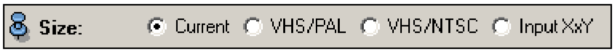
This port specifies the pixel format of the screenshots. If *RGBA* is chosen, the screenschots are generated with an 8 bit alpha channel blending out the scene background. Using the alpha feature has the advantage of smaller image files on complex scene backgrounds (cradient). The second advantage of movies containing an alpha channel is that one can choose another scene background during movie playback. Note that not all image or movie file formats are capable of storing a fourth channel. To create alpha image files choose the suffix *.tif* since TIFF is capable of storing alpha channels. The disadvantages of alpha movies are slower playback speed on some architectures and larger data files on non-complex backgrounds (uniform).

Tiles


Tiles: X 2 Y 2

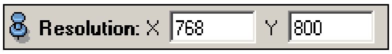
In order to create movies with a higher resolution than the screen, specify the number of tiles to render in each direction. Since the anti-aliasing feature of the former Movie Maker is gone, you can create high resolution single image files and downsample them in a batch to get smooth edges.

Size


Size: [✓ Current] VHS/PAL VHS/NTSC Input XxY

Radio box allowing you to adjust the size of the recorded images. If *current* is specified, the current viewer size adjusted to multiples of 16 pixels will be used. *VHS/PAL* means 720 by 576 pixels. *VHS/NTSC* means 640 by 480 pixels. Finally, if *Input XxY* is specified, the size can be specified explicitly.

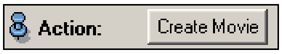
Resolution



Resolution: X 768 Y 800

This port will only be shown if *Input XxY* is chosen for the *Size* port. It allows you to set the image size of the resulting movie.

Action



Starts to create the movie. Since the movie is generated from snapshots, the viewer window must not be hidden by other windows while the movie is beeing created. Furthermore the Movie data with the module connected to its master connection port is altered with the settings of the newly created movie. If there is not such a Movie data object connected to the module, a new one is created.

6.87 MoviePlayer

This module allows you to play back a sequence of single images, a movie. These images can be stored as separate files in a 2D image format supported by *Amira* (for example *JPEG*, *TIFF*, *PNG*, or *PPM*) or they may packed together in one or more movie data files (moviename.amovstream). To create movies using *Amira*, use the MovieMaker.

6.87.1 The Amira Movie Format

This module is capable of playing back a sequence of images. This sequence is loaded from one or more so called streams. Each stream consists of an image sequence. During playback this module retrieves images from all streams and meshes them together for the final image sequence which gets then rendered to the screen. Imagine there are N streams. The first image comes from the first stream, image 2 from stream 2, image N from stream N, image N+1 again from stream 1, image N+2 from stream 2 and so on. (Expert note: this behavior can be changed by editing the *Amira* movie info file). After specifying the streams, the module needs to know what type of movie this image sequence forms, e.g., mono, stereo, stereo interlaced and so on. The structure of an *Amira* movie, i.e. the number and specification of streams, the type and the preferred compression type, is stored in a *Amira* movie info file (moviename.amov), which is a human readable and editable text file. In *Amira*, movies are specified in separate data objects connectable to this module. See Movie for details.

6.87.2 Optimized Amira Movies

This module is capable of converting a movie to an optimized format, i.e. a format that comes up with faster playback speed and sometimes smaller but mostly bigger but fewer data files. To perform this conversion, connect a Movie to the input connection. That's the source movie. Then connect a Movie to the Result connection port. That will be the destination movie. In the stream specification for that destination movie only *Amira* movie data files are allowed. The movie conversion is not able to write into single image files (which would not make sense anyway). After this, select *convert (overwrite)* for overwrite mode or *convert (append)* for append mode. To perform the conversion press one of the play buttons. Seeking does not affect the destination movie, so it is possible to seek to the part of interest in the source movie and start the conversion from there.

6.87.3 Which movie or image format should I use ?

To maximize the playback speed:

- Store many images into one or a few big *Amira* movie data files by performing the conversion process. This avoids the file searching overhead especially if there are thousands of single image files in a single directory.
- If the target system has limited CPU bandwidth, avoid using CPU-intensive operations like the post compression feature or a CPU-intensive image reader. To find out which image reader is the fastest you have to experiment, since it depends on the individual hardware and software configuration. If the systems *OpenGL* implements the extension *GL_ARB_texture_compression*, you can compress the images using *OpenGL* texture compression. This has the advantage, that the image size is reduced by factor 4 to 6 without the need for decompression by the CPU during playback, since the decompression is performed by the GPU.
- If the target sytem has limited harddisk bandwidth, compress the data by using high compressing single image formats like JPEG or the post compression feature in combination with the *OpenGL* texture compression feature if available. If the bandwith of a single harddisk limits the playback speed, but there are more harddisks available (like on many PCs), store the movie in multiple streams (described above), each of them stored on a different harddisk. On systems with RAID the movie should be stored in a single stream, since the RAID should distribute the movie data file over many disks, which should already increase the retrieval bandwidth. Using more than one stream jams the RAID strategy used for fast retrival of large disc files, which results in a slowly and bumpy playback.

To maximize the playback image quality:

- Choose a loss-free image format during movie creation outside of this module and also during movie conversion with this module. RGB(A) and gzip postcompression are loss-free. *OpenGL* texture compression is lossy.

Known Issues

- When playing single images with readers that are not thread safe, *Amira* may crash. In this case one can set the the value in the movies MaxThreads-port to 1 which sets the maximum number of reader-threads to one, what makes the playback speed slow but allows a conversion to the optimized format. On most platforms the JPG-reader should be thread-save. The BMP-reader is known to be not thread-save.

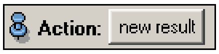
Connections

Data

Connect the Movie the module is going to play or convert.

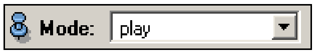
Ports

Action



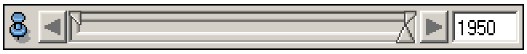
Press this button to create a new Movie data object connected to this player via the movie's Master port. Filename and type of this newly created movie are inititialized according to the properties of the input movie. The default number of streams is one. Alter the settings within the input movie object to match your needs before converting data into that movie.

Mode



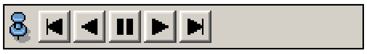
Set this to specify the action the module should perform. If set to *play*, the module will play the movie connected to the data connection port. If set to *play result*, the module will play the Movie connected with its *Master* port. If set to *convert (overwrite)*, the module performs the movie conversion from the source Movie to the destination Movie. The old contents of the destination Movie are deleted first. If set to *convert (append)*, the module appends the contents of the source Movie to the destination Movie.

Position



This slider displays the current position in the movie during playback. By pulling the slider it's possible to seek in the movie. With the two upper adjustable sub-range buttons one can limit the movie presentation to a smaller scene of interest. With the two outside buttons one can step through the movie, image by image, making it possible to use this module to present a series of single high resolution slides. By specifying an integer value in the text field one can jump directly to the specified frame. Note that the frame count begins with zero.

Playback



Push these buttons to seek to the beginning of the scene of interest, to its end, to play the movie forward or backward, and to pause or unpause playback.

Pixel format

Pixel format: RGB(A) [✓ GL-compress]

This setting gets evaluated during the movie conversion process. If set to *RGB(A)* the images are stored loss-free to the amira movie data files, 24 bits per pixel if RGB and 32 bits if RGBA. Note the alpha blending feature of this module which is enabled if the input image contains an alpha channel (amira is capable of creating screenshots with an alpha channel). If *GL-compress* is enabled and the systems *OpenGL* is capable of compressing textures, this module compresses the images using *OpenGL*. Note that this kind of compression is lossy. If it is not available on the current system, this option is disabled.

Post compression

Post compression: ☐ gzip

This setting is evaluated during the movie conversion process. Specify this for the destination movie. If enabled, the MoviePlayer module compresses the image, independent of the selected pixel format, with *GZIP*. The result is smaller movie data files that need more CPU power during playback. Note that only parts of the movie stream file are going to be compressed, so it is not possible to compress the whole file with an external zip encoder application.

Chunk size

Chunk size: MB 0

This setting is evaluated during the movie conversion process. Sometimes the resulting amira movie data file gets too large. With this option one can request the module to split this file into pieces during creation. The fragments are named filename.amovstream, filename.00000001, filename.00000002, ..., filename.0000000N. A value of 0 disables the split feature. A value greater 0 specifies the maximum fragment size in megabytes (each 1,048,576 bytes). One can split an amira movie stream file with external tools into fragments with arbitrary sizes. If the naming scheme matches the one mentioned above, the module will play it correctly. Note that there is an amira movie index file maintained by this module, which holds file numbers and byte offsets for every image on this stream. After manual splits or to force a recreation of this index file, simply remove it. At the next attempt to play or seek this stream, this module will scan the amira movie stream file and recreate the index file.

6.88 ObliqueSlice

The *ObliqueSlice* module lets you display arbitrarily oriented slices through a 3D scalar field of any type, as well as through an RGBA color field or a 3D multi-channel field. In the medical context such slices are known as multi-planar reconstructions (MPR).

The module is derived from ArbitraryCut. See the documentation of the base class for details about how to adjust position and orientation of a slice. Like in the OrthoSlice module three different methods are supported to map scalar values to screen colors, namely linear mapping, monochrome mapping based on adaptive histogram equalization, and pseudo-coloring.

Connections

Data `[required]`

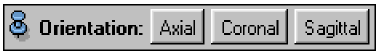
The 3D field to be visualized. 3D scalar fields, RGBA color fields, or 3D multi-channel fields are supported. The coordinate type of the input data doesn't matter. The data will be evaluated using the field's native interpolation method (usually trilinear interpolation).

Colormap `[optional]`

The colormap used to map data values to colors. This port is ignored when linear or histogram equalized mapping is selected, or when the module is connected to an RGBA color field or to a 3D multi-channel field.

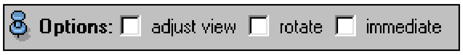
Ports

Orientation



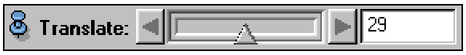
This port provides three buttons for resetting the slice orientation. *Axial* slices are perpendicular to the z-axis, *coronal* slices are perpendicular to the y-axis, and *sagittal* slices are perpendicular to the x-axis.

Options



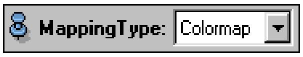
If the *adjust view* toggle is set, the camera of the main viewer is reset each time a new slice orientation is selected. With the *rotate* toggle you can switch the rotate handle for the cutting plane on and off. If the *immediate* toggle is set the slice is updated every time you drag it with the mouse in the 3D viewer. Otherwise only the bounding box of the cutting plane is moved and the update takes place when the mouse button is released.

Translate



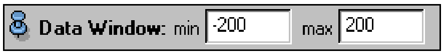
This slider allows you to select different slices. The slices may also be picked with the mouse and dragged directly in the 3D viewer.

Mapping Type



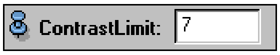
This option menu controls how scalar values are mapped to screen colors. In case of a *linear* mapping a user-defined data window is mapped linearly to black and white. If *historam* is selected, an adaptive histogram equlization technique is applied. This method tries to show all features of the data, even if a wide range of values is covered. Finally, *colormap* can be used to activate pseudo-coloring. The port is hidden if the the module is connected to a color field or a multi-channel field.

Data Window


Data Window: min -200 max 200

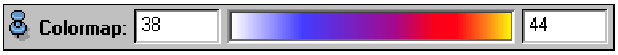
This port is displayed if linear mapping is selected. It allows you to restrict the range of visible data values. Values below the lower bound are mapped to black, while values above the upper bound are mapped to white.

Contrast Limit


ContrastLimit: 7

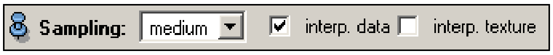
This port is only visible if *histogram* equalization is selected. The number determines the contrast of the resulting image. The higher the value, the more contrast is contained in the resulting image.

Colormap


Colormap: 38 44

This port is displayed if *colormap* is selected. Choose a colormap to map data to colors.

Sampling

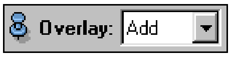

Sampling: medium [✓] interp. data [ ] interp. texture

This port controls the way how oblique slices are reconstructed. First, an option menu listing different sampling resolutions is provided. The four choices *coarse*, *medium*, *fine*, and *finest* correspond to an internal resolution of the underlying texture map of 128, 256, 512, and 1024 square pixels, respectively.

The next toggle, denoted *interpolate data*, will only be active if the scalar field to be visualized is defined on a uniform grid. In this case, if the toggle is off nearest neighbour interpolation is used. If it is on or if the data is not defined on a uniform grid, then the field's native interpolation method is used, e.g., trilinear interpolation on regular grids.

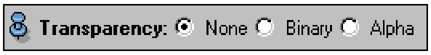
Finally, the toggle labeled *interpolate texture* controls how the slice is texture-mapped by the underlying OpenGL-driver. If the toggle is off, nearest-neighbour sampling is used. Otherwise, bilinear filtering is applied.

Overlay


Overlay: Add

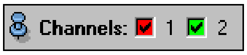
This port determines how optional *ColorWash* modules are mapped onto this slice.

Transparency


Transparency: [✓ None] Binary Alpha

This radio box port determines the transparency of the slice. The option is only available if the module is connected to an RGBA color field. *None* means that the slice is fully opaque. *Binary* means that black parts are fully transparent while other parts are opaque. *Alpha* means that opacity is taken as is.

Channels


Channels: ✓ 1 ✓ 2

This port is shown if the module is attached to a 3D multi-channel object. It allows you to toggle individual channels on or off. Each channel is mapped using a linear intensity ramp. The data window for each channel can be adjusted in the multi-channel object itself.

Commands

All commands described for ArbitraryCut can be applied to *ObliqueSlice* as well.

6.89 OrthoSlice

The *OrthoSlice* module is an important tool for visualizing scalar data fields defined on uniform Cartesian grids, e.g., 3D image volumes. Such data are visualized by extracting an arbitrary axial, frontal, or sagittal slice out of the volume. The data values can be mapped to colors or grey levels by one of three mapping methods. The most simple mapping technique uses a linear grey ramp together with two threshold values. This range determines which data values are mapped to black and which are mapped to white. Alternatively, a contrast limited histogram equalization technique may be applied. With this method, there is no unique correspondence between grey levels and data values any more. The method tries to visualize all features of an image. As a third mapping method, an external color map can be used in *OrthoSlice*.

The *OrthoSlice* module is also capable of extracting slices of an RGBA color field or of a 3D multi-channel field. In these cases the slices are displayed as is and no mapping method need to be chosen.

Connections

Data `[required]`

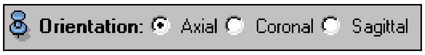
The 3D field to be visualized. Currently, regular scalar fields, multi-channel fields, and RGBA color fields with uniform or stacked coordinates are supported.

Colormap `[optional]`

Optional color map used to map scalar data to colors. This port is hidden when *linear* or *histogram* mapping is selected. See also Colormap.

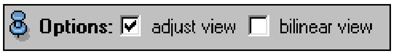
Ports

Orientation


Orientation: [✓ Axial] Coronal Sagittal

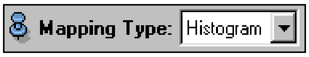
This port provides three buttons for resetting the slice orientation. *Axial* slices are perpendicular to the z-axis, *coronal* slices are perpendicular to the y-axis, and *sagittal* slices are perpendicular to the x-axis.

Options


Options: [✓ adjust view] bilinear view

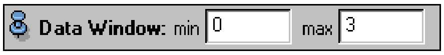
If the *adjust view* toggle is set, the camera of the main viewer is reset each time a new slice orientation is selected. The *bilinear* toggle determines how color is interpolated within a slice. By default, the toggle is off and constant interpolation is used. With constant interpolation individual pixels become visible. Bilinear interpolation often produces somewhat blurry results.

Mapping Type


Mapping Type: Histogram

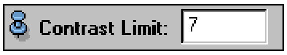
This option menu lets you select between the three different mapping methods, namely a *linear* grey ramp, *histogram* equalization, and *colormap* mode. The port is not available if the *OrthoSlice* module is connected to an RGBA color field or to a 3D multi-channel field.

Data Window


Data Window: min 0 max 3

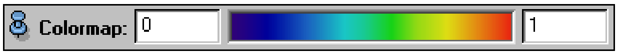
This port is displayed if *linear* mapping is selected. It allows you to restrict the range of visible data values. Values below the lower bound are mapped to black, values above the upper bound are mapped to white. In order to quickly change the data window a ContrastControl module can be attached to the *OrthoSlice*.

Contrast Limit


Contrast Limit: 7

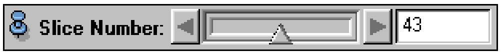
This port is displayed if *histogram* equalization is selected. The number determines the contrast of the resulting image. The higher the value, the more contrast is contained in the resulting image. A value of zero means that contrast will not be limited at all.

Colormap


Colormap: 0 1

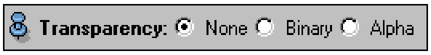
Choose colormap if *mapping type* is set to *colormap*.

Slice Number


Slice Number: 43

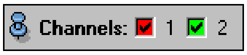
This slider allows you to select different slices. The slices may also be picked with the mouse and dragged directly in the 3D viewer.

Transparency


Transparency: [✓ None] Binary Alpha

This radio box port determines the transparency of the slice. *None* means that the slices are fully opaque. *Binary* means that black parts are fully transparent while other parts are opaque. *Alpha* means that opacity is proportional to luminance. If a colormap is used for visualization opacity values are taken from there.

Channels


Channels: [✓ 1] [✓ 2]

This port is shown if the module is attached to a 3D multi-channel object. It allows you to toggle individual channels on or off. Each channel is mapped using a linear intensity ramp. The data window for each channel can be adjusted in the multi-channel object itself.

Commands

`frame {0|1}`
This command lets you turn on or off the orange frame indicating the intersection of the plane with the bounding box of the 3D field.

`setFrameColor <color>`
Lets you change the color of the plane's frame.

`setFrameWidth <width>`
Lets you change the width of the plane's frame in pixels.

6.90 Parametric Surface

This module can be used to define and animate arbitrary parametric surfaces in two, three dimensions. There are some pre-defined surfaces like the Moebius strip, Klein bottle and Random mapping to illustrate the use of the text fields that define expressions for `X`, `Y`, and `Z`.

The expressions may depend on `u` and `v` which are Cartesian coordinates of the plane. They can also depend on `theta` and `r` which are their counterparts in polar coordinates. A separate spherical coordinate system is accessible by the variables `lambda` and `phi`. The variable `t` is linked to the time port and can be used to perform for example linear interpolations between two mappings like:

$$(t-1) * (\texttt{expression1}) + t * (\texttt{expression2}). \tag{6.3}$$

Surfaces can be exported by the `Draw Style` port (⇒ `more options` ⇒ `Create surface`).

Connections

Data `[optional]`

If a regular scalar field is supplied, the generated mesh will be scaled and shifted according to its bounding box. Using the transform editor of the data is a convenient way of scaling and shifting the generated mesh.

Colormap `[optional]`

The colormap is used to calculate the color of the mesh based on the expression in the Colorport (initially `r`, the radial distance of the u-v mesh point from its mean position).

Time `[optional]`

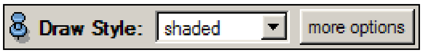
An additional variable which can be used in all the expression fields. It can be used to generate animations.

Cluster `[optional]`

If you connect to this module a Cluster object (points in space), the module will generate a second cluster object and apply the transformation in the expression fields to its `x` and `y` coordinates. This may be useful to visualize the u/v plane as a point cloud.

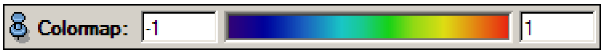
Ports

Draw Style


Draw Style: shaded

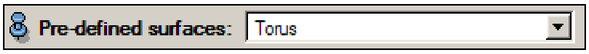
Select a specific draw style for the mesh currently on display. For higher resolution meshes usually shaded works best.

Colormap


Colormap: -1 1

This colormap will be used to map colors per vertex using by the values in the Color port.

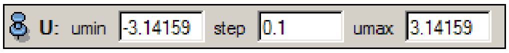
Pre-defined surfaces


Pre-defined surfaces: Torus

The pre-defined surfaces are some commonly used to illustrate analytically defined meshes. They are mainly here to illustrate the use of the different variables in the Colorports.

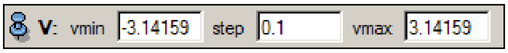
As a last entry you will find a Random generator which will generate random entries for the x, y and z ports, i.e. a random mapping. Currently there is a maximum depth of 10 for the randomly generated expressions.

U


U: umin -3.14159 step 0.1 umax 3.14159

This port describes the u resolution of the initial mesh (before the transformation was applied). Its values code the minimum value of u (step size) and the maximum value of u. Lowering the step-size will directly increase the resolution of the mesh.

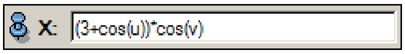
V



V: vmin -3.14159 step 0.1 vmax 3.14159

This port works the same way as the 6.90 port for the v parameter of the initial mesh.

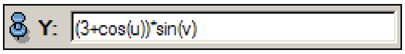
X



X: (3+cos(u))*cos(v)

Enter an expression to be evaluated in order to describe the x-position of a mesh in 3D space. Use the 6.90 port to see some useful settings for this port.

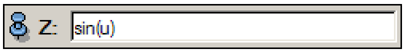
Y



Y: (3+cos(u))*sin(v)

Enter an expression to be evaluated in order to describe the y-position of a mesh in 3D space. Use the 6.90 port to see some useful settings for this port.

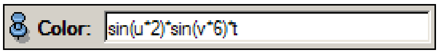
Z



Z: sin(u)

Enter an expression to be evaluated in order to describe the z-position of a mesh in 3D space. Use the 6.90 port to see some useful settings for this port.

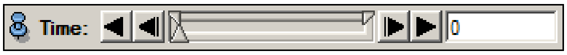
Color



Color: sin(u*2)*sin(v*6)*t

Enter an expression to be evaluated in order to describe the vertex color of the mesh. All variables of the other fields can be used in this field. By default it contains r which is the radial distance from the center position of the mesh.

Time



Time: 0

This slider is directly linked to the expression t which can be used in all x, y, z and Color ports.

DoIt

Press this button if you want to do an animation because you will need to re-compute the mesh every time t changes.

6.91 PlanarLIC

This module intersects an arbitrary 3D vector field and visualizes its directional structure in the cutting plane using a technique called *line integral convolution* (LIC). The LIC algorithm works by convolving a random noise image along the projected field lines of the incoming vector field using a piecewise-linear hat filter. The synthesized texture clearly reveals the directional structure of the vector field inside the cutting plane. As long as no valid LIC texture has been computed a default checkerboard pattern is displayed instead.

PlanarLIC is derived from ArbitraryCut. See the documentation of this module for details on how to adjust the position and orientation of the cutting plane. Click here in order to execute a script demonstrating the use of the *PlanarLIC* module.

Connections

Data `[required]`

Vector field to be visualized.

ColorField `[optional]`

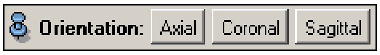
A scalar field which may be used for pseudo-coloring.

Colormap `[optional]`

Colormap used for pseudo-coloring. If no colormap is connected the default color of the colormap port will be used. The port is hidden if pseudo-color mode is set to *none*.

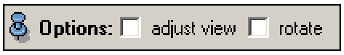
Ports

Orientation



This port provides three buttons for resetting the slice orientation. *Axial* slices are perpendicular to the z-axis, *coronal* slices are perpendicular to the y-axis, and *sagittal* slices are perpendicular to the x-axis.

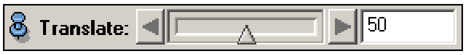
Options



If toggle *adjust view* is active then the camera of the 3D viewer will be reset whenever one of the orientation buttons is clicked.

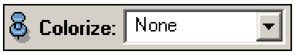
If the *rotate* toggle is active then a virtual trackball is displayed. By picking and dragging the trackball you may change the orientation of the plane. Remember that the viewer must be in interaction mode in order to do so. The ESC-key inside the viewer window toggles between navigation mode and interaction mode. The trackball of the last active ArbitraryCut can also be turned on and off by pressing the TAB-key inside the viewer window.

Translate



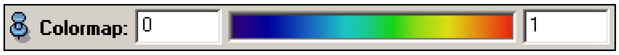
This port lets you translate the plane along its normal direction.

Colorize



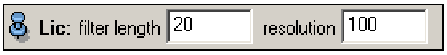
An option menu allowing to select different pseudo-color modes. If item *none* is selected the LIC texture will be displayed in greyscale only. If *magnitude* is selected each pixel of the LIC texture will be colored according to vector magnitude at this point. If *normal component* is selected then color denotes the signed length of the vector component perpendicular to the cutting plane. This length will be positive if the vector points upwards or negative if the vector points downwards. If *parallel component* is selected then color denotes the length of the vector component tangential to the plane. This length will always be greater or equal than zero. Finally, if *color field* is selected and if a scalar field is connected to port *ColorField* the external scalar field will be used for pseudo-coloring.

Colormap



Port to select a colormap.

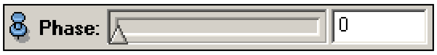
Lic



The input denoted *filter length* controls the one-sided length of the filter kernel used for line integral convolution. The larger this value is the more coherent is the greyscale distribution along the field lines. Often larger values are visually more attractive than smaller ones. A value of 0 lets you see an isotropic noise pattern without any directional information.

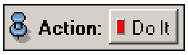
The second input of this port determines the *resolution* of an intermediate sampling raster used to compute field lines. Fine details of the vector field might be missed if the sampling resolution is too low. The resolution value also has an effect on the granularity of the resulting LIC image. The size of the LIC texture being computed is chosen to be the next power of two larger or equal than the sampling resolution. For example, if the resolution is set to 128 the size of the LIC texture will be 128x128. If the resolution is set to 129 then a LIC texture of size 256x256 will be computed, resulting in much finer structures.

Phase



This port will only be visible if a complex-valued vector field is connected to the module. It provides a phase slider controling which part of the complex 3D vectors is visualized. A value of 0 degree corresponds to the real part, while a value of 90 degrees corresponds to the imaginary part. Other values yield to intermediate vectors

Action



Starts computation of the LIC texture. If no LIC texture has been computed yet a default checkerboard pattern is displayed. This pattern is also displayed as soon as filter length or resolution are changed. Press the *DoIt* button again in order to update the texture.

Commands

`setNumSubPixels {1|2|3}`
Allows you to change an internal parameter of the LIC algorithm. Since LIC images contain very high spatial frequency components they are susceptible for aliasing. Aliasing can be almost eliminated by choosing the number of sub-pixels to be 2 or even 3. However, this is achieved at the expense of in increased computing time.

`writeTexture <filename>`
This command allows you to write the current LIC texture into a file in raw PPM format.

6.92 PointProbe

See Section Data Probing for details.

6.93 PointWrap

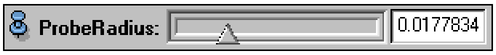
This algorithm performs a surface reconstruction from a set of unorganized points. It models a *probe sphere*, that is being 'dropped' onto and then 'rolled over' the set of points. Every three points the sphere rests on during this tour become a triangle in the resulting surface. The result is (almost) guaranteed to be an oriented manifold.

Connections

Data `[required]`
The input point set.

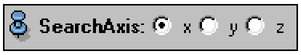
Ports

Probe Radius



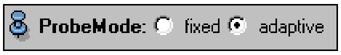
This slider specifies the radius of the probe sphere. It is only relevant if the module is run in the fixed radius mode and when looking for an initial triangle. If the algorithm is unable to find an initial triangle try increasing this value.

Search Axis


SearchAxis: [✓ x] y z

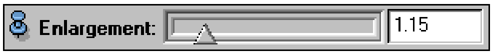
The direction the probe is initially dropped from to find a starting triangle. If no such triangle can be found, try a different axis.

Probe Mode


ProbeMode: fixed [✓ adaptive]

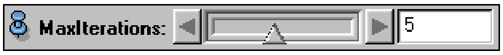
Here you can choose the probe radius to be fixed throughout the whole computation or to adapt it to local feature size. *Adaptive* probe size is faster than *fixed* probe size and gives better details on point sets that are relatively ’well behaved’. However, it is less robust.

Enlargement


Enlargement: 1.15

If the algorithm is run in *adaptive* probe size mode the local probe size might be too small to find any more triangles. If that happens, the size is enlarged by this factor and the local search is restarted.

MaxIterations


MaxIterations: 5

The iterative process described under *Enlargement* is repeated no more times than the number specified with this slider.

Action

The *DoIt* button triggers the computation.

6.94 ProbeToLineSet

This module can be connected (tight connection) to a LineProbe or SplineProbe module. It saves the probe line resp. spline into a LineSet data object. The LineSet contains all points where samples are taken as coordinates and the sampled values as data. Furthermore the spline controlpoints are saved in the parameters of the LineSet. Since a LineSet can hold more than one line additional probe lines will be saved whenever the DoIt button is hit.

If a LineSet is connected to the data port (optional) this LineSet is then copied and the probe lines are saved into the copy.

Connections

SplineProbe `[required]`

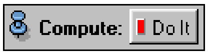
This is the controlling SplineProbe resp. LineProbe module.

Data `[optional]`

Can be optionally connected to a LineSet. A copy of that LineSet is used to save the probe lines.

Ports

Export



Pushing this button saves the current probe line into the LineSet.

Commands

6.95 ProjectionView

This module computes a shadow projection of a 3D uniform scalar field (an image volume) onto the three major planes (xy, xz, yz). Color fields and multi-channel fields are also supported. Maximum intensity projection or averaging can be chosen as projection method. This module is especially useful for data containing line-like structures like neurons or angiographic data.

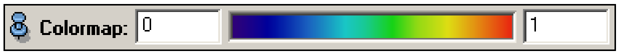
If attached to a 1-component scalar field, the location of voxels containing maximal values can be investigated interactively using the module Projection View Cursor.

Connections

Data `[required]`

The dataset to be projected. Uniform scalar fields, uniform color fields, and uniform multi-channel fields are supported.

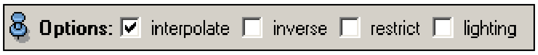
Colormap `[optional]`



An optional colormap which is used when the mapping type is set to *colormap* (see below).

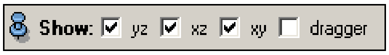
Ports

Options


Options: ✓ interpolate ☐ inverse ☐ restrict ☐ lighting

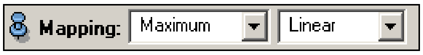
interpolate: Bilinearly interpolate between pixels on the projection planes. This in general gives better image quality and less aliasing. However sometimes the resulting image may appear somewhat blurred.
inverse: Invert gray values. Inversion is done *after* mapping data values to intensity values via the selected range.
restrict: If this toggle is set the projection is restricted to a subvolume of the original data set. The size of the subvolume can be adjusted via the minimum and maximum ports, or interactively via a tab-box dragger (see below).
lighting: Specifies whether the slices should be illuminated or not. If lighting is on the luminance of the slices changes when the scene is rotated.

Show


Show: ✓ yz ✓ xz ✓ xy ☐ dragger

The first three toggles can be used to turn off the yz-, xz-, or xy-slice. The fourth toggle (dragger) is only sensitive if the *restrict* option is enabled (see above). If set a 3D tab-box dragger is shown allowing you to adjust the size of the projection volume.

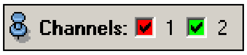
Mapping


Mapping: Maximum Linear

The first menu specifies the projection mode. *Maximum* searches the largest data value in each projection ray. *Average* computes the average data value for each projection ray. The options *depth* and *depth + max* are only available for scalar fields. If *depth* is chosen on each slice the depth of the pixel containing the maximum value is depicted, instead of the maximum value itself. If *depth + max* is chosen, first an ordinary maximum intensity projection is computed and mapped to a grey image. Then the grey image is multiplied with a pseudo-color image of the depth values. This allows to visualize both projected structures and depth information at once.

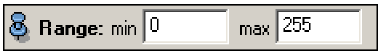
The second menu specifies how the projected data should be mapped to color or intensity. Three modes are available, *linear*, *histogram*, and *colormap*. These modes are identical to the ones described for the OrthoSlice module. *Colormap* is not available for color fields and for multi-channel fields.

Channels


Channels: ✓ 1 ✓ 2

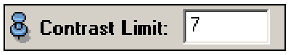
This port is shown if the module is connected to a multi-channel field. It allows you to turn individual channels on or of.

Range


Range: min 0 max 255

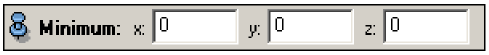
This port is only available in *linear* mapping mode. All data values smaller than the specified minimum are mapped to black, all values larger than the maximum are mapped to white, and the values in between are mapped linearly.

Contrast Limit


Contrast Limit: 7

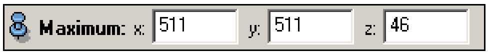
This port is only visible if *histogram* equalization is selected. The number determines the contrast of the resulting image. The higher the value, the more contrast is contained in the resulting image.

Minimum


Minimum: x: 0 y: 0 z: 0

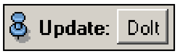
Only available if the *restrict* option is active. Specifies the lower left front corner of the projection subvolume.

Maximum


Maximum: x: 511 y: 511 z: 46

Only available if the *restrict* option is active. Specifies the upper right back corner of the projection subvolume.

Update


Update: DoIt

Only available if the *restrict* option is active. Press this button to recompute the projection after changing the size of the subvolume.

6.96 ProjectionViewCursor

This module can be attached to a Projection View module. It visualizes a 3D location on all three planes. The location is specified by picking (i.e. clicking on) any geometry, like an isosurface, in the viewer. If one of the three planes of the *Projection View* itself is picked the module will search for the brightest pixel to determine the missing coordinate. However, this will only work if a scalar field is connected to the *Projection View*, but not for color fields or multi-channel fields.

Connections

Module `[required]`

This module is attached directly to a ProjetionView module.

Ports

Color

Color of sphere and circles.

Options

If the 3D sphere option is selected, a sphere is shown at the 3D position of the selected location. Otherwise only the three projections of the sphere are shown.

Size

Radius of sphere and circles in percent of the bounding box diagonal length.

LandMarkSet

This port can be used to export the current cursor position into a landmark set data object. Such an object will be created automatically the first the the *Add* button is pressed. After that you can modify the cursor position and append the new position to the landmark set by pressing the *Add* button again. The *Clear* button removes all landmarks stored in the landmark set data object.

6.97 Registration

This module computes an affine transformation for registration of two image data sets, using an iterative optimization algorithm. A hierarchical strategy is applied, starting at a coarse resampling of the dataset, and proceeding to finer resolutions later on. Different similarity measures like Euclidean distance, mutual information and correlation can be chosen.

To use this module, you must connect it to two scalar fields. To follow the progress visually, a module like BoundingBox or Isosurface should be connected to each of them, but this may slow down the registration process. If the *Register* button of the *Action* port is pressed, the module starts by successively optimizing the transformation of the first input.

The optimization can be interrupted at any time. Interruption might take some seconds.

Connections

Model `[required]`

The model dataset to be transformed.

Reference `[required]`

The reference dataset to which the model is registered.

Ports

Metric

This port selects the similarity measure to be applied. *Euclidean* means the Euclidean distance, i.e. the mean squared difference between the gray values of model and reference. *Correlation* measures the correlation of the registered images. The *Mutual information* metrics, especially the normalized one, are recommended when medical images from different modalities, e.g., CT and MRT, are to be registered.

Transformation

At this port you can select the number of transformation parameters to be optimized. The numbers are 6 for *Rigid* (3 translations and 3 rotations), 7, 9, and 12 for *IsoScale*, *AnisoScale*, and *Shear*, respectively.

If you select for example *Rigid* and *AnisoScale*, on each resolution level the 6 parameters of a rigid transformation will be optimized first, followed by an optimization of all 9 parameters of an anisoscale transformation. In this way as much as possible is done using a rigid transformation.

If only *AnisoScale* is selected, all 9 parameters will be optimized at once. A larger contribution of the scaling parameters may be expected for this selection.

Extended options

If this toggle is unset, only the most important ports are visible. The other ports are set to default values which should work well in most cases.

If the toggle is selected, the additional ports *Optimizer*, *Optimizer step*, *QuasiNewton Optimizer on finest N levels*, *CoarsestResampling*, *Histogram range reference*, *Histogram range model*, and *Options* become visible (see below).

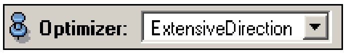
Action

If the *Align Centers* button is pressed, the centers of gravity of both datasets are computed, taking the image intensity as a mass density. The model dataset is translated in order to align both centers of gravity.

If the *Align Principal Axes* button is pressed, the centers of gravity and moments of inertia of both datasets are computed, again taking the image intensity as a mass density. The principal moments of inertia and corresponding principal axes are computed. The best of 24 possible alignments of the principal axes is determined according to the similarity measure as selected at the *Metric* port.

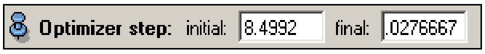
Pressing the *Register* button starts the actual registration process.

Optimizer


Optimizer: ExtensiveDirection

At this port you can choose between different optimization strategies. The *ExtensiveDirection* or *BestNeighbor* optimizers are well suited for coarse resolution levels, the *QuasiNewton* or *Line-Search* optimizers for the finer resolution levels. The default strategy uses the *ExtensiveDirection* optimizer on the coarse levels and the *QuasiNewton* optimizer on the finest levels.

Optimizer step

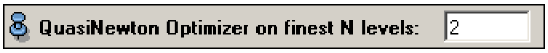

Optimizer step: initial: 8.4992 final: .0276667

This port sets the initial and the final value for the stepwidth to be applied in the optimizations. These stepwidths refer to translations. For rotations, scalings, and shearings appropriate values are chosen accordingly.

The default value for the initial stepwidth is 1/5 of the size of the bounding box. If both datasets are already reasonably aligned, you may choose a smaller initial stepwidth.

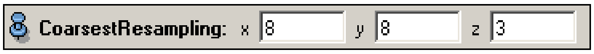
The default value for the final stepwidth is 1/6 of the voxel size.

QuasiNewton Optimizer on finest N levels


QuasiNewton Optimizer on finest N levels: 2

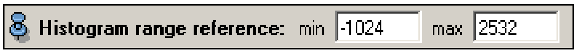
At this port you can select the number of resolution levels (between 0 and 2) where the *QuasiNewton* optimizer is applied. On the coarser levels the optimizer as selected at port Optimizer is applied. If the number of levels is less than or equal to the number selected at this port, the optimizer as selected at port Optimizer is applied at least at the coarsest level.

CoarsestResampling


CoarsestResampling: x 8 y 8 z 3

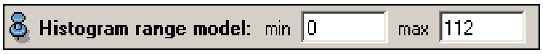
At this port you can define the resampling rate for the coarsest resolution level where registration starts. The resampling rate refers to the reference dataset. If the voxels of the reference dataset are *anisotropic*, i.e. have a different size in x-, y-, and z-direction, the default resampling rates are chosen in order to achieve isotropic voxels on the coarsest level. If the voxel sizes of model and reference differ, the resampling rates for the model are chosen in order to achieve similar voxel sizes as for the reference on the same level.

Histogram range reference


Histogram range reference: min -1024 max 2532

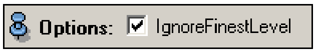
This port is only active if one of the *Mutual information* metrics has been selected. Here you can define the range of gray values for the histogram of the reference dataset. The essential information of the dataset should be within this range. It is a good idea to determine the range via a visualization of the dataset, e.g., using an *OrthoSlice* module.

Histogram range model


Histogram range model: min 0 max 112

The same as the previous port, now for the model dataset.

Options


Options: ☑ IgnoreFinestLevel

If toggle *IgnoreFinestLevel* is selected, registration will be performed on all but the finest (i.e. the original) resolution. In many cases a sufficient accuracy can be achieved in this way. Registration on the finest level may slightly improve the accuracy, but the computation time will typically increase by one order of magnitude.

6.98 Relabel

This module sorts the materials in a LabelField according to the material list of a template. Materials which are only present in the template are added. If materials are found in the input, which are not present in the template, a warning is issued.

The module is also capable of merging multiple label fields of the same size. This is useful if different parts of the same data set have been segmented independently. If corresponding voxels in the input are assigned different materials, which are both different from Exterior, the last input will be used. If the option *Conflict Material* is checked, such voxels will be assigned to a new material named *MergeConflict*.

This module can be used to fill a new, empty label field with materials from a previous segmentation.

Connections

Data `[required]`

Input LabelField.

Data2 `[optional]`

Additional LabelField to be merged with the first one.

Data3 `[optional]`

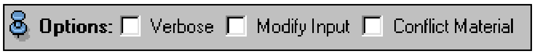
Additional LabelField to be merged with the first two ones.

Template `[optional]`

LabelField with the template material list.

Ports

Options



The **verbose** flag produces some information on the actual mapping. If the **Modify Input** option is chosen, the input data set itself is modified instead of duplicating it, do not use this option on a data set, while an editor is active. If the **Conflict Material** option is chosen in a merging action, voxels with different materials in the input data sets are assigned to a special material called *MergeConflict*.

Action

Trigger computation.

6.99 Resample

This module works on any 3-dimensional field with regular coordinates, e.g., complex and non-complex scalar or vector fields or RGBA color fields. It lets you *resample* the data, i.e., enlarge or shrink the dimensions of the regular grid while recalculating the data according to it. The first port displayed is *Input Resolution* which indicates the resolution of the incoming data field to be resampled. If the input field has uniform coordinate the voxelsize is also displayed. The other ports depend on whether the input field contains ordinary numbers or labels as used in image segmentation (c.f. LabelField).

For non-labeled data fields you see the *Filter*, the *Mode*, the *Resolution* port and if the input field has uniform coordinates you will see the *Voxel Size* port. In case of a labeled data field you see the *Average* port. In both cases the *DoIt* button completes the port list. Depending on the existence of labeled data the resampling operation is performed differently.

6.99.1 Resampling non-labeled data fields

For non-labeled input data the ports denoted *Filter, Mode*, *Resolution* and *Voxelsize* will be shown.

Filter provides an option menu allowing you to specify the filter kernel for the resampling operation. Usually the default kernel, *Triangle*, yields sufficiently good results for both minifications and magnifications. The following filters are supported:

- *Box* - simple replication of scalar values, shows considerably tiling or jaggies
- *Triangle* - computationally simple, still sharp transition lines

- *Bell* - smoothing filter
- *B-Spline* - no sharp transitions, but its width causes excessive blurring
- *Lanczos* - excessive ”ringing” effect
- *Mitchell* - no sharp transitions, good compromise between ”ringing” and ”blurring”
- *Minimum* - useful for down-sampling, preserves tiny dark features on a bright background
- *Maximum* - useful for down-sampling, preserves tiny bright features on a dark background
- *Cubic, width 6* - similar to Lanczos
- *Cubic, width 8* - performs similar to Lanczos

The *Minimum* and *Maximum* filters can only be used to downsample non-complex data fields.

Port *Mode* gives you choices how to specify the resolution of the output field. If you select ‘dimensions’ the port *Resolution* will be enabled to take your input. if you select ‘voxel size’ port *voxel size* is enabled.

If you connect a second field to the *Reference* connection the behavior will be a little bit different. If you select ‘dimensions’ the output field will have the same dimensions as the reference field. If you select ‘voxel size’ the voxel size will be taken from the reference. Additionally the choice ‘reference’ is enabled which resamples directly onto the lattice that is provided by the reference field.

To start resampling, press the *DoIt* button.

6.99.2 Resampling labeled data fields

Labeled data fields can only be down-sampled. Instead of *Filter* and *Resolution* a port denoted *Average* appears. This port allows you to enter the number of cells to average in every dimension. Note, that no enlargement is possible.

As above, the *DoIt* button initiates the resampling.

6.99.3 Coordinates of the resampled data set

Resampling can be performed on any 3-dimensional field with either uniform, stacked, rectilinear, or curvilinear coordinates. The resampling operation does not change the coordinate type. If you want to convert a data set with stacked, rectilinear, or curvilinear coordinates into one with uniform coordinates you should use the Arithmetic module instead of *Resample*. The coordinates of the resampled data set are obtained by a resampling operation on the coordinates of the input data set.

Note, that in general the bounding box of the resampled data set will be different from the one of the input data set. In particular, for uniform coordinates the bounding box will extend from the center of the first voxel to the center of the last one.

6.99.4 Resampling in progress

While resampling, the progress bar at the bottom of the work area indicates the percentage of resampling that is done. You may cancel the resampling calculation any time by pressing the stop button on the right of the progress bar.

Connections

Data `[required]`

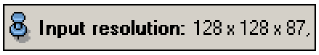
The underlying data field to be resampled.

Reference `[optional]`

A reference lattice. It can provide the new resolution, the new voxelsize or the new lattice.

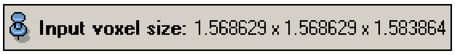
Ports

Input Resolution


Input resolution: 128 x 128 x 87,

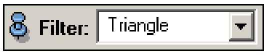
Displays the resolution of the input data set.

Input Voxel Size


Input voxel size: 1.568629 x 1.568629 x 1.583864

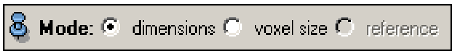
Displays the voxelsize of the input data set (if available).

Filter


Filter: Triangle

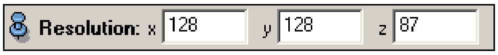
This lets you select one of the resampling filters mentioned above. This will only be visible if the input data set does not contain labels.

Mode


Mode: [✓ dimensions] voxel size reference

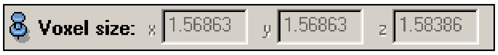
This lets you select whether you want to specify the new resolution or the new voxelsize. If a reference lattice is connected the values a taken from there. Additionally there is the option enabled to sample directly onto the points of the reference lattice.

Resolution


Resolution: x 128 y 128 z 87

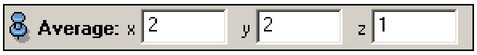
Specifies the resolution of the output data set. This port will only be visible if the input data set does not contain labels.

Voxelsize



Specifies the voxelsize of the output data set. This port will only be visible if the input data set does not contain labels and has uniform coordinates.

Average



Specifies how many labels should be averaged during down-sampling. This port will only be visible if the input data set contains labels.

Action

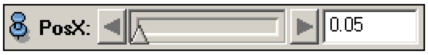
Starts resampling.

6.100 Scale

The Module displays a 2D coordinate System on top of the rendering area. The coordinate a calculated on a plane perpendicular to the viewing direction located a the focal distance of the camera. The focal distance is the point about which rotations are preformed. In orthogonal projections the displayed distances are valid for all parts of the picture, in perspective projection only for the mentioned plane. The location and size of the coordinate system can be controlled by the ports. The module scales things to get 'nice' numbers (no fractionals). Create an instance with the Edit-Create menu.

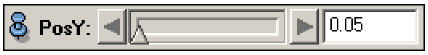
Ports

PosX



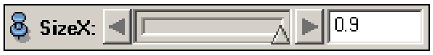
The position of the origin (fraction of horizontal size of the viewing area).

PosY



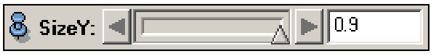
The position of the origin (fraction of vertical size of the viewing area).

SizeX



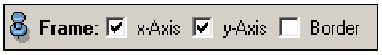
Maximal size of the coordinate system. (fraction of viewport size).

SizeY


SizeY: 0.9

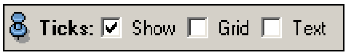
Maximal size of the coordinate system. (fraction of viewport size).

Frame


Frame: ☑ x-Axis ☑ y-Axis ☐ Border

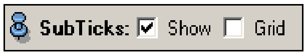
Select the parts of the frame which should be drawn.

Ticks


Ticks: ☑ Show ☐ Grid ☐ Text

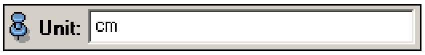
Select whether Ticks, Grid and Text should be drawn.

SubTicks


SubTicks: ☑ Show ☐ Grid

Select whether SubTicks and SubGrid should be drawn.

Unit


Unit: cm

A unit that is displayed at the axes.

Color

Color: ☐

Color of the coordinate system.

6.101 ScanConvertSurface

This module computes a volumetric representation of closed manifold and non-manifold surfaces. The result is a 3D uniform LabelField, whose bounding box and dimension can either be specified by filling in the corresponding ports or by a connecting a reference field. Each voxel on the inside of the surface is labelled according to material index of the surface. The user can choose to label all or pick out single materials.

Connections

Data `[required]`

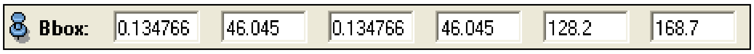
Surface to be converted.

Field `[optional]`

Reference regular 3D field, whose dimensions and bounding box are used to define the resulting field.

Ports

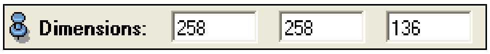
Bbox



0.134766 46.045 0.134766 46.045 128.2 168.7

Bounding box of the resulting field.

Dimensions



258 258 136

Dimensions of the resulting field.

Materials

All

Material(s) of the surface to be converted.

Options

If a reference LabelField was connected, the volume overlap of the materials of the resulting field with the reference field can optionally be computed. Its value will be written to the console window.

Action

Starts the computation.

6.102 SeedSurface

This module extends the vector field visualization module DisplayISL. It allows you to compute illuminated field lines of a vector field with seed points distributed across an arbitrary surface.

First, load both the vector field and the surface into amira. Then attach *DisplayISL* to the vector field. From the popup menu of *DisplayISL* choose *SeedSurface*. The *SeedSurface* module automatically connects itself to the first surface found in the object pool. Of course, you may change the surface connection at any time later on. Properties of the illuminated field lines such as base opacity, fade factor, or color will be determined by *DisplayISL*.

Connections

Data `[required]`

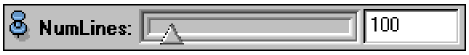
Module of type DisplayISL which is used to display the illuminated field lines.

DisplayISL `[required]`

The surface on which the field lines will be distributed.

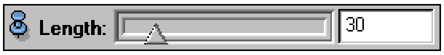
Ports

NumLines



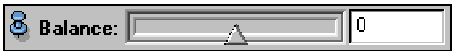
Number of field lines to be displayed. This port will only be visible if distribution mode *on surface* has been selected. The distribution algorithm tries to achieve a constant seed density per surface area.

Length



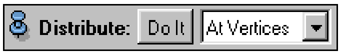
The length of the field lines, or more precisely, the number of atomic line segments, in forward respectively backward direction. The lines may stop earlier if a singularity (i.e. zero magnitude) is encountered or if the field's domain is left.

Balance



On default field lines are equally long in forward and backward direction, corresponding to a balance value of 0. This port allows you to change this behavior. A value of -1 indicates that field lines should extend in backward direction only, while a value of 1 indicates that field lines should extend in forward direction only.

Distribute



On the one hand, this port provides a *DoIt* button which is used to initiate distribution of seeds and recomputation of field lines. Once the incoming vector field has changed or you have modified the number of field lines or the line's length you have to press *DoIt* in order to update the display.

On the other hand, the port also provides an option menu specifying the seed distribution mode. If *at vertices* is chosen a field line is started at each vertex of the surface. If *on surface* is chosen a user-defined number of field lines will be uniformly distributed across the surface.

6.103 SelectRoi

This module defines a region-of-interest with the shape of an axis-aligned 3D box. This box can be used to restrict the output of many visualization modules like Voltex, ProjectionView, or all modules derived from ViewBase, e.g., Isosurface or HxSurfaceView.

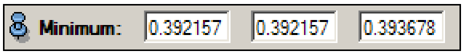
Connections

Data `[required]`

Connection to a 3D data object which defines the maximum size of the region-of-interest. Only the bounding box of the data object but not the data itself is interpreted by this module.

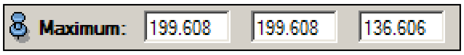
Ports

Minimum


Minimum: 0.392157 0.392157 0.393678

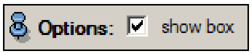
Minimum x-, y-, and z-coordinates of the region-of-interest.

Maximum


Maximum: 199.608 199.608 136.606

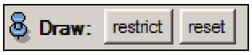
Maximum x-, y-, and z-coordinates of the region-of-interest.

Options


Options: [✓] show box

If the option *show dragger* is enabled a tab-box dragger is shown which allows you to interactively adjust the region-of-interest in the 3D viewer.

Draw



After pressing the *restrict* button you can draw a contour in the viewer window in order to restrict the region-of-interest. However, note that although you can draw an arbitrary contour the region-of-interest itself still remains a box. The *reset* button resets the box so that is covers the full bounding box of the connected data set.

6.104 Shear

This module shears a uniform scalar field by shifting each xy-slice in y direction. The shift is proportional to the z-coordinate of the slice, resulting in a shear-operation. The angle between the original and the sheared z-axis can be specified.

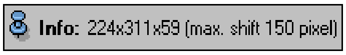
Connections

Data `[required]`

Connects to uniform scalar fields.

Ports

Info

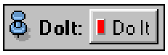

Info: 224x311x59 (max. shift 150 pixel)

The output field will be larger than the input field. This port gives the details.

Angle

Specifies the angle between the input z-axis and the sheared z-axis. Positive and negative values are allowed.

DoIt



Triggers the computation.

6.105 Smooth Surface

This module smoothes a surface by shifting its vertices. Each vertex is shifted towards the average position of its neighbors. Special care is taken in the case of boundary vertices, for which not all the neighbors are considered, but only those that are also on the boundary. In this way sharp boundaries are preserved.

The smoothing operation is controlled by two tunable parameters: the number of iterations to be performed and a *lambda* coefficient which should be in the range 0...1 and describes the step for each iteration.

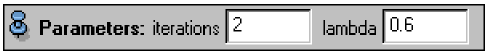
Connections

Data `[required]`

The surface to be smoothed.

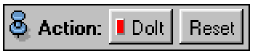
Ports

Parameters



Contains two parameters: *Iterations* specifies the number of steps for the smoothing, *lambda* is a shifting coefficient which should be in the range (0..1)

Action



The *DoIt* button starts the smoothing operation. Pressing *DoIt* once again causes the result to be smoothed further. The input is only accessed if no result is present. This behaviour is different from most other modules but it allows better interactive control of the smoothing operation.

Pressing the *Reset* button restores the original surface, i.e., the input is copied to the output.

6.106 Splats

This module visualizes scalar fields defined on tetrahedral grids using a direct volume rendering technique called cell projection splatting. The principle of this method is to display a kind of semi-transparent clouds. The higher the data values the brighter and more opaque these clouds are. Often, meaningful results are obtained if this technique is used in conjunction with a standard isosurface module.

In order to get correct results for linearly varying functions a special texture mapping technique is applied. On machines where texture mapping is not supported in hardware this might be quite slow. As an alternative untextured splats may be used. However, in this case the scalar field is assumed to be constant in each tetrahedron. To obtain such a piecewise constant function the vertex values of each tetrahedron are averaged. As a consequence artifical discontinuities might be observed.

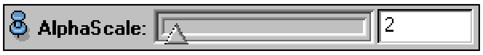
Connections

Data `[required]`

The tetrahedral scalar field to be visualized.

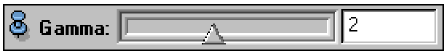
Ports

Alpha Scale


AlphaScale: 2

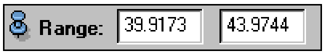
A global factor used to change the overall transparency of the individual splats. Higher values produce denser clouds.

Gamma


Gamma: 2

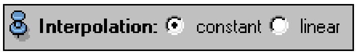
This value determines how the function value between the min and max values of port *Range* will be mapped to opacity and color. If the gamma value is 1 a linear mapping will be used. If the gamma value is smaller than 1 the overall appearance of the cloud gets more opaque. If the gamma value is bigger than 1 the cloud gets more transparent.

Range


Range: 39.9173 43.9744

Data range. Regions where the function value is below the min value will be completely transparent. Likewise, regions where the function value is above the max value will be opaque.

Interpolation


Interpolation: [✓ constant] linear

Lets you select the type of splats being used. *Constant* enables untextured splats. *Linear* enables textured splats. The latter setting will be able to correctly display linearly varying scalar fields.

Commands

```
setColor <color>
```

Lets you define the base color of the volume rendered clouds. On default an orange color will be used (0.8 0.6 0.1). The color may be specified by either an RGB triple in range 0...1 or by a common X11 color name, e.g., *green*.

6.107 SplineProbe

See Section Data Probing for details.

6.108 StandardView

The *StandardView* module displays a 3D image data set or, more precisely, a 3D scalar field with either uniform or stacked coordinates in three different 2D windows at once. The windows show coronal, sagittal, and axial views of the data, respectively. These views correspond to standard xy-, yz, and xy-orientations. Note, that for the axial (xy) view the origin is in the upper left corner. This is the standard orientation used in radiology. In each 2D view the position of the two other slices is indicated by a colored cross hair. You may click at any point in a 2D view in order to reposition the two other slices. The upper left part of the viewer window shows the usual 3D view.

Connections

Data `[required]`

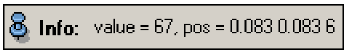
The 3D scalar field to be visualized.

OverlayData `[optional]`

If this port is connected to a second 3D image data set, an overlay of both images is shown in the 2D windows.

Ports

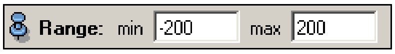
Info



Info: value = 67, pos = 0.083 0.083 6

This port gives some information about the value of the 3D scalar field at the point where the three 2D slices intersect, as well as its coordinates.

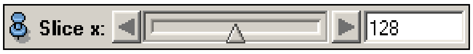
Range



Range: min -200 max 200

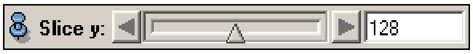
Controls the mapping of input data to gray values. Values below *min* are mapped to black, values above *max* are mapped to white. Values between *min* and *max* are mapped linearly.

SliceX


Slice x: 128

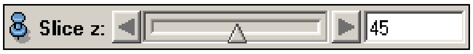
Number of slice currently displayed in sagittal (yz) window.

SliceY


Slice y: 128

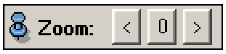
Number of slice currently displayed in coronal (xz) window.

SliceZ


Slice z: 45

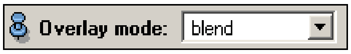
Number of slice currently displayed in axial (xy) window.

Zoom


Zoom: 0

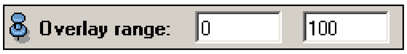
Allows you to decrease, reset, or increase the current image magnification factor. You may also press `[Ctrl][z]` to zoom down or `[Ctrl][Shift][z]` to zoom up.

Overlay mode


Overlay mode: blend

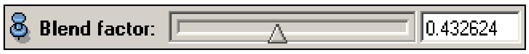
This port is only visible if a second image dataset has been connected to input port *OverlayData*. You can choose one of four overlay modes: in *blend*, *add*, and *maximum* mode a blending, the sum, or the maximum of both images is shown, respectively. In *checkerboard* mode the 2D windows are divided like a checkerboard showing both image datasets alternately.

Overlay range


Overlay range: 0 100

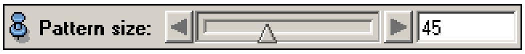
This port is only visible if a second image dataset has been connected to input port *OverlayData*. It controls the mapping of the second image dataset to gray values, similar to port *Range* for the first image dataset.

Blend factor


Blend factor: 0.432624

This port is only visible if overlay mode *blend* has been selected. It controls the blending of both image datasets. Blend factors 0 and 1 mean that only the first or only the second dataset is shown, respectively. For values between 0 and 1 a linear interpolation is done.

Pattern size



This port is only visible if overlay mode *checkerboard* has been selected. It controls the size of the checkerboard tiles.

6.109 StreamRibbons

This module displays stream lines or stream ribbons in a flow field. Stream ribbons are computed by tracing two individual stream lines and connecting them by triangles. The initial orientation of a stream ribbon is orthogonal to the normal direction of the flow field at the seed point. The seed points themselves are defined interactively by moving a seed shape in space (a line, a circle, or a filled square). The seed shape can be transformed using an Open Inventor transformer dragger. In amiraVR it is also possible to pick and transform the dragger using the 3D mouse.

Connections

Data `[required]`

The vector field to be visualized.

Colormap `[optional]`

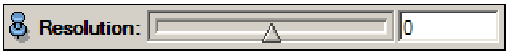
Colormap used to depict vector magnitude.

Ports

Colormap

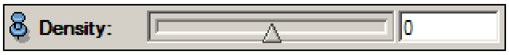
Colormap used to depict vector magnitude.

Resolution



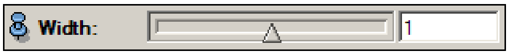
Logarithmic slider allowing to adjust the resolution of the stream line tracing algorithm. A value of 1 means a 10 times higher resolution compared to the default, while a value of -1 means a 10 times smaller resolution. The higher the resolution the more line segments and triangles are generated.

Density



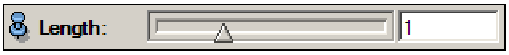
Logorithmic slider allowing to adjust the density of stream lines or stream ribbons. The higher the value the more lines or ribbons are traced.

Width



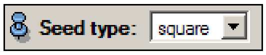
This slider can be used to adjust the width or thickness of the stream ribbons. The default thickness is set proportional to the length of the diagonal of the bounding box of the input data set.

Length



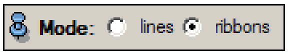
Adjusts the length of the stream lines or stream ribbons.

Seed type



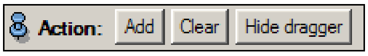
Defines the seed type: along a straight line segment, along a circular line, or inside a square regions.

Mode



Defines if simple stream lines or stream ribbons should be traced.

Action



The *Add* and *Clear* buttons have no meaning yet. The third button allows you to show or hide the Open Inventor dragger for manipulating the seed shape position.

Commands

Inherits all ports of Object.

`setBox -b xmin xmax ymin ymax zmin zmax`
Sets the position of the Open Inventor dragger so that it matches the specified box. The display is updated automatically.

`setBox [-t x y z] [-r x y z phi] [-s x y z]`
Alternate way of setting the Open Inventor dragger. The three optional arguments indicate the translational, rotational, and scaling part of the dragger's transformation matrix. With this command it is possible to set the dragger in an arbitrary way (it must not be axis-aligned). If all three parts are specified the option strings `-t`, `-r`, and `-s` can be omitted.

`getBox`
Returns the current transformation of the dragger in terms of translation (first three numbers), rotations (next four numbers), and scaling (last three numbers). The result can be used as the arguments of the `setBox` command.

6.110 StreamSurface

This module computes stream surfaces of an arbitrary 3D vector field. A stream surface consists of multiple interconnected stream lines started along a predefined line source. The algorithm automatically inserts new stream lines in divergent regions of the field. Likewise, in convergent regions stream lines are automatically removed. The resulting stream surfaces can be accumulated in a separate Surface object. In this way further post-processing is facilitated, for example, computation of a LIC texture using SurfaceLIC.

Connections

Data `[required]`

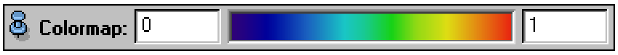
The 3D vector field to be visualized.

Colormap `[optional]`

Optional colormap. On default, the current stream surface will be display in wireframe mode and will be colored according to the arc-length of the stream lines forming the stream surface.

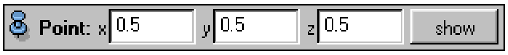
Ports

Colormap


Colormap: 0 1

Port to select a colormap.

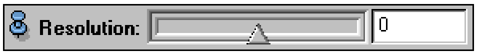
Origin


Point: x 0.5 y 0.5 z 0.5

This port defines the seed point of the stream surface. By pressing the toggle *show* a crosshair dragger can be activated which allows you to change the seed point interactively in 3D.

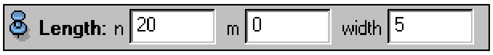
Actually, in order to compute a stream surface not only a seed point but a seed line is required. Starting from the seed point, such a line will be defined automatically taking into account the seed selection mode chosen in port *Action*.

Resolution


Resolution: 0

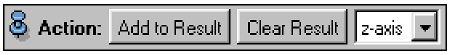
Controls the resolution of the discretized stream surface or, more precisely, the preferred edge length of the triangulation. Smaller values results in more details.

Length


Length: n 20 m 0 width 5

Value *n* determines how far the surface should be traced in backward direction. Value *m* determines how far the surface should be traced in forward direction. Value *width* determines the extent of the seed line. The actual lengths in physical space depend on the value of port *resolution*. All values may be changed using the right mouse buttons by means of a virtual slider.

Action



This port provides push buttons allowing you to store the current stream surface in a surface object (*add to result*), respectively to remove all triangles from this surface again (*clear result*). Moreover, the port provides an option menu allowing you to select the mode used for automatic seed line definition. The following modes are supported:

Binormal: The seed line will be traced in a direction perpendicular to both the stream line's tangent and normal (curvature) direction. Often this gives quite meaningful results. However, note that in general the stream surface's normal vectors do not conicide with the stream line's curvature vectors except at the seed line.

Normal: The seed line will be traced along the field line's normal (curvature) direction.

X-axis: The seed line is chosen in x-direction

Y-axis: The seed line is chosen in y-direction.

Z-axis: The seed line is chosen in z-direction.

Commands

`setLineWidth <width>`
Sets line width of wireframe model.

`setTolerance <eps>`
Sets relative tolerance used for field line integration. The default is 0.001 times the sampling width set in port *resolution*.

`doLineSet {0|1}`
If argument is 1 only stream lines will be displayed instead of the stream surface's triangulation.

`setDrawStyle {1|2}`
Allows you to set the draw style used in surface mode, i.e., when `doLineSet` is off. A value of 1 denotes wireframe mode, while a value of 2 denotes shaded surface mode. In order to display the surface in a more fancy way convert it into a surface object and use SurfaceView.

6.111 SurfaceArea

This module calculates the area of the individual patches of a surface (*patch mode*). The results are stored in a spread sheet data object.

In an alternative mode the area and the enclosed volume of the different regions defined in the surface are computed (*material mode*). In this mode the total surface area is twice as large since every triangle contributes to two regions. Note, that for this mode it is required that the surface is closed, i.e., that all regions are completely enclosed by triangles. Otherwise, the computed volumes will be incorrect.

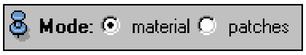
Connections

Data `[required]`

The surface to be investigated.

Ports

Mode



Toggles between *material mode* and *patch mode*.

In *material mode* the resulting spread sheet object contains one row for every non-empty region of the surface. Each row contains the name of the region, the number of triangles of the region's boundary, the surface area of the boundary, as well as the enclosed volume. Usually, for the *exterior* region the volume is negative.

In *patch mode* the resulting spread sheet object contains one row for every patch of the surface. Each row ontains the patch id, the patch's inner region name, the patch's outer region name, the number of triangles of the patch, and the surface area of the patch. In the following columns the total number of triangles and the total surface area are printed. If the surface data structure also contains the surface contour, the surface perimeter is displayed. Note that you might have to use the **recompute** command of the Surface module, to obtain the contour information.

Action

Starts computation.

6.112 SurfaceCut

The *SurfaceCut* displays a filled cross-section through a surface generated by the SurfaceGen module. The surface is supposed to separate different volumetric regions from each other without any holes. Within the cross section the different materials are indicated by their respective colors. If the surface does not form closed loops in the intersection plane these parts will not be shown. The module is derived from ArbitraryCut and thus provides the same methods for manipulating the position and orientation of the cross section as this base class. An analog module GridCut exists for displaying cross-sections in a tetrahedral finite-element grid.

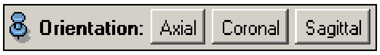
Connections

Data `[required]`

The surface to be visualized.

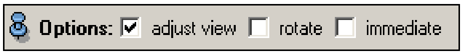
Ports

Orientation



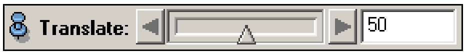
This port provides three buttons to specify the slice orientation. *Axial* slices are perpendicular to the z-axis, *coronal* slices are perpendicular to the y-axis, and *sagittal* slices are perpendicular to the x-axis.

Options


Options: [✓] adjust view [ ] rotate [ ] immediate

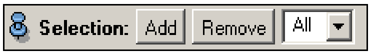
If the *adjust view* toggle is set, the camera of the main viewer is reset each time a new slice orientation is selected. With the *rotate* toggle you can switch on the rotate handle for the cutting plane and off again. If the *immediate* toggle is set the slice is updated every time you drag it with the mouse in the 3D viewer. Otherwise only the bounding box of the cutting plane is moved and the update takes place when you release the mouse button.

Translate


Translate: 50

This slider allows you to select different slices. The slices may also be picked and dragged directly in the 3D viewer.

Selection



This port maintains a list of materials to be displayed within the cross section. With the selection menu one can select a single material. The *Add* button adds the currently selected material to the list so the that is becomes visible and the *Remove* button removes the material so that is becomes invisible.

Commands

Inherits all commands of ArbitraryCut.

`selectMaterial <id1> [<id2> ...]`

Selects the materials with the specified ids so that intersections of these materials with the cutting plane will be shown. You need to call `fire` before changes take effect.

`unselectMaterial <id1> [<id2> ...]`
Unselects the materials with the specified ids so that intersections of these materials with the cutting plane will not be shown. You need to call `fire` before changes take effect.

6.113 SurfaceDistance

This module computes several different distance measures between two triangulated surface. For each vertex of one surface it computes the closest point on the other surface. From the histogram of these values the following measures are computed:

- mean distance
- standard deviation from the mean distance
- root mean square distance
- maximum distance (Hausdorff distance)
- medial distance
- area deviation: percentage of area that deviates more than a given threshold

By using an octree structure the computation is sped up, but may fail, when two surfaces are too different in their shape or location.

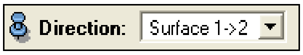
Connections

surface1 `[required]`
any triangulated surface

surface2 `[required]`
any other triangulated surface

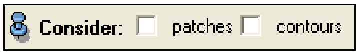
Ports

Direction


Direction: Surface 1->2

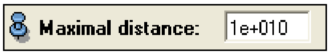
The distance measures are non–symmetric. Thus one can compute them from surface 1-¿2 or vice versa 2-¿1. One way to get symmetric measures is to join the histograms from both directions into a single histogram (Two–sided option). Note that the two-sided option takes twice as long to compute as their one-sided counterparts.

Consider


Consider: ☐ patches ☐ contours

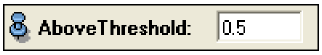
When computing the closest points, one can chose to respect either patches or patches and contours. This means, that closest points from any patch/contour will be restricted to lie on the same patch/contour of the other surface.

Maximal distance


Maximal distance: 1e+010

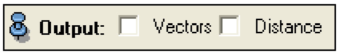
Value for the distance to be used when no closest point could be computed. This can happen when surfaces are too far apart or too different in their shape.

AboveThreshold


AboveThreshold: 0.5

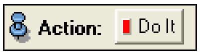
Threshold value for the computation of the area deviation.

Output


Output: ☐ Vectors ☐ Distance

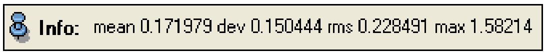
Optional output of the vectors from each point on one surface to their associated closest points on the other surface, or only their magnitude (distance option).

Action


Action: Do It

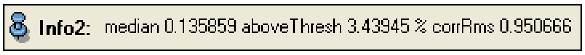
Start computation.

Info


Info: mean 0.171979 dev 0.150444 rms 0.228491 max 1.58214

Resulting distance measures will be displayed here: mean, standard deviation, root mean square and maximum.

Info2


Info2: median 0.135859 aboveThresh 3.43945 % corrRms 0.950666

Resulting distance measures will be displayed here: median, area deviation. If the connected surfaces have the same number of nodes. The root mean square distance between two vertices with the same index will be computed as well. This is of interest, when you have computed correspondences by some other method than the closest points computation.

6.114 SurfaceField

This module computes a *SurfaceField* from a Surface and a *UniformScalarField*. It can be choosen between encoding on nodes, on triangles, and on triangle nodes.

Connections

Surface `[required]`

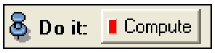
A surface that is the domain of the surface field to be generated.

Data `[required]`

A 3D scalar field.

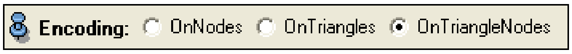
Ports

DoIt



starts computation of surface field

Encoding


Encoding: OnNodes OnTriangles [✓ OnTriangleNodes]

The encoding defines how the data is stored:

- On the the nodes of the surface.
- On the triangles of the surface.
- Three data values per triangle, one for each node.

6.115 SurfaceGen

This module computes a triangular approximation of the interfaces between different tissue types in a LabelField with either uniform or stacked coordinates. The resulting surfaces can be non-manifold surfaces. In earlier releases of amira this module was called GMC.

Depending on the resolution of the incoming LabelField the resulting triangular surface may have a huge amount of triangles. Therefore it is often recommended to start with a downsampled version of the LabelField, e.g., with a resolution of 128x128 pixels per slice. During the resampling process the Resample module will create or update the probability information of the LabelField. This way the loss of information caused by the resampling process will be minimized. The SurfaceGen module can use the probability information to generate smoother surfaces.

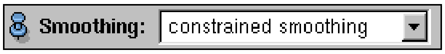
Connections

Data `[required]`

LabelField from which the interfaces should be extracted.

Ports

Smoothing

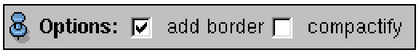

Smoothing: constrained smoothing

This port controls the way in which the module generates a smooth surface. If set to *none*, no sub-voxel weights are used and the resulting surface will look staircase-like. If set to *existing weights*, then pre-computed weights are used; such weights can be generated with the resample module or the smoothing filter in the image segmentation editor. The options *constrained smoothing* and *unconstrained smoothing* generate sub-voxel weights, such that the surface is naturally smooth; in the *constrained smoothing* mode, the module guarantees that no label be modified: any two voxel centers that have been labeled differently before the smoothing are separated by the generated surface afterwards. This is not necessarily the case for every small detail in the unconstrained case. The amount of smoothing can be controlled via the TCL interface. Type

```
SurfaceGen setVar SmoothKernelSize <value>
```

into the amira console, to change the default, which is 5 for the unconstrained and 4 for the constrained case. The values are not limited to integer variables.

Options

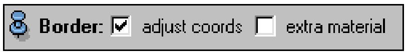

Options: [✓] add border [ ] compactify

This port provides two toggles.

The first toggle, *add border*, ensures that the resulting surfaces will be closed, even if some regions extend up to the boundary of the LabelField. Closed surfaces are required if a tetrahedral grid is to be generated later on.

If *compactify* is on then a specialized post-processing edge-contraction technique is used to reduce the number of triangles in the resulting surface. See the description of the port *Minimal Edge Length* below for a more detailed explanation.

Border


Border: [✓] adjust coords [ ] extra material

This port will only be visible if *add border* has been selected. It provides two toggle buttons labeled *adjust coords* and *extra material*.

It the first option is selected points belonging to triangles adjacent to boundary voxels will be moved exactly onto the nearest boundary face of the bounding box. In this way the resulting surface appears to be sharply cut off at the boundaries.

The second toggle indicates that triangles adjacent to boundary voxels will be inserted into separate patches. The outer region of these patches will be called *Exterior2*. If the toggle is off no such extra material will be created. Instead, the boundary is assumed to be labeled with 0, which usually corresponds to *Exterior*.

Minimal edge length

Minimal edge length: 0

A non-vanishing value indicates that short edges of the final surface should be contracted in order to increase triangle quality as well as to decrease the number of triangles. The value of the port indicates the minimal allowed edge length relative to the size of a unit grid cell. On default, values between 0 and 0.8 can be entered. Typically, a value of 0.4 already yields good results. However, note that intersections may be introduced during edge contraction. If you want to avoid this try to use the simplification editor. The editor applies some special strategies in order ensure topological consistency.

Action

Action: Triangulate

This port has a single button *Triangulate* which is used to start surface extraction. Depending on the size of the LabelField the algorithm requires up to a minute to finish.

6.116 SurfaceLIC

This module visualizes a vector field defined on an arbitrary triangular surface using line integral convolution (LIC). Alternatively, a 3D vector field projected onto such a surface can be visualized. The LIC algorithm works by convolving a random noise function along field lines tangential to the surface using a piecewise-linear hat filter. In this way for each triangle a small piece of texture is computed and mapped back onto the surface. The final surface texture clearly reveals the directional structure of the surface vector field. A similar 2D algorithm is implemented by the PlanarLIC module.

Click here to execute a script demostrating the *SurfaceLIC* module. Computation of the surface LIC texture may take about half a minute.

Connections

Data `[required]`

Surface for which a LIC texture is to be computed.

VectorField `[required]`

Surface vector field or 3D vector field to be visualized.

ScalarField `[optional]`

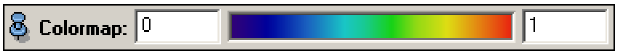
An optional scalar field which can be used for pseudo-coloring.

Colormap `[optional]`

Colormap used for pseudo-coloring. If no colormap is connected the default color of the colormap port will be used.

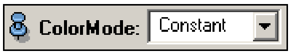
Ports

Colormap


Colormap: 0 1

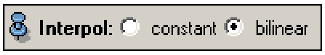
Port to select a colormap.

ColorMode


ColorMode: Constant

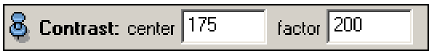
Three different color modes are provided. If *Constant* is selected then a uniform overall base color is used for the LIC texture. This will be the default color of the colormap port or the left-most color of the colormap connected to this port, if any. If *Magitude* is selected then the LIC texture will be colored according to the magnitude of the vector field. Finally, if *Color field* is selected and a scalar field is connected to the module then the LIC texture will be colored according to the values of this scalar field.

Texture Interpolation


Interpol: constant [✓ bilinear]

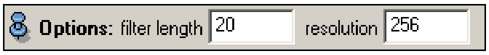
A radio box determining how the triangle textures are being filtered by the underlying OpenGL driver. Possible choices are *constant* or *bilinear* interpolation. Usually, you will not see a big difference unless you zoom up the image very much.

Contrast


Contrast: center 175 factor 200

This port provides two parameters controlling the amount of contrast of the final LIC texture. Input field *center* specifies the average grey value of the texture. Higher values result in brighter images. Input field *factor* determines the width of the grey value histogram, which is of Gaussian type. Higher numbers produce more contrast.

Options


Options: filter length 20 resolution 256

Parameter *filter length* specifies the one-sided length of the triangular filter kernel used for line integral convolution. The larger this value the more coherent the greyscale distribution along the field lines. Often larger values are visually more attractive than smaller ones.

The second input determines the *resolution* of the LIC texture. More precisely, the width of a single texture cell is chosen to be equal to the length of the diagonal of the incoming vector field's bounding box divided by the value of the *resolution* field.

Action

Starts computation of the surface LIC texture. Computation may take a minute or more depending on texture resolution and on the number of triangles of the surface.

Commands

`setAmbientColor <color>`
Allows you to change the ambient color of the surface.

`setDiffuseColor <color>`
Allows you to change the diffuse color of the surface.

`setSpecularColor <color>`
Allows you to change the specular color of the surface.

`setShininess <value>`
Allows you to change the shininess of the surface.

`setCreaseAngle <value>`
Neighboring triangles will share a common vertex normal if the angle between their face normals is smaller than the crease angle. The default value is 60 degrees. This command lets you overwrite this value. Note, that discontinuities will appear if the triangles of the input surface are not oriented in the same way. In order to fix this, use the command `fixOrientation` of the surface.

6.117 SurfaceView

This module allows you to visualize triangular surfaces, i.e., data objects of type Surface. Derived from the generic ViewBase class, the module provides an internal buffer of visible triangles. You can add triangles to this buffer by means of two special option menus. For example, if your surface contains regions FAT and MUSCLE, you may first highlight all triangles separating these two regions by choosing FAT and MUSCLE in port *Materials*. Highlighted triangles are displayed in red wireframe. By pressing button *Add* of port Buffer highlighted triangles can be added to the internal buffer, which causes them to be displayed in their own colors. You may restrict highlighting by means of an adjustable box. In order to resize the box pick one of the green handles at the corners of the box. Highlighted triangles may also be removed from the buffer by pressing button *Remove*. In addition, individual triangles may be removed from the buffer by *shift-clicking* them.

Connections

Data `[required]`
The surface to be visualized.

ColorField `[optional]`
In conjunction with a colormap an optional field can be visualized on top of the surface in pseudo-color mode. The field may be either of type *HxScalarField3* or of type *HxSurfaceScalarField*. In

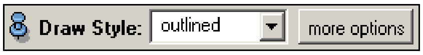
addtion to scalar surface fields also 3 and 4-component surface fields are supported. In this case, the field components are directly interpreted as RGB or RGBA values. The values should range from 0 to 1.

Colormap `[optional]`

The Colormap is used to visualize the data values of a scalar field connected to port *ColorField*.

Ports

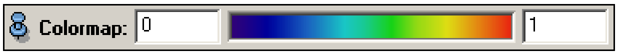
Draw Style



Draw Style: outlined

This port is inherited from the ViewBase class and therefore the description will be found there.

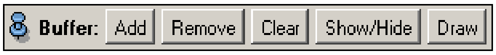
Colormap



Colormap: 0 … 1

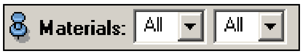
This port becomes visible only if a scalar field has been connected to the ColorField port.

Buffer



This port lets you *add* and *remove* highlighted triangles (being displayed in red wireframe) to an internal buffer. For a further description and for the functionality of each of the port buttons see ViewBase.

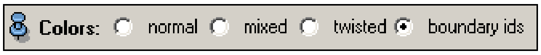
Materials



Materials: All All

This port provides two option menus listing all regions defined in the surface. By setting the menus properly, you can highlight all triangles separating two different regions. Highlighted triangles are displayed in red wireframe. You may restrict the set of highlighted triangles by means of an adjustable box. Use port *Buffer* to add or remove highlighted triangles to the internal buffer.

Colors



Colors: normal mixed twisted [✓ boundary ids]

There are four different color modes:

Normal: Each side of a triangle is colored according to the color of the opposite region.

Mixed: Both sides of a triangle are colored in the same color, which is a mixture of the colors of the two sides in *normal* mode.

Twisted: Each side of a triangle is colored according to the color of the adjacent region.

Boundary id: Color denotes the boundary ids of the triangles. For each boundary id a separate color can be defined in the surface's parameter section. Boundary ids can be set and removed using the surface editor.

BaseTrans

This port is visible if *transparent* has been chosen as the draw style. It allows you to modify the transparency of the surface. For each material an own transparency value can be defined in the surface's parameter section. If base transparency is 0.5, exactly this default transparency is taken. Larger values make all parts of the surface more opaque, smaller values the surface more transparent.

VR-Mode

This port is only available in VR mode (see amiraVR documentation). It allows you to choose how to interact with the surface using the 3D wand. In *query* mode information about the clicked point is displayed similar to what is displayed in standard amira when clicking onto a surface with the middle mouse button. The other modes are *select patches*, *highlight patches*, *select triangles*, and *highlight triangles*.

6.118 TetToHex

The *TetToHex* module converts a tetrahedral grid to a equivalent hexahedral grid. Each tetrahedron is converted to four hexahedra which have the same material ID as the 'parent' tetrahedron. The new hexahedral grid has not the same points set as the original tetrahedral grid. The new points set has more points, the new points being the center of each tetrahedron, the center of each face and the middle point of each edge.

The module also converts the data objects connected to the hexahedral grid.

Connections

Data `[required]`

Tetrahedral grid to be converted.

Ports

Options

Toggles wether the data objects connected to the tetrahedral grid are also converted or not.

Action

Press the *DoIt* button to make the conversion.

6.119 TetraCombine

This module takes two tetrahedral grids as input, puts them together and creates a new combined grid. The module can be used to create an *extended grid* from a patient grid and a grid representing the exterior space including parts of a treatment device or support to which the patient's body is attached. Triangles representing the boundary of the patient's body may be present in both input grids. *Combine* provides an option to remove any duplicated triangles and vertices from its output. The order in which the input grids are combined does not matter.

To make sure that both grids being combined fit exactly together, you can attach a GridVolume module to the combined grid. When invoking that module, all *exterior* triangles of the grid are highlighted, i.e., all triangles which are incident to only one tetrahedron. The displayed triangles should all be located at the outer boundary of the combined grid, not in its interior.

Connections

GridA `[required]`

The first input grid.

GridB `[required]`

The second input grid.

Ports

Options

Options: [✓ Remove duplicated]

Remove duplicated causes all duplicated points and triangles to be removed from the output.

Boundary

Boundary: [✓ Mark exterior faces] [✓ Check exterior faces]

Mark exterior faces sets `boundaryConditionId = 1` for all exterior faces. BoundaryConditionIds may be useful if a numerical simulation shall be performed on the resulting grid.

Check exterior faces checks whether all exterior triangles lie on a sphere around the center of the bounding box of the grid.

Action

The *DoIt* button triggers computation.

6.120 TetraGen

The *TetraGen* module creates a *volumetric tetrahedral grid*. Its input is a description of the 3D geometry by triangulated surfaces. The *advancing front method* is applied to fill each region defined by the surface data with tetrahedra. Tetrahedron generation can be performed *on-line* or as a *batch job*.

There are some 'reserved region names' for the exterior region that should not be filled with tetrahedra (otherwise the grid would extend to infinity): no name at all, 'Outside', and all names starting with 'Exterior'. If you choose a different name for the exterior region, tetrahedron generation will fail. Currently the SurfaceGen module creates correct names ('Exterior' and 'Exterior2'), but the module IvToSurface does not. You must edit the region names before applying TetraGen to a surface created by the latter module.

Connections

Data `[required]`

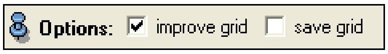
The input surface file (suffix `surf`).

Ports

Region

The menu of this port allows you to specify whether tetrahedra should be generated for all regions or just for one selected region. The latter choice is needed for testing purposes only.

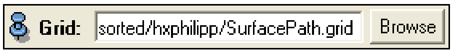
Options


Options: [✓] improve grid [ ] save grid

If toggle 'improve grid' is set, the quality of the resulting tetrahedral grid will be improved in a post-processing step. A combined smoothing will be applied which includes moving inner vertices and flipping inner edges or faces of the grid.

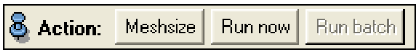
If toggle 'save grid' is set, an additional port *Grid* will occur where you can enter a filename. The resulting tetrahedral grid will be automatically saved under that filename. This toggle must be set if tetrahedron generation shall be performed as a batch job.

Grid


Grid: sorted/hxphilipp/SurfacePath.grid

This port is only visible if toggle 'save grid' at port *Options* is set. Here you can define the filename for the resulting tetrahedral grid. We recommend to include a suffix `grid` into the filename.

Action



If you press the *Meshsize* button an editor window appears. It allows you to define a desired mesh size for each region. Note that there are predefined values for some materials in amira's material database. Check wether these values are appropriate for your application.

After you have setup the simulation, you can commit it by pressing the *Run batch* or the *Run now* button. If the file specified at port *Grid* did already exist, a warning message is issued. If you don't want to overwrite the file, press *Cancel* and change the filename.

If you press the *Run batch* button, the *job dialog* window appears, showing the status of the job queue. If you press the *Start* button, the first pending job of the queue starts running.

If you press the *Run now* button, tetrahedron generation will be performed *on-line*. This may take some time for large surfaces. The progress bar informs you about the current region and which part of its volume is already filled with tetrahedra.

6.121 TetraQuality

The *TetraQuality* module creates a histogram of qualities for tetrahedral volume grids, e.g., tetrahedral patient models. For this purpose it has to be attached to a Grid Volume module. Quality is calculated for all tetrahedra selected by that module. On default, the histogram is shown with a logarithmic scale to direct the focus on the tetrahedra with worst quality.

Connections

Data `[required]`

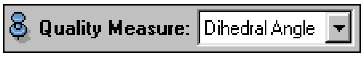
The tetrahedral volume.

GridVolume `[required]`

A Grid Volume module that selects the tetrahedra for which the quality is calculated.

Ports

Quality Measure



This option menu lets you select between different quality measures:

- **Diameter Ratio:** Ratio of diameters of circumscribed and inscribed sphere. The optimal (minimal) value is 3.

- **Aspect Ratio:** Aspect ratio = 3 / diameter ratio. The optimal (maximal) value is 1.
- **Dihedral Angle:** For each tetrahedron edge the dihedral angle is defined as the angle between its adjacent faces. For an equilateral tetrahedron all dihedral angles are about 70 degrees.
- **Solid Angle:** For each tetrahedron vertex the solid angle is defined as the part of the unit sphere which is occupied by the tetrahedron. For an equilateral tetrahedron all solid angles are about 30 degrees.
- **Edge Length:** If you choose this measure, a histogram of edge lengths is created for all selected tetrahedra.

Select Angle

Select Angle: All

If the quality measure is dihedral or solid angle, you can select whether a histogram is created for

- all angles,
- the minimal angle,
- the maximal angle, or
- minimal and maximal angle

of each tetrahedron.

Samples

Samples: 180

This slider lets you select the number of samples for the histogram. Normally, the default values should be adequate.

Histogram

Histogram: ☐

If you select this toggle, a plot window appears showing the histogram of qualities for all tetrahedra selected by the *Grid Volume* module. If no tetrahedra have been selected, the plot window will not be shown.

6.122 TimeSeriesControl

This module is created automatically if files are imported via the *Load Time Series...* option of the main window's *File* menu. The module assumes that all files selected in the file browser represent the same data object at different time steps. Instead of loading all files at once a slider is provided allowing the user to select the current time step. Whenever a new time step is loaded data objects associated with a previous time step are replaced. The replacement is performed in such a way that connections to down-stream modules are retained.

The time series module is also able to linearly interpolate between subsequent time steps. However, this only works for certain types of data objects, namely surfaces, tetrahedral grids, hexahedral grids, fields defined on surfaces, tetrahedral or hexahedral grids, or data objects derived from the vertex set base class. In addition, it is required that for each time step the same number of data objects is created, that corresponding data objects are created in the same order in all time steps, and that corresponding data objects have the same number of vertices or elements. For example, you cannot easily interpolate between surfaces with a different number of triangles (Amira provides other modules to support this). If multiple data objects are created for each time step interpolation of particular objects can be suppressed by switching off the orange viewer toggle of these objects.

In addition to the step number the module can also display the physical time associated with each time step, provided the physical time is specified as a *Time* parameter for each data object. It is required that the physical times of subsequent time steps are monotonously increasing. In order to use physical time mode it the relevant time steps must be loaded in index mode first.

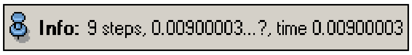
Connections

Time `[optional]`

Connection to a global time object.

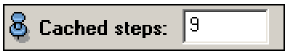
Ports

Info


Info: 9 steps, 0.00900003...?, time 0.00900003

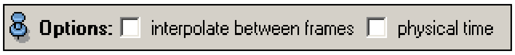
This port displays the total number of time steps. If the data objects provide a physical time in a *Time* parameter, the physical time range is displayed too. If the last time step has not yet been loaded, a question mark is printed instead of the maximum time value. In addition, if a physical time is provided the current physical time or the current time step index is displayed, depending on the value of the *physical time toggle*.

Cached steps


Cached steps: 9

Allows to adjust the cache size. On default every time step is cached, i.e., the number of cached steps is equal to the total number of time steps. If the number of cached steps is zero the cache is disabled. Data objects associated with a previous time step are deleted before a new time step is loaded. Interpolation mode requires a cache size of two, i.e., in addition to the current interpolated time step at least two additional time steps have to be stored in memory.

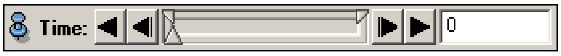
Options


Options: ☐ interpolate between frames ☐ physical time

The first toggle is used to activate *interpolation mode*. In this mode fractional time steps can be specified and a linear interpolation between two subsequent time steps is performed. Note that interpolation can only be performed if certain requirements are met (details are descibed above).

The second toggle is used to activate *physical time mode*. Physical time mode is only available if the loaded data objects provide a *Time* parameter. In physical time mode the time slider displays the physical time instead of the current time step.

Time



Specifies the current time step or the current physical time. The port provides a popup menu (right mouse button click) which can be used to configure settings like animation mode or subrange interval. The two outer buttons allow to automatically animate the time step or the physical time in forward or backward direction. Animation speed in physical time mode can be controlled via the increment value in the configure dialog.

6.123 TissueStatistics

This module takes a uniform or stacked label field as well as an optional scalar field as input and computes some statistical quantities for the regions defined in the label field.

If the quantities are computer per material or per connected component, the different columns of the spreadsheet have the following meaning:

- **Number:** Enumerates all materials (regions) of the label field.
- **Material:** Denotes which material (region) is stored in a row.
- **Count:** Number of voxels contained in a region.
- **Volume:** Number of voxels times size of a single voxel.
- **CenterX:** X-coordinate of the region’s center.
- **CenterY:** Y-coordinate of the region’s center.
- **CenterZ:** Z-coordinate of the region’s center.

If an additional scalar field is connected to the module, then four more columnns will be generated:

- **Mean:** Mean value of the field in a particular region.
- **Deviation:** Standard deviation of the field in a region.
- **Min:** Min value of the field in a particular region.
- **Max:** Max value of the field in a particular region.

The scalar field will be evaluated at the center of each voxel.

Connections

Data `[required]`

Label field defining the regions.

Field `[optional]`

Optional scalar field, for example the image data the label field is associated with. If the scalar field has the same dimensions as the label field the voxels are accessed directly. Otherwise, a lookup using the field's native interpolation method is performed.

Voi `[optional]`

Optional label field used as a volume of interest. It is only used if the statistics are set to *Volume per VOI*.

Ports

Select

The quantities can be computed in different modes:

- per material (Material)
- per connected components (Region)
- per slice (Volume per Slice)
- per material in another attached label field (Volume per VOI)

The results are stored in a spreadsheet object.

For example, in a medical application you may have two kidneys both assigned to the same material. In the *Material* mode, the two kidneys will be interpreted as one object, in the *Region* mode as two objects.

In mode *Volume per slice* the slices are stored as rows in the spreadsheet. The columns are labeled by the materials and the cells contain the volume of these materials in each slice. The scalar field is ignored.

In mode *Volume per VOI* (Volume of Interest) another label field has to be attached to the connection named Voi. The rows of the spreadsheet are labeled with the materials in this label field. The columns are labeled with the material of the main label field and the cells contain the volume of a material in the main label field at positions where the Voi contains the material indicated by the row.

Pixels

This port allows to exclude small regions from the statistics.

Action

Action: Do It

Triggers the computation.

6.124 TriangleQuality

The *TriangleQuality* module computes the triangle qualities for a triangular surface. You can attach to the result a SurfaceView module to visualize the triangle qualities or a *Histogram* module to plot a histogram of triangle qualities. You can also use the *TriangleQuality* module in conjunction with the Surface Editor to detect the worst triangles and manually repair them.

Connections

Data `[required]`

A triangular surface.

Ports

Average edge length

Average edge length: 0.0126384

Displays the average length of triangle edges in the input.

Output

Output: R/r

This option menu lets you select between different quality measures:

- **R / r:** Ratio of diameters of circumscribed and inscribed circle. The optimal (minimal) value is 2.
- **Largest Angle:** Computes the largest angle [degree] of each triangle. For numerical applications obtuse angles above 90 degree should be avoided.
- **Dihed Angle:** Computes the dihedral angle [degree] for each pair of adjacent triangles. Small dihedral angles below 5 - 10 degree should be avoided.
- **Triangle Number:** For testing purposes only: assigns the triangle number to each triangle.

Action

Action: Do It

Press the *DoIt* button to start calculation of triangle qualities.

6.125 VRML_Export

This module enables you to export a triangular surface and optionally the 3D dataset from which the surface is derived into a VRML scene. The scene consists of a coordinate system and some little animations: the 3D dataset is displayed by three orthoslices. To get an idea how the basic 3D dataset is related to the derived surface you can set the orthogonal slices via sliders, scrolling through the triangular surface. Moreover, the materials of the surface are clickable objects, which means that they can be hidden or shown by a mouse click.

You can publish such a VRML scene directly on the web, just put a link to it into your homepage. To view the file by yourself, make sure that a VRML viewer plugin like *cosmoplayer (version 2.1 or newer)* plugin from SGI is installed for your web browser.

If connected to the surface view module, a special simple export mode becomes avalilable which produces a VRML-scene containing only a single IndexedFaseSet node. This is necessary e.g., for some 3D-printers with limited VRML parsing abilities.

If no surface is connected, only the slices may exported to VRML.

Connections

Data `[required]`

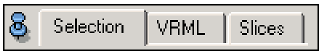
The surface to be inserted into the VRML scene could either be a amira Surface dataset or the surface view module.

Image `[optional]`

A scalar field of type *UniformScalarField3* or of type *UniformColorField3* is needed to export the orthoslices. If ommited, no slices are displayed in the VRML scene.

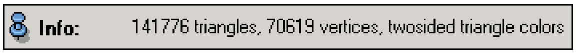
Ports

Tabbar



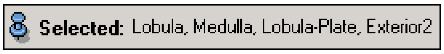
Tab bar to navigate through the user interface sections. The *Slices* tab remains disabled as long no image data is connected.

Info


Info: 141776 triangles, 70619 vertices, twosided triangle colors

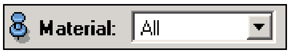
In simple export mode, the coloring mode and the number of primitives is shown here. (only avail. in simple mode)

Selected


Selected: Lobula, Medulla, Lobula-Plate, Exterior2

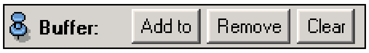
This port shows the selected materials. (not avail. in simple mode)

Material


Material: All

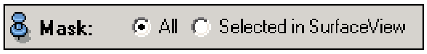
With this port you can choose a material in order to add or to remove it from the current selection. (not avail. in simple mode)

Buffer



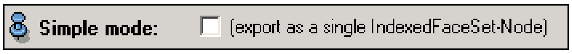
With this port the selection of materials can be modified. To remove a material from the list, choose this material from the *Material* menu and hit *Remove*. Similarily you can use *Add*. *Clear* clears the selection list. Initially, all materials are selected. (not avail. in simple mode)

Mask


Mask: [✓ All] Selected in SurfaceView

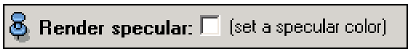
If the simple export mode was chosen, no selection by material is provided. Instead one can use the selection mechanism of the connected surface view module. Toggle *Selected* to export only the primitives selected there, i.e., the visible ones. (only avail. in simple mode)

Simple mode


Simple mode: ☐ (export as a single IndexedFaceSet-Node)

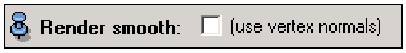
If connected to the surface view module, a special simple export mode becomes avalilable which produces a VRML-scene containing only a single IndexedFaseSet node. This is necessary e.g., for some 3D-printers with limited VRML parsing abilities.

Render specular


Render specular: ☐ (set a specular color)

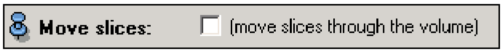
Adding a specular color to the scene which makes the surface more look like plastic.

Render smooth


Render smooth: ☐ (use vertex normals)

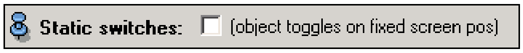
Gives the scene some smoother look by using vertex normals.

Move slices


Move slices: ☐ (move slices through the volume)

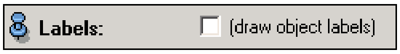
If clicked, the slices will actually be moved through the volume, when the sliders are operated. Otherwise, only the textures mapped onto the slices are cycled.

Static switches


Static switches: ☐ (object toggles on fixed screen pos)

If clicked, the small objects used to switch on and off the surface parts will be fixed on the screen. Note that this requires at least *CosmoPlayer 2.1* or similar.

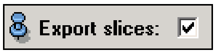
Labels



Labels: ☐ (draw object labels)

Toggle this for displaying the material names near the material toggles in the VRML scene.

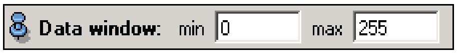
Export slices



Export slices: ☑

If toggled, the slice images taken from the image data are exported. If no image data is connected no slices are exported regardless of this toggle.

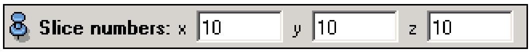
Data window



Data window: min 0 max 255

Needed to map data values of input field to gray values. See module OrthoSlice for explanation.

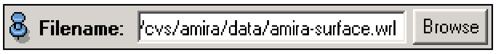
Slice numbers



Slice numbers: x 10 y 10 z 10

Enter the numbers of slices per direction you want to be selectable in the VRML scene. Only available on SGI and when image data is connected.

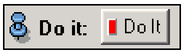
Filename



Filename: /cvs/amira/data/amira-surface.wrl

Name of the File, the VRML code is written to. If export of slices is selected, the slice images are stored to the same directory.

Do it



Do it:

Hit this button to start the export.

6.126 VectorProbe

The *VectorProbe* module allows you to interactively investigate a 3D vector field by moving around a dynamic vector field probe. The probe displays certain quantities associated to the first order derivative of the field in an intuitive way. This kind of probe has been originally proposed by W.C. de Leeuw and J.J. van Wijk in *A Probe for Local Flow Field Visualization, Proceedings of Visualization'93, pp. 39-45*. It looks as follows:

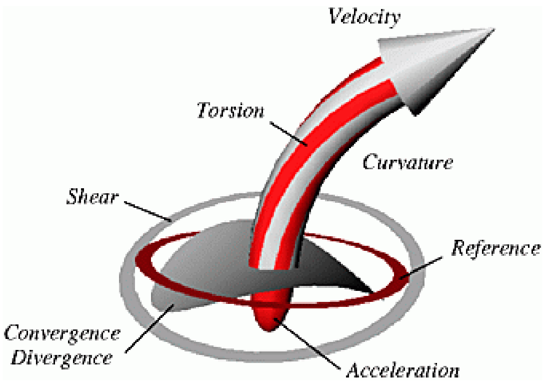


Figure 6.5: Components of the vector fi eld probe.

Connections

Data `[required]`

The 3D vector field to be visualized.

Ports

Dragger

Shows or hides the dragger and the vector field probe attached to it. The dragger provides a cylinder handle and a square plate handle. The cylinder handle allows you to translate the icon along the center axis. The orientation of the cylinder can be changed using the `[Ctrl]` key while the mouse is located somewhere over the dragger.

Buffer

Adds the current vector field probe to an internal buffer. In this way multiple probes can be displayed at once.

Scale

Scales the overall size of the vector field probe.

Length

Adjusts the length of the probe's arrow part.

6.127 Vectors

This is a so-called *overlay module* which can be attached to any module defining a cutting plane, e.g. OrthoSlice or ArbitraryCut. Inside this plane a 3D vector field can be visualized using a regular array of vector arrows.

Connections

Data `[required]`

The 3D vector field to be visualized.

Module `[required]`

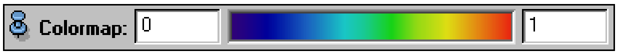
The module which defines the cutting plane where the arrows are placed.

Colormap `[optional]`

An optional colormap used for pseudo-coloring.

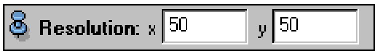
Ports

Colormap



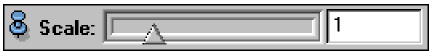
Port to select a colormap.

Resolution


Resolution: x 50 y 50

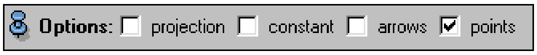
Provides two text inputs defining the resolution of the regular array of vector arrows in the plane's local x- and y-direction. The larger these values are the more arrows are displayed.

Scale


Scale: 1

Scaling factor used to control the length of the vector arrows.

Options


Options: ☐ projection ☐ constant ☐ arrows ☑ points

This port provides the following toggle buttons.

Projection: If this option is set then 3D vectors are projected into the current plane. Otherwise, the arrows will indicate the true direction of the vector field.

Constant: If this option is set then all arrows will be of equal length. Otherwise, the length of the arrows is chosen to be proportional to vector magnitude.

Arrows: If this option is set then true arrows will be displayed. Otherwise, only simple line segments will be drawn.

Points: If this option is set then a little dot will be drawn at the bottom of an arrow. This is useful to highlight a sampling point at locations where the vector magnitude is so low that the arrow vanishes completely.

Colorize

An option menu allowing to control how arrows are colored. In order to see an effect a colormap must be connected to the module. If *Magnitude* is chosen then the vector magnitude is used to lookup a color from the colormap. If *Normal component* is chosen then color denotes the signed length of the vector component perpendicular to the cutting plane. This length is positive if the vector points upwards or negative if the vector points downwards. Finally, if *Parallel component* is chosen then *color* denotes the length of the vector component tangential to the plane. This length will always be greater or equal than zero.

Phase

This port will only be visible if a complex-valued vector field is connected to the module. It provides a phase slider controlling which part of the complex 3D vectors is visualized by the arrows. A value of 0 degree corresponds to the real part, while a value of 90 degrees corresponds to the imaginary part. The display can be animated with respect to the phase by the *cycle* button, this way polarization properties of the field can be revealed or wave phenomena become visible.

Commands

`setLineWidth <value>`

Allows to change the line width of the arrows. By default arrows are drawn two pixels wide.

6.128 Vectors (Tetrahedra)

This display module can be attached to a GridVolume or a GridBoundary module as well as to a 3D vector field defined on a tetrahedral grid. In the latter case all vectors are displayed otherwise the vector field is displayed on the surface nodes or on all nodes of the selected tetrahedra (*GridVolume*) resp. triangles (*GridBoundary*).

The vector representation can be done using simple *lines* or *arrows*. The vector lines/arrows can be coloured using a colormap taking the magnitude of the vectorfield in to account.

Connections

Data `[required]`

The 3D tetrahedral vector field to be visualized.

PortModule `[required]`

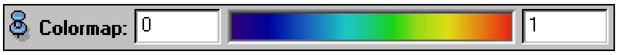
The GridVolume resp. GridBoundary master module.

Colormap `[optional]`

An optional colormap used for colouring the magnitude of the vectors.

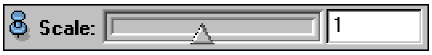
Ports

Colormap


Colormap: 0 1

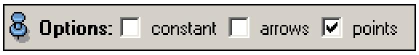
Choose a colormap to control how lines/arrows are coloured. In order to see an effect the colormap must be connected to the module. The vector magnitude is used to lookup a color from the colormap.

Scale


Scale: 1

Scaling factor used to control the length of the vector lines/arrows.

Options

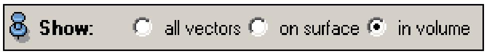

Options: ☐ constant ☐ arrows ☑ points

Constant: If this option is set then all lines/arrows will be of equal length. Otherwise, the length is chosen to be proportional to vector magnitude.

Arrows: Choose between simple lines and arrows

Points: If this option is set then a little dot will be drawn at the bottom of an arrow. This is useful to highlight a point at locations where the vector magnitude is so low that the arrow vanishes completely.

Show

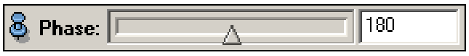

Show: ○ all vectors ○ on surface ● in volume

All Vectors: If this radio option is on the vectors from all nodes are displayed.

On Surface: Only those vectors are displayed which lie on the surface of the selected boundaries.

In Volume: All vectors of all selected tetrahedrons are shown.

Phase



This port will only be visible if a complex-valued vector field is connected to the module. It provides a phase slider controlling which part of the complex 3D vectors is visualized by the arrows. A value of 0 degree corresponds to the real part, while a value of 90 degrees corresponds to the imaginary part. The display can be animated with respect to the phase by the *cycle* button, this way polarization properties of the field can be revealed or wave phenomena become visible.

Commands

`setLineWidth <value>`

Allows to change the line width of the arrows. By default arrows are drawn two pixels wide.

`setPointSize <value>`

Allows to change the size of the points at the bottom of the arrows. By default points are drawn two pixels wide.

`setLogScale 0|1`

Switches logarithmic scaling on or off.

6.129 Vectors/Normals (Surface)

This is a multifunctional display module having the following applications:

- attached to a SurfaceView module, it shows the normals on the Surface displayed by the module. Only the normals corresponding to selected patches/triangles are displayed; the normal binding in the SurfaceView module (triangle normals, vertex normals or direct normals) is respected.

- attached to a 3D vector field defined on a Surface,it visualizes the whole vector field

- connected to both SurfaceView module showing a Surface and a 3D vector field defined *on the same* Surface, it displays the vector field only in the selected regions of the surface.

The representation can be done using simple *lines* or *arrows*. In the case of vector fields it can be chosen between a constant length or a magnitude - dependent length of the lines/arrows. The lines are coloured using the colormap.

Connections

Data `[optional]`

The 3D vector field to be visualized. If there is no vector field, the normals on surface are displayed.

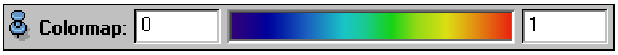
Surface `[optional]`

The SurfaceView master module.

Colormap `[optional]`
An optional colormap used for coloring the magnitude of the vectors.

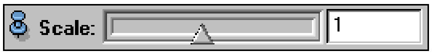
Ports

Colormap



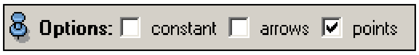
Choose a colormap to control how lines/arrows are colored. In order to see an effect the colormap must be connected to the module. The vector magnitude is used to lookup a color from the colormap.

Scale



Scaling factor used to control the length of the vector lines/arrows.

Options



Options: [ ] constant [ ] arrows [✓ points]

Constant: (Only for vector fields) If this option is set then all lines/arrows will be of equal length. Otherwise, the length is chosen to be proportional to vector magnitude.

Arrows: Choose between simple lines and arrows

Points: (Only for vector fields) If this option is set then a little dot will be drawn at the bottom of an arrow. This is useful to highlight a point at locations where the vector magnitude is so low that the arrow vanishes completely.

Commands

`setLineWidth <value>`
Allows to change the line width of the arrows. By default arrows are drawn two pixels wide.

`setPointSize <value>`
Allows to change the size of the points at the bottom of the arrows. By default points are drawn two pixels wide.

`setArrowSize <value>`
Allows to change the size of the arrows.

6.130 Vertex Morph

This module takes two Vertex Set objects as input, e.g. two surfaces or two tetrahedral grids, and computes an output surface by linearily interpolating the vertex positions. This can be used to create a smooth transition between the two objects. Note, that the two input data sets must be related in order to produce meaningful results. For example, the second input could actually be a copy of the first one with an applied transformation.

Connections

Input1 `[required]`

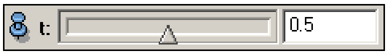
First input data set. This data set is duplicated to produce the output.

Input2 `[required]`

Second input data set, must have at least as many points as the first input.

Ports

t



Interpolation parameter. For t=0 the output will be identical to the first input. For t=1 output will be second input.

6.131 Vertex View

This module allows you to visualize arbitrary *vertex sets*. Vertex sets occur as part of objects of other types, such as Surfaces, Tetrahedral Grids, Line Sets or Molecules. The vertices can be displayed in three different modes: *spheres*, *plates* and *points*. The vertices may be colored according to a scalar field and a color map. Alternatively, colors may also be defined via the command interface of the *VertexView* module. Furthermore an internal buffer exists that allows you to view only those vertices that are of interest to you.

Connections

Data `[required]`

The data object from which the vertex set is read.

ColorField `[optional]`

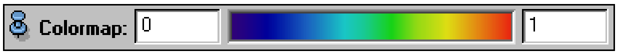
3D scalar field which is used along with a color map to color the vertices according to the value of the scalar field at the position of a vertex.

Colormap `[optional]`

Used to color the vertices in connection with the *ColorField*.

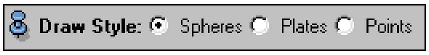
Ports

Colormap


Colormap: 0 1

Port to select a colormap.

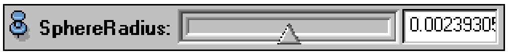
Draw Style


Draw Style: [✓ Spheres] Plates Points

Vertices may be drawn in three different styles:

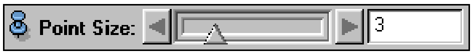
- *Spheres:* Points are drawn as triangulated spheres with equal radius.
- *Plates:* Points are drawn as quadrats with mapped-on image of a sphere.
- *Points:* Points are drawn as points. The size of the points does not differ according to the distance of the vertex from the viewer; they all have the same size.

SphereRadius


SphereRadius: 0.0023930

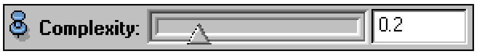
Specifies unique radius for all spheres. This port is only visible if *Spheres* or *Plates* is chosen as draw style.

Point Size


Point Size: 3

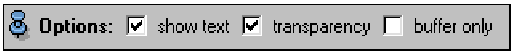
Size of points for draw style *Points*. Only visible in *Points* mode.

Complexity


Complexity: 0.2

Set complexity of displayed spheres. Reducing the complexity leads to coarser spheres and improved rendering performance. If draw style is set to *plates* the complexity controls the size of the texture maps containing the sphere's images. The smallest texture size is 32x32, the biggest is 512x512. The port is not visible for draw style *points*.

Options:

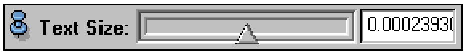

Options: [✓ show text] [✓ transparency] buffer only

A toggle list of options.

- *Show text:* If selected vertex numbers are drawn. This might be quite slow, especially for large vertex sets. Alternatively, you may select a vertex by clicking on it. Then its number is printed in the console window.

- *Transparency:* Optionally, the spheres and plates may be drawn transparent. Note that the vertices are not sorted along the z-axis. Thus the appearance of the vertex set may look flawed.
- *Buffer only:* The default is to display all spheres of the vertex set. By clicking this toggle you can display only those vertices that were previously added to the buffer.

Text Size



Specify textsize if *show text* is active.

Transparency

Allows you to specify the degree of transparency of the spheres.

Buffer:

A list of buttons that allow to manipulate the internal buffer. In order to actually see the buffer content activate the *buffer only* option.

- *Add:* adds selected vertices to buffer.
- *Remove:* removes selected vertices from buffer.
- *Clear:* removes all vertices from buffer.
- *Show/Hide Box:* controls box to select vertices.
- *Invert:* exchanges buffer content.

Commands

`setHighlightColor <red> <green> <blue>`

Set the color that is used to highlight selected spheres.

`setDefaultColor <red> <green> <blue>`

Set default color.

`setColorHighlighted <red> <green> <blue>`

Color all highlighted spheres.

`setColor [<first-vertex-number> [<last-vertex-number>]] <red> <green> <blue>`

Set color for all spheres from *first-vertex-number* to *last-vertex-number*. If *last-vertex-number* is omitted the color is set for the sphere with index *first-vertex-number*. If *first-vertex-number* is omitted too, the color for all spheres is set. *Red, green, and blue* range from 0 to 1.

`highlight <first-vertex-number> [<last-vertex-number>]`
Highlight spheres from *first-vertex-number* to *last-vertex-number*.

`unhighlightAll`
Unhighlight all spheres.

`addToBuffer <first-vertex-number> [<last-vertex-number>]`
Add all spheres ranging from *first-vertex-number* to *last-vertex-number* to buffer. If *last-vertex-number* is omitted, only the sphere with index *first-vertex-number* is added.

`removeFromBuffer <first-vertex-number> [<last-vertex-number>]`
Remove all spheres ranging from *first-vertex-number* to *last-vertex-number* from buffer. If *last-vertex-number* is omitted, only the sphere with index *first-vertex-number* is removed.

`getNumVertices`
Print number of vertices.

`getCoords <vertex-number>`
Print coordinates of vertex with number *vertex-number*

`setTextSize <size>`
Set size with which the text is displayed.

`setTextOffset <x> <y> <z>`
Set offset which is added to the text position.

`setTextColor <red> <green> <blue>`
Set text color.

`updateTextures`
To both *plates* and *spheres* textures are applied to make the spheres look smoother. If the direction from which the light comes changes, those textures need to be recomputed. Updating is invoked by this command only.

6.132 VertexDiff

The module computes the displacement field. A vector field on a surface is computed by the difference of the vertex positions of corresponding vertices in both surfaces.

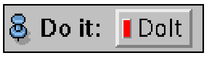
Connections

Data `[required]`
Surface 1

Surface2 `[required]`
Surface 2

Ports

Do it



Compute the displacement field.

6.133 VertexShift

The module displaces the vertices of a surface. The displacements are given by one or more vector fields. The computed vector fields automatically connect to the domain surface.

Connections

Data `[required]`

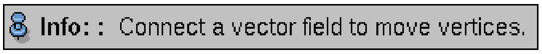
The surface to be displaced

Vector field `[required]`

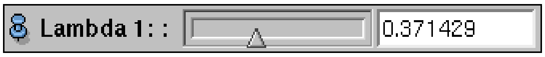
The displacement field

Ports

Info

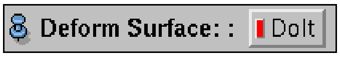

Info: : Connect a vector field to move vertices.

Lambda 1



For each displacement field, a parameter *lambda* controls the displacement. When the mouse is moved over the slider, an icon indicates which field the slider belongs to.

Deform Surface:



Compute the displaced surface.

6.134 ViewBase

This module is the base class for several other amira modules displaying a set of triangles, like Isosurface, SurfaceView, GridVolume, and others. *ViewBase* is not useful on its own but provides special features common to all derived modules. In particular, these features comprise the following:

- A dedicated port allowing the user to modify the draw style of a triangular surface in an easy and consistent way. All modules derived from *ViewBase* thus have a similar GUI. Among the supported draw styles is a physically correct transparency mode.
- A generic buffer concept allowing the user to select triangles by means of an Open Inventor tab-box dragger. Only triangles added to the internal buffer will be displayed. Thus, complex surfaces may be decomposed into smaller pieces.
- An arbitrary 3D scalar field may be visualized on top of the triangular surface by means of pseudo-coloring. Pseudo-coloring may be achieved via Gouraud shading (mapped vertex colors will be interpolated) or, more accurately, via texture mapping (vertex values will be interpolated linearly and mapped to color afterwards).
- The set of triangles currently being visible can be converted automatically into a Surface object. This is useful for example in order to post-process isosurfaces or to extract parts from a bigger surface.
- Common Tcl-commands allow to control many parameters of modules derived from *ViewBase*. This includes line width, outline color, highlight color, specular color, shininess, alpha mode, normal binding, and more.

Connections

Color Field

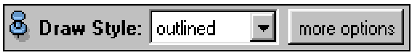
Arbitrary scalar field to be visualized via pseudo-coloring. The color field will be evaluated at the surface's vertex positions and the vertex color will be set approriately.

Colormap

Colormap used for pseudo-coloring. To connect the port to a colormap use the popup menu under the right mouse button. To change the port's default color click it with the left mouse button. See also Colormap.

Ports

Draw Style


Draw Style: outlined

This port determines the draw style of the surface. Five major styles may be selected via an option menu:

Outlined: Opaque shaded display with edges superimposed.

Shaded: Opaque shaded display without edges being visible.

Lines: Shaded wireframe display.

Points: Triangle vertices only.

Transparent: Semi-transparent display.

Transparent mode implies physically correct transparencies, i.e., triangles appear more opaque if they are viewed under a small angle. It also implies approximate depth-sorting, i.e., triangles are roughly rendered from back to front in order to obtain correct blending results. Note, that in some cases visual artifacts may occur for long and thin triangles, for self-intersecting surfaces, or for multiple semi-transparent surfaces being displayed simultaneously.

The drawstyle may be fine-tuned by means of an additional popup menu which is activated by clicking on the button labeled *more options*. This menu contains the following items:

Shaded: Enables or disables specular highlights. Specular color and shininess may be changed via the Tcl-command interface.

Gouraud: Indicates that if a color field is connected pseudo-coloring is performed via Gouraud shading. First, colors are looked up at the traingles' vertices, then these colors are interpolated inside the triangles.

Texture: Indicates that if a color field is connected pseudo-coloring is performed via texture mapping. The color field's values are interpolated linearly before color lookup is performed. In this mode no specular colors can be used.

Opaque: All triangles will be rendered opaque.

Const alpha: Opacity values of a triangle will be taken as is. Usually, if no pseudo-coloring is done, all parts of the surface will have equal opacity.

Fancy alpha: Enables physically correct transparencies. The triangle's opacity values will be modified according to their orientation with respect to the viewing direction. Causes the silhouette of the surface to be fully opaque, thus enhancing perception for very transparent surfaces.

Depth sorting: Enables approximate depth sorting. The triangle's centers are presorted along the major coordinates axes.

Create surface: Creates a Surface object containing the set of currently visible triangles.

The following items are only present for a subset of derived modules, e.g., Isosurface or SurfaceView:

Both faces: Indicates that triangles are rendered both from back and front.

Front face: Enables back face culling. Increases rendering speed but may lead to artifacts for non-closed surfaces.

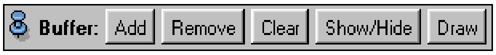
Back face: Enables front face culling. Increases rendering speed but may lead to artifacts for non-closed surfaces.

Triangle normals: Enables per-triangle normals. Shading will be discontinuous at the triangles' edges.

Vertex normals: Enables per-vertex normals. An average normal is computed for all triangle vertices.

Direct normals: An averaged normal is computed for every vertex a triangle. No averaging is performed if two neighboring triangles form an angle bigger than the crease angle set via the Tcl-command `setCreaseAngle`. The default is 30 degrees.

Buffer



This port can be used to modify the list of currently visible triangles. All triangles being visible are stored in an internal buffer. You may add or remove triangles from this buffer via an Open Inventor tab-box dragger. Triangles selected by this dragger will be highlighted, i.e., displayed in red wireframe. If the dragger is not visible, click on one of the buttons to activate it.

Add: Adds highlighted triangles to the buffer. If the Shift key is held down while pushing the button the buffer will be cleared before adding. The Shift key is especially useful in conjunction with the *Draw* selection.

Remove: Removes highlighted triangles from the buffer.

Clear: Removes all triangles from the buffer.

Show/hide: Shows or hides the tab-box dragger without modifiying the internal buffer.

Draw: Activates a lasso style selection mechanism: Using the mouse you can draw a curve in the viewer. All triangles within this curve will be highlighted. If CTRL is pressed while drawing, the triangles within the curve are un-highlighted.

Commands

`createSurface [name]`

Converts set of visible triangles into a surface.

`setAlphaMode {opaque|constant|fancy}`

Triangles may be drawn either opaque or transparent. Two transparent modes are possible: with a constant alpha value *(constant)* or an alpha value varying according to triangle normal *(fancy)*.

`setNormalBinding {perTriangle|perVertex}`

Normals can be bound either per triangle or per vertex. In the first mode the triangles appear flat.

`setPointSize size`

Sets the size of points.

`setLineWidth width`

Sets the width of lines.

`setOutlineColor color`

Sets color of lines in outlined mode.

`setHighlightColor color`

Sets wireframe color of selected triangles.

`setEmissiveColor color`

Sets the emissive color of the surface.

`setSpecularColor color`
Sets the specular color of the surface. This will only take effect if specular lighting has been enabled in port *Draw Style*.

`setShininess shininess`
Sets the shininess of the surface.

`showBox`
Shows box that is used for selecting triangles.

`hideBox`
Hides box that is used for selecting triangles.

6.135 VolPro-1000

This module is available for Windows only. If you need it on other platforms, contact us at `amira@indeed3d.de`.

Note that this module requires special hardware.

This module provides high quality real time volume rendering by exploiting the TetraRecon Volume-Pro 1000 board (compare *www.terarecon.com*). The volume rendering algorithm is implemented in hardware on the VolumePro board. Hence, the rendering can be performed at interactive frame rates even on low-end machines. For general information about volume rendering see the description of the Voltex module.

The *VolPro-1000* module exploits most of the features provided by the VolumePro 1000 system, such as transfer functions, lighting, super sampling, several blend and modulation modes, cropping and cut planes in the three main orientations. Rendering can be combined with other polygonal geometry, as long as no transparent polygons are involved.

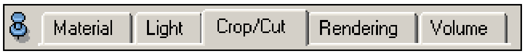
Connections

Data `[required]`
The scalar field to be visualized. The module is also able to handle objects of type LargeDiskData.

Colormap `[optional]`
The transfer function represented by a color map.

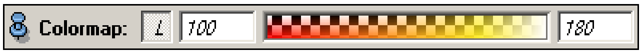
Ports

Tabbar



This port facilitates the navigation in the user interface by grouping the ports.

Colormap



Colormap: 100 180

This port allows you to select one of the predefined color maps and to specify a range of the scalar values which the colors of the color map should be mapped to.

Alpha scale

Alpha scale: 1

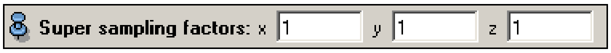
This value scales the opacity of the object.

Super sampling space

Super sampling space: object [✓ camera]

Supersampling is a technique for improving the quality of the rendered image. This port allows you to choose among two supersampling spaces, i.e. camera and object space. Supersampling in object space is especially useful if the distance between the voxels varies strongly along the three main axes. At rendering time object space supersampling factors are transformed into camera space factors. Supersampling in camera space in the x and y directions results in more samples in the base plane. This is done by multipass rendering, which drastically decreases the performance. Supersampling in the z direction results in more slices in the viewing direction, which is supported by the hardware and, hence, hardly influences performance.

Super sampling factors



Super sampling factors: x 1 y 1 z 1

This port allows to specify the degree of supersampling in the x, y and z direction.

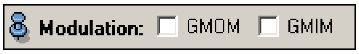
Blend mode

Blend mode: [✓ FrontToBack] MIP MINIP

Three blending modes are supported:

- *FrontToBack:* Going along the ray from the front of the volume to the back, the final color value is accumulated from all the samples.
- *MIP (Maximum Intensity Projection):* Chooses the voxel with the greatest intensity along the ray.
- *MINIP (Minimum Intensity Projection):* Chooses the voxel with the lowest intensity along the ray.

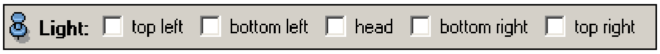
Modulation



Modulation: GMOM GMIM

- *GMOM (Gradient Magnitude Opacity Modulation):* The original opacity is scaled by the magnitude of the sample gradient. Thus only surfaces will be perceived.
- *GMIM (Gradient Magnitude Illumination Modulation):* Multiplies the computed diffuse or specular color with the magnitude of the sample gradient. As a result the illumination from non-surfaces, which have small gradients, is reduced. GMIM is only effects the image if light is applied to the scene.

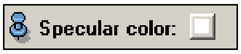
Light



Light: ☐ top left ☐ bottom left ☐ head ☐ bottom right ☐ top right

Up to five lights can be applied to the scene at the same time.

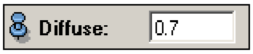
Specular color



Specular color: ☐

Specify the color of the light(s).

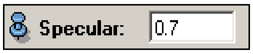
Diffuse



Diffuse: 0.7

Material coefficient.

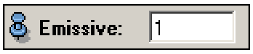
Specular



Specular: 0.7

Material coefficient.

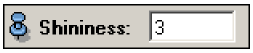
Emissive



Emissive: 1

Material coefficient.

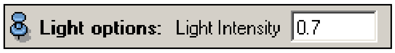
Shininess



Shininess: 3

Material coefficient.

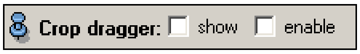
Light options



Light options: Light Intensity 0.7

Light intensity.

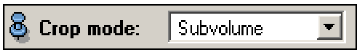
Crop dragger



Crop dragger: ☐ show ☐ enable

A dragger box can be used to crop parts of the volume. With the *show* toggle you hide or show the box, which can then be scaled and translated in the viewing window. With the *enable* toggle cropping is switched on or off.

Crop mode

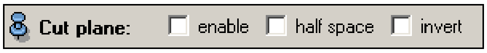

Crop mode: Subvolume

There exist six different crop modes:

- *SubVolume:* Only the volume within the box is shown.
- *3DCross:* The sides of the box are projected to the bounding box and everything within within this 3d cross is displayed.
- *3DFence:* All voxels within the x-, y-, OR z-range of the box are displayed.

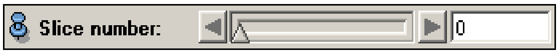
The results of these three modes can also be inverted.

Cut plane


Cut plane: ☐ enable ☐ half space ☐ invert

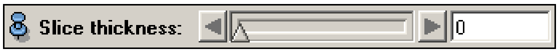
With the help of the cut plane slices of varying thickness can be cut out of the volume. Either the part of the volume defined by this slice or the rest of the volume except this slice might be visualized separately. Apart from slices half spaces can be clipped. The *Cut Plane* port has three toggle buttons, which determine how clipping is done. The first button enables or disables the cut plane. If the second toggle button, *halfspace*, is pushed, not a plane is selected, but a whole halfspace, i.e. the volume on one side of the plane is clipped, the other one is shown. The toggle *invert* inverts clipping, e.g., if it is off and *halfspace* is off too, a slice with a certain thickness is shown. If you then switch the *invert* toggle on, the slice that could formerly be seen is now clipped and the rest of the volume can be seen.

Slice number


Slice number: 0

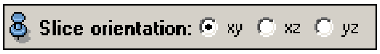
Determines the number of the cut plane slice.

Slice thickness


Slice thickness: 0

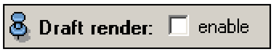
Determines the thickness of the cut plane. Notice, that the thickness grows only into one direction.

Slice orientation


Slice orientation: [✓ xy] xz yz

With this port you can specify the plane which the cut plane should be parallel to.

Draft render


Draft render: ☐ enable

If checked the rendering process gets simplified during object animation. The desired effect ist to increase the rendering speed. The loss of quality is the price for the more of interactivity. If the objects stops to transform a maximum quality still picture is renederd.

Draft render quality

Draft render quality: 0.5

With this slider one can adjust the amount of simplicity or quality loss when draft rendering is enabled. The lower the value the faster but messier the render result. No matter which value was chosen, the depth buffer feature (providing correct occlusion with polygonal objects) is always turned off during draft rendering.

Multiboard mode

Multiboard mode: replicate

A volume buffer can be replicated or spread over multiple VolumePro 1000 boards. This port can be either set to *single*, *replicate* or *split*.The default is *single* disabling multiboard rendering for that volume. Multiboard mode *replicate* replicates the volume data onto all boards in the system. Each board will render the entire volume for one horizontal band of the output image buffer. This will speed up rendering when the slowest part of the rendering process is the ray casting process, but will slow down volume updates because each update must go to all boards. Multiboard mode *split* splits the volume into pieces and loads one piece onto each board in the system. This allows very large volumes to be rendered, since two boards will be able to support a volume greater than one board. However, the images from each hardware rendering operation are full size images that must be blended together in a separate rendering process; this adds extra overhead for rendering.

Multiboard min size

Multiboard min size: 256

Minimum size to enable multiboard capabilities for a volume.

Multiboard overlap

Multiboard overlap: 64

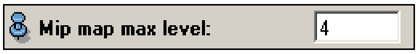
For multiboard rendering mode *split*, this specifies how much overlap there will be between the split volume data. The minimum is 2, which is fine for non-mipmap volumes. The larger the overlap, the more mipmap levels can be generated correctly. For example, an overlap of 64 will allow levels 1 through 5 even with the mipmap shrink factor set to 50

Mip map min level

Mip map min level: 0

Range of mipmap levels to use when rendering. Valid values are 0 to 32.

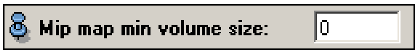
Mip map max level



Mip map max level: 4

Range of mipmap levels to use when rendering. Valid values are 0 to 32.

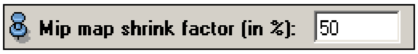
Mip map min volume size



Mip map min volume size: 0

Minimum size to enable mipmap capabilities for a volume.

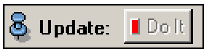
Mip map shrink factor (in %)



Mip map shrink factor (in %): 50

Shrink factor (in percent) to be used for automatic mipmap level generation. The valid range is 50

Update



Press this button ro reload the volume to the VolumePro 1000 board(s). This may be necessary after changing some crucial ports. As long there is no need to reload the volume, the button is disabled. The default behaviour of this module at its creation time is to load the volume into a single board. This can be stopped by pressing the stop button in the work area beside the progress bar. Then one could change the multiboard parameters and press this button to reload the volume.

6.136 Voltex

Direct Volume Rendering is a very intuitive method for visualizing 3D scalar fields. Each point in a data volume is assumed to emit and absorb light. The amount and color of emitted light and the amount of absorption is determined from the scalar data by using a color map which includes alpha values. Default colormaps for volume rendering are provided with the distribution and can be edited using the colormap editor. Then the resulting projection from the ”shining” data volume is computed.

This module provides you with a hardware accelerated implementation, which uses 2D or 3D texture hardware, to allow for real-time rendering. Note that this currently is not supported by all grahics hardware. Currently hardware acceleration for 2D and 3D textures is available on e.g., SGI Octane, SGI Reality and InfiniteReality, SGI High/Max Impact, HP fx/4, fx/6, and fx/10. The SGI O2 supports 2D texturing. Most PC graphics cards support 2D texture mapping. Older SGI systems, like Indigo2 Extreme, and many Linux drivers currently do not offer hardware texture acceleration. Using this module on the latter platforms can be extremely slow.

Note that on some systems a significant slowdown can occur if the data set is larger than the available texture memory (which typically is 4 - 16 MB).

Connections

Data `[required]`

The 3D scalarfield to visualized. Alternatively an RGBA data volume (Colorfield) can be connected. In this case no colormap is used, but the color and opacity values are taken directly from the data. As a third mode the module can operate on multi-channel fields. Here the transfer function for each channel is computed automatically based on the channels native color, the channels data range, and the value of the Gamma port (see below).

ROI `[optional]`

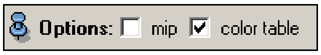
Connection to a module providing a region-of-interest, like SelectRoi. If such a module is connected, only the selected part of the volume will be displayed.

Colormap `[optional]`

Colormap used to vizualize the data.

Ports

Options

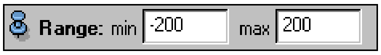


Options: [ ] mip [✓ color table]

mip stands for *maximum intensity projection*. When this option is selected, the brightest data value along each ray of sight is displayed instead of the result of the emission absorption model described above. This mode is especially useful for very ”sparse” data sets, for example: angiographic data or images of neurons.

The *color table* is supported by some graphics boards only, e.g., SGI Onyx InfiniteReality and nVidia boards. When this option is activated only a quarter of the texture memory is needed for RGBA rendering and the color table and its range can be modified in real-time (i.e. without pressing *DoIt*). Note that due to incomplete OpenGL implementations some graphics boards which claim to support color tables, they do not. If you see artifacts or only plain white cubes, disable this option.

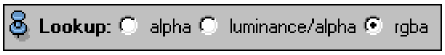
Range



Range: min -200 max 200

This port is only available if the module operates on a 3D scalar field and no colormap is connected. In this case data values are mapped according to this range. Values smaller than the minimum are mapped to completely transparent (no absorption and no emission). Values larger than the maximum appear completely opaque and emit the maximum amount of light. Values in between are mapped proportionally.

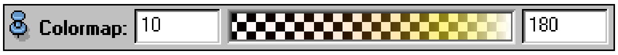
Lookup



Lookup: alpha luminance/alpha [✓ rgba]

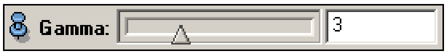
Only available if a colormap is connected. In *Alpha* mode, the colormaps alpha value is used for both absorption and emission. In *LumAlpha* mode, the colormaps alpha value is used for absorption, while the luminance is taken for (uncolored) emission. In *RGBA* mode, colored images are generated by using all four channels of the colormap.

Colormap



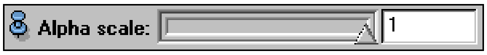
Port to select a colormap.

Gamma



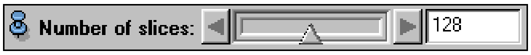
Controls the shape of the transfer function when multi-channel fields are visualized. The opacity value is taken to be $\alpha = x^\gamma$, x = 0 . . . 1 (proportional to data values). The smaller the gamma value is, the more prominent regions with small data values will be.

Alpha scale



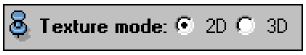
A global factor to change the overall transparency of the object independent of the data value.

Number of slices



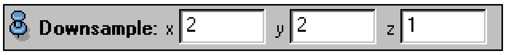
Only available in 3D texture mode. The larger this number, the better the image quality and the less the rendering performance.

Texture mode



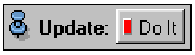
2D texture mode requires some precomputation time but also works on machines which do not support hardware accelerated 3D texturing, e.g., SGI O2. *3D* mode needs less setup time and sometimes provides superior quality on high-end machines. 3D texture mapping is not available on the Windows platform.

Downsample



You can specify integer downsample factors to reduce the size of the data set on-the-fly. e.g., downsampling by 2 in each direction would decrease the size of the data set by a factor of 8. This can dramatically improve rendering performance.

Update



Click on this button in order to trigger the computations necessary to display the volume. Most parameter changes require pressing this button again.

Commands

`showSlices on`

If on is nonzero, the slices used to display the volume are drawn outlined instead of full textured. This mode is mainly useful for debugging.

`setInterpol <value>`

Enables or disables linear interpolation of texture values. If linear interpolation is disabled a nearest neighbour lookup is performed. On default, linear interpolation is enabled.

`getInterpol`

Checks if linear texture interpolation is enabled or not.

`setColorTableInterpol <value>`

Enables or disables linear interpolation within the color table. The setting will be ignored if color table mode is off or if color table rendering is done using paletted textures. The default value is on.

`getColorTableInterpol`

Checks if linear color table interpolation is enabled or not.

`setColorTableMode 0|1|2|3|4`

Sets the type of OpenGL extension used for color table mode. The modes are encoded as follows:
0 = Don’t use any extension, turning off color table mode.
1 = GL_SGI_texture_color_table.
2 = GL_EXT_paletted_texture.
3 = GL_NV_fragment_program.
4 = GL_ARB_fragment_program.

`getColortableMode`

Returns the type of OpenGL extension used for color table mode.

`doBricking <value>`

Enables or disables bricking in 3D texture mode. If bricking is enabled the volume is rendered in smaller blocks if it is larger than the 80 percent of the total amount of texture memory. Since it is difficult to determine precisely the total amount of texture memory can be specified using the environment variable AMIRA_TEXMEM (the value of this variable is interpreted as megabytes).

`verbose <value>`

If value is 1 additional message for debugging are printed.

6.137 VolumeEdit

This module provides tools for the interactive modification of 3D image volumes. This is particularly useful for removing noise or undesired objects in a 3D image before applying isosurfaces, volume rendering or other image segmentation tools.

The module takes a scalar field as input and produces a new data set as output which can be modified iteratively. The module does not display any geometry in the viewer. It is typically being used in conjunction with a Voltex or an Isosurface module.

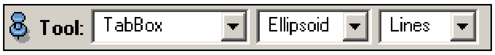
Connections

Data `[required]`

The input dataset to be edited (uniform scalar field).

Ports

Tool



The module provides two different types of tools: a lasso or draw tool and 3D dragger tools.

The lasso or draw tool lets you encircle a specific region in the 3D viewer which then can be cleared in the output data set. Alternatively the part not encircled (exterior) can be cleared (cut away), or the original data values can be restored in the encircled region. In order to use the draw tool, first press one of the action buttons *cut interior*, *cut exterior*, or *restore*. Then draw a line around the specific region in the 3d viewer.

The dragger tools let you specify a region to be modified by dragging, rotating and resizing a 3D shape (box, ellipsoid, cone, or cylinder). A cut or restore operation can then be applied to the interior or exterior part of that shape.

ZeroLevel

This port specifies by which data value voxels in selected regions are replaced when either *cut interior* or *cut exterior* is pressed.

Cut

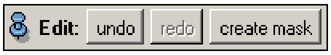
If the button *interior* is pressed, voxels inside the region selected by a dragger shape are replaced by the zero level port. If the button *exterior* is pressed, voxels outside this region are replaced. If the *draw tool* is active, you have to encircle the region to be replaced after pressing one of the buttons.

Restore

If the button *interior* is pressed, voxels inside the region selected by a dragger shape are replaced by the original data values. If the button *exterior* is pressed, voxels outside this region are replaced. If the *draw tool* is active, you have to encircle the region to be replaced after pressing one of the buttons.

If the button *all* is pressed, the entire volume is reset to its original state.

Edit



This port provides two buttons for undoing or redoing the last cut or restore operation. The *create mask* buttons creates a binary label field in which all voxel with a modified data value are set.

6.138 VoxelView

This module can be attached to an OrthoSlice module. It allows you to visualize contiguous 3D regions of a LabelField or of some other uniform scalar field with integer values. The regions are selected by clicking onto the *OrthoSlice* with the middle mouse button. Starting from the selected pixel a 3D flood fill process is performed. Multiple regions can be selected by shift-clicking multiple seeds.

On default the regions to be visualized are taken from the same input object the *OrthoSlice* module is attached to. However, optionally an independent scalar field may be connected to the *VoxelView* module. For example, an *OrthoSlice* module may be used to visualize a stack of CT images, while a *VoxelView* attached to it is used to display segmented regions defined in a label field.

Connections

Slice `[required]`

The OrthoSlice module which provides the slice where seed points have to be selected using the middle mouse button.

Data `[optional]`

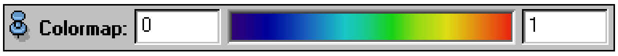
Optional scalar field. If set contiguous regions of this field will be displayed instead of regions of the field the *OrthoSlice* module is attached to.

Colormap `[optional]`

The colormap used to color the 3D regions.

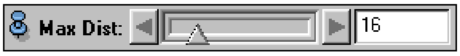
Ports

Colormap



Port to select a colormap.

Max Dist



This port limits the region growing process. At most the given number of slices are considered in upward or downward direction. May be useful on slow machines in order to limit the number of triangles.

Draw Style

Draw Style: Filled

Three different draw styles are provided, *filled*, *lines*, and *points*. Due to the regular structure of the voxel data only three different face orientations occur. Thus only three different colors will be used to render the voxel regions.

Floodfill Type

Floodfill Type: 6 neighbors

Specifies the kind of flood fill algorithm to be applied (neighbours at faces, faces and edges, or faces, edges, and corners). If *take all* is selected all voxel with the same value as the seed voxel will be selected and displayed.

Action

Provides a button to hide all selected 3D regions.

Chapter 7

Alphabetic Index of Ports

7.1 Port

This is the base class of all ports in amira.

Ports are used to interact with an amira object. There are several different types of ports, e.g., option menus, sliders, or toggle buttons. If an object is selected its ports are displayed in the lower part of the amira main window, the so-called working area. An object may choose not to display all ports at all times, depending on the internal state of the obejct. Most ports have a pin, allowing to keep the port visible even if the the object the port belongs to is deselected.

Connections are special ports, allowing to specify dependencies between objects. In contrast to other ports connections usually don't have their own user interface, but they are represented by corresponding lines in the object pool.

Every port provides its own Tcl command interface. This allows script programmers to interact with the port in an automated way. In addition to the commands defined by a port itself also all commands of the port's base class are available (this is the same as for amira modules and data classes). Port commands are called using the following syntax:

```
<modulename> <portname> <commandname> [optional parameters]
```

In order to get a list of all ports of an object you can use the command `<modulename> allPorts`. In most cases, but not in all cases, the name of a port used in Tcl commands is equal to the port's label as displayed in the graphical user interface, with the only exception that the name usually starts with a lower case letter and that white spaces are omitted.

Commands

`help`

Displays all commands available for the port. The same result can be obtained by just calling the port without any command.

`isNew`

Returns `1` if the port was changed after the object was fired the last time, and 0 otherwise.

`getState`

Returns a string which later on can be used to restore the state of a port using the command `set-State`. The format of the state string is not specified and it may be different for each port. Usually the state string contains the same information which is also stored in amira network files, but not for example label strings which might have been modified using the script interface as well.

`setState <state-string>`

Restores the state of a port from a string previously obtained using the `getState` command.

`getLabel`

Returns the label of the port displayed in graphical user interface, for example *Options:*.

`setLabel <label>`

Sets the label of the port.

`getLabelWidth`

Returns the width of the port's label in pixels.

`setLabelWidth <width>`

Sets the width of the port's label in pixels, see also `align`.

`align <port2> <port3> ...`

Align this and all other ports specified as arguments so that all labels have the same width.

`getPin`

Check if the port has been pinned or not.

`setPin <value>`

Set or unset the port's pin button. If the port is pinned it will be displayed in the work area even if the owner of the port is deselected.

`touch`

Touch the port to simulate a change of value.

`untouch`

Untouch the port so that `isNew` will return 0 the next time even if the port was changed.

`object`

Returns the name of the object the port belongs to.

`send`
Fires the object the port belongs to as well as all downstream objects.

`show`
Shows the port, provided the owning object is selected.

`hide`
Hides the object, provided the owning object is selected.

`isVisible`
Returns 1 if the port is shown, or 0 if it is hidden.

`reposition <index>`
Changes the position of the port within the object's port list and in the graphical user interface. Note that connection ports are counted too even if they may not have any controls.

`isOfType <typeid>`
Checks if this port if of the specified type or derived from it.

`getTypeId`
Returns the port's type name.

7.2 Connection

This port represents a connection from one object to another. It is commonly used to specify on which input objects some other object should depend on. A connection does not provide its own user interface like other ports, but is visually represented in the object pool by a line between the source object and the object the connection port belongs to. In order to change the connection this line can be picked with the mouse and dragged to some other source object. All connection ports of an object are listed in a special popup menu which is activated by clicking with the right mouse button in the little connection area (left most rectangle) of the object's icon.

Commands

Inherits all commands of Port.

`source`
Returns the name of the source object connected to this port. If there is no source object connected an empty string is returned.

`connect <source>`
Tries to connect the port with the specified source object. If the connection cannot be established a Tcl error is generated.

`disconnect`
Disconnects the port from any source object.

`setTightness {0|1}`
Sets the tightness flag of the port. If the connection is tight no connection line is drawn in the object pool. Instead, the icon of the owning object is glued together with the icon of the source object, so that both icons comprise an icon group. Each object can only have one connection port with tightness set to 1.

`isTight`
Checks if the connection is tight or not (see above).

`setVisibility {0|1}`
Sets the visibility flag of the port. If visibility is switched off the connection will not be represented by a line in the object pool.

`isVisible`
Checks if the connection is visible or not (see above).

`allowEditing {0|1}`
Allow or disallow interactive editing of the connection. If editing is disallowed the connection cannot be changed interactively by moving the connection line to some other source object.

`isEditable`
Checks if the connection can be edited interactively or not.

`validSource <source>`
Checks if the specified object can be connected to this port. If this is the case 1 is returned, otherwise 0 is returned.

7.3 MasterConnection

This port represents a special connection port which is provided by every data object. It allows to connect the data object to a particular slot of a compute module or to an editor. A master connection is not available for script objects.

Commands

Inherits all commands of Connection.

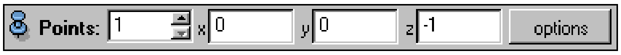
`editor`
Returns the name the editor which currently has control over the data object, i.e., which possibly modifies the data object.

`connect <source> [<slot>]`
If this command is called with only one argument, it behaves like the `connect` command of an ordinary connection. If this command is called with two arguments and if the source object is a compute module, it connects the data object the master connection belongs to to the specified result slot of the compute module. Result slots are used to allow compute modules to have multiple result objects.

7.4 Port3DPointList

This port lets you enter an arbitrary number of 3D points. The points can be specified either by typing in their coordinates in appropriate text fields or by activating a dragger in the 3D viewer. In order to move the dragger you have to put the viewer into interaction mode (press the **ESC** key). Then move the mouse over one of the dragger's crosshairs and push the left mouse button. The color of the picked crosshair changes and a grey transparent plane is shown. The moving area of the dragger is restricted to this plane. The new coordinates are delivered to the application when you release the mouse button, or, if *immediate mode* is on, while you are moving. If the *orthogonal mode* is on, all other points are moved in sync. You can only move one point, the current point, at a time. Type **TAB** in the viewer window to cycle through all points if you want to change the current point. You can also hit the up or down arrows of the port's point index spin box. Theses arrows are shown only if the port manages a list of at least 3 3D points. The current point is shown as a red sphere, all other points as yellowish spheres.

Note: You can pick a location with the middle mouse button on a pickable object in the scene, e.g., an OrthoSlice, to set the current point to that location.



This port is used for example by the data probing modules.

Commands

Inherits all commands of Port.

`getNumPoints`
Returns the current number of points defined by this port.

`getValue [<index>]`
Returns the coordinates of point `<index>` or of the current point, if no argument was specified.

`setValue [<index>] <x> <y> <z>`
Sets the coordinates of point `<index>` or of the current point, if no argument was specified. The coordinates are automatically clipped against the bounding box of the dragger.

`getBoundingBox`

Returns the bounding box of the dragger. The command returns six number, namely the xmin, xmax, ymin, ymax, zmin, and zmax coordinates of the bounding box in that order.

`setBoundingBox <xmin> <xmax> <ymin> <ymax> <zmin> <zmax>`

Sets the bounding box of the dragger. Most modules adjust the bounding box if the module is connected to a new input data object.

`getFormat`

Returns the format used for the coordinate text fields.

`setFormat <format>`

Sets the format used for the coordinate text fields in printf syntax. The default is `%g`.

`getImmediate`

Checks whether immediate mode is enabled or not.

`setImmediate {0|1}`

Enables or disables immediate mode. If immediate mode is enabled the module the port belongs to is fired permanently while the dragger is moved. Otherwise the module is fired only once after the dragger was moved.

`getOrtho`

Checks whether orthogonal mode enabled or not.

`setOrtho {0|1}`

Enables or disables orthogonal mode. In orthogonal mode all points of the port are moved at the same time when the dragger is moved. Otherwise only the current point is moved.

`showDragger`

Shows the dragger at the current point.

`hideDragger`

Hides the dragger.

`showPoints`

Shows the points. The points are represented by little red spheres.

`hidePoints`

Hides the points.

`getPointSize`

Returns the current point size (radius) in absolute coordinates.

`setPointSize <radius>`

Sets the point size (radius). The point size is automatically adjusted if the dragger's bounding box changes.

`getPointScale`
Returns the current point size scale factor.

`setPointScale <factor>`
Sets the point size scale factor. On default the scale factor is 1. If you find that the default size of the points is too small or too big you can adjust this factor.

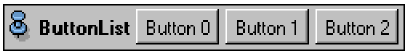
`appendPoint [<x> <y> <z>]`
Appends a new point to the port's point list. If no coordinates are specified a point with some default coordinates will be appended. If the number of points exceeds some limit (which cannot be modified via the script interface) nothing happens and -1 is returned. Otherwise the index of the appended point is returned.

`insertPoint <index> [<x> <y> <z>]`
Inserts a new point at position `index` in the port's point list. If no coordinates are specified a point with some default coordinates will be inserted. If the number of points exceeds some limit (which cannot be modified via the script interface) nothing happens and -1 is returned. Otherwise `<index>` is returned.

`removePoint <index>`
Removes point `<index>` unless a minimal number of points (which cannot be modified via the script interface) is reached. If the point couldn't be removed 0 is returned. Othwerwise 1 is returned.

7.5 PortButtonList

This port provides an arbitrary number of push buttons. The buttons are implicitely numbered 0, 1, 2, ... from left to right. When a button is pushed the port's new flag is set. The index of the pushed button can be obtained using the command `getValue`. After the module was fired the port's new flag is unset again and `getValue` returns -1. This means that the port does not store a permanent state. A push button is frequently used to trigger some action.



Modules using this port are for example GridVolume or ObliqueSlice.

Commands

Inherits all commands of Port.

`getValue`
Returns the index of the pushed button. If no button was pressed since the module was fired the last time -1 is returned. The method does not interprete the snap toggle. In order to handle snapped buttons better use the command `wasHit` (see below).

`setValue <index>`
Marks button `index` as being pushed.

`wasShiftDown`
Checks if the shift key was down the last time a button was pressed.

`wasCtrlDown`
Checks if the control key was down the last time a button was pressed.

`wasAltDown`
Checks if the alt key was down the last time a button was pressed.

`setShiftDown {0|1}`
Sets the shift key modifier flag.

`setCtrlDown {0|1}`
Sets the control key modifier flag.

`setAltDown {0|1}`
Sets the alt key modifier flag.

`getSensitivity [<index>]`
Checks whether the first button or button `index` is enabled or disabled.

`setSensitivity [<index>] {0|1}`
Enables or disables the first button or button `index`. If a button is disabled it is grayed out and cannot be pushed anymore.

`getLabel [<index>]`
Returns the label of button `index` or the port's label.

`setLabel [<index>] <label>`
Sets the label of button `index` of the port's label.

`setCmd [<index>] <command>`
Sets a Tcl command which is executed when the specified button is pressed. This feature is especially useful in script objects since it avoids writing long Tcl code with many if statements. If a command has been set for a button the owning module will not be fired and the port will not be touched when the button is pressed. The command will be executed in the global Tcl namespace. The variable `this` refers to the owning module.

`getCmd [<index>]`
Returns the Tcl command associated with the specified button or an empty string if no Tcl command has been set.

`getNumButtons`
Returns the number of buttons of the port.

`setNumButtons <number>`
Sets the number of buttons of the port. New buttons are created with an empty label.

`enableSnapToggle [<index>] {0|1}`
Enable or disable the snap toggle for the first button or for button `index`.

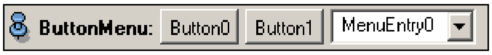
`snap [<index>] {0|1}`
Snap or unsnap a button. The button's snap toggle must be enabled.

`isSnapped [<index>]`
Check whether the specified button is snapped or not.

`wasHit [<index>]`
Returns 1 if the specified button was hit or if its snap toggle is down.

7.6 PortButtonMenu

This port combines an arbitrary number of push buttons with one or more option menus containing an arbitrary number of options. The port is derived from PortButtonList and hence inherits its behaviour.



This port is used for example by the modules FieldCut or DisplayISL.

Commands

Inherits all commands of PortButtonList.

`setNumOptEntries [<menu>] <number>`
Sets the number of entries in the option menu.

`getNumOptEntries [<menu>]`
Returns the number of entries in the option menu.

`setOptValue [<menu>] <index>`
Selects item `<index>` in the option menu.

`setOptValueString [<menu>] <label>`
Selects the item with the specified label string in menu `<menu>` or in the first option menu, if only one argument is given. If no item with such a label exists, nothing happens and 0 is returned. Othwerwise 1 is returned.

`getOptValue [<menu>]`
Returns the index of the selected item in the option menu.

`setOptLabel [<menu>] <index> <label>`
Sets the label of the option menu item specified by `<index>`.

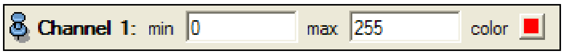
`getOptLabel [<menu>] <index>`
Returns the label of the option menu item specified by `<index>`.

`setOptSensitivity [<menu>] <index> {0|1}`
Enables or disables a particular item in the option menu.

`getOptSensitivity [<menu>] <index>`
Check whether a particular item in the option menu is enabled or not.

7.7 PortChannelConfi g

This is a special port used exclusively by multi-channel fields. There is one such port for each channel of a multi-channel field. Derived from connection, this port manages the link to the actual channel object. In addition, it provides two text fields allowing the user to define a data range used for mapping the channel data to gray values. The channel's color can be set using a color button. Clicking on this button pops up the amira color dialog. This port is not available for script objects.



Commands

Inherits all commands of Connection.

`getMinValue`
Returns the lower value of the channel's data window.

`setMinValue <value>`
Sets the lower value of the channel's data window.

`getMaxValue`
Returns the upper value of the channel's data window.

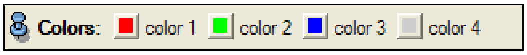
`setMaxValue <value>`
Sets the upper value of the channel's data window.

`getColor`
Returns the channel's color as an RGB tuple of floating point values.

`setColor <color>`
Sets the channel's color. The color can be specified either as a tuple of three RGB integer values in the range 0...255, or as a tuple of three RGB floating point values in the range 0...1, or as a text string (e.g., `blue` or `red`).

7.8 PortColorList

This port provides one or more color buttons which can be used to define a constant color. Colors can be easily edited using the amira color dialog which is popped up when one of the color buttons is pushed.



This port is used for example by the Axis module.

Commands

Inherits all commands of Port.

`setColor <index> <color>`
Sets the color of button `<index>`. The color can be specified either as a tuple of three RGB integer values in the range 0...255, or as a tuple of three RGB floating point values in the range 0...1, or as a text string (e.g., `blue` or `red`).

`getColor <index>`
Returns the color of button index as an RGB tuple of three floating point numbers in the range 0...1.

`setAlpha <index> <alpha>`
Sets the alpha value of button `<index>`.

`getAlpha <index>`
Returns the alpha value of button `<index>`.

`setLabel [<index>] <label>`
Sets the label of button `index` of the port's label, if only one argument is specified.

`getLabel [<index>]`
Returns the label of button `index` or the port's label, if no argument is specified.

`setNumButtons <number>`
Sets the number of color buttons of the port.

`getNumButtons`
Returns the number of color buttons of the port.

`setContinuousMode {0|1}`
Enables or disables continuous mode. If continuous mode is activated the owning module is fired permanently when changing a color using the color dialog.

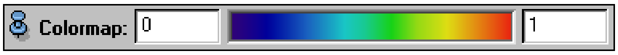
`getContinuousMode`
Checks whether continous mode is enabled or not.

7.9 PortColormap

This port provides a connection to a colormap object. In contrast to a standard connection port this port has a user interface, i.e., it is represented in the work area as any other port if the owning object is selected. In particular the contents of the connected colormap as well as its range are shown. New colormaps can be quickly connected to the port via a context menu which pops up when pressing the right mouse button over the colormap area of the port.

The context menu also allows you to switch between global and local range mode. In global range mode the coordinates used to map data values to colors are taken from the colormap itself. If the same colormap is used by two different modules and if the range then is modified, both modules are updated. In contrast, in local range mode the coordinates are defined by the port itself. Thus, although the same colormap might be used by two different modules, the ranges still can be different.

If no colormap is connected to the port, still a default color and a default alpha value can be defined. The default colors can be modified via the amira color dialog which is popped up by double-clicking on the colormap area with the left mouse button. If a colormap is connected a left mouse button double-click makes the icon of the colormap visible in the object pool.



See also: Colormap, ColormapEditor

Commands

Inherits all commands of Connection.

`setDefaultColor <red> <green> <blue>`
Sets the default color of the port. An RGB tuple of three floating point values between 0...1 must be specified.

`getDefaultColor`
Returns the default color of the port.

`setDefaultAlpha <alpha>`
Sets the default alpha (opacity) value of the port.

`getDefaultAlpha`
Returns the default alpha (opacity) value of the port.

`enableAlpha {0|1}`
Enables or disables the display of the alpha channel in the port's colormap area. Even if the alpha channel is not displayed by the port it is up to the owning module to interprete alpha values or not.

`isAlphaEnabled`
Checks is alpha values are displayed by the port or not.

`setLocalRange {0|1}`
Enables or disable the local range feature (see discription above).

`getLocalRange`
Checks is the local range feature is enabled or not.

`setLocalMinMax <min> <max>`
Sets the coordinate range used for mapping data values to colors when the local range feature is enabled. The port is touched but the network is not fired.

`getLocalMinValue`
Returns the lower bound of the local range.

`setLocalMinValue <min>`
Sets the lower bound of the local range.

`getLocalMaxValue`
Returns the upper bound of the local range.

`setLocalMaxValue <max>`
Sets the upper bound of the local range.

`setMinMax <min> <max>`
Sets the coordinate range used for mapping data values to colors. If local range is enabled or if no colormap is connected the local range is updated. Otherwise the range of the connected colormap is updated. In both cases the network is fired, i.e., the module owning this port or modules connected to the modified colormap are updated.

7.10 PortDoIt

This port is essentially identical to PortButtonList. The only difference is that this port always comes up with a single button initially and that this button has a snap toggle by default.

This port is mainly used by compute modules, for example Arithmetic or CastField.

Commands

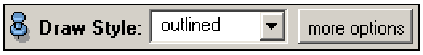
Inherits all commands of PortButtonList.

7.11 PortDrawStyle

This is a special port allowing to adjust the drawstyle of display modules derived from ViewBase. The port provides a combo box (option menu) offering the following drawstyles:

- outlined
- shaded
- lines
- points
- transparent

In addition, the port allows to set additional options vai a popup menu which is activated by pressing the *more options* button. For a detailed description of these options please refer to the documentation of ViewBase. This port is not available for script objects.



Commands

Inherits all commands of Port.

`getValue`
Returns the index of the current draw style.

`setValue <index>`
Sets the current draw style. The draw styles are indexed in the same order as they appear in the combo box, i.e., 0=outlined, 1=shaded, 2=lines, 3=points, 4=transparent.

`isTexture`
Checks if 1D-textures for pseudo-coloring are enabled (as opposed to Gouraud shading).

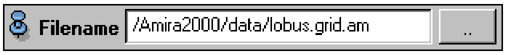
`setTexture {0|1}`
Enables or disables 1D-textures for pseudo-coloring.

`getNormalBinding`
Returns an index describing the current normal binding mode.

`setNormalBinding {0|1|2}`
Sets the normal binding. The indices have the following meaning: 0=triangle normals, 1=vertex normals, 2=direct normals (computed using a crease angle criterium, compare ViewBase).

7.12 PortFilename

This port is used to define a file name. The file name may be either manually typed in the text field, or it may be selected using the amira file browser. The file browser can operate in three modes, one allowing to select any file, one allowing to select one existing file only, and one allowing to select multiple existing files. However, if the file name is typed in manually no check is performed to ensure that the file really exists. Typically the first mode is used for saving while the second and third modes are used for loading. Associated with every filename is a filetype.



This port is used for example by the TetraGen module.

Commands

Inherits all commands of Port.

`getFilename`
Returns the filename or a list of filenames.

`getFileType`
Returns the filetype or a list of filetypes.

`setFilename <filename> <filetype>`
Sets the filename.

`setFilenames [list <filename1> <filename2> ...] [list <file-type1> <filetype2> ...]`
Sets a list of filenames in *multi file* mode.

`getValue`
Returns the filename (same as `getFilename`) or a list of filenames.

`setValue <filename> <filetype>`
Sets the filename (same as `setFilename` or `setFilenames` in *multi file* mode).

`getMode`
Returns 0 if the file browser operates is *any file* mode, 1 if it operates in *existing file* mode, or 2 if it operates in *multi file* mode (see description above).

`setMode {0|1|2}`
Sets the mode of the file browser (0 for *any file*, 1 for *existing file*, 2 for *multi file* (see description above).

`registerFileType [<formatname>] [<extension>]`
If the file browser is opened in *any file* mode, it shows a combo box allowing the user to specify an optional format to be used to save a data set. This command allows you to register format names and optional file extensions which are added to the file type combo box. Calling this command without any arguments removes all previously registered file types.

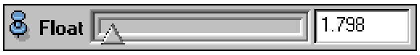
`exec`
Pops up the file browser. The file browser is opened in modal mode, i.e., the command blocks until the user closed the dialog again. There is no way to close an open file browser using a Tcl command.

The method returns 0 if the browser was closed using the cancel button and the port's filename field was not updated. Otherwise it returns 1.

7.13 PortFloatSlider

This port provides a slider allowing the user to specifiy a floating point number. The floating point number is limited to the current range of the slider. The range can be changed either using the Tcl command `setMinMax` or interactively using a configure dialog, which is be popped up by pressing the right mouse button over the slider.

Optionally the slider may have arrow buttons allowing the user to decrease or increase the value by some predefined increment. The slider may also have so-called subrange buttons allowing the user to restrict the actual range of the slider to a smaller subrange. In this way it is possible to interactively adjust the slider's value with high precision. In order to numerically adjust the lower (left) subrange value the command `L <value>` can be typed in the slider's text field. In order to numerically adjust the upper (right) subrange value the command `R <value>` can be typed in the slider's text field.



This port is used for example to specify the threshold of the Isosurface module.

Commands

Inherits all commands of Port.

`getValue`
Returns the float value.

`setValue <float>`
Sets the float value.

`getMinValue`
Returns the lower bound.

`getMaxValue`
Returns the upper bound.

`setMinMax <min> <max>`
Sets the lower and upper bounds of the slider.

`setFormat <format>`
Sets the format used for the slider's text field in printf syntax. Typically, the default format is `%g`.

`getFormat`
Returns the printf format used for the slider's text field.

`setIncrement <increment>`
Sets the increment. The increment is used for adjusting the slider value using the optional arrows buttons. It is also used to constrain the slider value in discrete mode (see below).

`getIncrement`
Returns the slider's increment.

`setButtons {0|1}`
Enables or disables the display of the optional arrow buttons.

`hasButtons`
Checks whether the optional arrow buttons are enabled or not.

`setSliderWidth <width>`
Sets the width of the slider in pixels.

`getSliderWidth`
Returns the width of the slider in pixels.

`setNumColumns <columns>`
Sets the width of the slider's text field in characters.

`getNumColumns`
Returns the width of the slider's text field in characters.

`setTracking {0|1}`
Enables or disables tracking mode. If tracking mode is enabled the network is fired permanently while the user moves the slider. Otherwise it is fired only once when the slider is released.

`getTracking`
Checks whether tracking mode is enabled or not.

`setDiscrete {0|1}`
Enables or disables discrete mode. If discrete mode is enabled the slider value will always be a equal to the lower bound plus an interger multiple of the current increment.

`getDiscrete`
Checks whether discrete mode is enabled or not.

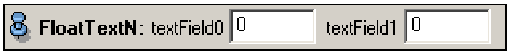
`setSubRangeButtons {0|1}`
Enables or disables the subrange buttons (see description above),

`hasSubRangeButtons`
Checks whether the subrange buttons are enabled or not.

7.14 PortFloatTextN

This port represents a variable number of bounded float values which can be edited interactively. The actual number of float fields can be changed at run-time. Each float field has a label, a printf-style format string, and a lower and upper bound used to constrain the value of the text field. minimum and maximimum values, as well as the sensitivity of each field.

In order to quickly change the value of a float field, a *virtual slider* is provided. The virtual slider is activated by clicking into a text field with the shift key hold down, and then moving the mouse up or down.



This port is used for example by the Resample module.

Commands

Inherits all commands of Port.

`getValue [<index>]`
Returns the value of the text field `<index>` or of the first text field, if no argument is specified.

`getMinValue [<index>]`
Returns the lower bound of the text field `<index>` or of the first text field, if no argument is specified.

`getMaxValue [<index>]`
Returns the upper bound of the text field `<index>` or of the first text field, if no argument is specified.

`setValue [<index>] <value>`
Sets the value of the text field `<index>` or of the first text field.

`setMinMax [<index>] <min> <max>`
Sets the lower and upper bound of the text field `<index>` or of the first text field.

`setValues <value1> <value2> ... <valueN>`
Sets the values of all text fields. The number of arguments must match the number of text fields.

`setMinMaxAll <min1> <max1> ... <minN> <maxN>`
Sets the lower and upper bounds of all text fields. The number of arguments must be equal to twice the number of text fields.

`getFormat [<index>]`
Returns the printf format string of the specified text field.

`setFormat [<index>] <format>`
Sets the printf format string of the specified text field.

`getSensitivity [<index>]`
Checks if the specified text field is enabled or not. Unsensitive text fields are grayed out and they accept do not accept user input.

`setSensitivity [<index>] {0|1}`
Enables or disables the specified text field.

`getLabel [<index>]`
Returns the label of text field `index` or the port's label.

`setLabel [<index>] <label>`
Sets the label of text field `index` of the port's label.

`setPart <index> {0|1}`
Shows or hides the specified text field. A hidden text field is not displayed at all (in contrast to an insensitive one), but still exists and stores a value.

`getNum`
Returns the current number of text fields.

`setNum <value>`
Sets the current number of text fields.

7.15 PortGeneric

This is a generic port which can be dynamically configured in various way. Several different predefined user interface components can be used:

- text fields for integer values
- text fields for floating point values
- check boxes
- groups of radio buttons
- combo boxes (option menus)
- push buttons
- labels

Interface components inserted into this port are identified using a unique ID, which has to be specified when creating the component. This ID is used in all commands to set or get the value of the particular component.

The design goal of this class was primarily ease-of-use. Thus flexibility is somewhat limited. If you want to create a port with custom GUI components you should use amiraDev and derive a new class from Port using Qt widgets.

This port is used for example by the TransformEditor.

Commands

Inherits all commands of Port.

`getValue <id>`
Returns the value of the specified component. For combo boxes and radio groups the index of the selected item is returned.

`setValue <id> <value>`
Sets the value of the specified component. If the specified component is a combo box or a radio group and if value is not a number than the value is interpreted as a label and the combo box item or the radio button with that label is selected.

`getColor <id>`
Returns the color of a color button inserted via the command `insertColorButton`. If the component `<id>` is not a color button the result is undefined.

`setColor <id> <color>`
Sets the color of a color button with the specified id. If the component `<id>` is not a color button the command returns immediatly.

`isItemNew <id>`
Checks if the specified component was modified since the owning module was fired the last time.

`deleteItem <id>`
Deletes the specified item.

`insertIntText <id> [<columns> [<index>]]`
Inserts an integer text field. In order to set the text field to e.g., 27, use `setValue <id> 27`. In order to query the value of the text field, use `getValue <id>`. `<columns>` specifies the width of the text field. `<index>` specifies the position where the text field should be inserted. If this argument is omitted the text field is appended after all other elements.

`insertFloatText <id> [<format> [<columns> [<index>]]]`
Inserts a floating point text field. In order to set the text field to e.g., 5.5, use `setValue <id> 5.5`. In order to query the value of the text field, use `getValue <id>`. `<format>` specifies the printf format of the text field. Usually `%g` is a good choice. `<columns>` specifies the width of the text field. `<index>` specifies the position where the text field should be inserted. If this argument is omitted the text field is appended after all other elements.

`insertCheckBox <id> [<label> [<index>]]`
Inserts a check box with a label. In order to set the check box use `setValue <id> {0|1}`. In order to query the value of the check box use `getValue <id>`. `<index>` specifies the position where the check box should be inserted. If this argument is omitted the text field is appended after all other elements.

`insertRadioGroup <id> <num> <label1> ... <labelN> [<index>]`
Inserts a group of n radio buttons. The button labels are specified by `<label1>` to `<labelN>`. The number of labels must match the number of radio buttons. In order to set for example the second radio button (and unset all others) use `setValue <id> 1`. In order to query which radio button is checked use `getValue <id>`. `<index>` specifies the position where the radio group should be inserted. If this argument is omitted the radio group is appended after all other elements.

`insertComboBox <id> <num> <label1> ... <labelN> [<index>]`
Inserts a combo box with n items. The item's labels are specified by `<label1>` to `<labelN>`. The number of labels must match the number of radio buttons. In order to select for example the second item (and unset all others) use `setValue <id> 1`. In order to query which item is selected use `getValue <id>`. `<index>` specifies the position where the combo box should be inserted. If this argument is omitted the combo box is appended after all other elements.

`insertPushButton <id> <label> [<index>]`
Inserts a simple push button. `<label>` specifies the label of the button. In order to simulate a button press event use `setValue <id> 1`. In order to check if the button was pressed use the command `isItemNew <id>`. `<index>` specifies the position where the button should be inserted. If this argument is omitted the button is appended after all other elements.

`insertColorButton <id> [<index>]`
Inserts a color button. The button can be used to specify an RGB color. Pushing the button opens the color dialog, In order to set the color use the command `setColor <id> <color>`. In order to query the color use the command `getColor <id>`. `<index>` specifies the position where the button should be inserted. If this argument is omitted the button is appended after all other elements.

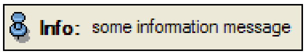
`insertLabel <id> <label> [<index>]`
Inserts a label. `<index>` specifies the position where the label should be inserted. If this argument is omitted the label is appended after all other elements.

`setSensitivity <id> {0|1}`
Sets the sensitivity of an element. If an element is insensitive it is grayed out and accepts no user input. On default, all new elements are sensitive.

`getSensitivity <id>`
Returns the sensitivity of an element.

7.16 PortInfo

This port displays a simple single line information message. It does not allow any user interaction, and thus never causes the owning object and the network to be fired.



This port is used for example by the Interpolate module.

Commands

Inherits all commands of Port.

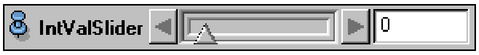
`getValue`
Returns the text displayed by the port.

`setValue <text>`
Sets the text displayed by the port.

`printf <fmt> <arg1> <arg2> ...`
Sets the text displayed by the port using a C-style printf command. `<fmt>` is a message string which may contain `%s` format specification fields. These fields are substituted by the arguments. Note, that only `%s` string type fields must be used, but no numerical format fields like `%d` or `%g`.

7.17 PortIntSlider

Slider port representing an integer value. This class is derived from PortFloatSlider and it provides exactly the same features and commands. The only exception is that the values returned by `getValue`, `getMinValue`, and `getMaxValue` will be rounded to the nearest integer value. In addition, the dafault format string used by the slider's text field is `%.0f` which is suitable for integer values.



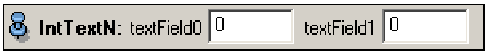
Modules using this port are for example OrthoSlice or Vectors.

Commands

Inherits all commands of PortFloatSlider.

7.18 PortIntTextN

This port provides an arbitrary number of text fields for entering integer values. The port is derived from PortIntTextN and it provides exactly the same features and commands. The only exception is that the values returned by `getValue`, `getMinValue`, and `getMaxValue` will be rounded to the nearest integer value. In addition, the dafault format string used for the text fields is `%.0f` which is suitable for integer values.



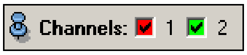
This port is used for example by the Resample module.

Commands

Inherits all commands of PortFloatTextN.

7.19 PortMultiChannel

This port is commonly used by modules operating on multi-channel fields. The port provides a set of colored toggle buttons, one for each channel of the multi-channel field. Using these toggle buttons individual channels of the multi-channel field can be quickly switch on or off. The color of a toggle button represents the color of the channel it is representing.



This port is used for example by the ProjectionView module.

Commands

Inherits all commands of Port.

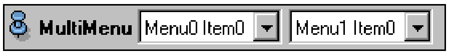
`getValue <index>`
Checks whether channel `<index>` is switched on or off.

`setValue <index> {0|1}`
Switchs channel `<index>` on or off.

`getNum`
Returns the number of channels controlled by this port.

7.20 PortMultiMenu

This port provides one or more combo boxes with a variable number of items each. At any time exactly one item is selected in each combo box.



Modules using this port are for example SurfaceView or GridBoundary.

Commands

Inherits all commands of Port.

`getValue [<menuindex>]`
Returns the index of the item currently being selected in the specified menu or in the first menu, if no argument is given.

`setValue [<menuindex>] <itemindex>`
Selects item `<itemindex>` in the specified menu or in the first menu, if only one argument is given.

`setValueString [<menuindex>] <label>`
Selects the item with the specified label string in menu `<menuindex>` or in the first menu, if only one argument is given. If no item with such a label exists, nothing happens and 0 is returned. Othwerwise 1 is returned.

`setNum [<menuindex>] <numitems>`
Sets the number of items in menu `<menuindex>` or in the first menu, if only one argument is given.

`getNum [<menuindex>]`
Returns the number of items in menu `<menuindex>` or in the first menu, if only one argument is given.

`setLabel [<menuindex>] [<itemindex>] <newlabel>`
If only one argument is given, this command sets the port's label. If two arguments are given, this command sets the label of item `<itemindex>` in the first menu. If three arguments are given, this command set the label of item `<itemindex>` in menu `<menuindex>`.

`getLabel [<menuindex>] [<itemindex>]`
If no argument is given, this command returns the port's label. If one argument is given, this command returns the label of item `<itemindex>` in the first menu. If two arguments are given, this command returns the label of item `<itemindex>` in menu `<menuindex>`.

`setMenuSensitivity [<menuindex>] {0|1}`
Enables or disables menu `<menuindex>` or the first menu, if `<menuindex>` is missing. If a menu is disabled it is grayed out and it does not accept any user input.

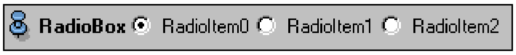
`getMenuSensitivity [<menuindex>]`
Checks whether the specified menu is enabled or not.

`setSensitivity [<menuindex>] <itemindex> {0|1}`
Enables or disables item `<itemindex>` in menu `<menuindex>` or in the first menu, if `<menuindex>` is missing. If an item is disabled it is no longer listed in the menu.

`getSensitivity [<menuindex>] <itemindex>`
Checks whether the specified item in menu `<menuindex>` is enabled or not.

7.21 PortRadioBox

This port provides multiple toggle buttons with a radio behaviour, i.e., only one button can be switched on at a time. Thus the port facilitates a one-of-many choice. For a larger number of possible choices usually a combo box as provided by PortMultiMenu is more suitable.



Modules using this port are for example OrthoSlice or SplineProbe.

Commands

Inherits all commands of Port.

`getValue`
Returns the index of the button currently being selected.

`setValue <index>`
Selects button `<index>` and deselects the previously selected button.

`setLabel [<index>] <label>`
If only one argument is given, this command sets the port's label. If two arguments are given, this command sets the label of radio button `<index>`.

`getLabel [<index>]`
If no arguments are given, this command returns the port's label. If one argument is given, this command returns the label of radio button `<index>`.

`setSensitivity <index> {0|1}`
Enables or disables radio button `<index>`. A disabled button is grayed out and cannot be selected interactively.

`getSensitivity <index>`
Checks whether radio button `<index>` is enabled or not.

`getNum`
Returns the total number of radio buttons of the port.

7.22 PortSeparator

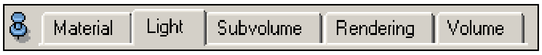
This port displays a horizontal line. It is used to visually separate different groups of ports from each other. In contrast to most other ports this port does not provide a pin button and thus cannot be pinned. It does not accept any user input.

Commands

Inherits all commands of Port.

7.23 PortTabBar

The QTabBar class provides a bar consisting of a list of labeled, selectable tab elements, e.g., for use in tabbed dialogs. Only one tab can be switched at a time. Thus the port facilitates a one-of-many choice. This port has all methods of a PortRadioBox with the additional ability to assign a list of port objects to every single tab element. If a tab element is chosen, all ports included in the tabs port list and which are also marked as visible, are scheduled for drawing. If a tab gets deselected, the ports included in it's port list become hidden. It's possible to add the same port to multiple tab elements and even to multiple QTabBar objects but it gets only visible if all its parent tab elements are selected, which is never the case for two tabs at the same QTabBar.



Modules using this port are for example VolPro.

Commands

Inherits all commands of PortRadioBox.

`portAdd <index> <objectLabel> <portLabel>`
Adds the given port `<object><port>` to the tab objects `<index>` port list.

`portRemove <index> <objectLabel> <portLabel>`
Removes the given port `<object><port>` from the tab objects `<index>` port list.

7.24 PortText

This port provides a simple text input field.

The port is used for example in the Arithmetic module.

Commands

Inherits all commands of Port.

`getValue`
Returns the contents of the text field.

`setValue <string>`
Sets the text in the text field to `<string>`.

`getNumColumns`
Returns the width of the text field in characters.

`setNumColumns <numcolumns>`
Sets the width of the text field so that `<numcolumns>` characters fit in.

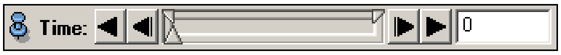
`getSensitivity`
Checks if the text field is enabled or not.

`setSensitivity {0|1}`
Enables or disables the text field. If the text field is disabled it is grayed out and it does not accept any user input.

7.25 PortTime

This port defines a floating point value like PortFloatSlider. However, the value is typically interpreted as a time value. It can be automatically animated in forward or backward direction, or it can be synchronized with any other object providing a time port, for example a global time object. In order to facilitate this kind of synchronization, *PortTime* is derived from Connection. If the port is connected to some other object providing a time port, the value of that port is used, i.e., the time port becomes an alias of the connected time port. The range of the time port as well as its increment (which is relevant for animation) can be edited in a little dialog which is pops up when pressing the right mouse button over the slider.

In addition to the main button the time slider also provides so-called subrange buttons. These buttons allow you to temporarily restrict the range of the slider. For animations only this restricted range is considered. The subrange buttons can be set to an exact value via the port's popup dialog or by typing `L <value>` or `R <value>` in the sliders text field. The letters `L` or `R` indicate that the left or right subrange button should be set, and not the main button.



Commands

Inherits all commands of Connection.

`getValue`
Returns the current time value.

`setValue <value>`
Sets the current time value.

`getMinMax`
Returns two numbers indicating the full range of the time port.

`setMinMax <min> <max>`
Sets the range of the time port.

`getSubMinMax`
Returns two numbers indicating the subrange of the time port. The subrange is relevant for animations.

`setSubMinMax <min> <max>`
Sets the subrange of the time port.

`getIncrement`

Returns the increment of the time port. During animations or when pressing the forward or backward buttons the time value is increased or decreased by the increment.

`setIncrement <value>`

Sets the increment of the time port.

`getDiscrete`

Checks if the time port is in discrete mode or not. In discrete mode the time can only take on values *min* + *n**increment*, where *min* is the minimum value of the range, *increment* is the port's increment value, and *n* is an integer value.

`setDiscrete {0|1}`

Sets discrete mode on or off.

`setFormat <format>`

Sets the format used for the slider's text field in printf syntax. Typically, the default format is `%g`.

`setTracking {0|1}`

Turns tracking on or off. If tracking is enabled the object the port belongs to is fired permanently while the moving the slider with the mouse. If tracking is off the object is only fired once when the mouse button is released.

`play [-forward|-backward] [-once|-loop|-swing]`

Starts to animate the time value. Additional arguments can be specified for the animation direction (forward or backward) and for the animation mode (play once, loop forever, play forth and back forever). If no arguments are specified forward play and the last animation mode are assumed.

`stop`

Stops a running animation.

`setRealTimeDuration <duration_in_seconds>`

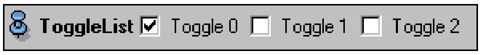
This command enables real time mode. In real time mode a full animation cycle takes as long as specified by this command. The increment added to the current time value after each animation step depends on the time required for that step. When real time mode is enabled discrete mode will be automatically disabled.

`getRealTimeDuration`

Returns the time duration of a full animation cycle in real time mode. If real time mode is disabled, 0 is returned. When the min or max time value is changed while real time mode is enabled the time duration is automatically adjusted.

7.26 PortToggleList

This port provides multiple toggle buttons. The buttons are implicitly numbered 0, 1, 2, ... from left to right and each button can be switched independently. Usually each toggle represents a particular option affecting the operation of a module.



The port is used for example in the LabelVoxel module.

Commands

Inherits all commands of Port.

`getValue <index>`
Returns 1 if toggle `<index>` is checked or 0 if it is not.

`setValue <index> {0|1}`
Sets the state of the button `<index>`.

`getNum`
Returns the number of toggle buttons of the port.

`setNum <numbuttons>`
Sets the number of toggle buttons of the port.

`getLabel [<index>]`
If no arguments are given, this command returns the port's label. If one argument is given, this command returns the label of toggle button `<index>`.

`setLabel [<index>] <label>`
If only one argument is given, this command sets the port's label. If two arguments are given, this command sets the label of toggle button `<index>`.

`getLabel [<index>]`
If no arguments are given, this command returns the port's label. If one argument is given, this command returns the label of toggle button `<index>`.

`setLabel [<index>] <label>`
If only one argument is given, this command sets the port's label. If two arguments are given, this command sets the label of toggle button `<index>`.

`getSensitivity <index>`
Checks whether toggle button `<index>` is enabled or not.

`setSensitivity <index> {0|1}`
Enables or disables toggle button `<index>`. A disabled button is grayed out and cannot be modified interactively.

Chapter 8

Alphabetic Index of Data Types

8.1 AnnaScalarField3

This data class represents a user defined 3D scalar field based on a regular grid. It provides a port *Expression* by which an arithmetic expression depending on uniform cartesian coordinates x, y, z can be entered that defines the value for each point of the unit cube. The range of values with respect to this (default) domain is indicated as *Component Range*. A data object of type *AnnaSccalarField3* has two additional input ports that can be connected to other data objects representing scalar fields on a regular grid, e.g., to image data objects. The predefined variables *a* and *b* are available for referencing such connected data objects in the arithmetic expression, this way a scalar field depending on other regular scalar fields can be defined. Whenever an expression evaluation is triggered to compute the value for a point (x,y,z), the variables *a* and/or *b* will be be substituted by the corresponding input values at the same point (x,y,z). For instance the value at a single point may be obtained by attaching a PointProbe module to the *AnnaScalarField3* data object and entering the point's cordinates by the *Coord* port of that module.

An expression consists of variables and mathematical and logical operators, the syntax is basically the same as for C expressions. The following variables are always defined:

- **x:** the x-coordinate of the current point
- **y:** the y-coordinate of the current point
- **z:** the z-coordinate of the current point

If data objects *ScalarField A* or *ScalarField B* are connected to port *InputA* resp. *InputB* the following variables are also defined:

- **a:** the values of input object A
- **b:** the values of input object B

The same C style mathematical and logical operators as well as built-in functions that are availabe for arithmetic expressions can be used here, see Arithmetic module description.

Connections

InputA `[optional]`

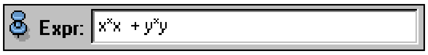
Optional scalar field.

InputB `[optional]`

Optional second scalar field.

Ports

Expr



```
x*x + y*y
```

Input field for arithmetic expression defining the scalar field values.

8.2 AnnaVectorField3

This data class represents a user defined 3D vector field based on a regular grid. It provides ports *X*, *Y*, *Z* by which arithmetic expressions can be entered that define the component mappings with respect to uniform Cartesian coordinates x, y, and z. The (default) domain is the unit cube, thus for each point (x,y,z) in it, the associated vector (X,Y,Z) is given by scalar functions X(x,y,z), Y(x,y,z), and Z(x,y,z) which can be specified by the user in the same way as a function for a HxAnnaScalarField3 data object.

The range of vector magnitudes, defined as Euclidean vectors lengths, is indicated as *Magnitude*.

A data object of type *HxAnnaVectorField2* has three additional input ports named *InputA*, *InputB*, *InputC* that can be connected to other data objects representing scalar or vector fields on a regular grid. There are some predefined variables for referencing such connected data objects in the arithmetic expressions, namely *a*, *b*, *c* for scalar fields and *ax*, *ay*, *az*, *bx*, *by*, *bz*, *cx*, *cy*, *cz* for x-, y-, resp. z-components of vector fields. This way a vector field depending on other regular scalar and/or vector fields can be defined. Whenever an evaluation of the three expressions is triggered to compute the vector associated to a point (x,y,z), each of the variables mentioned that occurs in them will be substituted by the corresponding input values at the same point (x,y,z). For instance the component values of a vector associated to a single point may be obtained by attaching a PointProbe module to the *HxVectorField3* data object, entering the point's cordinates by the *Coord* port of that module and setting the *Vector* toggle to *all*.

An expression consists of variables and mathematical and logical operators, the syntax is basically the same as for C expressions. The following variables are always defined:

- **x** - the x-coordinate of the current point

- **y** - the y-coordinate of the current point
- **z** - the z-coordinate of the current point

If a scalar data object is connected to any of the ports *InputA*, *InputB*, *InputC* a corresponding variable for access to the data values is also defined, namely:

- **a** - the values of input object A
- **b** - the values of input object B
- **c** - the values of input object C

If a vector data object is connected to any of the ports *InputA*, *InputB*, *InputC* corresponding component variables for access to the data values are also defined, namely:

- **ax, ay, az** - the xyz vector components of input object A
- **bx, by, bz** - the xyz vector components of input object B
- **cx, cy, cz** - the xyz vector components of input object C

The same C style mathematical and logical operators as well as built-in functions that are availabe for arithmetic expressions can be used here, see Arithmetic module description.

Connections

InputA `[optional]`

Optional scalar or vector field.

InputB `[optional]`

Optional second scalar or vector field.

InputC `[optional]`

Optional third scalar or vector field.

Ports

X

X: 5

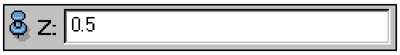
Input field for arithmetic expression defining the x-component field values.

Y

Y: 10

Input field for arithmetic expression defining the y-component field values.

Z



Input field for arithmetic expression defining the z-component field values.

8.3 CameraRotate

This object can be created from the main window's *Create* menu. It lets you rotate the cameras of all viewers activated in the object's viewer mask. This is useful in order to create simple animations. The animations can be stored in MPEG movie files by attaching a MovieMaker module to this object. More complex camera animations can be created using the *CameraPath* object and its associated editor

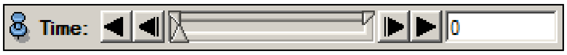
Connections

Time `[optional]`

Optional connection to some other object providing a time source. This allows you to synchronize multiple time dependent objects.

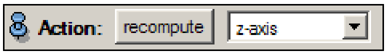
Ports

Time



The current time value. Changing the time modifies the cameras in all viewers activated in the object's viewer mask, i.e., with the orange viewer mask button being set.

Action



This port lets you specify the orientation of the camera rotation. Whenever the *recompute* button is pressed or a new menu option is selected the camera path is recomputed, i.e., the center of rotation and the radius are determined by analysing the camera in the main viewer.

8.4 Cluster

Data objects of type *Cluster* are used to represent sets of 3D points with additional data variables associated to them. For each point at least the x-, y-, and z-coordinate as well as an additional index is stored. The index is an arbitrary number which can be used to identify corresponding points in different data sets.

Moreover, an arbitrary number of additional data columns may be defined. The elements of a data column may be stored as 32-bit floats, as 32-bit integers, or as 8-bit characters. Each data column also has a string specifying its name. Optionally, a symbol may be defined, which is used to denote the column in an arithmetic filter expression as provided by the modules ClusterView and ClusterDiff.

Commands

`resetIds`

Resets all point indices. The index of the first point will be set to 0, the index of the second point will be set to 1, and so on.

`computeConnectivity`

Computes connectivity information required to display bonds between neighbouring points. Bond detection does not take into account any chemical information. Instead, merely nearest neighbours are computed. Each point may have at most 12 such neighbours. In addition the length of the longest bond of a point may not be larger than 1.4 times the length of the shortest one.

8.5 ColorField3

An RGBA color field is a regular 3D field with 4 data components per voxel. Each data component is an 8-byte value. The first 3 components are interpreted as red, green, and blue color values. The fourth component represents an opacity value (alpha). Color fields usually have uniform coordinates, i.e., equal slice distances in all directions, but other coordinate types are possible as well.

Color fields can be visualized using the slicing modules OrthoSlice and ObliqueSlice. They are especially useful in combination with a Voltex module for direct volume rendering. Since the color field stores an RGBA tuple for each voxel, no additional transfer function is required. This allows you for example to visualize different data sets in a single volume rendered image at once. The conversion of one or multiple scalar fields into a color field is accomplished using the ColorCombine module.

Commands

`alphaAverage [min] [max]`

Sets the alpha value of all voxels equal to the luminance 0.3*R + 0.59*G + 0.1*B, i.e., brighter objects become more opaque. If min and max are specified the alpha values are scaled so that they fill this range. On default min and max are 0 and 255, respectively.

`alphaSet <alpha>`

Sets the alpha value of all voxels to the specified value.

`alphaThreshold <luminance>`

Sets the alpha value of all voxels with a luminance smaller than the specified value to 0. The alpha of all other voxels is set to 255. Luminance is computed as 0.3*R + 0.59*G + 0.1*B.

`swapRGBA`

Swaps ABGR tupels into RGBA tupels or vice versa.

8.6 Colormap

A *Colormap* is a sequence of RGBA-tupels, where every tupel specifes a color by a red, green and blue value in the *RGB color model*. Each value ranges from 0.0 to 1.0 and is represented by a floating point value. A fourth value, the so-called alpha value, defines *opacity*. It also ranges from 0.0 to 1.0, where 0.0 means that the color is fully transparent, and 1.0 that the color is fully opaque. A colormap usually stores 256 different RGBA-tupels, but other sizes and even procedurally defined colormaps are possible too.

Beside the raw RGBA values the colormap also stores two *coordinates*, defining a range used for color interpolation. Color lookup requests for an argument smaller than the minimum coordinate evaluate to the first colormap entry. Requests for an argument greater than the maximum coordinate evaluate to the last entry.

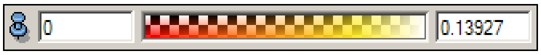
Connections

Datafield

Connection to a data field from which the min-max values of the colormap are taken.

Ports

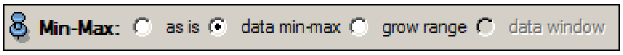
Colormap



0 0.13927

This port displays the contents of the colormap. Transparent values are drawn over a checkerboard background. The coordinate range can be edited via the two text fields left and right from the graphics area.

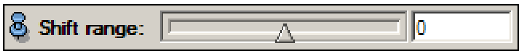
Min-Max



Min-Max: as is [✓ data min-max] grow range data window

This port is only visible if an input is connected to port *Datafield*. If this is the case and *data min-max* is selected, the coordinate range of the colormap is automatically adjusted so that it matches the min max values of the connected data field. If *grow range* is selected the coordinate range is enlarged if the min max values of the connected data field fall outside the current range. The option *data window* becomes active only if the connected data field contains a parameter *DataWindow*. If the option is selected the coordinate range is set to the range specified by the *DataWindow* parameter (see also Section 3.2.7 (Parameters)).

Shift range



Shift range: 0

This slider allows you to shift the range assumed for a connected data field. Instead of the true min max values shifted values are used if the value is different from 0.

Scale range

This slider allows you to scale the range assumed for a connected data field. Instead of the true min max values scaled values are used if the value is different from 1. The scaling is applied relative to the shifted center of the range. The shifted and scaled range will always be clamped to the original range. The shift and scale ports are useful to investigate certain subrange of a data set in detail.

Commands

Inherits all commands of Data.

`setMinMax <min> <max>`
Sets the coordinate range of the colormap.

`minCoord`
Returns the lower bound of the coordinate range.

`maxCoord`
Returns the upper bound of the coordinate range.

`isTransparent`
Checks whether the colormap contains transparent values or not.

`getRGBA <u>`
Returns the interpolated RGBA values for the parameter `<u>`, which is a value between 0.0 and 1.0. The value 0.0 corresponds to the first colormap entry, while 1.0 corresponds to the last colormap entry.

`getRGB <u>`
Similar to the above command, but returns only three values (RGB, not alpha).

`makeSteps <numsteps>`
Discretizes the colormap so that only *¡numsteps¿* different colors remain.

`setInterpolate {0|1}`
Turns interpolation on or off. When interpolation is on, the colors of two neighbouring colormap entries are interpolated when a color is looked up. When interpolation is off, the color of the nearest colormap entry is returned. This is be useful is a colormap modified with the `makeSteps` command is used.

`getInterpolate`
Checks wheter interpolation is on or off.

`makeRandom`
Replaces all colors of the colormap by random values.

`makeVolren <r> <g> <b> <power> <huewidth>`
Replaces the colormap by something useful for volume rendering. The first three arguments specify the base color of the colormap. *¡power¿* denotes an exponent used to compute an alpha curve. If this is 1 the colormap gos linearly from fully opaque to fully transparent. `<huewidth>` indicates the width of the color spectrum around the base color between 0 and 1.

`interpol <map1> <map2> <u>`
Replaces the colormap by the weighted average of two other colormaps `<map1>` and `<map2>`. The interpolation parameter `<u>` should be chosen between 0 and 1.

8.7 Data

This is the base class of all amira data objects. Data objects are usually represented by green icons in the object pool. In contrast to modules, data objects can be duplicated and saved to a file. They also provide a hierachical list of parameters or attributes. This list can be edited using the parameter editor.

Commands

Inherits all commands of Object.

`touch`
Touches the data object, marking it as modified. When the network is fired modules connected to a data object will only be invoked if the data object was modified.

`duplicate`
Duplicates the data object and returns the name of the duplicated obejct.

`save [format filename]`
Saves the data object. If no arguments are specified the command does the same as choosing *Save* from the *File* menu, i.e., it saves the object under the same name as it was saved before. Otherwise, a format and a file name must be specified. The format should be the name of a format as it is displayed in the file dialog's file type menu, e.g., `"Amiramesh ascii"` or `"HxSurface binary"`. When an object is saved its name is replaced by a possibly modified version of the filename. The command returns the actual name of the object after it has been saved.

`parameters [options ...]`
Provides access to the data object's parameter list. This command takes several different options allowing you query and set parameters. A list of all folders containing parameters too can be obtained using `list`. The name of each folder is used to access that folder. For example, to get a list of all materials of a surface or of a label field, use `parameters Materials list`.

A parameter value can be set using `setValue <name> <value>`, and it can be returned using `getValue <name>`. For example, to set the color of the material *Exterior* of a surface or of a label field, use `parameters Materials Exterior setValue Color <color>`.

8.8 Field3

This class is the base class for all 3D fields in amira, e.g., scalar fields, vector fields, or color fields with uniform, stacked, or some other coordinates, or for fields defined on unstructured finite-element grids. This class provides a transparent interface to evaluate the field at any position without needing to know how the field is actually represented. This interface can be accessed via the Tcl command `eval` described below. A field may have an arbitrary number of data variables which can be queried using the Tcl command `nDataVar`. For example, a scalar field has one data variable, while a vector field has three.

Commands

Inherits all commands of SpatialData.

`nDataVar`
Returns the number of data variables of the field.

`eval <x> <y> <z>`
Evaluates the field at the position `<x> <y> <z>`. On success the command returns as many numbers as there are data variables. The command may fail because the specified position lies outside of the grid the field is defined on. In this case the string `domain error` is returned.

`getRange`
Returns the minimum and maximum data component of the field. For field with more than one data variable this is not the magnitude range of the field.

`primType`
Returns the primitive data type of the field, i.e., the way how the values are represented internally. A number with the following meaning is returned: 0 = bytes, 1 = 16-bit signed integers, 2 = 32-bit signed integers, 3 = 32-bit floating point values, 4 = 64-bit floating point values, 7 = 16-bit unsigned integers.

8.9 IvData

This is a simple data object which encapsulates an Open Inventor scene graph. The scene graph can be displayed in amira using the module IvDisplay. However, it cannot be edited or processed further. Instead, amira provides a separate data type Surface for representing triangular surfaces with connectivity information and with an optional patch structure. An Open Inventor scene graph can be converted into an amira surface object using the module IvToSurface.

8.10 LabelField3

Data objects of type *LabelField* are used to represent the result of a segmentation applied to a 3D image volume.

A *LabelField* is a regular cubic grid with the same dimensions as the underlying image volume. For each voxel it contains a label indicating the region that the voxel belongs to. Use module LabelVoxel to create a *LabelField* from an image data stack. You can manually modify a *LabelField* using amira's image editor GI.

In addition to the labels themselves a *LabelField* may also contain *weights* indicating the degree of confidence of the label assignment made for each voxel. Such weights are calculated automatically when you choose option *sub-voxel accuracy* in LabelVoxel, when you apply the *smoothing filter* of GI, or when you resample a *LabelField* to a smaller resolution using the Resample module.

You can visualize a *LabelField* by attaching an OrthoSlice module to it. If a *LabelField* contains weights, the port *Primary Array* allows you to choose whether the labels or the probabilities are to be displayed.

Connections

Master `[unused]`

ImageData `[required]`

Connection to the image data that the segmentation results refer to. You cannot connect this port to an image object with dimensions different from that of the *LabelField*, except the *LabelField* has been newly created via the *Edit Create* menu. In this case, the LabelField will be resized so that it matches the dimensions of the image object.

Ports

Primary Array

An option menu which only appears if the *LabelField* contains weights. In this case the menu lets you select whether the labels or the weights are the primary data array. The primary data array is the default array visible to modules expecting an ordinary uniform scalar field like OrthoSlice or Arithmetic.

Commands

`hasMaterial <name>`

Returns true if the specified material is defined in the material section of the *LabelField*.

`makeColormap`

Creates a new colormap object in amira's object pool, containing the default colors of all materials of the *LabelField*.

`relabel`
Computes new labels so that the materials are numbered in consecutive order starting from 0.

`deleteAltData`
Deletes the weight information if it is present.

8.11 LandmarkSet

This data type represents specific points or markers in 3D space. It can be used to flag markers in medical MRI images, or to specify pairs or n-tuples of corresponding points in multiple data sets.

An empty set of landmarks can be created by typing

```
create HxLandmarkSet
```

into the amira console window, cf. Section 3.1.8. Individual landmarks can be interactively added, repositioned, or removed from a landmark set by means of the landmark editor.

Commands

`setNumSets <n>`
Sets the number of point sets contained in this data object. Upon creation a landmark set contains one set of points. In order to represent pairs of corresponding points two sets are required.

`getNumSets`
Returns the number of point sets in this data object.

`setPoint <index> <x> <y> <z> [<set>]`
Sets the coordinates of the specified marker `<index>` in a particular set. If `<set>` is omitted the first set is used.

`getPoint <index> [<set>]`
Returns the coordinates of the specified marker `<index>` in a particular set. If `<set>` is omitted the first set is used.

`setOrientation <index> <x> <y> <z> <rad> [<set>]`
Sets the orientation of the specified marker `<index>` in a particular set. If `<set>` is omitted the first set is used. The orientation is specified by an axis plus an angle of rotation around this axis in radians.

`getOrientation <index> [<set>]`
Returns the orientation of the specified marker `<index>` in a particular set. If `<set>` is omitted the first set is used. The orientation is returned as an axis plus an angle of rotation around this axis in radians.

`appendLandmark <x> <y> <z>`
Appends a new marker to the data object. The new marker will have the same coordinates in all sets.

`removeLandmark <index>`
Removes the specified marker from all sets, reducing the number of points by one.

`swapSets [<set1> <set2>]`
Exchanges the coordinates of the specified sets. If no arguments are given the first and the second set are swapped, provided both sets exist.

`computeRigidTransform [<src-set> <dst-set>]`
Computes a rigid transformation which move the points of the first set as close as possible onto the points of the second set (the sum of the squared distances between corresponding points is minimized). The result is returned as a 4x4 transformation matrix, which for example can be used to transform some other data object using the `setTransform` command.

`translateCoords <x> <y> <z> [<set>]`
Translate the landmarks of the specified set by the given displacement vector. If `<set>` is omitted the landmarks in all sets are translated.

`scaleCoords [-center <x> <y> <z>] <xscale> [<yscale> <zscale>] [<set>]`
Scales the coordinates of the landmarks in the specified set. If `<set>` is omitted the landmarks in all sets are scaled. The optional argument `-center` defines the center of the scaling operation. If no center is specified the origin (0,0,0) will be used.

8.12 LargeDiskData

A LargeDiskData object is useful for large image data. It allows to extract subvolumes loaded as a usual field object. In this way amira can manage data bigger than main memory. Note that only uniform coordinates are supported.

A couple of fileformats might be loaded as LargeDiskData (AmiraMesh, Raw Data, Stacked-Slices, LargeDiskData).

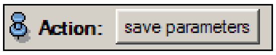
To access the data you have to attach an Access module. It provides an interface to load a subblock into amira. You can use all the amira visualization techniques on this subblock.

As every other amira data object a LargeDiskData object has parameters associated with it. In contrast to most other data types the LargeDiskData can not be saved directly. But some of the mentioned file formats allow to save the actual parameters to the file defining the LargeDiskData. This is done by hitting the *save paramters* button. It is only visible if you have write access to the file.

The *Save As* menu entry allows to export the data into a common image file format or as raw data. It does not save parameters. This might look strange at a first glance, but in contrast to all other data formats the LargeDiskData are not in main memory. The actual data reside only on disk and can not be saved to another place by amira.

Ports

Action



Save paramters to the file defining the LargeDiskData.

8.13 Lattice3

This class represents regular 3D data arrays. Every node of a regular data array can be addressed by an index tuple (i,j,k). The data array is characterized by its dimensions (the number of nodes in each direction), the primitive data type (e.g., bytes or shorts), the number of data variables per node, and by its coordinates. In amira uniform, stacked, rectilinear, and curvilinear coordinates are supported, compare Section 3.2.3 (Coordinates and Grids). *Lattice3* is a simple but powerful data type. In particular, all 2D and 3D images in amira are represented by this type.

As a technical detail it should be mentioned that in contrast to other data types *Lattice3* is not a data class by itself, i.e., it is not derived from Data or Object. Instead it is a so-called interface class which is used by other classes such as *RegScalarField3*, *RegVectorField3*, or *RegColorField3*. Usually this fact will not be important for end-users, but only for *amiraDev* users.

Commands

Data objects using this class inherit all commands of Field3.

`getDims`
Returns three numbers indicating the number of nodes in each direction of the 3D array.

`coordType`
Returns a number indicating the coordinate type of the lattice, 1 = uniform, 2 = stacked, 3 = rectilinear, 7 = curvilinear.

`getValue <i> <j> <k>`
Evaluates the field at the index position `<i> <j> <k>`. As many numbers are returned as there are data variables in the lattice.

`setValue <i> <j> <k> <value1> [<value2> ...]`
Sets the field values at the index position `<i> <j> <k>`. The number of values specified by this command must match the number of data variables of the field.

`swapByteOrder`
Swaps the byte order of the lattice's data values from litte endian to big endian or vice versa.

`clearSlice <k>`
Sets all values of slice `<k>` to zero.

`exchangeSlices <k1> <k2>`
Swaps the contents of the slices `<k1>` and `<k2>`.

`crop <imin> <imax> <jmin> <jmax> <kmin> <kmax> [<value>]`
Crops the lattice. The first six arguments specify the index bounds of the subvolume to be cropped. It is possible to enlarge the data set by specifiying negative lower bounds or upper bounds exceeding the current size of the lattice. In this case the last slice is replicated unless `<value>` is specified. If this is the case the new slices are initialized with `<value>`.

`flip {0|1|2}`
Flips the lattice in i-, j-, or k-direction, depeinding on whether the argument was 0, 1, or 2.

`swapDims <iIdx> <jIdx> <kIdx>`
Performs a kind of rotation about 90 degrees. The arguments tell at which position an index was before, i.e., they must be a permutation of 0, 1, 2. For example, to convert ijk into jki you have to use the arguments `1 2 0`.

`setBoundingBox <xmin> <xmax> <ymin> <ymax> <zmin> <zmax>`
Sets the bounding box of the lattice. The bounding box encloses the centers of all voxels of the lattice (not the complete voxels). In case of stacked, rectilinear, and curvilinear coordinates the coordinates of inner points are scaled appropriately.

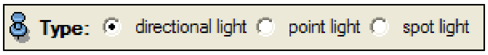
8.14 Light

Light objects are used to define additional lights in the amira viewer windows. Actually light objects are neither modules nor standard data objects. Nevertheless, they are displayed in the object pool. Like modules they provide some ports allowing the user to adjust the object's properties. In particular, three different light types are supported, namely directional lights, point lights, and spot lights. Lights can be define d in scene coordinates or in camera coordinates (camera slave mode). In order to interactively change the light parameters appropriate Open Inventor draggers can be activated.

Note, that the amira viewers define separate headlighta on default. Light objects represent additional light sources and are not related to the viewer's head light. New lights can be interactively created using the View Lights menu of the amira main window. This menu also allows to activate new light settings consisting of multiple lights. A light setting is stored as an amira script in the subdirectory `share/lights` in the amira installation directory. New settings can be added dynamically by copying new scripts into this directory.

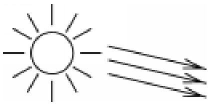
Ports

Type

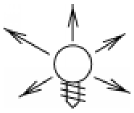

Type: [✓ directional light] point light spot light

This radio box determines the light type. Three different types are supported:

A *directional light* emits parallel light. It is faster to compute than the other lights. Another advantage is that it has no particular location (although the dragger used to edit the light is located somewhere). Therefore light settings consisting of directional lights only can be easily applied to new scenes, regardless of the actual size or position of the objects in that scene.

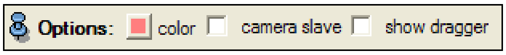


The second type is a *point light*. A point light is specified by its location only. It emits light symmetrically in all directions.



The third type is a *spot light*, which has a position like a point light, but which also defines cone restricting the shape of the light being emitted.

Options

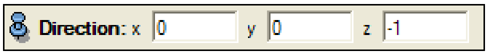

Options: 🟥 color ☐ camera slave ☐ show dragger

First, this port provides a color button indicating the color of the light. Pressing the button pops up the color dialog and lets you change the light's color.

The next toggle called *camera slave* specifies, whether the light remains fixed relative to the camera. If not, the light's position and direction are fixed with respect to other objects in the scene.

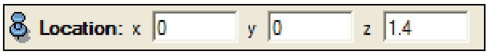
Finally, the last toggle called *show dragger* allows you to activate an Open Inventor dragger which can be used to move the light or to modify its direction.

Direction


Direction: x 0 y 0 z -1

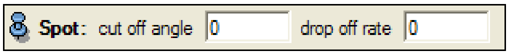
Shows the direction of the emitted light. The values represent the direction in world or camera coordinates depending on the *camera slave* setting. The port is not available for point lights.

Location



Shows the location of the light source. The values represent the location in world or camera coordinates depending on the *camera slave* setting. The port is not available for directional lights.

Spot



Here the two additional parameters for spot lights are specified:

The *cut off angle* determines the spread of the cone of the emitted light, measured from one edge of the cone to another.

The *drop off rate* controls how concentrated the light is. The light's intensity is highest in the center of the cone. It's attenuated toward the edges of the cone. A value of 0 produces very sharp edges, A value of 1 produces very soft edges.

Commands

Inherits all commands of Object.

`getColor`
Returns the color of the light as an RGB tuple of floating point number.

`setColor <color>`
Sets the color of the light. The color can be specified either as a tuple of three RGB integer values in the range 0...255, or as a tuple of three RGB floating point values in the range 0...1, or as a text string.

`getIntensity`
Returns the intensity of the light.

`setIntensity <value>`
Sets the intenisty of the light. The intensity modulates the light's color. Instead of modifying the light's intensity the brightness of the light's color could be changed as well.

`getDirection`
Returns the direction of the light (undefined for a point light).

`setDirection <x> <y> <z>`
Sets the direction of the light. Has no effect for a point light.

`getLocation`
Returns the location of the light. For a directional light the location of the associated light dragger is returned.

`setLocation <x> <y> <z>`
Sets the location of the light. For a directional light the location of the associated light dragger is set.

8.15 LineSet

A *LineSet* data object is able to store independent line segments of variable length. Optionally, for each vertex one or more scalar data items can be stored. *LineSet* objects inherit the vertex set interface, cf. Section 3.2.5. In order to visualize the line segements of a *LineSet* object the module LineSetView can be used. *LineSets* sets can be stored using the AmiraMesh file format.

Commands

Inherits all commands of VertexSet. In particular, the methods `getNumPoints`, `getPoint`, and `setPoint` are inherited.

`setNumPoints <num>`
Sets the number of points of the line set. The coordinates of new points need to be initialized afterwards using `setPoint`. Care must be taken that only existing points are referenced, i.e., that no point index is be bigger than `n-1`.

`addPoint <x> <y> <z>`
Adds a new point to the line set's vertex array. The new point will not yet be referenced by any line segment. The method returns the index of the new point.

`getNumDataValues`
Returns the number of data values per point.

`setNumDataValues <num>`
Sets the number of data values per vertex. New data values need to be initialized afterwards using `setData`.

`getNumLines`
Returns the number of lines.

`getLineLength <line>`
Returns the number of points of the specified line.

`getLineVertex <line> <point>`
Returns the index of point `<point>` of line `<line>`.

`getData <line> <point> [<set>]`
Returns the data value in set `<set>` of point `<point>` of line `<line>`. If `<set>` is omitted the first data value at that point is returned.

`setData <line> <point> <value> [<set>]`
Sets the data value in set `<set>` of point `<point>` of line `<line>`. If `<set>` is omitted the first data set is used. Before using this method the number of data sets has to be set using `setNumDataValues`.

`addLine <p1> [<p2> [<p3> ...]]`
Adds a new line consisting of the points `<p1>`, `<p2>`, `<p3>` ... to the line set. The index of the new line is returned.

`deleteLine <line>`
Deletes the specified line without changing the number of points of the line set.

`deleteAllLines`
Deletes all lines of the line set.

`removeLineVertex <line> <point>`
Delete point `<point>` of line `<line>`. The method does not remove the point from the global vertex array even if the point is not referenced by any other line.

`addLineVertex <line> <point> [<pos>]`
Adds an additional vertex to line `<line>`. The second argument `<point>` is the index of the referenced point. `<pos>` specifies the position of the new vertex within the line. If this argument is omitted the new vertex will be appended after all other vertices of the line.

`smooth <factor>`
Smooths the lines by replacing the coordinates of each vertex by the weighted average of its neighboring vertices. The bigger `<scale>` the more are the vertices smoothed.

`getRange`
Returns the min and the max of all data values of the line set.

8.16 Movie

This data module stores a movie description. The general concept of *Amira* movies is described in the manual section of the MoviePlayer module.

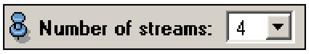
Connections

Master

If this port is connected to a MoviePlayer modules result port, this movie can be used as destination movie during a movie conversion process.

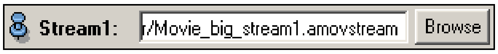
Ports

Number of streams


Number of streams: 4

Select the requested number of streams. After this, the appropriate number of the following file name fields will appear.

Stream1


Stream1: :/Movie_big_stream1.amovstream Browse

Source specification of the image sequence for stream one. This may be an *Amira* movie data file (.amovstream), a single image file or a sequence of images specified by a wildcard expression. For example specify c:/mymovie/left*.jpg to get all JPEG-images from directory c:/mymovie starting with ”left” in the filename. Possible wildcards are * and ?. * matches a sequence of arbitrary characters. ? matches a single character. If specifying an *Amira* movie data file (which is an *Amira* specific file containing a series of images) wildcards are not permitted. For experts: to specify a whole list of wildcard patterns or single image files edit the *Amira* movie info file (.amov).

Stream2

Stream2: :/Movie_big_stream2.amovstream Browse

Identical to above for stream number 2.

Stream3

Stream3: :/Movie_big_stream3.amovstream Browse

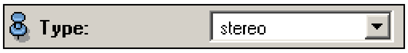
Identical to above for stream number 3.

Stream4

Stream4: :/Movie_big_stream4.amovstream Browse

Identical to above for stream number 4.

Type

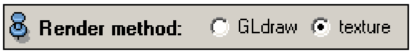

Type: stereo

Select here how the MoviePlayer module has to interpret and render the final image sequence.

- *mono* - every image forms a single frame.
- *stereo* - two images form a stereo frame.
- *stereo(interlaced)* - every image forms a stereo frame. The left eye channel is stored in the even lines and the right in the odd lines. Internally the module resorts the lines to get an image of the type *stereo(up/down)* .
- *stereo(up/down)* - every image forms a stereo frame. The left eye channel is taken from the upper half of the image and the right from the lower one.

- *stereo(left/right)* - every image forms a stereo frame. The left eye channel is taken from the left half of the image and the right from the right one.

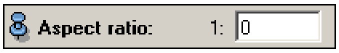
Render method



Render method: GLdraw [✓ texture]

This option affects the playback behavior. If set to *GLdraw*, the images are copied directly to the *OpenGL* frame buffer by using the function *glDrawPixels()*. If set to *texture*, the images are first transfered into an *OpenGL* texture object and then rendered as polygons textured with this texture. If an image was compressed using the *OpenGL* texture compression feature by a preceding movie conversion process, this image gets rendered as textured polygon independently of how this port is set. Be warned, that *OpenGL* texture compression is not available for all systems. *SGI* for example seems to implement it less often in it's *OpenGL*. On *PC* systems *OpenGL* texture compression seems to be a standard, due to the limited bandwidth and memory storage. Which render method performs better depends on the individual hardware and software conditions.

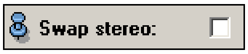
Aspect ratio



Aspect ratio: 1: 0

Force the the aspect ratio of the rendered images to a fixed value. If set to *0* the aspect ratio is taken from the individual image resolutions.

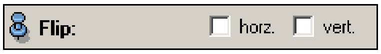
Swap stereo



Swap stereo: ☐

Swap left and right images for stereo movies.

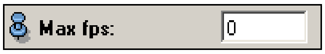
Flip



Flip: ☐ horz. ☐ vert.

Flip the movie in X- and/or Y-direction.

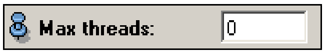
Max fps



Max fps: 0

Limit the playback speed to a maximal value of frames per second. Set this value also if the movie playback looks jerky, that gives the content retrival more time between the single frames. A value of *0* disables this feature.

Max threads



Max threads: 0

On multiprocessor systems the image retrieval and decompression is performed on default with so many threads as processors are available. That gives much speed but can hamper other users or tasks on this machine. Set this option to a value greater than *0* (which is to disable limitations) to change the number of retrival threads. Set this to 1 if the movie includes images read by non thread-save readers.

8.17 MultiChannelField3

Multi-channel objects are used to group multiple grey level images of the same size. Display modules such as OrthoSlice, ProjectionView, or Voltex then can be directly connected to the multi-channel object, thus allowing to operate on all channels simultaneously.

Multi-channel objects are created automatically when reading microscopic image files containing multi-channel information, e.g., Zeiss TIF files or Leica image files. Alternatively, channels can be manually attached to a multi-channel object. Details of how to work with multi-channel objects are described in a separate tutorial.

For each scalar field attached to a multi-channel object a special-purpose port is show, allowing you to define the channel's data window as well as its prefered color. The data window usually is interpreted in such a way that the lower data value is mapped to black while the upper is mapped to the channel's prefered color.

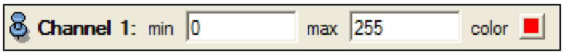
Connections

Channel 1 `[required]` Used to attach the first scalar field to the multi-channel object. This input determines the dimensions and the data type of the multi-channel object.

Channel 2 `[optional]` Used to attach a second scalar field to the multi-channel object. The second input must have the same dimensions and the same data type as the first one. After a second input has been connected a new input called *Channel 3* will be created, and so on. In this way an arbitrary number of channels can be conected.

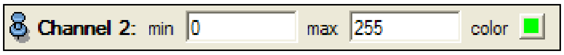
Ports

Channel 1



Determines the prefered data window of a channel as well as its color. For example, if the lower bound of the data window is set to 100, voxels with values smaller or equal than 100 will be drawn in black by the slicing modules OrthoSlice or ObliqueSlice.

Channel 2



Specifies the settings of the second channel.

8.18 Object

All amira objects represented by icons in the object pool are derived from this base class. The class provides some basic Tcl commands allowing to select or deselect on object, or to show or hide its icons. The class is not of interest for end users, but only for script programmers and developers.

Commands

`hasInterface <typename>`
Checks if the object provides an interface matching the specified type. For more information about interfaces please refer to the amira Programmer's Guide.

`showIcon`
Makes the object's icon visible.

`hideIcon`
Hides the object's icon. Although the object is no longer displayed the object itself is still contained in the object pool.

`iconVisible`
Checks if the object's icon is visible or not.

`select`
Selects the object, so that the ports are shown in the work area.

`deselect`
Deselects the object, hiding the ports in the work area.

`setLabel <name>`
Renames the object. If another object with the same name already exists, the specified name is modified so that it becomes unique. In any case, the new name of the object is returned.

`fire`
Updates the object and all downstream objects.

`compute`
Updates the object by calling its update and compute methods. In contrast to `fire` downstream objects are not updated.

`allPorts`
Returns a list of all ports of an object.

`connectionPorts`
Returns a list of all connection ports of an object.

`downStreamConnections`
Returns a list of all objects connected to this object. For each object also the name of the corresponding connection port is reported. That is, each element of the returned list in turn is a list containing the the name of the connected object and the name of the connection port.

`setIconPosition <x> <y>`
Sets the position of the object's icon in the object pool.

`getIconPosition`
Returns the position of the object's icon in the object pool.

`clipGeom <PlaneModule>`
Causes all geometry display of the object to be clipped by the plane defined by `<PlaneModule>`. For example, a plane module is any module derived from Arbitrary Cut. The geometry of an object might be clipped by up to six clipping planes. Also see `unclipGeom` below.

`unclipGeom <PlaneModule>`
Undos the effect of the `clipGeom` command described above.

`destroy`
The object is removed, as well as certain dependent objects.

`getTypeId`
Returns the type name of the object.

`help`
Displays all commands specific to that object.

`setLabel <name>`
Changes the name of the object to `<name>`. If already some other object with the same name exists, `<name>` will be automatically modified.

`setViewerMask <mask>`
This command is used to show a possible 3D output of the object in certain viewer windows and to hide it in other viewers. The bits in `<mask>` controls the viewers, e.g., a mask value of 2 shows the output in viewer 1 and hides it in viewer 0.

8.19 ScriptObject

Note: If you have worked with script objects prior to amira *version 3.0, please read the compatibility notes at the end of this file.*

amira is fully scriptable via its built-in Tcl interface (see Section 5 (Scripting)). The *ScriptObject* module allows the user or a custom solution provider to create scripts that fit seamlessly into the amira user interface, and define their own user interface components.

A script object is an object showing up in the object pool similar to an *OrthoSlice* or an *Axis* module. It can have ports, like sliders or buttons. The ports are defined by Tcl code and the reaction to a change of the ports is also implemented in Tcl. The Tcl code consists of a portion for initialization and a Tcl procedure that is called whenever the script has to react to changes of input parameters, the `compute` procedure.

Any amira script can be turned into a script object by putting a special header line into the script. Write an amira script (using your favorite text editor) and put a special header for script objects in the first line:

```
# Amira-Script-Object V3.0
echo "Hello world, a script is called."
```

Load this file into amira. A blue icon appears. Each time you click on the *Restart* button, the script will be read and executed and the above message appears in the console window.

During the execution of a script object, the global variable `$this` always contains the name of the currently active script object. This allows to easily access object-specific commands, the so-called *methods*. Declaring a method is very similar to declaring an ordinary Tcl procedure:

```
$this proc name args body
```

Just like the Tcl command `proc`, you can declare a method by using `$this proc`. Methods can be executed by calling `$this name args`. The syntax is completely analogous to global Tcl procedures (see Section 5.2 (Introduction to Tcl)). The special point about a method is that inside the method the `$this` variable is set appropriately. Example:

```
# Amira-Script-Object V3.0
$this proc sayHello {} {
   echo "module $this is greeting you"
}
```

If you load this script object, nothing will happen visibly. However, if your script object is called `MyScript.scro`, you can type

```
MyScript.scro sayHello
```

in the amira console window, and you will get a personalized greeting line as the result.

There are several methods with a special meaning:

- `$this proc constructor {} {...}` defines a method that is called when the script object is created. This is used for creating user interface elements and initializing the object.

- `$this proc destructor {} {...}` defines a method that is executed when the object is deleted or restarted. Used for cleaning up or terminating communications.
- `$this proc compute {} {...}` defines a method that is called whenever a user interface component of the script object is changed by the user. See examples below.

Here is an example that uses the `constructor` and `compute` methods in order to define a simple user interface and to query the current state of that user interface:

```
# Amira-Script-Object V3.0

$this proc constructor {} {
  $this newPortIntSlider myValue
  $this myValue setLabel "Value:"
}

$this proc compute {} {
  set val [$this myValue getValue]
  echo "The value is $val"
}
```

In the example, the constructor creates a new port, an integer slider which will appear in the user interface. The port has the internal name *myValue* and its visible label is set to *Value*. Whenever the user modifies the value of the slider, the `compute` method is called, and outputs the current port value.

In addition to defining methods, a script object also allows to define member variables. Analogous to Tcl variables, a member variable is a placeholder for a certain value, but a member is local to each script object. If you have two script objects `A` and `B`, both can have a member variable `x`, and the values of these two variables is kept separately. In order to define and query member variables, use the commands `$this setVar` and `$this getVar` (see below).

You can save amira networks containing script objects. When loading the saved network into amira, the following things will happen:

- the script object is created
- the saved value of all member variables is restored
- the `constructor` method is called
- the `compute` method is called
- the value of all ports is restored
- the `compute` method is called again

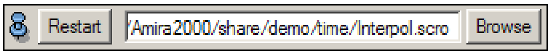
Connections

Data `[optional]`

Can be connected to any data object. Can be used by the script.

Ports

Script



This port is available for any script object. The *Restart* button deletes all dynamically created ports, sets the `isFirstCall` flag to `1` and calls the script. The text field indicates the location of the script file.

Commands

`newPortButtonList <name> <number-of-buttons>`
Creates a new button list port.

`newPortButtonMenu <name> <number-of-buttons> <number-of-options>`
Creates a new button menu port.

`newPortColormap <name>`
Creates a new colormap port.

`newPortFilename <name>`
Creates a new filename port.

`newPortFloatSlider <name>`
Creates a new float slider port.

`newPortFloatTextN <name> <number-of-fields>`
Creates a new float text port.

`newPortMultiMenu <name> <num-options-1> [<num-options-2> ...]`
Creates a new multi menu port.

`newPortInfo <name>`
Creates a new info port.

`newPortIntSlider <name>`
Creates a new integer slider port.

`newPortIntTextN <name> <number-of-fields>`
Creates a new integer text port.

`newPortRadioBox <name> <number-of-toggles>`
Creates a new radio box port.

`newPortSeparator <name>`
Creates a new separator port.

`newPortText <name>`
Creates a new text port.

`newPortTime <name>`
Creates a new time port.

`newPortToggleList <name> <number-of-toggles>`
Creates a new toggle list port.

`newPortConnection <name> <type-name>`
Creates a new connection port. The type name specifies what type of objects can be connected to the port. The type name of an existing object can be obatined using the Tcl command `getTypeId`.

`deletePort <name-of-port>`
Deletes a port which has been created using one of the *newPort* commands.

`proc name args body`
Define a Tcl member procedure (see above). The syntax is analogous to the global Tcl proc command. This command is not specific to the ScriptObject, but it is available in all amira objects.

`setVar <variable> <value>`
Variables stored in this way keep their values between successive calls of the `compute` procedure. Ordinary Tcl variables get lost. This command is not specific to the ScriptObject, but it is available in all amira objects.

`getVar <variable>`
Returns the value of a variable set using `setVar`. This command is not specific to the ScriptObject, but it is available in all amira objects.

`testBreak`
This commands checks if the stop button has been pressed. If so the execution of the script is automatically terminated. Use this command inside long animation loops or similar constructs.

In addition to the script object extensions, each script objects inherits a number of methods from the general amira object type.

Compatibility Note: The semantic of script objects has slightly changed with amira 3.0 compared to older amira versions. If the first line of the script contains the line "# Amira-Script-Object V0.1", the old behavior is enforced.

8.20 SpatialData

In amira all data objects embedded in 3D space are derived from this class. Every spatial data object provides a 3D bounding box as well as an optional transformation matrix. The transformation matrix allows the user to translate, rotate, or scale the object and the geometry of any display modules attached to it. Transformations can be defined interactively using the Transform Editor, or or from a script using the Tcl commands described below.

Commands

Inherits all commands of Data.

`getBoundingBox`
Returns the bounding box of the data object. The bounding box consists of 6 values denoting the xmin, xmax, ymin, ymax, zmin, and zmax coordinates in that order. For data objects defined by a set of discrete points like point clusters, surfaces, tetrahedral or hexahedral grids, the bounding box is the smallest box containing all points. For 3D images it is the smallest box containing all voxel centers, but not all voxels as is.

`getTransform [-d]`
Returns the transformation matrix of the data object. The transformation matrix is a 4x4 matrix which can be applied to a 3D vector in homogeneous coordinates. It encodes a translation, rotation, and scaling operation. If the `-d` option is specified, the transformation matrix is returned in *decomposed* form, i.e., the translation, rotation, and scaling operations are separated.

`setTransform [<a11> <a12> ...  <a44>]`
Sets the transformation matrix of the data object. If no arguments are given the transformation is reset to the identity matrix.

`getInverseTransform`
Returns the inverse transformation of the data object as a 4x4 matrix.

`getTranslation`
Returns the translation part of the decomposed transformation matrix of the object.

`setTranslation [<x> <y> <z>]`
Sets the translation part of the decomposed transformation matrix of the object. If no arguments are given the translation part is reset to zero.

`getRotation`
Returns the rotation part of the decomposed transformation matrix of the object. Four number are returned. The first three number denote the axis of rotation. The fourth number denotes the angle of rotation in degrees (0...360).

`setRotation [-center <x> <y> <z>] <x> <y> <z> <degrees>`
Sets the rotation part part of the decomposed transformation matrix of the object. The rotation is specified by a rotation axis and an angle of rotation. The optional argument `center` can be used to specify the center of rotation.

`getScaleFactor`
Returns the scaling part of the decomposed transformation matrix of the object. Three number are returned, denoting the scaling in x-, y-, and z-direction.

`setScaleFactor [<x> <y> <z>]`
Sets the scaling part of the decomposed transformation matrix of the object. If no arguments are given the scaling part is reset to unity.

`translate [-l|-w] <x> <y> <z>`
Translates the object by modifying its transformation matrix. The optional argument `-l` indicates that the translation is applied in local coordinates (after the existing transformation). This is the default. The optional argument `-w` indicates that the translation is applied in world coordinates (before the existing transformation).

`rotate [-lx|ly|-lz|-wx|-wy|-wz| [-l|-w] <x> <y> <z>] <degrees>`
Rotates the object by modifying its transformation matrix. The object can be rotated around the local x-, y-, or z-axis or around the world x-, y-, or z-axis (as indicated by the arguments `-lx` to `-wza`). Alternatively, the obejct can be rotated around a user-specified axis in either local or world coordinates. `degrees` specifies the angle of rotation in degrees (0...360).

`scale [-l|-w] <x> <y> <z>`
Scales the object by modifying its transformation matrix. The optional argument `-l` indicates that the scaling is applied in local coordinates (after the existing transformation). This is the default. The optional argument `-w` indicates that the scaling is applied in world coordinates (before the existing transformation).

`multTransform [-l|-r] <a11> <a12> ... <a44>`
Multiplies the current transformation matrix with the specified matrix. The arguments `-l` and `-r` indicate whether the matrix should be multiplied from left or from right. Multiplication from left means that the matrix is applied to the objects local coordinates.

8.21 SpreadSheet

This data type represents a spreadsheet. A spreadsheet will be created e.g., by the module TissueStatistics.

8.22 Surface

In amira data objects of type *Surface* are used to represent non-manifold triangulated surfaces. Such surfaces are required as an intermediate step in generating a tetrahedral patient model from the results of segmentation of a 3D image stack.

Surfaces mainly consist of a list of triangles as well as a list of 3D coordinates. Each triangle is defined by three indices pointing into the list of coordinates. Moreover, triangles are grouped into so-called *patches*. Conceptually, a patch describes the boundary between two adjacent regions (tissue types). These two regions, called *inner region* and *outer region*, are represented by indices into the surface's *material list*. Although required for grid generation, the patch structure of a surface does not necessarily define a valid space partitioning. However, any surface must have at least one patch.

Surfaces may also contain additional data, such as edges, boundary contours, or connectivity information. Because these data can be computed online they are usually not written into a file. If the data are not already present but a certain module requests them, they are recomputed automatically. You may notice a small time delay in this case. Recomputation of the connectivity information can be enforced by the Tcl-command `recompute`.

You may use the SurfaceGen module to extract boundary surfaces from a LabelField describing the results of s segmentation. You can visualize surfaces by attaching a SurfaceView module to it.

There are two editors that can be applied to modify surfaces: the Surface Simplification Editor and the Surface Editor. Use the former once to reduce the number of triangles contained in the surfaces. The latter allows you to perform an intersection test and to modify the surfaces manually.

Commands

`recompute`
Recomputes any additional data such as edges, boundary contours, or connectivity information from scratch. If the surface contained patches consisting of unconnected groups of triangles these patches are automatically subdivided into new patches consisting of connected triangles only.

`fixOrientation [patch]`
Checks if all triangles of a given patch are oriented in the same way. If this is not the case some triangles will be inverted in order to fix the orientation. If no patch number is specified all patches of the surface will be processed in this way.

`invertOrientation`
Inverts all triangles of the surface.

`makeOnePatch`
Puts all triangles of the surface into a single patch.

`cleanup`
Removes any additional data such as edges, boundary contours, or connectivity information from the surface.

`getArea <i>`
Compute area of all surface patches incident on material `i`.

`getVolume <i>`
Compute volume enclosed by material `i`.

`setColor <material> <color>`
Defines the color of a material used in module SurfaceView. The material may be specified by

either a material name or by a material index. The color may be specified by either an RGB triple in range 0...1 or by a common X11 color name, e.g., *red* or *blue*.

`setTransparency <material> <t>`
Defines the transparency of a material used in module SurfaceView when draw style is set to transparent. The material may be specified by either a material name or by a material index. The transparency value `t` must by a floating point number in range 0...1.

`add -point <x> <y> <z>`
Adds a new point to the surface. The method returns the index of the new point.

`add -triangle <p1> <p2> <p3>`
Adds a new triangle to the surface and returns its index. The triangle will be inserted into the first patch of the surface. If no patch yet exists one will be created.

`refine`
Refines the surface by subdividing all edges. After this operation the surface will contain four times the number of triangles.

8.23 TetraGrid

A data object of type *TetraGrid* represents an unstructured finite-element grid composed of tetrahedra. The geometric information is stored in terms of vertices, edges, faces, and tetrahedra. For instance such data objects are useful as patient models. Like a LabelField with its uniform hexahedral grid structure a tetrahedral grid also contains a 'dictionary' of different material types or regions. In addition to the material names the dictionary may contain colors and other parameters related to material properties.

amira is able to reconstruct tetrahedral grids from 3D image data. This procedure involves several steps, including image segmentation, extraction of boundary faces, surface simplification, and finally grid generation. The tutorial in Section 2.7 of the user's guide illustrated this process in more detail. The actual grid generation step is performed by the computational module TetraGen. The quality of a tetrahedral grid may be improved by applying certain operations provided by the Grid Editor.

Commands

`hasMaterial <name>`
Returns true if the specified material is defined in the material section of the *TetraGrid*.

`hasDuplicatedNodes`
Returns the number of duplicated nodes, i.e., nodes with exact identical coordinates. Such nodes may be used in order to represent discontinous piecewise linear fields.

`removeDuplicatedPoints`
Removes all duplicated points from the grid. No field object must be connected to the grid.

`add <othergrid>`
Copies all vertices and tetrahedra from an other tetrahedral grid into this one.

`removeTetra <n>`
Marks the tetrahedral cell specified by `<n>` as obsolete.

`cleanUp`
Removes all obsolete tetrahedra from the grid.

`fixOrientation`
Fixes the orientation of all tetrahedra so that the enclosed volume is positive.

8.24 Time

Modules such as Time Series Control or other data objects dealing with time-dependent data provide a time port, i.e., a special slider which also can be animated. Multiple such modules can be synchronized by connecting them to one global *Time* object. The time object provides the same functionality as the time port in the modules themselves. In fact, the time value can be changed in both the time port or in an upstream time object. A time object can be created by choosing *Time* from the main window's *Edit Create* menu. Alternatively, it can be created by choosing *Create time* from the popup menu of a time port (press the right mouse button over a time slider in order to activate this menu).

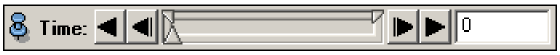
Connections

Time `[optional]`

Connection to an upstream time object. Usually there will only be one instance of a time object and this port will not be connected.

Ports

Time



This slider specifies the current time value. Additional settings can be modified via a popup menu which is activated by pressing the right mouse button over the slider. The inner buttons proceed one step in backward or forward direction, respectively. The outer buttons activate animation mode. The popup menu lets you choose between simple animation (play once) and two endless modes (loop and swing). Animation is always restricted to a subrange of the whole time interval. The subrange can be controlled graphically via the two upper arrow buttons. All settings can also be adjusted in a configure dialog which can be activated via the popup menu, too.

8.25 VertexSet

The *HxVertexSet* class is an abstract base class. It is derived by many other amira data objects containing a list of 3D vertices, for example landmark sets, surfaces, or tetrahedral grids. *HxVertexSet* provides an interface allowing other modules to access the vertices in a transparent way.

In order to visualize the vertices of a vertex set you may use the VertexView module.

Commands

`applyTransform`

This command changes the coordinates of the vertices of the data object according to the object's current transformation matrix. This matrix can been defined using the Transform Editor. After the vertices have been transformed the transformation matrix is reset to the identity.

This command is useful in order to make transformations permanent. In particular, it should be issued before a transformed data object is written to a file. Otherwise, the transformation will be ignored by most file formats.

`translate <dx> <dy> <dz>`

Translates all vertices by a constant amount.

`scale {<f> | <fx> <fy> <fz>}`

Scales all vertices by a common factor. If three arguments are specified the x-, y-, and z-coordinates are scaled by different factors.

`jitter {<d> | <dx> <dy> <dz>}`

The coordinates of the vertices are jittered randomly. The arguments indicate the maximal amount of jitter. Each coordinate of a vertex will be changed change by at most $\pm d/2$. The method is useful in order to resolve problems due to degenerate configurations in certain geometric algorithms.

`getNumPoints`

Returns the number of vertices of the data object.

`getPoint <n>`

Returns the coordinates of the specified vertex.

`setPoint <n> <x> <y> <z>`

Sets the coordinates of the specified vertex.

Chapter 9

Alphabetic Index of File Formats

9.1 ACR-NEMA

The ACR-NEMA file format is the predecessor of the DICOM standard file format for medical images. amira can import both old ACR_NEMA files and new DICOM files. Actually, both formats are interpreted by the same reader. Therefore, for more information please refer to the documentation of the DICOM reader. In contrast to DICOM files ACR_NEMA files can not be exported by amira.

9.2 AVS Field

This format provides a bridge between amira and AVS, the *Advanced Visual System* software. It enables you to read and write data in the native AVS field format.

Note that AVS supports fields of arbitrary dimensions and with an arbitrary number of components per node. For some combinations there are no corresponding amira data objects. Thus, the following restrictions apply:

- Only three-dimensional fields will be handled.
- AVS fields with a one-component data vector at each voxel will be loaded as a *regular scalar field*.
- Two-component fields will become *regular complex fields*.
- Three-component fields will become *regular vector fields*.
- Four-component fields will be loaded as *RGBA color fields*, provided they are defined on a uniform lattice. If not, they are rejected.
- Six-component fields will be loaded as *regular complex vector fields*.

- Since AVS does not support *stacked coordinates* these will be written as rectilinear fields and therefore appear as such when reloaded into amira.

Amira identifies AVS Field files by the file name suffix `.fld`.

9.3 AVS UCD Format

The AVS UCD (*Unstructured Cell Data*) format can be used to represent finite-element grids and associated data fields in 2 and 3 dimensions. Currently, in amira only 2D triangular cells, 3D tetrahedral cells, and 3D hexahedral cells are supported. Grids with these cells types will be converted into objects of type Surface, TetraGrid, or HexaGrid, respectively. Corresponding data fields will be converted into appropriate amira objects as well, provided the data is defined on a per-node basis. Cell data are currently not supported.

Two variants of the AVS UCD format exist, an ASCII version and a binary version. amira is able to read and write both of them. amira identifies AVS UCD data files by the file name suffix `.inp`.

9.4 Amira Script

amira scripts are written in the Tcl command language. Scripts allow you to start demos, to perform routine tasks, or to create animations. Networks are saved as script files too. When such a file is executed the network is restored. A detailed description of the amira scripting facilities is contained in the user's guide in Chapter 5 (Scripting). A script file is recognized either by the special comment `# Amira Script` in the first line of the file, or by the file extension `.hx`.

9.5 Amira Script Object

amira script objects are custom module with user-defined GUI elements (ports) written in the Tcl command language. Files describing script objects must obey certain requirements as stated in the data types section about ScriptObjects. When a file describing a scipt object is loaded an instance of a *ScriptObject* module is created and initialized as requested. Files describing script object are recognized by the special comment `# Amira Script Object V3.0` in the first line of the file, or by the file extension `.scro`. Note, that in previous versions of amira a different syntax was used for script objects. In order to make clear that a new-style script object is defined the comment mentioned above should be used.

9.6 AmiraMesh Format

AmiraMesh is Amira's native general-purpose file format. It is used to store many different data objects like fields defined on regular or tetrahedral grids, segmentation results, colormaps, or vertex sets such as landmarks. The format itself is very flexible. In fact, it can be used to save arbitrary multi-dimensional arrays into a file. In order to create an Amira data object from an AmiraMesh file the contents of the file are analysed and interpreted. For example, a tetrahedral grid is expected to have a one-dimensional array *Nodes* containing entries of type `float[3]` called *Coordinates*, as well as a one-dimensional array *Tetrahedra* containing entries of type `int[4]` called *Nodes*. If the Amiramesh file contains an entry *ContentType* in its parameter section the value of this parameter directly determines what kind of Amira data object is to be created.

A first example. In order to describe the syntax of an AmiraMesh file we first give a short example. This example describes a scalar field defined on a tetrahedral grid. Concrete examples of how to encode other data objects are given below.

```
# AmiraMesh ASCII 1.0

define Nodes 4
define Tetrahedra 1

Parameters {
    Info "This is an AmiraMesh example",
    Pi 3.1459
}

Materials { {
    Name "Stone",
    Color 0.8 0.3 0.1
} {
    Name "Water",
    Color 0 0.3 0.8
} }

Nodes { float[3] Coordinates } = @1
Tetrahedra { int[4] Nodes } = @4

Nodes { float Values } = @8
Tetrahedra { byte Materials } = @12

Field { float Example } = Linear(@8)

@1
0 0 0
1 0 0
0 1 0
0 0 1
```

```
@4
1 2 3 4

@8
0 0 0 1

@12
1
```

The first line of an AmiraMesh file should be a special comment including the identifier `AmiraMesh`. Moreover, if the tag `ASCII` is given in this line all data arrays are stored in plain ascii text. If the tag `BINARY` is given, the data arrays are stored in IEEE big-endian binary format. Note, that the header section of an AmiraMesh file is always given as ascii text.

The statement `define Nodes 4` defines a one-dimensional array of size 4. Later on, this array can be referenced using the name `Nodes`. Similarly, a statement `define Array 100 100` defines a two-dimensional array of size 100 x 100. The actual kind of data stored per array element will be specified later on.

The optional section `Parameters` allows the user to define arbitrary additional parameters. Each parameter consists of a name (like `Pi`) and a value (like `3.1459`). Values may be one or multiple integer or floating point numbers or a string. Strings have to be quoted using a pair of `"`-characters.

The optional section `Materials` allows the user to define additional material information. This is useful for finite element applications. The material section consists of a comma-separated list of parameters just as in the `Parameters` section.

The statement `Nodes { float[3] Coordinates } = @1` specifies that for each element of the array `Nodes` defined earlier three floating point numbers (floats) should be stored. These data are given the name `Coordinates` and a tag in this line and will appear below as a tagged block with the marker `@1`. Such data markers must always begin with the letter `@`.

Similar, the following lines define additional data to be stored in the arrays called `Nodes` and `Tetrahedra`. The primitive data types must be one of `byte`, `short`, `int`, `float`, `double`, or `complex`. Vectors of primitive data types are allowed, aggregate structs are not, however.

The statement `Field { float Example } = Linear(@8)` defines a continuous scalar field with the name `Example`. This field will be generated by linear interpolation from the data values `Values` defined on the nodes of the tetrahedral grid. Other interpolation methods include `Constant(@X)` and `EdgeElem(@X)`.

After the marker `@1` the coordinate values of the grid are stored. Likewise, the other data arrays are given after their corresponding markers. In case of a `BINARY` file the line containing the marker is read up to the next new line character. Then the specified number of bytes is read in binary format. It is assumed that sizeof(short) is 2, sizeof(int) is 4, sizeof(float) is 4, sizeof(double) is 8, and sizeof(complex) is 8. Multidimensional arrays indexed via `[k][j][i]` are read with `i` running fastest.

Backward compatibilty. For backward compatibility the following statements are considered to be equal:

`nNodes 99` is equal to `define Nodes 99`
`nTriangles 99` is equal to `define Triangles 99`
`nTetrahedra 99` is equal to `define Tetrahedra 99`
`nEdges 99` is equal to `define Edges 99`

`NodeData` is equal to `Nodes`
`TriangleData` is equal to `Triangles`
`TetrahedraData` is equal to `Tetrahedra`
`EdgeData` is equal to `Edges`

Other data objects. Of course, not only scalar fields defined on tetrahedral grids can be encoded using the Amiramesh format. Many other data objects are supported as well. In each case there are certain rules about what data arrays have to be written and how these arrays have to be named. Below, we describe how to encode the following data objects:

- Fields with uniform coordinates
- Fields with stacked coordinates
- Fields with rectilinear coordinates
- Fields with curvilinear coordinates
- Label fields for segmentation
- Landmarks for registration
- Line segments
- Colormaps

9.6.1 Fields with uniform coordinates

In order to encode 3D scalar or vector fields defined on a uniform grid you first have to define a 3D AmiraMesh array called *Lattice*. The field's data values are stored on this array. The coordinate type of the field as well as the bounding box are specified in the parameter section of the AmiraMesh file. This is illustrated in the following example:

```
# AmiraMesh ASCII 1.0

# Dimensions in x-, y-, and z-direction
define Lattice 2 2 2
```

```
Parameters {
    CoordType "uniform",
    # BoundingBox is xmin xmax ymin ymax zmin zmax
    BoundingBox 0 1 0 1 0 1
}

Lattice { float ScalarField } = @1

@1
0 0 1 1 0 0 2 2
```

Use `float[3]` in order to encode a vector field instead of a scalar field. Likewise, you may modify the field's primitive data type. For example, 3D images are commonly encoded using `byte` or `short`. Instead of `ScalarField` you may use any other name in the data definition statement.

The field's bounding box is given by the minimum and maximum x-, y-, and z-coordinates of the *grid nodes* or voxel centers, not of the voxel boundaries. Amira will always assume the width of a single voxel to be *(xmax-xmin)/(dims[0]-1)*. For degenerated 3D data sets with one dimension being 1 choose equal minimum and maximum coordinates in that direction.

9.6.2 Fields with stacked coordinates

A field with stacked coordinates has uniform pixel spacing in x- and y-direction, but slices may be arranged arbitrarily in z-direction. This type of coordinates is commonly used to encode 3D medical images with non-uniform spacing.

In order to encode a 3D scalar or vector field with stacked coordinates you have to define a 3D array called *Lattice* and 1D array called *Coordinates*. The field's data values are stored at *Lattice* while the slices' z-positions are stored at *Coordinates*. The coordinate type of the field as well as the bounding box in xy are specified in the parameter section of the AmiraMesh file. Here is an example:

```
# AmiraMesh ASCII 1.0

define Lattice 2 2 3
define Coordinates 3

Parameters {
    CoordType "stacked",
    # BoundingBoxXY is xmin xmax ymin ymax
    BoundingBoxXY 0 1 0 1
}
```

```
Lattice { byte Intensity } = @1
Coordinates { float z } = @2

@1
0 0 0 0
1 1 1 1
2 2 2 2

@2
0 0.75 2
```

Use `float[3]` in order to encode a vector field instead of a scalar field. Likewise, you may modify the field's primitive data type. For example, 3D images are commonly encoded using `byte` or `short`. Instead of `Intensity` you may use any other name in the data definition statement.

The field's xy bounding box is given by the minimum and maximum x- and y-coordinates of the *grid nodes* or pixel centers, not of the pixel boundaries. Amira will always assume the width of a single pixel to be *(xmax-xmin)/(dims[0]-1)*. For degenerated 3D data sets with one dimension being 1 choose equal minimum and maximum coordinates in that direction.

9.6.3 Fields with rectilinear coordinates

A field with rectilinear coordinates still has axis-aligned grid cells. However, the x-, y-, and z-coordinates of the grid nodes are specified explicitely for each direction.

In order to encode a 3D scalar or vector field with rectilinear coordinates you have to define a 3D array called *Lattice* and 1D array called *Coordinates*. The field's data values are stored at *Lattice* while the x-, y-, and z-positions for each direction are stored at *Coordinates* in subsequent order. The size of the *Coordinates* array must be equal to the sum of the sizes of *Lattice* in x-, y-, and z-direction. The coordinate type of the field is specified in the parameter section of the AmiraMesh file. Here is an example:

```
# AmiraMesh ASCII 1.0

define Lattice 2 2 3
define Coordinates 7 # This is 2+2+3

Parameters {
    CoordType "rectilinear"
}

Lattice { float ScalarField } = @1
```

```
Coordinates { float xyz } = @2

@1
0 0 0 0
1 1 1 1
2 2 2 2

@2
0 1     # x coordinates
0 1.5   # y coordinates
-1 1 2  # z coordinates
```

Use `float[3]` in order to encode a vector field instead of a scalar field. Likewise, you may modify the field's primitive data type. Instead of `ScalarField` you may use any other name in the data definition statement.

9.6.4 Fields with curvilinear coordinates

A field with curvilinear coordinates consists of a regular array of grid cells. Each grid node can be addressed by an index triple (i,j,k). The coordinates of the grid nodes are specified explicitly.

In order to encode a 3D scalar or vector field with curvilinear coordinates you have to define a 3D array called *Lattice*. This array is used to store the field's data values as well as the grid nodes' coordinates. The coordinates must be given as a `float[3]` vector containing x-, y-, and z-values. The coordinate type of the field is specified in the parameter section of the AmiraMesh file. Here is an example:

```
# AmiraMesh ASCII 1.0

define Lattice 2 2 3

Parameters {
    CoordType "curvilinear"
}

Lattice { float ScalarField } = @1
Lattice { float[3] Coordinates } = @2

@1 # 2x2x3 scalar values
0 0 0 0
1 1 1 1
2 2 2 2
```

```
@2 # 2x2x3 xyz coordinates
0 0 0
1 0 0
0 1 0
1 0 0
0 0 1
1 0 1
0 1 1
1 0 1
0 0 2
1 0 2
0 1 2
1 0 2
```

Use `float[3]` in order to encode a vector field instead of a scalar field. Likewise, you may modify the field's primitive data type. Instead of `ScalarField` you may use any other name in the data definition statement.

9.6.5 Label fields for segmentation

Label fields are closely related to ordinary scalar fields with uniform coordinates. However, the data values at each voxel are interpreted as labels denoting the different materials or regions the voxels have been assigned to during a segmentation process. Therefore, the most important difference of label fields compared to uniform scalar fields is the occurence of a *Materials* section in the AmiraMesh file. Whenever such a section occurs and elements of type *byte* denoted *Labels* are found the AmiraMesh file is interpreted as a label field. Here is a simple example of a label field containing two different materials:

```
# AmiraMesh ASCII 1.0

define Lattice 2 2 2

Parameters {
    CoordType "uniform",
    # BoundingBox is xmin xmax ymin ymax zmin zmax
    BoundingBox 0 1 0 1 0 1
}

Materials {
    { Id 0, Name "Exterior" }
    { Id 4, Name "Something" }
}

Lattice { byte Labels } = @1
Lattice { byte Probability } = @2

@1
0 0 0 4
0 0 4 4

@2
255 255 130 180
255 200 190 230
```

Each material is supposed to have a parameter *Id* specifying the correspondence between labels and materials. In the example above all voxels labeled with 0 belong to material *Exterior*, while all voxels labeled with 4 belong to material *Something*.

Optionally, label fields may contain probability information or weights as shown in the example above. These weights denote the degree of confidence of the labeling. This information is used by the GMC module when extracting boundary surfaces.

9.6.6 Landmarks for registration

The data type Landmark Set is useful for registration and alignment of multiple 3D image data sets. It allows you to store multiple sets of corresponding marker positions. The data type can also be used to represent a simple list of 3D points in Amira. In this case you would only specify a single set of markers. Consider the following example:

```
# AmiraMesh ASCII 1.0

define Markers 3

Parameters {
    ContentType "LandmarkSet",
    NumSets 2
}

Markers { float[3] Coordinates } = @1
Markers { float[3] Coordinates2 } = @2

@1
38.5363 15.2135 20.3196
```

```
35.1264 14.0106 37.155
31.6494 14.2791 31.0932

@2
40.2112 15.907 20.3119
35.9551 13.8241 40.4785
30.1375 13.7279 28.9235
```

In this example first the number of markers or points is defined to be 3. In the parameter section of the AmiraMesh file the content type is specified, as well as the number of marker sets. The marker coordinates of the first set are denoted `Coordinates` (xyz-values stored as `float[3]`). Likewise, the marker coordinates of the second set are denoted `Coordinates2`. If more sets are defined the coordinate values must be called `Coordinates3`, `Coordinates4`, and so on.

It is also possible to define additional data values for each marker such as `MarkerTypes` or `Orientations`. How these values are interpreted in detail will be specified in a future release of Amira.

9.6.7 Line segments

The data type Line Set is used to represent a generic set of indexed line segments, i.e., line segments defined by an index into a vertex list. Optionally, an arbitrary number of scalar data values may be associated with each vertex.

In order to store line sets in an AmiraMesh file two 1D arrays have to be defined, namely *Lines*, used to store the indices, and *Vertices*, used to store the vertex coordinates as well as additional vertex data. Here is an example:

```
# AmiraMesh ASCII 1.0

define Lines 15
define Vertices 12

Parameters {
    ContentType "HxLineSet"
}

Vertices { float[3] Coordinates } = @1
Vertices { float Data } = @2
Lines { int LineIdx } = @3

@1 # 12 xyz coordinates
0.9 0 0
```

```
1 0 0.1
1 0 1.9
0.9 0 2
0 0.9 0
0 1 0.1
0 1 1.9
0 0.9 2
-0.9 0 0
-1 0 0.1
-1 0 1.9
-0.9 0 2

@2 # 12 data values
1 1 1 1 2 2 2 2 1 1 1 1

@3 # 15 indices, defining 3 line segments
0 1 2 3 -1
4 5 6 7 -1
8 9 10 11 -1
```

Lines are defined using vertex indices as shown above. The index of the first vertex is 0. An index value of -1 indicates that a line segment should be terminated. An arbitary number of additional vertex data values can be defined. Multiple values should be distinguished by denoting them `Data2`, `Data3`, and so on.

9.6.8 Colormaps

An Amira colormap consists of a one-dimensional array of RGBA components accompanied by two numbers *min max* specifying which data window should be linearly mapped to the RGBA values. The RGBA array should have 256 elements in order to be able to edit the colormap using the colormap editor.

Colormaps are encoded in an AmiraMesh file as follows:

```
# AmiraMesh ASCII 1.0

define Lattice 256

Parameters {
    ContentType "Colormap",
    MinMax 10 180
}
```

```
Lattice { float[4] Data } = @1

@1
1 0 0 0
1 0.00392157 0 0
1 0.00392157 0 0.00392157
1 0.0117647 0 0.00392157
1 0.0196078 0 0.0196078
1 0.027451 0.00392157 0.0196078
...
```

The RGBA values are stored in floating point format. A component value of 0 means no intensity (black), while a component value of 1 means maximum intensity (white). The fourth component denotes opacity (*alpha*). Here a value of 0 indicates that the color is completely transparent while a value of 1 indicates that the color is completely opaque.

9.7 AmiraMesh as LargeDiskData

Image data stored in an uncompressed AmiraMesh file can be loaded as LargeDiskData. Use the File Dialog's Popup Menu to force the file format to "AmiraMesh as LargeDiskData".

The file is opened readonly. You cannot change the imagedata nor add or modify parameters.

9.8 Analyze 7.5

This format was used by older versions of the Analyze medical imaging software system. Today it has been widely replaced by the AnalyzeAVW format. The format is used to store 3D medical images. The actual image data and the header information are stored in two different files. In order to import the data in amira, you have to select just the header file in the Amira file browser. The header file is recognized by the extension `.hdr`. If it has some other extension you have to manually select the file format via the file browser's popup menu.

The header file contains the dimensions of the 3D image, the voxel size, and the primitive data type (8/16/32 bit grayscale, 24 bit color). In addition some other information such as a short data description or a patient id are contained in the header. This information will be stored in the parameter section of the generated amira data object.

9.9 AnalyzeAVW

This format is used by newer versions of the Analyze medical imaging software (version 2.5 or higher). Previous versions used the Analyze 7.5 format, where image data and header information were stored in different files. The *Analyze AVW* format can be used to store 2D, 3D, and 4D medical images, but the 4D time series option is currently not supported in amira. Additional attributes stored in the files (like patient name or examination time) are stored in the parameter section of the generated amira data objects. *Analyze AVW* files are safely recognized by inspecting the file header. The common file name extension is `.avw`.

Besides *Analyze AVW* image files sometimes also so-called *Analyze AVW* volume files are used, which contain a list of 2D file names forming a 3D image stack. Such volume files currently cannot be interpreted by amira. However, note that amira provides a very similar concept with the Stacked-Slices format.

Amira exports scalar and color fields with uniform coordinates to the *Analyze AVW* format. Other coordinate types can be written as well, but in these cases the coordinate information will be lost and no accurate recovery will be possible.

9.10 BMP Image Format

BMP is a standard image format mainly used on the Microsoft Windows platform. Image data is stored with 8 or 24 bits per pixel without applying any compression. When writing RGBA color fields the alpha channel will be discarded. When reading BMP images an alpha value of 255 (full opacity) will be assumed. BMP files are automatically identified by the file name extensions `.bmp`.

Regarding the import and export of multiple slices the same remarks apply as for the TIFF image format. When reading BMP images the channel conversion dialog is popped up. This dialog is also is also described the TIFF section.

9.11 Bio-Rad Confocal Format

The Bio-Rad confocal file format is used to store 3D image data from confocal microscopy. It essentially consists of a 76 byte header section followed by the image data in big endian raw format. amira recognizes Bio-Rad files automatically by the suffix `.pic`. In order to load Bio-Rad files from the command line use `load -biorad <filename>`.

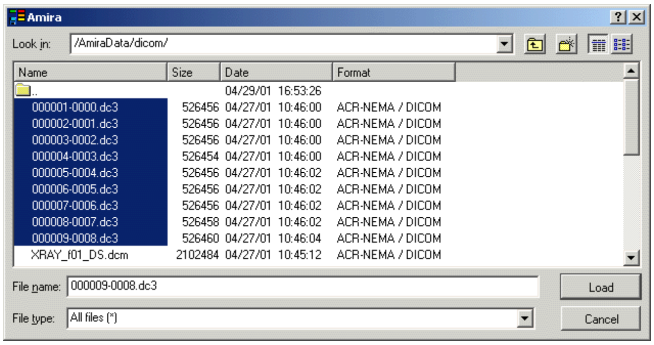
Since the header section of the format doesn't contain full information about the voxel size, the bounding box of the 3D image has to be adjusted manually for the resulting uniform scalar field using amira's crop editor. Note, that Bio-Rad confocal files can only be read but not be written by amira.

9.12 DICOM

The DICOM data format and its predecessor the ACR-NEMA format are widely used to exchange medical image data, provided by various modalities. DICOM stands for **D**igital **I**maging in **Co**mmunications and **M**edicine, and it was originally designed as pure transfer format between imaging modalities and image retrieval systems (client/server). The data stream has a so called *tagged* format, with a variable amount of tags (DICOM data elements). Each element is defined by a unique group-element identifier. These group-element pairs are always sorted in ascending order within the DICOM data stream.

In amira the import of sequences of axis-aligned CT or MRI images *stored* in DICOM or ACR-NEMA format is supported. It will be checked whether the spacing between subsequent slices is constant or not. In the first case an image stack with uniform z-coordinates is created, in the latter one so called stacked coordinates are used. Nonuniform stacks can be easily converted into uniform stacks using the arithmetic module. Both, image stacks, either uniform or stacked, can be segmented using the image segmentation editor. Such labelled data can be converted into polygonal models using the SurfaceGen module, and furthermore into tetrahedral models using the TetraGen module.

When reading DICOM or ACR-NEMA image stacks, all files of the data volume have to be selected simultaneously within the file browser. This is done by selecting each file with a mouse click holding the control button down or by clicking the first file and then shift-clicking the last one. amira automatically identifies files in DICOM or ACR-NEMA format if the file name suffix is `.dcm`, `dc3`, `.ima` or `.ani`, if the file name matches a DICOM unique instance ID, e.g., `1.3.12.2.1107.5.1.2.20395.19980429...` or if the `DICM` sequence can be found at byte position 128 within the data stream. Individual files are automatically uncompressed if they were compressed using `gzip` or `compress`, and if the file name ends with the suffix `.gz` or `.Z`.



Look in: /AmiraData/dicom/

Name	Size	Date	Format
..		04/29/01 16:53:26	
000001-0000.dc3	526456	04/27/01 10:46:00	ACR-NEMA / DICOM
000002-0001.dc3	526456	04/27/01 10:46:00	ACR-NEMA / DICOM
000003-0002.dc3	526456	04/27/01 10:46:00	ACR-NEMA / DICOM
000004-0003.dc3	526454	04/27/01 10:46:00	ACR-NEMA / DICOM
000005-0004.dc3	526456	04/27/01 10:46:02	ACR-NEMA / DICOM
000006-0005.dc3	526456	04/27/01 10:46:02	ACR-NEMA / DICOM
000007-0006.dc3	526456	04/27/01 10:46:02	ACR-NEMA / DICOM
000008-0007.dc3	526458	04/27/01 10:46:02	ACR-NEMA / DICOM
000009-0008.dc3	526460	04/27/01 10:46:04	ACR-NEMA / DICOM
XRAY_f01_DS.dcm	2102484	04/27/01 10:45:12	ACR-NEMA / DICOM

File name: 000009-0008.dc3

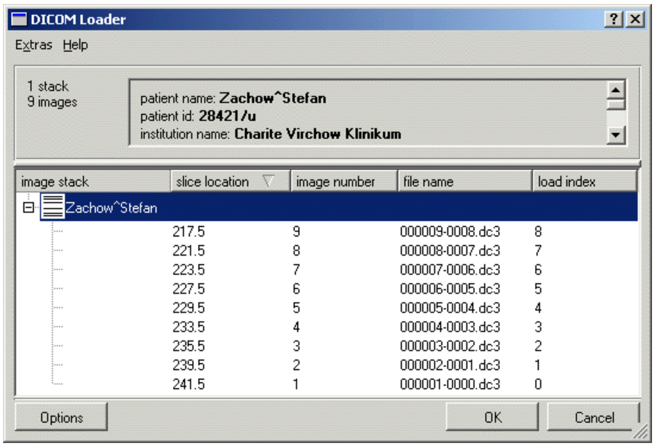
File type: All files (*)

Figure 9.1: Selecting multiple DICOM files in the file dialog.

After chosing files from the file selection dialog, a list of images is displayed within the DICOM loader dialog view (see below), which allows you to adjust several parameters for image stack generation. This intermediate step is necessary because the conversion of sequences of DICOM images into image stacks can be ambiguous, although in many cases the standard settings will produce the desired results. The evolution of the ACR-NEMA/DICOM file format is fast and striking. However, the primary goal of reading image stacks into amira is the consistency of the entire data volume. The Loader has to handle *retired* data elements, different *transfer syntaxes*, explicit or implicit *value representation*, *image compression* and *multi-frame data*, to name a few specialities. Furthermore, the availability of certain tags is not always guaranteed, and so called *private* data elements can be added at will by any creating instance. Further information on the DICOM 3.0 data format can be found in NEMA: Standards Publications PS3.x

The Dicom Load Dialog

After selection of DICOM/ACR-NEMA files from the file selection dialog, all images stored within these files are listed within the DICOM Loader dialog. The images are initially sorted by location in ascending order. Images of different studies are grouped into separate image stacks. Each stack is represented by a stack symbol and the patient's name if available. Stacks with equally distributed images are depicted with a uniform stack symbol, and stacks with variable spacing have a non-uniform stack symbol. Single slices are represented by image icons within the list view. Clicking on a stack or on one of it's images will display additional information in the top area of the dialog's view.



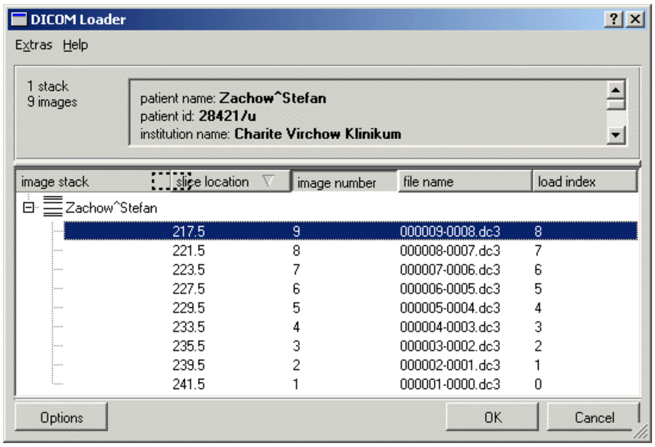
1 stack
9 images

patient name: **Zachow^Stefan**
patient id: **28421/u**
institution name: **Charite Virchow Klinikum**

image stack	slice location	image number	file name	load index
Zachow^Stefan				
	217.5	9	000009-0008.dc3	8
	221.5	8	000008-0007.dc3	7
	223.5	7	000007-0006.dc3	6
	227.5	6	000006-0005.dc3	5
	229.5	5	000005-0004.dc3	4
	233.5	4	000004-0003.dc3	3
	235.5	3	000003-0002.dc3	2
	239.5	2	000002-0001.dc3	1
	241.5	1	000001-0000.dc3	0

Figure 9.2: The DICOM load dialog.

Sorting order and policy can be modified by clicking on the column headers or by moving columns from one position to another. The order of the columns' precedence defines the siginificance of the sort keys. Clicking on a column's header toggles the sorting order between ascending and descending, depicted by an arrow within the column's header. The leftmost column has the highest priority. Rows with equal contents, e.g., equal slice locations, can be subsorted depending on the order and contents of all remaining columns.


DICOM Loader

Extras Help

1 stack
9 images

patient name: **Zachow^Stefan**
patient id: **28421/u**
institution name: **Charite Virchow Klinikum**

image stack	slice location	image number	file name	load index
Zachow^Stefan				
	217.5	9	000009-0008.dc3	8
	221.5	8	000008-0007.dc3	7
	223.5	7	000007-0006.dc3	6
	227.5	6	000006-0005.dc3	5
	229.5	5	000005-0004.dc3	4
	233.5	4	000004-0003.dc3	3
	235.5	3	000003-0002.dc3	2
	239.5	2	000002-0001.dc3	1
	241.5	1	000001-0000.dc3	0

Options OK Cancel

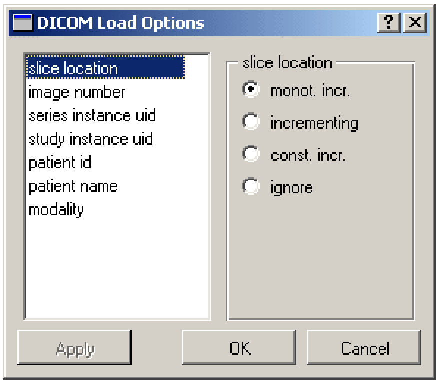
Figure 9.3: Reordering columns in the load dialog.

Sorted sequences of images are automatically broken into substacks when certain image parameters do not coincide, i.e. the image/pixel size or the bits/samples per pixel vary. Furthermore there are stack criteria like patient name, patient id, series instance uid etc. that are supposed to enforce stack consistency but can be overridden or additionally be set by the user.

Building image stacks for 3D reconstruction requires unique per-slice locations, thus image stacks of a single study are automatically broken into substacks if duplicate slice locations occur. This is especially the case, when large areas were scanned in several steps. Such duplicate slices can be removed by right clicking on the image row, choosing `remove image` from the popup menu. Remaining images are automatically combined to one stack if they are not affected by any other stack break criterion. Stack break criteria can be modified or disabled within the DICOM Load Options dialog, that shows up after pressing the `Options` button from the DICOM Loader.

Load options can also be removed completely by choosing the appropriate column, right clicking on the column's header and selecting `remove column` from the popup menu showing up at this point. Options are disabled when `ignore` is selected from the Load Options dialog. After any modification of load options the new settings can be applied to the list of images by either pressing the `Apply` button or accepting all options by leaving the dialog with `Ok`. Any changes are immediately visible through rearrengement of images and stacks within the list view.

Initially the major stack criteria, like slice location, image number, patient name, patient id, series id and date as well as the file name and the load index are shown in different columns. If any of the parameters remain constant for all images (except the slice location), the respective column will not show up. Columns can be manually removed by clicking on the column's header with the right mouse button and chosing `remove column` from the popup menu. New columns can also be added to the list view.

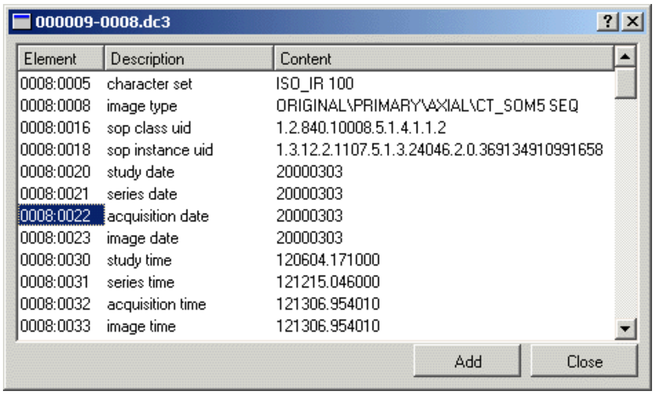


DICOM Load Options

slice location	slice location
image number	[✓ monot. incr.]
series instance uid	incrementing
study instance uid	const. incr.
patient id	ignore
patient name	
modality	

Figure 9.4: Dialog for specifying stack break criteria.

Imagine that you were loading a time series of images showing cardiac motion or the distribution of nuclear tracers. The location could be the same for all images but there might be any other parameter of interest describing the order of acquisition. Usually the image number or even the file name will suffice to represent such an order, but there are other parameters that might be of interest, too. Clicking on any image row with the right mouse button shows a popup menu where the menu option `DICOM parameters` will open up a list of all DICOM data elements for the belonging image series.



000009-0008.dc3

Element	Description	Content
0008:0005	character set	ISO_IR 100
0008:0008	image type	ORIGINAL\PRIMARY\AXIAL\CT_SOM5 SEQ
0008:0016	sop class uid	1.2.840.10008.5.1.4.1.1.2
0008:0018	sop instance uid	1.3.12.2.1107.5.1.3.24046.2.0.369134910991658
0008:0020	study date	20000303
0008:0021	series date	20000303
0008:0022	acquisition date	20000303
0008:0023	image date	20000303
0008:0030	study time	120604.171000
0008:0031	series time	121215.046000
0008:0032	acquisition time	121306.954010
0008:0033	image time	121306.954010

Figure 9.5: Dialog listing all DICOM parameters.

The parameter list is sorted by the group-element pairs as described in the DICOM format. It can be sorted by element description or parameter values as well, for easily finding a certain data element. Any parameter can be appended as stack criterion by pressing the `Add` button. Moving this parameter column to the first position will break sequences of images according to the effective load option, that can be either `constant` for alphanumeric values or incrementing for numerical values.

If you would like to force all images into one stack of a given sorting order, remove all insignificant columns, respectively set their load options to `ignore`. This will only fail if any of the stringent stack criteria (image size etc.) varies. If the sorting order counteracts to the order of slice locations, or no slice locations are given at all, you will be prompted for the position of the first image within the list view and the spacing between two adjacent images. Note that this will always end up in stacks with uniform image distribution.

The Dicom Save Dialog

amira is able to export 3D images with uniform or stacked coordinates consisting of either 8-bit or 16-bit values in the DICOM 3.0 format. If the data set to be exported originally has been read from DICOM files, the DICOM attributes (which are stored in the parameter section of a data object) will be exported too. Othwerwise, default values for the required attributes will be used. The image dimensions, the voxel size, and the slice location are written correctly in any case.

When writing a 3D image in the DICOM 3.0 format, the Dicom save dialog pops up, allowing you to define certain attributes of the exported files, such as the image modality.

9.13 DXF

The *DXF* file format is a general-purpose format used by the AutoCAD (TM) software, The format is able to store a large variety of 2D and 3D geometries. Currently, amira only exports files in plain ASCII format containing 3D triangular surfaces and 3D line sets. The information defined in the parameter section of an amira data object will only be saved in case of 3D trianglar surfaces.

3D line sets can be exported in two different ways: the whole line set in a single file or on a per-slice basis (*DXF slice-by-slice*). The latter format can only be selected if the line set contains a special parameter describing the structure of the individual slices. Groups of lines belonging to the same slice, i.e., having the same x, y, or z coordinate, are saved in the same *DXF* file. Line sets of this special type are produced by the module ComputeContours.

When importing *DXF* files amira will create an Open Inventor scene graph object. You may use the module IvToSurface in order to convert the scene graph into an amira surface. Currently, only *DXF* files in ASCII format can be read.

DXF files are identified by the file name extension `.dxf`.

9.14 Encapsulated Postscript

Amira is able to save snapshots as well as individual slices of a 3D image data set in *Encapsulated Postscript* format. You may directly send EPS files to a Postscript printer, or you may include these files in many standard desktop publishing programs. The EPS files produced by amira contain bitmaps rather than vector information.

The import of EPS files is not supported.

9.15 FIDAP NEUTRAL

The *FIDAP NEUTRAL* describes 3D simulations on geometries consisting of 3D vertices and a number of geometry elements based on them. amira can only read *FIDAP NEUTRAL* files in plain ASCII format containing 3D triangular/quad surfaces, and tetrahedral grids.

After describing the geometry, the *FIDAP NEUTRAL* file contains a number of time steps, each time step specifying the same subset set of data sets defined on the 3D nodes (velocity, pressure, temperature etc.). amira loads the geometry and then displays a dedicated module called *FIDAPControl* which allows the user to select the desired timestep. In this way, the evolution of the data in time can be followed.

9.16 Fluent / UNS

The *Fluent* file format is used for storing 2D and 3D geometries, such as unstructured finite-element grids. The format is quite powerful. Currently, amira only supports *Fluent* files in plain ASCII format containing 3D triangular surfaces, tetrahedral or hexahedral grids.

When exporting *Fluent* files, additional information defined in Amira's surface or grid structures such as the material section is saved as well. However, other applications won't be able to read or interprete these additional data.

Fluent files are identified automatically by analyzing the file header.

9.17 HTML

HTML files loaded via the amira file dialog are displayed in the online help viewer. The help viewer supports certain but not all HTML formatting tags. It also is not able to resolve web links. Only links to local files are resolved. Nevertheless, it is very convenient to describe projects and to invoke demo scripts from an HTML page displayed in the amira help viewer. Whenever a file with the extension `.hx` is linked, this file is interpreted as an amira script. The script is executed when the link is clicked. As an example you may look at the demo pages provided with the online user's guide, for example `share/usersguide/recon.html` or other files listed in `share/usersguide/AmiraIndexDemo.html`.

9.18 HxSurface

Simplified version. The surface format has been designed to represent triangular non-manifold surfaces. The triangles of such a surface are grouped in patches. In addition, information about so-called boundary contours and branching points can be stored is a surface file. However, since this information can be recomputed automatically if required, we discuss a simplified version of the format first. Here is an example:

```
# HyperSurface ASCII

Parameters {
    Info "GMC: 3 colors, case 13"
}

Materials { {
    color 0.83562 0.78 0.06,
    Name "Yellow" }
{
    color 0.21622 0.8 0.16,
    name "Green" }
{
    color 0.8 0.16 0.596115,
    name "Magenta" }
}

Vertices 11
    1.000000 0.666667 0.500000
    0.666667 0.500000 1.000000
    1.000000 0.500000 0.000000
    0.500000 1.000000 0.000000
    0.000000 0.000000 0.500000
    0.000000 1.000000 0.500000
    1.000000 1.000000 0.500000
    0.500000 0.000000 1.000000
    1.000000 0.500000 1.000000
    0.500000 1.000000 1.000000
    0.523810 0.523809 0.500000

Patches 3
{   InnerRegion Green
    OuterRegion Yellow
    Triangles 7
      3 1 11
      4 3 11
      6 4 11
      5 6 11
      8 5 11
```

```
        2 8 11
        1 2 11
  }
  {   InnerRegion Magenta
      OuterRegion Yellow
      Triangles 1
        9 2 1
  }
  {   InnerRegion Magenta
      OuterRegion Green
      Triangles 2
        2 10 7
        7 1 2
  }
```

The first line is required and identifes the surface format. Additional comments starting with a hashmark may appear at any point in the file. Next, an optional parameter section and a material section follow. These sections have the same format as in an AmiraMesh file. The parameter section may contain an arbitrary number of name-value items. The material section contains additional information about imaginary regions the surface patches are supposed to separate. In contrast to an AmiraMesh file individual materials need not to have an *Id* since they are referenced via their names.

The statement `Vertices 11` indicates that the x-, y-, and z-coordinates of 11 vertices follow. Likewise, the statement `Patches 3` indicates that 3 patches follow. The definition of each patch is enclosed by a pair of brackets { and }. Inside these brackets `InnerRegion` and `OuterRegion` indicate the two regions the patch is supposed to separate. If you don't want to generate a tetrahedral grid from your surface you may omit these statements or you may choose both regions to be the same. Finally, the triangles of a patch are specified by indexing vertices defined in the vertex section. Like in an AmiraMesh file indices start at 1, not at 0.

Extended version. In its extended version the surface format is able to store additional topological information of a surface. Before we discuss this in detail let us first make some definitions and introduce the underlying concepts.

- **Region:** In finite element applications regions are usually called materials. A region is defined by its surrounding surface, which may consist of multiple patches. Each region must have a unique name.
- **Surface:** A surface is defined by the boundary of a 3D region and therefore must be closed. This means that for example each edge must be connected to an even number of triangles. In one half of the triangles the edge is referenced in forward orientation, in the other half in backward orientation. A surface may consist of multiple pieces, so-called patches. In the file format the patches of a surface are given by a list of signed indices. A negative index means that the patch

has negative orientation in the current surface.

- **Patch:** A part of a surface which separates exactly two differnt regions. Patches are built from triangles. The triangles are all oriented in a unique way so that we can define an inner region and an outer region. The triangle normals point into the outer region. For each patch an `InnerRegion` and an `OuterRegion` may be specified in the file format. If one of these specifiers is missing it is assumed that the patch delimits the exterior space. Otherwise the exterior region should be called `OUTSIDE`. Patches may be closed or may be delimited by so-called boundary curves. Boundary curves are specified by a list of signed integers. The sign denotes the orientation of the curve segment for a particular patch. In addition for each patch inner branching points may be specified, if necessary.
- **BoundaryCurve:** At a boundary curve multiple patches join. The curves are defined by a set of vertex indices. For closed curves the first and the last index must be equal.
- **BranchingPoint:** A point where multiple regions join. The start and end points of a boundary curve are branching points. In addition there may be also branching points inside a patch which are not part of a boundary curve.
- **Vertices:** These are the points from which the surface triangles are built. In the file format the vertex coordinates are specified after the indentifier `Vertices` followed by the number of vertices. First branching points should be specified, then points on boundary curves, and then inner points of the patches. In the definition of boundary curves and patches the vertices are referenced by indices starting from 1, not from 0.

The following example describes the surfaces of three connected tetrahedra which are assigned to three different regions. Some of the definitions are optional. Really necessary are only the list of vertex coordinates as well as the definition of the patches. Boundary curves and surfaces may be reconstructed from this.

```
# HyperSurface ASCII

Vertices  6
     0.0  0.0  0.0
     0.0  0.0  1.0
     0.0  1.0  0.0
    -1.0  0.0  0.0
     1.0  0.0  0.0
     0.0 -1.0  0.0

NBranchingPoints 2
NVerticesOnCurves 2

BoundaryCurves 3
{
    Vertices 3
```

```
        1  3  2
} {
    Vertices 2
        1  2
} {
    Vertices 3
        1  4  2
}

Patches 5
{
    InnerRegion Material1
    OuterRegion OUTSIDE
    BranchingPoints 0
    BoundaryCurves 2
        1  -2
    Triangles 3
        5  1  3
        1  5  2
        5  3  2
} {
    InnerRegion Material2
    OuterRegion OUTSIDE
    BranchingPoints 0
    BoundaryCurves 2
        3  -1
    Triangles 2
        3  1  4
        2  3  4
} {
    InnerRegion Material3
    OuterRegion OUTSIDE
    BranchingPoints 0
    BoundaryCurves 2
        -3  2
    Triangles 3
        4  1  6
        2  4  6
        1  2  6
} {
    InnerRegion Material1
    OuterRegion Material2
```

```
        BranchingPoints   0
        BoundaryCurves 2
            2  -1
        Triangles 1
            3  1  2
    } {
        InnerRegion Material2
        OuterRegion Material3
        BranchingPoints 0
        BoundaryCurves 2
            -3  2
        Triangles 1
            1  2  4
    }

    Surfaces 3
    {
      Region Material1
      Patches 2
         1  4
    } {
      Region Material2
      Patches 3
         2  -4  5
    } {
      Region Material3
      Patches 2
         3  -5
    }
```

9.19 Hypermesh

The *Hypermesh (TM)* file format is used by the *Altair HyperWorks* product family. The format describes 3D geometries consisting of 3D vertices and a number of cells based on them. amira can only read *Hypermesh* files in plain ASCII format containing 3D triangular surfaces, tetrahedral grids or so-called *tria6* components (a special kind of traingles). If tetrahedral components are present the result of the import is a TetraGrid, else it is a 3D surface.

Tetrahedral grids and 3D surfaces can also be exported in *Hypermesh* format. Every 3D surface component (patch) is saved with its interior-exterior material names, separated by a ”-”. However, it is not guaranteed that other applications understand this coding. A tetrahedral grid is saved together with its boundary surface. The patch structure of the boundary surface is retained. In order to separate the boundary into different patches the boundary condition ids are interpreted. The material ids of every tetrahedron are also saved.

Amira identifies *Hypermesh* files by the extensions `.hm` or `.hmascii`. The decision whether the file contains a 3D surface or a tetragrid is made after reading and analysing the content.

9.20 IDEAS universal format

The *I-DEAS universal* file format is a general format originally used by SRDC's I-DEAS Master Series software to encode CAD/CAM models and FEM simulation results. The format is able to store lots of different data entities and FEM cell types. In amira the following subset of so-called *universal dataset numbers* is supported when importing I-DEAS files:

- **15:** Nodes with single precision coordinates.
- **781, 2411:** Nodes with double precision coordinates.
- **71, 780, 2412:** Cell definition. Lines, triangles, quads, tetrahedra, hexahedra, and prisms with linear shape functions are supported. Quads are decomposed into triangles. Prisms are decomposed into triangles as well. Hexahedra are decomposed into tetrahedra if tetrahedra or prisms appear too (Amira currently is not able to handle meshes with mixed element types or elements with higher-order shape functions).
- **55, 2414:** Data at nodes.

The common file name extension of this format is `.unv`. For *I-DEAS* files with some other extension the file format has to be specified manually via the popup menu of the file dialog. Surfaces, tetrahedral grids, and hexahedral grids can also be exported into an *I-DEAS universal* file.

9.21 Icol

This is a simple ASCII file format coming in two variants, indexed and non-indexed, to store colormaps. The *Icol* file format and the *Icol Colormap Editor* originate from the Graphics and Visualization Lab (GVL) of the Army High Performance Computing Research Center (AHPCRC), Minnesota. The Colormap Editor in amira shares many ideas with the original AHPCRC tool. The structure of an *Icol* format can be immediatly understood by looking at the examples in amira's demo data directory.

9.22 JPEG Image Format

JPEG is a standard format which can be used to store RGB and greyscale images in a highly compressed form. However, note that the compression algorithm is lossy. The quality of the compression for file export can be specified by typing the command

```
set AmiraJPEGQuality <quality>
```

into the Amira console prior to the export, where `<quality>` ranges from 0 to 100. The default is 90. When writing RGBA color fields the alpha channel will be discarded. When reading RGB images an alpha value of 255 (full opacity) will be assumed. JPEG files are automatically identified by the file name extensions `.jpg` or `.jpeg`.

Regarding the import and export of multiple slices the same remarks apply as for the TIFF image format. When reading JPEG images the channel conversion dialog is popped up. This dialog is also is also described the TIFF section.

9.23 LargeDiskData

This is the native amira fileformat for blockwise access of imagedata stored on disk as described in LargeDiskData. It supports read/write operations, multiresolution access and saving of changed parameters. The data are stored in a couple of files, e.g.:

```
visMaleCT
visMaleCT-SUPR.dat
visMaleCT-0001.dat
visMaleCT-0000.dat
visMaleCT-0002.dat
```

The first one is recognized by amira as an AmiraMesh file. It contains the paramters and a link to the other files. It can be loaded into amira with the File Dialog. The other files contain the actual image data.

You can create such a fileset with the ConvertToDiskData module.

9.24 Leica 3D TIFF

This reader is able to read 3D TIFF files containing a whole stack of 2D images. In particular, the 3D TIFF format is used by newer Leica laser scanning microscopes.

In addition to the image data itself special parameters like pixel size or slice distances may be stored in a 3D TIFF file. If such information is found it will be interpreted in order to create a uniform scalar field of the proper type. However, if no bounding box information is encountered, the channel conversion dialog described in the 2D TIFF section will be popped up.

9.25 Leica Binary Format (.lei)

This is the Leica binary file format used by Leica laser scanning microscopes. It consists of an *lei* file as well as several TIFF slices. In order to read these files, select only the *lei* file. Parameters like pixelsize or slice distance are read from the *lei* file.

9.26 Leica Slice Series (.info)

This is the file format used by the older Leica laser scanning microscopes. It consists of an *info* file as well as several raw or TIFF slices, which all must reside in the same directory. In order to read these files, select only the *info* file. Parameters like pixelsize or slice distance are read from the *info* file. If the file contains colormaps, they will be read, too.

9.27 Metamorph STK Format

The MetaMorph Stack (STK) file format is used to encode 3D image data, e.g. from confocal microscopy. It is a special version of the TIFF file format. Thus, STK files are indicated as TIFF files in the format column of the file dialog.

STK files can be read just as ordinary 3D TIFF files. The channel conversion dialog is popped up, letting the user decide how to proceed with multiple channel images and letting him define the bounding box of the 3D image. Note, that size hints stored in the STK file itself are currently not interpreted. Also note, that amira can only read but not write STK files.

9.28 Open Inventor

The *Open Inventor* file format is used for storing 3D geometries. The format is very powerful. The VRML format known from the World Wide Web is very similar to Open Inventor. Since amira is built on top of Open Inventor, it naturally supports this format. However, amira has added a lot of special purpose nodes to Open Inventor. Therefore currently some geometries which can be displayed in amira's 3D viewer cannot be saved in Open Inventor file format.

When reading an Open Inventor file a data object of type IvData will be created. This data object stores the Open Inventor scene graph and can be visualized using a IvDisplay module. IvData objects can be saved again in ASCII or binary Open Inventor format. In addition amira surfaces can be exported in Open Inventor format.

9.29 PNG Image Format

PNG stands for *Portable Network Graphics*. The format is mainly used for internet applications. Usually, image data is stored in compressed form using a lossless compression algorithm. The format is able to store an alpha channel besides the ordinary color channels. PNG files are automatically identified by the file name extensions `.png`.

Regarding the import and export of multiple slices the same remarks apply as for the TIFF image format. When reading PNG images the channel conversion dialog is popped up. This dialog is also is also described the TIFF section.

9.30 PNM Image Format

This format includes the PPM, PGM and PBM image formats. These formats are used to store RGB color images, greyscale images, as well as black and white images, respectively. For each of the three formats there is a binary and an ASCII version. amira is able to read all six of them, but will only write binary PPM and PGM files. Black and white PBM images can only be read if the image width is a multiple of eight. PNM files will be automatically identified by their file headers.

Regarding the import and export of multiple slices the same remarks apply as for the TIFF image format. When reading PNM images the channel conversion dialog is popped up. This dialog is also is also described the TIFF section.

9.31 PSI format

The PSI format stores a set of 3D points with additional data variables associated to them in a column-oriented way. In amira PSI files are represented by data objects of type HxCluster.

A PSI file starts with an (optional) header section. This section describes the contents of the file, i.e., the number and the meaning of the individual data columns. The file syntax is illustrated in the following example:

```
# PSI Format 1.0
#
# column[0] = "x"
# column[1] = "y"
# column[2] = "z"
# column[3] = "Energy"
# column[4] = "Grain"
# column[5] = "Id"
# column[6] = "Coordination Number"
# column[7] = "Crystallographic Class"
#
# symbol[3] = "E"
# symbol[4] = "g"
# symbol[6] = "n"
# symbol[7] = "c"
#
# type[3] = float
# type[4] = byte
# type[6] = byte
# type[7] = { PFCC, GFCC, PHCP, GHCP, OT12, OTHR }

5 2694 115001
1.00 0.00 0.00
0.00 1.00 0.00
0.00 0.00 1.00

-0.748  -0.748  -0.777  -4.3840  15  327909  12  PFCC
-0.735  -0.739  -0.757  -4.3840  15  327910  12  PFCC
-0.742  -0.784  -0.754  -4.3400  15  328800  12  GFCC
-0.757  -0.769  -0.766  -4.3823  15  328812  12  PFCC
-0.747  -0.762  -0.745  -4.3638  15  328813  12  PFCC
```

The very first line indicates that this is a PSI file. The statement `column[0] = "x"` indicates that the first data column represents the x-coordinates of the points. In this example in total eight data columns are defined. Note, that the names `"x"`, `"y"`, `"z"`, and `"Id"` have a special meaning. These columns are required.

Next, the statement `symbol[3] = "E"` defines a symbol for the data column labeled `"Energy"`. This symbol may be used in an arithmetic filter expression as provided for example by the modules ClusterView and ClusterDiff. Note, that symbol definitions for the four special columns labeled `"x"`, `"y"`, `"z"`, and `"Id"` have no effect.

Finally, the statement `type[3] = float` indicates, that internally energy values should be stored as 32-bit floating point numbers. Alternatively, data values may be stored as 32-bit integers (`int`) or as 8-bit characters (`byte`). As a special feature, data values may also be represented by textual tokens. In this case, the set of allowed tokens should be specified in a comma-separated list as shown in the above example.

The header section may also contain any other user-defined comment, provided the first character of a comment line is a `#`. The first non-blank line after the header section specifies the number of points, as well as two other parameters, wich are ignored by amira. Next, the bounding box of the data set is specified. However, amira will ignore this definition. Instead, the bounding box will be calculated from the point coordinates itself.

If a *PSI* file contains no header section at all, amira assumes that the file contains exactly eight data columns and that these data columns are arranged like in the example above.

9.32 Plot 3D Single Structured

Plot3D is a simple binary file format, to represent structured curvilinear grids and scalar or vector fields defined on these grids. This format originates from the Plot3D program developed by Pieter Buning at

NASA Ames.

To read data in this format, you have to select a *Grid* file, and optionally additional *scalar*, *vector*, or *solution* files. The preferred file name suffix for Plot3D files is `.p3d`. In order to load Plot3D files from the command line use `load -plot3d`.

To write a data object in Plot3D format first create the grid file by choosing *Plot3D Grid File* in amira's file dialog. Then write the data itself by choosing *Plot3d Data File*. From the command line you may use `save -plot3dgrid` and `save -plot3ddata`, respectively.

In detail Plot3D files must have the following structure (all binary 32-bit big endian format):

Grid File:

```
3 integers:

[IDIM, JDIM, KDIM]

IDIM x JDIM x KDIM (call this product NPOINTS) floating-point X co-
ordinates (brackets
indicate one record):

[x1,x2,...,xNPOINTS,

NPOINTS floating-point Y coordinates:

y1,y2,...,yNPOINTS,

NPOINTS floating-point Z coordinates:

z1, z2,...,zNPOINTS]
```

Solution File:

```
3 integers:

[IDIM, JDIM, KDIM]

4 floating-point conditions:

(free-stream mach number, angle-of-attack, Reynold's num-
ber, and integration time)

[FSMACH, ALPHA, RE, TIME]
```

```
IDIM x JDIM x KDIM (call this product NPOINTS) floating-
point Q1 values (brackets indicate
one record):

(Q1 is density (RHO))

[q11, q12,...,q1NPOINTS,

NPOINTS floating-point Q2 values:

(Q2 is x momentum (RHO*U))

q21,q22,...,q2NPOINTS,

NPOINTS floating-point Q3 values:

(Q3 is y momentum (RHO*V))

q31,q32,..., q3NPOINTS,

NPOINTS floating-point Q4 values:

(Q4 is z momentum (RHO*W))

q41,q42,...,q4NPOINTS,

NPOINTS floating-point Q5 values:

(Q5 is total energy per unit volume (E))

q51,q52,...,q5NPOINTS]
```

Scalar File:

```
4 integers:

[IDIM, JDIM, KDIM, 1]

IDIM x JDIM x KDIM (call this product NPOINTS) floating-
point F values:

[f1, f2, ..., f1NPOINTS]
```

Vector File:

```
4 integers:

[IDIM, JDIM, KDIM, 3]

IDIM x JDIM x KDIM (call this product NPOINTS) floating-
point F values:

[fx1,fx2,...,fxNPOINTS,

NPOINTS floating-point FY values:

fy1,fy2,...,fyNPOINTS,

NPOINTS floating-point FZ values:

fz1,fz2,...,fzNPOINTS]
```

9.33 Ply Format

The *Ply* format was developed at Stanford University Computer Graphics Lab. It is used for storing points and geometries. In amira it is possible to read and save surfaces in this format. When writing amira surfaces in this format additional information such as the material section of the surface will be written into *Ply* files as well. However, other application will not be able to interprete this additional data.

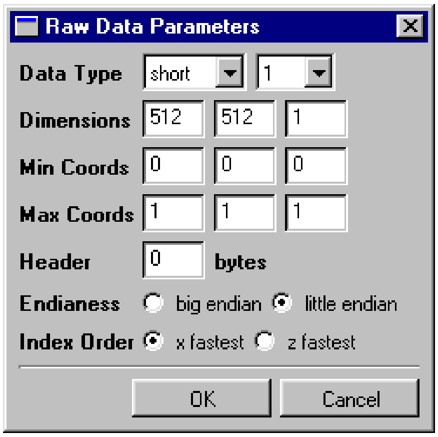
amira identifies a *Ply* file by its header.

9.34 Raw Data

Sometimes you may want to read data defined on uniform lattices in raw format, i.e. plain three dimensional arrays of data. Tomographic images might be given in this way, and raw data is often the easiest format to produce with e.g., custom simulation programs.

To read in raw data, use the *Load/Load Data* menu, select the file in the file browser and click *OK*. Since Amiracan not recognize the file format automatically, a dialog will popup. Within this dialog choose *Raw Data* as file format and click *OK*. Since the raw data file does not contain any information on how to format the data, some user specifications are required. amira will bring up a dialog:

Now adjust the parameters:


Raw Data Parameters

Data Type: short, 1

Dimensions: 512, 512, 1

Min Coords: 0, 0, 0

Max Coords: 1, 1, 1

Header: 0 bytes

Endianess: big endian [✓ little endian]

Index Order: [✓ x fastest] z fastest

Figure 9.6: Amira’s raw data read dialog.

- **Data Types.** Primitive type of data: byte for 8 bit data, short for 16 bit data, int32 for 32 bit integer data, float for 32 bit float data, double for 64 bit floating point data.

  In addition the number of data values per data point has to specified: 1 for scalar data, 3 for vector data, 6 for complex vector fields.
- **Dimensions.** Size of the three dimensional array. If wrong dimensions are specified, the data will be scrambled.
- **Min/Max Coords.** The bounding box of the data. These parameters are not as critical as the other ones. In particular the bounding box can be adjusted afterwards using the crop editor.
- **Header.** Many file formats consist of a raw data block with a prepended header. If such files are read with this method, the size of the header can be specified here. The header will then simply be skipped, when reading the file
- **Endianess.** The byte order of data types other than byte is system dependend. If you read your files on the same type of processor as on which they have been produced, the default setting will be ok. But if you read data produced e.g., under Linux (little Endian), on an SGI (big Endian), you have to specify the correct byte order of the data to be read (little Endian in this case).
- **Index Order.** Order in which the data points are read.

The raw data format is a very powerful tool, especially for quick-access/prototyping use. However it may sometimes be tricky to figure out the parameters. Some tools which may help are `vi` or `od -c` (to examine the header of files).

After loading click on the icon. If the data range is obviously completely wrong, then you have either specified a wrong data type, or a wrong endianess, or a to short header. If most of the volume looks ok, but is shifted in x-direction, then you probably have specified a wrong header. If the data range looks ok, but the data seems scrambled, then you have specified wrong dimensions, or a wrong index order.

9.35 Raw Data as LargeDiskData

This file format allows to load subvolumes out of one large raw data block on disk. Use the File Dialog's Popup Menu to force the file format to ”Raw Data as LargeDiskData“. The format of the file and the paramters you have to provide are described in Raw Data. The file will be loaded as a LargeDiskData objekt.

The file is opened readonly. You cannot change the imagedata nor add or modify parameters.

9.36 SGI-RGB Image Format

This is the SGI file format for RGB and greyscale images. When writing an RGBA color field the alpha value will be discarded. When reading RGB images full opacity is assumed. 16 bit data values, even though allowed by the SGI file format specification, are not supported. SGI-RGB files are automatically identified by the filename extensions `.sgi`, `.rgb`, or `.bw`.

Regarding the import and export of multiple slices the same remarks apply as for the TIFF image format. When reading SGI-RGB images the channel conversion dialog is popped up. This dialog is also is also described the TIFF section.

9.37 STL

STL is a CAD format for Rapid Prototyping. It is a faceted surface representation, i.e. a list of the triangular surfaces with no adjacency information. Currently the ASCII version of the format is supported for writing Amira objects of type Surface.

9.38 Stacked-Slices

This file format allows to read a stack of individual image files with optional z-values for each slice. The slice distance need not to be constant. The images must be one-channel images in an image format supported by Amira (e.g., TIFF). The reader operates on an ASCII description file, which can be written with any editor. Here is an example of a description file:

```
    # Amira Stacked Slices

    # Directory where image files reside
    pathname C:/data/pictures

    # Pixel size in x- and y-direction
    pixelsize 0.1 0.1
```

```
# Image list with z-positions
picture1.tif 10.0
picture7.tif 30.0
picture13.tif 60.0
colstars.jpg 330.0
end
```

Some remarks about the syntax:

- `# Amira Stacked Slices` is an optional header, which allows Amira to automatically determine the file format.
- `pathname` is optional and can be included in case the pictures are not in the same directory as the description file. A space separates the tag ”pathname” from the actual pathname.
- `pixelsize` is optional too. The statement specifies the pixel size in x- and y-direction. The bounding box of the resulting 3D image is set from 0 to pixelsize*(number_of_pixels-1).
- `picture1.tif 10.0` is the name of the slice and its z-value, separated by a blank.
- `end` indicates the end of the description file.
- Comments are indicated by an hashmark character (#).

9.39 Stacked-Slices as LargeDiskData

Stack of image files can be loaded as LargeDiskData. This file format allows to read a stack of individual image files with optional z-values for each slice. The slice distance need not to be constant. The images must be one-channel images in an image format supported by Amira (e.g., TIFF). The reader operates on an ASCII description file, which can be written with any editor.

Here is an example of a description file:

```
# Amira Stacked Slices as ExternalData

# Directory where image files reside
pathname C:/data/pictures

# Pixel size in x- and y-direction
pixelsize 0.1 0.1

# Image list with z-positions
picture1.tif 10.0
picture7.tif 30.0
picture13.tif 60.0
colstars.jpg 330.0
```

```
    end
```

The format of the file and the paramters you have to provide are described in StackedSlices.

The image files are opened readonly. You cannot change the imagedata but you can add or modify parameters in the description file. The file will be loaded as a LargeDiskData object.

9.40 TIFF Image Format

This is the 2D TIFF file format. It can be used to read and write one or more 2D images. If multiple images of equal size have been selected in the file dialog they will be combined into a single 3D image volume, i.e., a uniform scalar field of bytes or an RGBA color field. TIFF files will be automatically identified by looking at the file header, irrespectively by the actual file name extension.

Likewise, if a 3D image data set is to be saved in TIFF format, in fact for each slice a separate file will be created. If you choose the 2D TIFF format in the file dialog's format menu a sequence of hashmark characters `[#]` will be automatically inserted into the filename. When saving the images the hashmark sequence will be replaced by the current slice number (formatted with leading zeros). For example, if the base filename if `image.####.tif` the files actually being written will be named `image.0000.tif`, `image.0001.tif`, and so on.

Note, that not all variants of the TIFF format are supported. In particular, the following limitations apply:

- The number of channels must be 1, 3 or 4.
- The number of bits per pixel must be 8 or 16 (1 channel images only).
- Images defined in YbCbR colorspace can not be read.
- Tiled images can not be read.
- Only scalar fields consisting of bytes can be saved.

The Channel Conversion Dialog

When reading 2D image files a special dialog window will be popped up. This dialog asks the user to specify how the 2D images should be converted into Amira data objects. In addition, the world coordinates of the resulting 3D data object can be adjusted. The channel conversion dialog looks as depicted in Figure 9.7.

First of all, the dialog displays the number of files to be read, the number of 2D slices (in most cases equal to the number files), the size of a 2D slice in pixels, as well as the number of channels stored in the files. An option menu lets you select whether a 1-component uniform scalar field should be created or a 4-component RGBA color field. Depending on the type of input not all options may be active. The meaning of the individual items is described below.

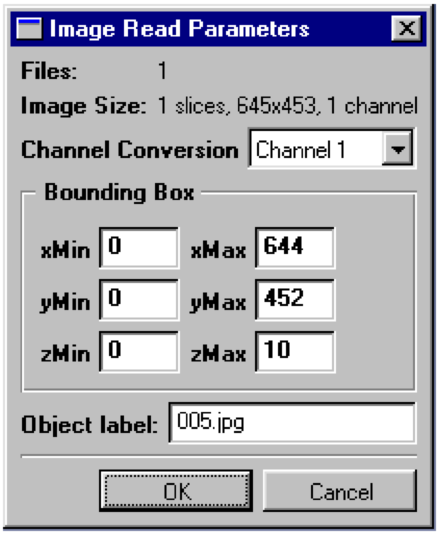


Image Read Parameters

Files: 1

Image Size: 1 slices, 645x453, 1 channel

Channel Conversion: Channel 1

Bounding Box

xMin 0 xMax 644

yMin 0 yMax 452

zMin 0 zMax 10

Object label: 005.jpg

Figure 9.7: Amira's channel conversion dialog.

- **Maximum.** The maximum value of the red, green, and blue channel is stored in a uniform scalar field (3 and 4 channel input only).
- **Weighted Average.** A grayscale uniform scalar field is created according to the NTSC formula, `I=.3*R+.59*G+.11*B` (3 and 4 channel input only).
- **Channel 1.** The first channel of the input is converted into a uniform scalar field (will always be active).
- **Channel 2.** The second channel of the input is converted into a uniform scalar field (3 and 4 channel input only).
- **Channel 3.** The third channel of the input is converted into a uniform scalar field (3 and 4 channel input only).
- **Channel 4.** The fourth channel of the input is converted into a uniform scalar field (4 channel input only).
- **Color Field.** An RGBA color field is created (4 channel input or 1 channel input with additional colormap).

If the first image file contains a colormap the map will be loaded as a separate object if *Channel 1* is selected, or it will be used to compute an RGBA color field if *Color Field* is selected.

Moreover, the dialog allows you to adjust the bounding box of the resulting image data set. The bounding box specifies the world coordinates of the center of the lower left front voxel (min values) and the center of the upper right back voxel (max values). For example, if your input is 256 pixels wide and the size of each voxel is 1mm, then you may set *xMin* to 0 and *xMax* to 255. The bounding box of a data object may also be changed later using the ImageCrop Editor.

9.41 VRML

VRML is a file format for storing 3D geometries. The format is especially popular for web or internet applications.

If you are using an amira version that is linked with the TGS Open Inventor toolkit 2.5 or higher you may import VRML files as scene graph objects. Use IvToSurface in order to convert the surface parts of the scene graph into a surface object.

On the other hand, surface objects may be exported into a VRML files using the VRML Export module. Unlike most other formats VRML Export is implemented as module rather than an export filter. The reason is that in addition to the surface itself other data objects, e.g., 3D image data, can be included into the VRML file.

Chapter 10

Alphabetic Index of Editors

10.1 CameraPath Editor

The *CameraPath Editor* allows you to edit camera paths defined by a number of keyframes. To get started choose **CameraPath** from the *Create* menu and click on the editor button of the new camera path object. The editor will appear and automatically open an additional viewer.

The original viewer (viewer 0) is the camera view, i.e., it shows the same as if you were glancing through the camera, while the additional viewer will show you all keyframe-cameras from a bird's eye view.

In order to start your camera path, choose an appropriate view in the original viewer (see Section 3.1.7 (Viewer Window) to learn how to do that) and click on the *add* button. Your first keyframe has been saved. The time slider should contain one red line and it should have advanced to 10. Now change the view in the original viewer and click on add again. A second keyframe appears and so on.

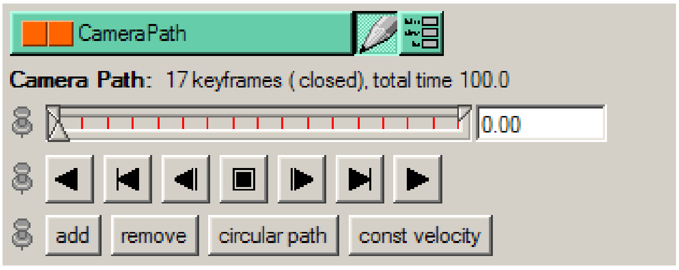

Camera Path: 17 keyframes (closed), total time 100.0

Figure 10.1: Control panel of the camera path editor. The arrow shape control buttons can be used to play forward and backward, to jump from keyframe to keyframe or to step frame by frame. When the editor is off the camera path provides a time port.

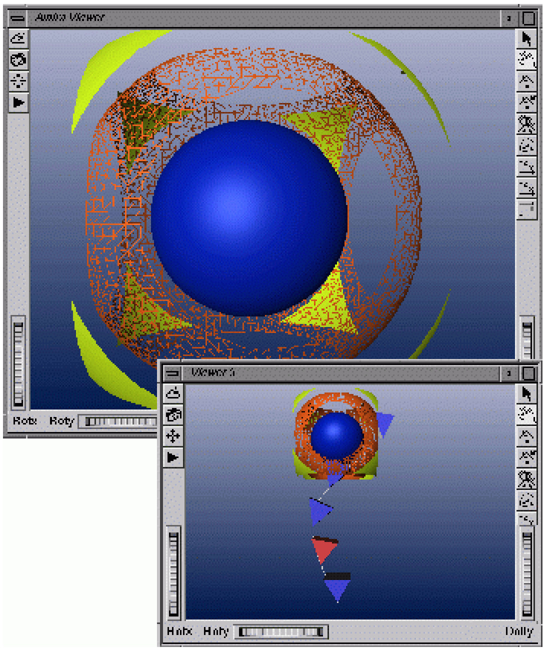


Figure 10.2: The camera path is shown in an extra viewer. You can click on a keyframe and modify the camera. The changes will be displayed in the original viewer simultaneously. The redish icon represents the current camera. In general this is not a keyframe but an interpolated camera.

Once you have played your camera path using the play button, you may continue adding keyframes, by simply typing the time for the new keyframe into the text field or by moving the slider and then clicking *add*. Note: The original view will not be changed, when the slider is moved, or the text in the text field is changed.

To **change the time of a keyframe**, jump to this frame, click on *remove*, type in the new time, and click on *add* again.

If you want to have a constant camera velocity on its path, push the *constv* button. Dependending on the distances between the camerapositions at the keyframes the keyframe times are changed in order to get equidistant inbetween camerapositions.

Caution: Pushing the *circular path* button will create a new camera path destroying the old one. The new camera path lies on a circle around one of the major axes with the center at the point the camera in the original window is looking at. The radius is determined by the distance of the cameraposition and the point the camera looks at.

10.2 Color Dialog

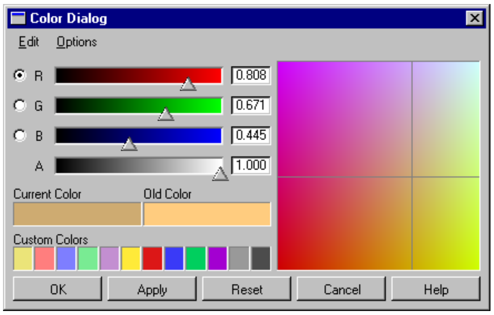
The *Color Dialog* provides an interface for selecting colors. It pops up whenever a single color is to be changed in amira. The dialog provides serveral different interface components: a menu bar, color sliders, default color cells, a color picker, and five buttons.

The menu bar allows you to set some options for working with the color dialog. The sliders (and the corresponding text fields and arrow buttons) let you choose a color by hue, saturation, and value (HSV), or by the levels of red, green, and blue (RGB). The color picker allows you to pick a color by sight. Moreover, you can choose a predefined color from a palette of custom colors cells. The buttons are used to apply or reset changes and to quit the dialog. A detailed description of the interface components is given below.

Menu Bar: The menu bar holds two pop up menus: the *Edit* and the *Options* menu. There are two items in the Edit menu allowing the user to toggle between different editing modes: the *Immediate Mode* and *WYSIWYG Mode*.

If *Immediate Mode* is selected, each modification of the color to be changed is effective immediately, e.g., if you are changing the background color of the 3D viewer and *Immediate Mode* is active, the effect of every slider or picker operation can be observed directly in the viewer.

The *WYSIWYG Mode* (meaning *what you see is what you get*) determines the background of the color sliders. If *WYSIWYG Mode* is active the background of a slider shows a color range representing how the current color would change by moving this slider.



[✓ R] 0.808
G 0.671
B 0.445
A 1.000

Figure 10.3: Amira color dialog.

The user has the option of displaying and manipulating two different combinations of sliders: RGB and HSV. The Options menu holds two items representing these two combinations. Selecting *RGB Sliders* provides controls over RGB color space. Selecting *HSV sliders* provides sliders to manipulate the HSV color space.

Color Sliders. The color sliders present the color range in hue, saturation, value (HSV) or red, green, blue (RGB) color spaces. If the color to be changed has an alpha component an additional slider is automatically displayed. Each slider controls one color component. The color sliders are visible depending on the user selection in the Option Menu. Each of the color sliders is accompanied with a text edit field to view the exact value of the current color component and to set its numerical value. The component values are always in the range from 0 to 1. Furthermore, two arrow buttons can be used to move each slider respectively.

On the left side of each slider (except the alpha slider) an additional check button is displayed allowing the user to select one color component to control the appearance and functionality of the color picker.

Color Cells. The color cells are subdivided into three different groups: One cell named *Current Color* displays the color depending on the current settings. Each color manipulation is shown by this cell immediately. The cell named *Old Color* displays the original color the dialog has been invoked with. The old color does not change until current setting are applied by activating the *Apply* button. The other cells labeled *Custom Colors* provide a user defined palette for storing colors. The custom colors stay resident between successive color dialog pop ups.

The *Drag and Drop mechanism* is applied to store and restore colors. The colors can be copied arbitrary between all cells (except that a color could not be stored to the old color cell). To drag and drop a color

1. move your mouse cursor on top of a color cell (source),
2. press down on the left mouse button and keep it pressed,
3. move your mouse cursor on top of another cell (destination),
4. release the mouse button.

Buttons. The four buttons named *OK*, *Apply*, *Reset* and *Cancel* are for quitting the dialog and applying the changes to the underlying color (OK), applying the changes without quitting (Apply), resetting the current color to the old color, i.e. discarding last changes (Reset) and quitting the dialog without applying the changes (Cancel). The fifth button named *Help* is for displaying this documentation in the amira help window.

Color Picker. The Color Picker provides visual selection of a color. Depending on the selected color component (done by the toggle button on the left side of each slider) the two other components of the color can be set with the picker. One component corresponds with the vertical extent of the color picker and the other with the horizontal. The selected component can not be changed with the picker. To select a color using the color picker

1. Move your mouse cursor on top of the color picker.
2. Press down on the left mouse button and move your mouse. While moving the mouse around the current color is set to the color at the mouse position.
3. Release the mouse when you are done.

10.3 Colormap Editor

To modify existing colormaps push the edit button (pencil icon) which is visible when the colormap is selected. The colormap editor pops up. As you can see there is a menu bar near the top border of the window with the sub menus *Edit*, *Mode* and *Brush*, which help you to control the behavior of the colormap editor and to change components of the chosen colormap.

Below the menu bar a so-called ”color chart” is displayed. The red, green and blue lines are the graphs of the *color channel* components of the colormap, the underlying color model is RGB (at startup). The axes are not shown directly, the x-axis ranges from the lowest to the highest colormap index and the y-axis from 0.0 to 1.0 according to the RGB model. Thus, a point on a color line indicates the amount of color for the corresponding color channel and color index. You can manipulate the course of a line by setting a brush onto any of its points and moving it up or down. A brush is set by pressing a mouse button, the left one for the red line, the middle one for the green line and the right one for the blue line.

Below the color chart a ”color bar” is displayed which is a larger version of the one used for display in the colormap port. Its only function is to show you the actual appearance of the modified colormap in a smoother way by linearly interpolating the colors between the indices.

The largest area of the editor window is occupied by the ”color buttons” which represent the color in every position (index) of the chosen colormap, each position is called a ”color cell”. You can set the *focus* which is the target index of modifications applicable by the color sliders (see below), and it is also possible to modify the colormap by dragging a color cell. The index of the focus color cell is shown below the color buttons.

In the lower area of the window there are some ”color sliders”, one for each color channel by which you can modify color channel values with respect to the focus cell. This means that the values of neighboring cells are affected as well as you can see in the color chart. The sliders are manipulated by dragging the small triangles, alternatively values in fix-point format may be entered directly into the text fields to the right of the sliders.

The last three buttons named *OK*, *Apply*, and *Cancel* are for quitting the editor, applying your changes to the underlying *Colormap* data object, and quitting the editor without applying your changes.

Interactive editing of a colormap is facilitated by two user interface elements, the *focus* and *colormap knots*.

Focus. The focus marks the active color cell which can be edited using the color sliders. It is represented by a black-and-white box drawn around the active color cell and by a black vertical line in the color chart.

You set the focus by clicking with the left mouse button on a colormap entry in the color button panel. A focus cell also gets a *knot* marker (see below); clicking again on a focus cell removes the knot and places the focus on to the leftmost color cell, just setting the focus to a different cell does not remove a knot.

Knots. Knots are fixed points in the colormap, i.e., they retain their values while the colormap is being manipulated by snapping (see ”Menu bar” - ”Edit” - ”Snap”) or by dragging the focus. Knots are represented by a small black box with a white surrounding in a color cell and a white vertical line in the color chart.

You set a knot by setting the focus and remove a knot (with the focus) by clicking another time on a color cell with the focus. The knots on the first and last cells of the colormap cannot be removed.

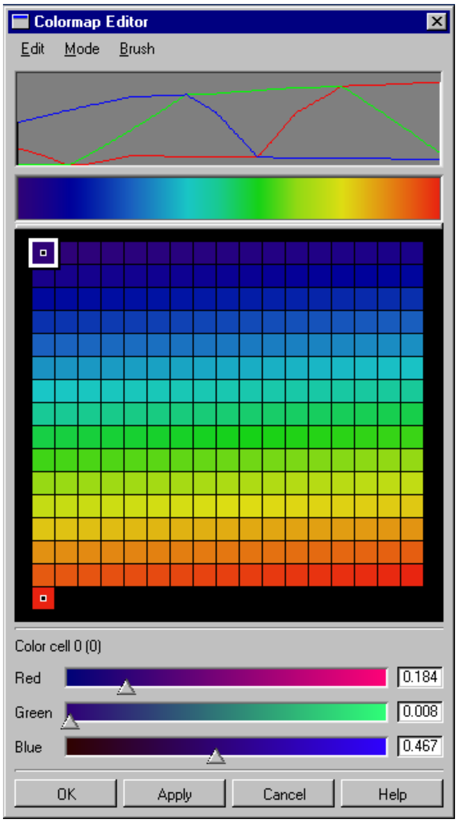


Figure 10.4: Amira colormap editor.

10.3.1 Description of the user interface elements

A. **Menu Bar**
The menu bar consists of the three sub menus ”Edit”, ”Mode” and ”Brush”.

a) **Edit Menu**
This sub menu offers two opportunities for editing the colormap. The first group deals with the undoing and the opposite - redoing - changes you made in the colormap. The second group ”snaps” one or more channels like tightening a rope between the left and right neighboring knots of the focus.

1. **Undo.** If you think that your last changes to the colormap have been a mistake this entry lets you take back your last changes. You can easily go back to a point of editing when you were content with the colormap and start editing again. The colormap editor memorizes up to 50 of your last changes.
2. **Redo.** If you took back too much this entry undoes the last undo, i.e. redoes the undone changes. This is the reverse function of ”Undo”. You may undo/redo your changes as often you wish but if you modify the colormap by another function (not ”Undo”/”Redo”) you lose the opportunity to redo the last undos. ”Redo” is only activated if your last action was invoking ”Undo”.
3. **Snap All.** By selecting this entry all colormap entries will be tightened like a rope (”snapped”) between their left and right neighboring knots leaving the knots itself untouched. This means that lines are drawn between all pairs of successive knots for each color channel. This function manipulates all color channels simultaneously.
4. **Snap.** This operation behaves much like a local ”Snap All”, i.e., snapping occurs only between the left and the right neighboring knots of the focus by tightening all color channels as if they were ropes between the knots.
5. **Snap Red or Hue.** Depending on the chosen color model (see ”Mode”) the first color channel ”Red” or ”Hue” is snapped between the left and right neighboring knots of the focus leaving the other color channels untouched. See ”Snap” for an explanation of snapping.
6. **Snap Green or Saturation.** Like in ”Snap Red” this entry manipulates the second color channel between the left and the right knot leaving the other color channels untouched.
7. **Snap Blue or Value.** Like in ”Snap Red” this entry manipulates the third color channel between the left and the right knot leaving the other color channels untouched.
8. **Snap Alpha.** This entry is only shown if ”Show Alpha” (see ”Mode”) has been activated. By selecting it the alpha channel will be locally snapped as described in ”Snap” between the left and the right neighbouring knots of the focus leaving the other color channels untouched.

b) **Mode Menu**
This menu allows you to select the color model used to modify the colormap. **RGB Sliders** chooses the RGB model where colors are represented by a red, green, and blue component. **HSV Sliders** chooses the HSV model where colors are represented by a hue, saturation, and value (intensity) component. In addition, an **Immediate Mode** toggle is provided. If this toggle is active all changes are immediately applied to the colormap and downstream modules are immediatly fired so that they can update their display.

c) **Brush Menu**
This has only relevance for the color chart. Here you can set the brush type used for editing a color channel curve. Four different brushes are supported. Their shapes are visually represented by corresponding icons.

d) **Extra Menu**
This menu allows you to replace the whole colormap by a set of predefined map. Currently four such predefined maps are available, a **Gray ramp** ranging from black to white, a **Hue ramp** ranging

from blue over green and yellow to red, a **Hot iron** map ranging from red over yellow to white, and a **Glow** map ranging from black over red and yellow to white. This last map is frequently used in epi-fluorescence microscopy. Two additional options are provided to subdivide the current colormap into a discrete set of colors (**Make steps**) and to define an alpha curve with a predefined gamma value (**Alpha curve**). If one of these options is chosen an additional dialog window pops up, allowing you to performs the appropriate operations.

B. **Color Chart**
The color chart shows the graphs of the colormap's colorchannel components. The red curve shows the first color channel (red or hue), the green curve shows the second channel (green or saturation), the blue curve shows the third channel (blue or value), and the black curve shows the foirth channel (alpha). The knots and the focus are represented by white and black vertical lines, respectively. The left edge of the color chart shows the values of the leftmost index of the colormap and the right edge those of the rightmost index.

C. **Color Bar**
Like in the colormap port this color bar displays a continuous form of the chosen colormap by linearly interpolating the colors between the colormap entries. If *Show Alpha* is enabled alpha values are represented by a certain amount of transparency in the colors, i.e., you see the colors translucent over a white and black chequer. For an alpha value of 0.0 you see no color information but only the white and black chequer, for a value of 1.0 you do not see a chequer because the colormap entry is not translucent.

D. **Color Buttons**
An entry of the color button panel is also called a color cell. Each color cell shows the color associated to a color index without the alpha value. The indices are counted from left to right and from top to botom. Thus the color cell in the upper left corner shows the color of the leftmost colormap entry and the color cell in the lower right corner shows the color of the rightmost colormap entry. The index of the focus cell is displayed below the color button panel. Just after the index the data value which corresponds to the focus cell is shown in brackets. The data value depends on the current range of the colormap.

E. **Color Sliders**
In the lower area of the *Colormap Editor* window there are four color sliders allowing you to modify the values of the focus cell. Depending of the current color model the first three sliders are associated to red, green, and blue or to hue, saturation, and value, respectively. The fourth slider always modifies the alpha value. In front of each color slider there is a toggle button. If the toggle is activated the corresponding color can be edited using the left mouse button in the *color chart* (see below).

F. **Control Buttons**
These three buttons let you choose whether the changes to the colormap should be kept or not. By pressing the "OK" button the changes made with the editor are written back to the colormap object that you are working on. This action also causes the editor to exit. If you just want to write back the changes without exiting, e.g., if you want to see how your changes take effect, just press the *Apply* button. If *Apply* is done in the *Immediate* mode however, the previous state of the colormap cannot be

restored by *Cancel*.

If you think that your manipulations have gone totally wrong you can always decide to keep the old colormap and throw away your changes. This is done by pressing the *Cancel* button which also closes the editor window. *Cancel* restores the colormap to the last state when the *Apply* button was pressed, or to the initial, if *Apply* was left untouched.

10.3.2 How to modify a colormap

This section describes how to modify a colormap in various ways. For the effects of invoking the various menu items see *Menu bar*.

A. **Using the color chart**
A color channel can be edited by modifying the corresponding curve with a brush. Values are increased by approaching the curve from the bottom side with a brush while holding down the left mouse button, and vice versa. Only one curve can be edited at a time. The curve to be edited is determined by the toggle button in front of the four color sliders. For example, if you want to modify the alpha curve you first have to select the toggle button in front of the alpha slider.

Four different brushes can be chosen in the *Brush* menu, namely a small square, a bigger square, a circle, and a diamond. The shape of the brush determines how a curve will be modified when approaching it with the mouse. When the brush touches the curve it moves the color channel up or down (depending from which side it comes) to the first pixel outside the brush's area. Colors are clamped to the channel's minimum and maximum allowed values.

The new colors are displayed instantaneously in the color bar and in the color buttons. If you modify the focus cell the color slider corresponding to the cuve being edited will also be updated.

B. **Using the color buttons**
This item offers you the most opportunities to modify the colormap. You can set the focus, set / unset a knot or modify a certain region of the colormap by dragging the focus cell. Due to the fact that a focus always exists a special operation for unsetting the focus is not necessary because you unset a focus simply be setting a new one.

1. **Setting the focus**
   You simply select a focus by clicking once in a non-focus cell. The new focus cell will be surrounded by a black and white border while the border of the old focus cell disappers. The other displays will be refreshed, i.e. the color index displayed below the color buttons will be set to the focus cell's index, the color chart will display a black vertical line in a position corresponding to the new focus cell and the color sliders will show the color channels' values in the focus cell. As mentioned above you simply unset a focus by setting a new one.
2. **Setting a knot**
   Because every focus cell must have a knot you set a knot simply by setting the focus, i.e. by clicking once in a non-focus cell without a knot. The knot appears as a small black box with a white surrounding in the middle of the new focus cell. The color chart displays it with the

focus as a black vertical line. If you change the focus the knot still remains in the color cell but without the focus it is displayed in the color chart as a white vertical line.

3. **Unsetting a knot**
   You unset a knot by clicking another time into a focus cell. This does two things: First the knot is removed from the color buttons and the color chart. Second the focus is set to the first color cell, i.e. to the upper left corner of the color button panel and to the left edge in the color chart. The displays are refreshed, i.e. no small black box in the color cell and no vertical line at the corresponding position in the color chart will be shown.

4. **Dragging the focus cell**
   When you click the first time in a non-focus cell, you can hold the mouse button (be careful: a click in a focus cell unsets a knot) and drag the focus cell over the color buttons. The corresponding color values will be temporarily copied to its new position and the color channels between the next knots to the left and to the right from the actual position will be snapped i.e. the color channels will be tightened like a rope between the focus and the left or right knot. When you move in a new region between two knots the old region will be restored and the new region will be affected by snapping. When you release the button the focus cell's color values (the one which you dragged) will be copied to its last position and the colormap remains in this state, i.e., with the copied color values in the focus cell and the snapped color values between the focus and the left or right knot. While moving the mouse the color chart is updated appropriately. The color sliders stay the same because the focus cell values remain the same (remember: you drag the focus and copy its cell's values).

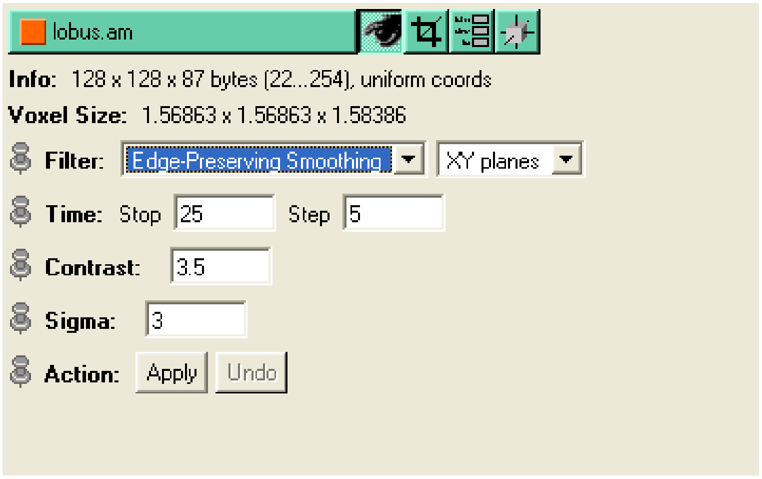
C. **Using the color sliders**
The color sliders offer you three ways to modify the values associated to the focus cell, either by setting the value explicitly in the text field, by dragging the small triangle or by clicking somewhere into the slider area.

All modifications have effect on the surrounding colormap entries between the next knots to the left and to the right of the focus cell. A modification of a value changes the surrounding colormap entries relative to their distances to the focus. This is similar to raising / lowering a rubber band between the left and right knot in the focus position. You can clearly see the effect by looking at the curves displayed in the color chart. The values and displays are refreshed in the same way as described in the previous sections.

10.4 Digital Image Filters

This editor provides digital image filters for 3D image data sets, such as smoothing, unsharp masking, or morphological operations. Some filters operate in 3D while others are applied to two-dimensional slices. In the latter case, the orientation of the 2D slices can be selected via an option menu. Currently, the following filters are supported:

- Minimum Filter


lobus.am

Info: 128 x 128 x 87 bytes [22...254], uniform coords

Voxel Size: 1.56863 x 1.56863 x 1.58386

Filter: Edge-Preserving Smoothing XY planes

Time: Stop 25 Step 5

Contrast: 3.5

Sigma: 3

Figure 10.5: Editor for applying digital image fi lters.

- Maximum Filter
- Unsharp Masking
- Laplacian Zero-Crossing Filter
- Median Filter
- Gauss Filter
- Sobel Filter
- Histogram Filter
- Edge-Preserving Filter
- Lanczos Filter
- Sigmoid Filter
- Brightness/Contrast Filter
- Moments Filter

Ports

Filter

Specifies the filter and its domain. It allows you select whether the filter should be applied to the XY, XZ, or YZ slices or to the entire three-dimensional image.

Action

Pressing button *Apply* starts the computation. The *Undo* button allows to undo the last filter operation.

10.4.1 Minimum Filter

The minimum filter replaces the value of a pixel by the smallest value of neighboring pixels covered by a NxN mask. The size of the mask can be adjusted via the input field *kernel size*. A value of 3 denotes a 3x3 mask (respectively 3x3x3 in 3D). If applied to a binary label field the minimum filter implements a so-called erosion operation. It reduces the size of a segmented region by removing pixels from its boundary.

10.4.2 Maximum Filter

The maximum filter replaces the value of a pixel by the largest value of neighboring pixels covered by a NxN mask. The size of the mask can be adjusted via the input field *kernel size*. A value of 3 denotes a 3x3 mask (respectively 3x3x3 in 3D). If applied to a binary label field the maximum filter implements a so-called dilation operation. It enlarges the size of a segmented region by adding pixels to its boundary.

10.4.3 Unsharp Masking

This filter sharpens an image using an unsharp mask. The unsharp mask is computed by a Gaussian filter of size *kernel size*.

Then, the smoothed image is subtracted from the original image such that only high contrast remains. The weighted difference of the original image (weight: $\frac{c}{2c-1}$) and the blurred image (weight: $\frac{1-c}{2c-1}$) is calculated afterwards using the *sharpness parameter c*. It determines the relation between original and blurred image effectively controlling the amount of sharpness. c can be adjusted via a text input field and should be in the range of 0.6 to 0.8. A value of 1 leaves the image unchanged.

10.4.4 Laplacian Zero-Crossing Filter

The Laplacian filter is a rotation invariant edge detection filter. The algorithm finds zero crossings of the second derivation, i.e. changes of the sign of the first derivation of the ”image function” which may indicate an edge.

10.4.5 Median Filter

The median filter is a simple edge-preserving smoothing filter. It may be applied prior to segmentation in order to reduce the amount of noise in an image. The filter works by sorting pixels covered by a NxN mask according to their grey value. The center pixel is then replaced by the median of these pixels, i.e., the middle entry of the sorted list. The size of the pixel mask may be adjusted via the text field labeled *kernel size*. A value of 3 denotes a 3x3 or mask (respectively 3x3x3 in 3D). An odd value is required.

10.4.6 Gauss Filter

The Gauss filter smooths or blurs an image by performing a convolution operation with a Gaussian filter kernel. The text fields labeled *kernel size* allow to change the size of the convolution kernel in each dimension. A value of 3 denotes a 3x3 kernel (respectively 3x3x3 in 3D). Odd values are required. The text fields labeled *sigma* allow to adjust the width of the Gauss function relative to the kernel size.

10.4.7 Sobel Filter

The Sobel-Filter is a rotation variant edge detection filter. It convolutes the image with 4 different filter kernels representing horizontal, vertical and two diagonal orientations. Each kernel is constituted of a combination of gaussian smoothing and the differentiation in the proper orientation.

10.4.8 Histogram Filter

This filter performs a so-called *contrast limited adaptive histogram equalization (CLAHE)* on the data set. The CLAHE algorithm partitions the images into *contextual regions* and applies the histogram equalization to each one. This evens out the distribution of used grey values and thus makes hidden features of the image more visible. Parameter *Clip Limit* determines the contrast limit for the CLAHE algorithm.

10.4.9 Edge-Preserving Smoothing

This is a smoothing filter that models the physical process of diffusion. Similarly to the Gaussian filter, it smooths out the difference between grey levels of neighbouring voxels. This can be interpreted as a diffusion process, in which energy between voxels of high and low energy (grey value) is leveled. In contrary to the Gaussian filter, it does not smear out the edges because the diffusion is reduced or stopped in the vicinity of edges. Thus, edges are preserved.

Note that in the 3D modus the computation can take rather long time if the data is large. A faster preview is always possible by switching to the 2D mode.

The stop *time* determines how long the diffusion runs. The longer it runs, the smoother the image becomes. The *time step* determines how accurately this process is sampled.

The *contrast* parameter determines how much the diffusion process depends on the image gradient, i.e., how much the smoothing is stopped near edges. A value of 0 makes the diffusion independent of the image gradient and smooths out the edges, a large value prevents smoothing in all edge-like regions.

In order to make the diffusion process more stable, the image is prefiltered by a Gaussian filter with parameter *sigma*. All features of size *sigma* are removed. This allows to remove noise from the image. But a too large value may also remove relevant features.

10.4.10 Lanczos Filter

The Lanczos filter can be used to sharpen images. It performs a convolution with a Lanczos kernel:

$$f(x) = \frac{\sin(\pi x)\sin(2\pi x)}{2\pi^2 x^2},\ |x| < 2$$

The kernel size in each dimension can be adjusted using the parameter inputs *kernel size*. A value of 3 denotes a 3x3x3 kernel. Odd values are required.

Parameter *Sigma* determines the effective size of the Lanczos function. For large values of Sigma the effect of the filter will be a smoothing rather than a sharpening.

10.4.11 Sigmoid Filter

This filter operates on single voxels (kernel size 1) and is used to raise a specific intensity range. This us useful as preprocessing step in image segmentation. The intensity range is described by its center β and it width α. The target image range is given by the interval [*min*, *max*].

$$f(z) = (max - min) \cdot (1 + \exp(-\frac{z-\beta}{\alpha}))^{-1} + min$$

10.4.12 Brightness and Contrast Filter

Modifies the the image brightness and contrast.

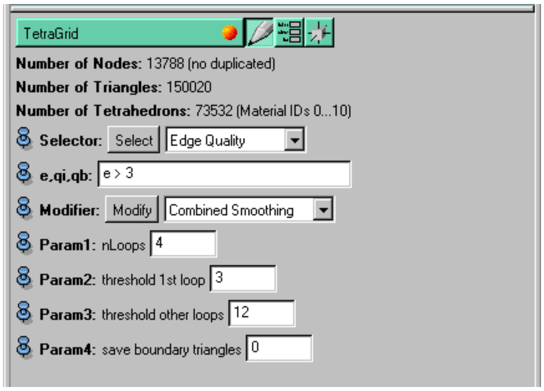
10.4.13 Moments Filter

This filter calculates the n-th centralized moment of the data in a gliding window. The centralized moments of order n are defined by:

$$f(x) = \frac{1}{N-1}\sum_{i=0}^{N}(x_i - \bar{x})^n.$$

The second moment ($n = 2$) is therefore the local variance in the data. For some data sets this can be used to mask out noisy regions or to detect edges.

Because of the n-th power involved the computation you may want to use CastField to do a conversion of your data set to floats or doubles first.


TetraGrid

Number of Nodes: 13788 (no duplicated)
Number of Triangles: 150020
Number of Tetrahedrons: 73532 (Material IDs 0...10)
Selector: Select | Edge Quality
e,qi,qb: e > 3
Modifier: Modify | Combined Smoothing
Param1: nLoops 4
Param2: threshold 1st loop 3
Param3: threshold other loops 12
Param4: save boundary triangles 0

Figure 10.6: Grid editor for improving the quality of tetrahedral grids.

10.5 Grid Editor

The Grid Editor allows you to analyze the quality of a tetrahedral grid according to different quality measures and to semi-automatically improve the grid quality.

To do this push the edit button of a selected tetrahedral grid (see the icon above). Then a user interface composed of different button controls (Figure 10.6) will appear in amira's Working Area. In case you select some other *Selector* or *Modifier* other ports might be shown. The editor works in conjunction with the GridVolume module. If such a module is not already active, an instance of it will be automatically created when the editor is invoked.

Tetrahedra can be distorted in different ways (see Figure 10.7):

- *Slivers* contain four nearly coplanar vertices forming a quadrangle.
- *Caps* consist of a triangle and a fourth vertex 'just above' it.
- *Needles* consist of a triangle and a fourth vertex 'far away'.
- *Wedges* are characterized by one sharp edge.

Using the tools of the *Select* menu

At the menu of port *Selector* you can choose one of several selection criteria. At the next port you can enter an expression which will be evaluated for each tetrahedron or edge. When you press the *Select* button, the number of selected tetrahedra or edges is shown in the *Console Window*, and the selected tetrahedra are shown by the *GridVolume* module.

- **Index Selector**
  This tool selects tetrahedra according to their index (i). This is mainly for debugging puprposes.
- **Tetra Quality Selector**

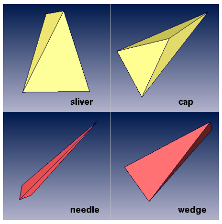


Figure 10.7: Four examples of distorted tetrahedra.

This tool selects tetrahedra according to different quality measures.

'd' is the determinant of the matrix composed of the vectors pointing from vertex 0 to the vertices 1, 2, and 3. The tetrahedron volume is 1/6 of the absolute value of d. d approaches zero if the four vertices are nearly coplanar. d ¡ 0 detects tetrahedra with an inverted orientation, which should normally not occur.

'R' is the ratio of the radii of an inscribed and a circumsribed sphere. R reaches its optimal (maximal) value 1/3 for an equilateral tetrahedron. All types of distorted tetrahedra are detected by a small value of R. Therefore R is taken as the quality measure for all modifiers of the *Grid Editor* (see below). R shouldn't be smaller than approximately 1/10 of the optimal value.

'r1' is the ratio of the smallest to the largest edge length. r1 reaches its optimal (maximal) value 1 for an equilateral tetrahedron. Needles and wedges are detected by a small value of r1.

'r2' is the tetrahedron volume divided by the third power of the largest edge length, normalized to an optimal (maximal) value 1 for an equilateral tetrahedron. All types of distorted tetrahedra are detected by a small value of r2.

- **Dihed/Solid Angle Selector**
  This tool selects tetrahedra according to their minimal and maximal dihedral angles (d,D) or solid angles (s,S). For each edge of a tetrahedron the dihedral angle is defined as the angle between its adjacent faces. The solid angles are related to the tetrahedron vertices; they measure the part of the unit sphere which is occupied by the tetrahedron. The solid angle at a tetrahedron vertex can be maximally 2π, or 360 deg. For an equilateral tetrahedron all dihedral angles are approx. 70 deg, all solid angles are approx. 30 deg.

  You can combine different selections, e.g., the selection `(d<10) && (D>140)` detects *slivers*, and the selection `S>180` detects *caps*.

- **Edge Quality**
  This tool primarily selects edges, and then shows all tetrahedra adjacent to that edges. Edges are selected according to their length (e) or to the 'edge quality' q, which is computed as follows: for each tetrahedron adjacent to the edge the length l of the opposite edge and the dihedral angle d at the opposite edge are determined. The contribution of the tetrahedron is $l \cot(d)$. q is the sum of those contributions over all adjacent tetrahedra. qi refers to the inner edges, qb to the boundary edges of the tetrahedral grid.

For an ’ideal’ grid, all qi and qb should be positive. Such a grid would be well suited for a numerical simulation, because the Finite Element stiffness matrix (probably for the Laplacian operator) is an M-matrix if q is positive for all edges.

Using the tools of the *Modify* menu

At the menu of port *Modifier* you can choose one of several modifiers. When you press the *Modify* button, the grid modification will start. Some informations will be given at the *Console Window*.

There is a certain inconsistency between the tetra quality selector and the modifiers of the *Grid Editor*, because the tetrahedron quality criterion for all modifiers is the *inverse* of the radius ratio R as defined above. In applying the modifiers, keep in mind that the optimal (minimal) value of 1/R is 3 and distorted tetrahedra are detected by large values of 1/R.

The modifiers *Laplace Smoothing*, *Optimization Smoothing* and *Flip Edges and Faces* are mainly for debugging purposes, because the *Combined Smoothing* modifier is a combination of them which should give the best results in most cases. The modifiers *Repair Bad Tetras* and *Bisect Inner Edges* are still in an experimental state. We recommend to apply the *Remove Inner Vertices* and the *Combined Smoothing* modifiers.

- **Laplace Smoothing**
  This tool improves mesh quality by moving inner vertices. For each inner vertex the center of mass of the adjacent vertices is calculated. The vertex is moved to that location if this improves the quality 1/R of the adjacent tetrahedra. Otherwise the midpoint between the old location and the center of mass is examined. Parameter *nLoops* defines the number of smoothing loops (maximally 10). Laplace smoothing will in general improve the mesh quality 1/R, but in most cases an *Optimization Smoothing* will be superior.
- **Optimization Smoothing**
  This tool improves mesh quality by moving inner vertices. For each inner vertex a new location is determined that optimizes the quality 1/R of the adjacent tetrahedra. Parameter *nLoops* defines the number of smoothing loops (maximally 10), parameter *threshold* defines a threshold for tetrahedron quality which is applied starting with the second loop. If all adjacent tetrahedra have a quality better than *threshold*, the vertex position is not changed. The default value 12 (quadruple of the optimal value) leaves tetrahedra unchanged which should be acceptable in most cases. Selecting a smaller value will induce opimization of more vertex locations.
- **Flip Edges and Faces**
  This tool improves mesh quality by flipping edges and faces. For each inner (triangular) face the adjacent tetrahedra are determined. It is examined whether the face is a boundary face and whether the adjacent tetrahedra form a convex polyhedron. Depending on this classification a suitable type of edge or face flipping is selected. The flip operation is only performed if it improves the quality 1/R of the tetrahedra involved. Parameter *threshold* defines a threshold for tetrahedron quality. If all adjacent tetrahedra have a quality better than *threshold*, a face is not examined for flipping. If parameter *save boundary triangles* is set to 1, the edges and faces of the exterior grid boundary and the interior boundaries between different materials will not be

flipped.

- **Repair Bad Tetras**
  This tool tries to repair *slivers* and *caps*. For a sliver, two opposite edges with obtuse dihedral angles are bisected. If the distance between the new vertices is small compared to the sliver's mean edge length, the edge connecting them is collapsed. For a cap, the triangle opposite to the vertex with largest solid angle is determined. If that triangle is part of the outer boundary, the tetrahedron is removed. Otherwise, it is examined if the cap can be removed by a face flip. Parameter *threshold* defines a threshold for tetrahedron quality 1/R. Only tetrahedra with a quality worse than the given threshold will be examined for repair.
- **Remove Inner Vertices**
  This tool improves mesh quality by removing inner vertices. In a tetrahedral grid, the mean number of tetrahedra incident on an inner vertex is 24. If a vertex is incident on less than 10 tetrahedra, it is very likely that the mesh quality can be improved by removing that vertex and reconnecting the hole.

  All inner vertices incident on a number of tetrahedra less or equal the value of parameter *max num neighbours* are inspected for removal. They will be removed, if this improves tetrahedral quality 1/R. If a value ¿ 0 is set for parameter *max edge length*, this defines an upper bound for the edge length. An inner vertex will not be removed, if this would imply creation of a longer edge.
- **Bisect Inner Edges**
  This tool improves mesh quality by bisection of inner edges. An inner edge is bisected if for its vertices different boundary conditions are defined. After bisection, apply *Optimization smoothing* to improve the position of the new vertices.
- **Combined Smoothing**
  This tool combines edge and face flipping and optimization smoothing. Parameter *nLoops* defines the number of 'large loops' (maximally 10), the other parameters define thresholds for tetrahedron quality 1/R which are applied in the first loop and the other loops, respectively. Setting the first threshold to 3 means that all edges and faces will be inspected for flipping and all vertices for optimization in the first loop. If parameter *save boundary triangles* is set to 1, the edges and faces of the exterior grid boundary and the interior boundaries between different materials will not be flipped.
- **Flip/Bisect Long Edges**
  This tool tries to remove inner edges depending on their length or edge quality q as defined above. Parameter *max edge length* defines the maximal allowed edge length and parameter *min edge quality* the minimal allowed edge quality, which must be less or equal to zero. Setting a minimal quality of zero means that edge quality is ignored in edge selection. Parameter *threshold* sets an upper limit for tetrahedron quality 1/R. The selected edges are removed, preferably by an edge flip, or by an edge bisection, if all newly generated tetrahedra will have a quality below that threshold.

  This tool should be applied repeatedly, until the number of flips as reported in the *Console Window* goes down to 0, and applying an Optimization smoothing in between may improve the results.

10.6 ImageCrop Editor

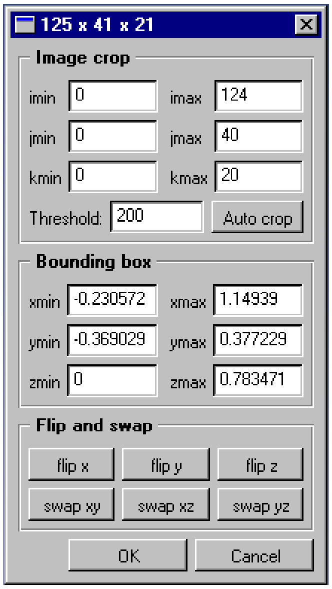

125 x 41 x 21

Image crop

imin 0 imax 124
jmin 0 jmax 40
kmin 0 kmax 20
Threshold: 200

Bounding box

xmin -0.230572 xmax 1.14939
ymin -0.369029 ymax 0.377229
zmin 0 zmax 0.783471

Flip and swap

Figure 10.8: The ImageCrop editor allows you a crop a 3D image and to modify its bounding box or voxel size.

This editor works on 3D fields with an arbitrary number of components defined on a regular grid, e.g., 3D images or 3D vector fields with uniform, stacked, rectilinear, or curvilinear coordinates. It allows you crop such a field, i.e., to discard all voxels lying outside of a specified subvolume in index space. Alternatively, you may adjust the physical size of the field by changing its bounding box. Note that this does not mean resampling, i.e., the number of voxels will not be changed.

Each node of a regular grid can be addressed by an index triple *(i,j,k)*. Each index ranges from zero to the number of slices minus one in that direction. This editor also allows you to add new slices on every side in every dimension. Last but not least you can flip and rotate the slices with respect to the global x-, y-, or z-direction.

When activating the crop editor the dialog shown in Figure 10.8 pops up. If the editor is activated for a field with uniform, stacked, or rectilinear coordinates in addition a tab box dragger is displayed in the viewer window. The dragger allows you to specify the subvolume to be cropped interactively in 3D. When resizing the dragger the index bounds of the current subvolume are permanently updated in the corresponding fields in the dialog window.

The *Auto crop* button automatically adjusts the subvolume to be cropped by taking into account the value of the *threshold* field. Slices at the boundaries of the original data volume are cropped if they contain only data values smaller than the specified threshold. In this way it is easy to isolate a bright object in a 3D image with a large dark background.

In order to add slices you have to set the indices in the text fields beyond the limits of the total data volume. When slices are added, on default the values of the last slice in that direction are replicated. If you switch off the *replicate* toggle the text field *pixel value* becomes active. You can then specify a data value which is used to initialize the new slices.

In order to manipulate the bounding box of the data object new values have to be entered in the corresponding text fields. For data objects with uniform coordinates two modes are available, *bounding box* and *voxel size*. See section below for details.

All changes take effect if you press the *OK* or *Apply* buttons. They are discarded by activating the

Cancel button. As usual the editor is closed by pressing *OK* or *Cancel*, *Apply* will leave it open, *Cancel* restores the previous state.

10.6.1 Cropping an image by dragging and moving the box

Some experience with manipulating *draggers* is assumed. Using the draggers you can reduce or enlarge the data volume you want to preserve, enlarging is, of course, possible only up to the size of the bounding box. The size of the dragging box can be manipulated intuitively in every direction of the three axes, i.e. along the x-, the y- or the z-axis. The values in the text fields of the editor change according to your manipulations. A certain minimum thickness is preserved while reducing a dimension of the box, for further reductions the text fields have to be used.

Using the draggers you control the size of the subset of the data volume which has to be preserved, but the location of it in the total data volume is set by moving the box around to the desired location.

10.6.2 Cropping an image by setting values explicitly in the text fields

The text fields in the cropping area of the editor show the indices of the slices that have to be preserved for every dimension. You can easily define a subset by setting a range of indices explicitly in the text fields of an axis - this defines the size as well as the location of the data subset. Notice that a slice index refers to an axis that is perpendicular to the slice.

10.6.3 Adding new slices

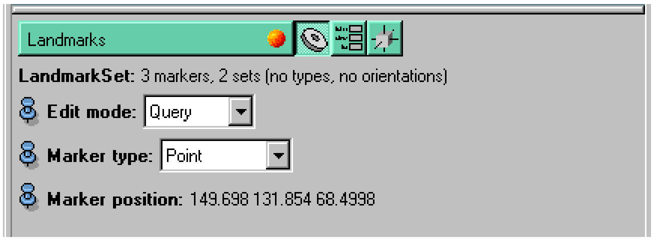
Adding new slices is only possible by setting an index that does not fall into the range given by the minimum and maximum presently shown in the *Image Crop* panel, thus a negative index must be specified if the minimum is zero. Use the *Min* or *Max* text field for the required axis text field to enter a new slice index.

A warning dialog pops up whether you really want to add new slices, and the slices are added on your confirmation to the volume at the specific end and according to the axis selected. The new slices contain the data of the last slice in this direction, e.g., if you add two slices before the slice with index 0 along the x-axis, they contain exactly the same data as the slice with index 0.

10.6.4 Changing the size of the bounding box

You change the size of the bounding box by setting new values explicitly in the bounding box fields. This does *not* affect the stored data, but its representation as displayed by a viewing module, e.g., OrthoSlice, by way of a different scaling for the specified dimensions.

The boundig box of a data set with uniform coordinates specifies the world coordinates of the center of the lower left front voxel (min values) and the center of the upper right back voxel (max values). For


Landmarks

LandmarkSet: 3 markers, 2 sets (no types, no orientations)

Edit mode: Query

Marker type: Point

Marker position: 149.698 131.854 68.4998

Figure 10.9: User interface of the landmark editor.

example, if your input is 256 pixels wide and the size of each voxel is 1mm, then you may set *xMin* to 0 and *xMax* to 255.

If a data set with uniform coordiantes is being edited, instead of the bounding box also the *voxel size* can be modified. You can switch between the two modes and see how bounding box and voxel size influence each other.

10.6.5 Flipping slices in one dimension

Invoke one of the three flip buttons to revert the order of slices in the corresponding dimension, e.g., click on *FlipX* to flip the slices in the x-dimension.

10.6.6 Exchanging two dimensions

Invoke one of the three exchange buttons to interchange the corresponding dimensions, e.g., click on *X-Y* to interchange the x- and y-dimension. CAUTION: The exchange operations are not rotations; they change the spatial orientation of the dataset.

10.7 LandmarkEditor

This component allows you to edit landmark sets. It operates by directly interacting with the 3D geometry in the viewer window, cf. Section 3.1.7. In order to do so the viewer has to be switched to interaction mode, e.g., by hitting `ESC` as described in its documentation.

The editor has different **Edit modes**, which are described in the following:

- **Add.** In this mode new landmarks are added by clicking onto a 3D geometry (e.g., an Isosurface or an OrthoSlice). Multiple clicks are required, if the landmark set contains more than one point set.
- **Move.** Markers can be moved in this mode by first clicking on the marker (selecting it) and then clicking to a new position.

- **Transform.** In this mode, markers can be moved using a dragger. Click on one of the markers, then use the dragger to manipulate it.
- **Flip.** Flip geometry of marker. Only makes sense, if the marker type is not *Point*.
- **Remove.** The marker you click on will be removed. If more than one point set is present the marker will be removed from all sets.
- **Query.** After clicking on a marker its type and its xyz-position are shown. The marker type can be changed via an option menu. The default marker type is a *Point*, represented by a little sphere. In addition, a number of predefined medical marker types can be selected. In contrast to point-type markers, medical markers have fixed predefined sizes (world coordinates assumed to be given in centimeters).

See also documentation for Landmark Set.

10.8 LineSet Editor

This component allows you to edit linesets. It operates by directly interacting with the 3D geometry in the viewer window, cf. Section 3.1.7. In order to do so the viewer has to be switched to interaction mode, e.g., by hitting `ESC` as described in its documentation. Points and lines can be selected by clicking on them in the viewer. Selected points or lines will be highlighted in red. Multiple actions can be performed by pushing one of the buttons in the *Action* port. They are described below.

The *Selected* port informs you how many points and lines are selected at the moment.

In the *Display* port you may choose in which way the lineset is displayed. By clicking on the toggles optionally *all points*, *endpoints* and *lines* will be displayed in the viewer. Only displayed geometry is selectable. Default is endpoints and lines.

The *Select* port provides the following selection modes:

- **All.** Clicking this button all lines will be selected.
- **By Length.** Clicking this button an additional dialog pops up where you can enter the minimal and maximal numbers of points in lines that are to be selected. Pushing the *OK* buttons actually selects the requested lines.
- **Clear.** Pushing this button the current selection will be cleared.

The *Action* port provides the following actions:

- **Connect.** Pushing this button two lines can be connected. In order to do this exactly two points have to be selected, otherwise this button will not be active.
- **Delete.** Selected points and lines will be deleted from the lineset.
- **Stretch.** Pushing this button all selected lines will be smoothed.
- **Split** Pushing this button all lines will be splitted at selected points.

10.9 Parameter Editor

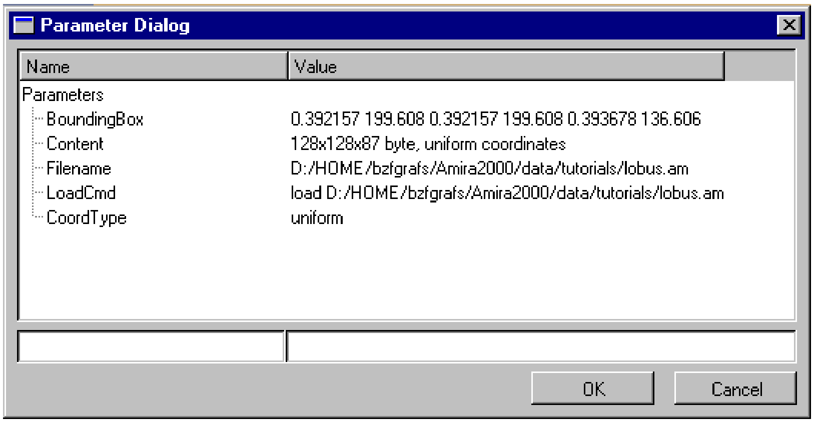
The *Parameter Editor* lets you view and modify the parameter list associated with a data object. A parameter is a name/value pair which contains some extra information related to the data. Some parameters have a special meaning within amira, see Section 3.2.7 of the user's guide.

To change the value or name of an existing parameter, click on the respective line and change the name/value in the text fields. To add a new parameter right click on the root of the parameter tree, labeled *Parameters* and select *New Parameter*. Then select the new parameter to change its name and value.

Note, that there are many standard file formats which do not support parameter information to be written. If in doubt, use amira's native AmiraMesh format. This format does write parameters.

10.10 Plot Tool

The *Plot Tool* is a special-purpose viewer designed to display 1-dimensional data, e.g., function curves. Instead of being a separate amira module the *Plot Tool* will be invoked implicitely by certain modules such as the data probing modules or the LabelVoxel module. Each of these modules generates 1-dimensional data such as a function plot along a line or a histogram. While the data-generating modules control the initial settings of the plot window the user can freely adjust these settings afterwards.



Name	Value
Parameters	
BoundingBox	0.392157 199.608 0.392157 199.608 0.393678 136.606
Content	128x128x87 byte, uniform coordinates
Filename	D:/HOME/bzfgrafs/Amira2000/data/tutorials/lobus.am
LoadCmd	load D:/HOME/bzfgrafs/Amira2000/data/tutorials/lobus.am
CoordType	uniform

Figure 10.10: User interface of amira's parameter editor.

10.10.1 Plot basics

The layout of the plot is determined by objects and groups of objects. These entities are processed by a plot engine. In particular, there are objects for

- Curves
- Cartesian Axes
- Polar Axes
- Plot Areas
- Legends
- Annotations
- Markerlines
- Lattices
- Colormaps

The plot engine processes the objects top to bottom (see Figure below). The sequence of objects determine their behavior. Objects of type *PlotArea* define the area within the plot window where objects are placed. This area is given in normalized coordinates with the origin at the lower left corner. Besides this a PlotArea object also acts as a grouping object. An axis is placed in such an area and establishes together with the preceding PlotArea a window-viewport (world coordinates to normalized coordinates) transformation. Notice that the objects have to be kept in the following sequence: 1. PlotArea, 2. Axis, 3. *Curves* and *Markerlines*. Legends display all succeeding curves in the sequence and can be placed whereever needed. It is possible to have more than one plot area or axis in a plot setup.

Every object is identified by a unique name. You need to know these names if you want to change certain object attributes via the command interface of the *Plot Tool*.

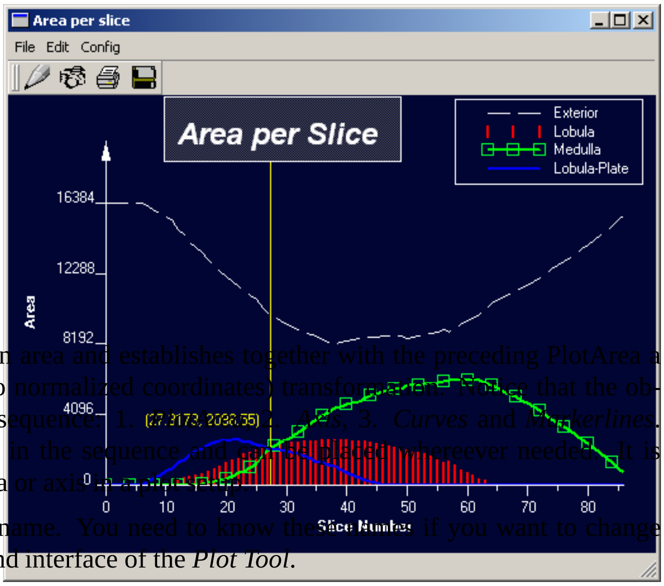
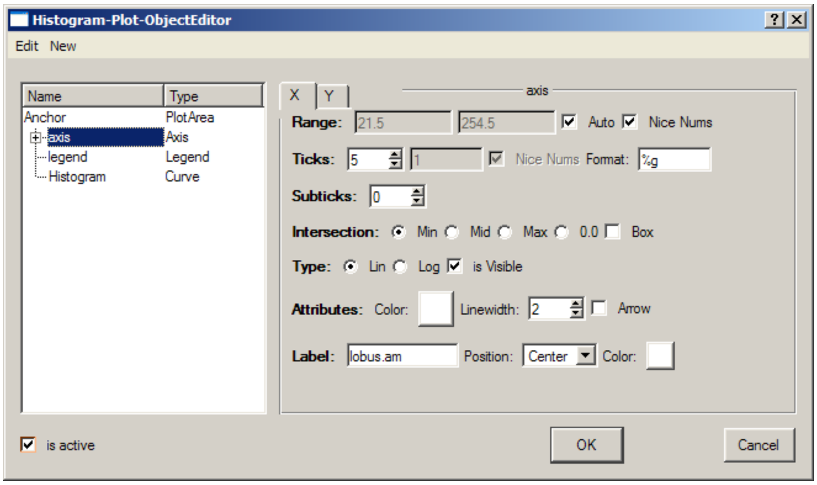


Figure 10.11: Amira's plot window.

10.10.2 Editing parameters

In order to change the parameters of a plot object select the *Edit Objects...* item from the *Edit* pulldown menu of the *Plot Tool*. A window appears with the list of names of all objects currently in use. By selecting one of these objects the parameters of that object are displayed and can be changed. The *is active* toggle near the lower left corner of the window can be used to switch the processing of the selected object on or off. If the processing of an object is switched off which has group functionality (Plot Areas, Plot Groups), all objects within that group are also not processed.



Name	Type
Anchor	PlotArea
axis	Axis
legend	Legend
Histogram	Curve

X Y axis

Range: 21.5 254.5 [✓] Auto [✓] Nice Nums

Ticks: 5 1 [✓] Nice Nums Format: %g

Subticks: 0

Intersection: (●) Min ( ) Mid ( ) Max ( ) 0.0 [ ] Box

Type: (●) Lin ( ) Log [✓] is Visible

Attributes: Color: Linewidth: 2 [ ] Arrow

Label: lobus.am Position: Center Color:

[✓] is active

Figure 10.12: Dialog for editing plot objects. On the right hand side the controls for editing axis parameters are shown.

In the following sections all not self-describing parameters are documented.

10.10.2.1 Editing axis parameters

To choose the x- or y-component of an axis just click on the appropriate tab. If you want to change the range you have to set off the *Auto* toggle. Now the range fields are sensitive. The *Nice Num* toggle sets the range to the next 'good looking' boundaries. A *tick delta* can be typed in after the *number of ticks* are set to -1.

It is possible to zoom interactively into your data using the mouse. For this purpose drag a rectangle within the plot area by pressing the shift key *and* the left mouse button while moving the mouse. To go back to the automatic ranges click into the plot area with the left mouse button while the alt-key is pressed at the same time.

Every axis object has a hidden grid child object which is not active by default. To show the grid just open the grid child and select it. Then set the *is active* toggle to on.

10.10.2.2 Editing annotation parameters

To position an annotion on your plot you can switch between world coordinates or normalized coordinates. The usage of world coordinates is useful if you like to annotate a certain feature e.g., an extrema of a curve. In this case you have to insert the annotion after that curve object. You can also position an annotation in the plot window interactivly by pressing the left mouse button on the annotation, move the mouse and release it at the new position.

10.10.2.3 Editing legend parameters

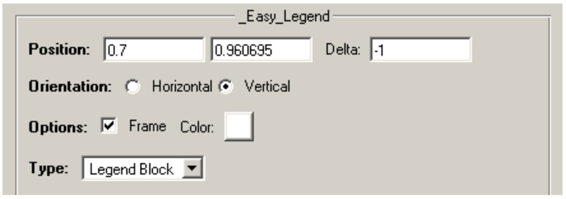

_Easy_Legend
Position: 0.7 0.960695 Delta: -1
Orientation: Horizontal [✓ Vertical]
Options: [✓ Frame] Color:
Type: Legend Block

Figure 10.14: Editable legend parameters.

There are three types of legends possible: A *Legend Block* displays an entry for each curve together with a short line depicting the appearance of the curves. A *Name Block* is just a list of the curve names. The position and orientation for both types can be manipulated easily in the editor window. The position denotes the coordinates of the first legend item. Positions of legends are always normalized coordinates. The delta parameter applies only to the vertical (y) coordinate. You can also position a legend in the plot window interactivly by pressing the left mouse button on one of the names in the legend, move the mouse and release it at the desired position. The third legend type (*Name List*) displays a list of curve names, too. But in contrast to the latter type you can move around every name separately in the same way like annotations (see above).

10.10.2.4 Editing markerline parameters

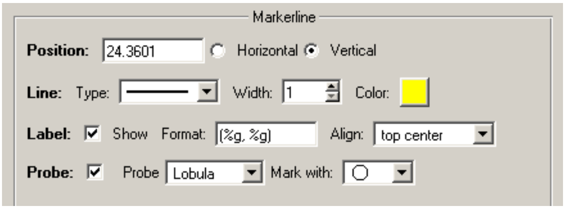

Markerline
Position: 24.3601 Horizontal [✓ Vertical]
Line: Type: ——— Width: 1 Color: 🟨
Label: [✓ Show] Format: (%g, %g) Align: top center
Probe: [✓ Probe] Lobula Mark with: ○

Figure 10.15: Editable markerline parameters.

The position of a markerline has to be given in world coordinates. Also be sure that the markerline is inserted after the appropriate axis. You can shift a markerline horizontally resp. vertically in the plot window by pressing the left mouse button on the markerline, move the mouse and release it at the new position. Markerlines can be used to probe those curves which belong to the same group (PlotArea) as the markerline. You can display the value where the markerline intersects the selected curve the first time. This intersection can also be indicated with a marker symbol.

10.10.2.5 Editing lattice parameters

A lattice is a two-dimensional field which is rendered as an image by default. If both dimensions are less or equal than 32 a lattice can also be rendered as colored dots in a grid (*Gridded*) or as dots of different sizes (*Spot*) to represent the values. For lattices rendered as images a gamma value can be set to enhance the image. Lattices have a colormap (Default: black to white) which can be set through the appropriate amira modules. Global colormaps are not supported within the amira context.

10.10.2.6 Editing colormap parameters

Colormaps have only a few editable parameters: The minimum and maximum values can be given or taken from the axis if there is one. Furthermore the colormap can be reversed. The colors itself can only be changed with the amira colormap editor.

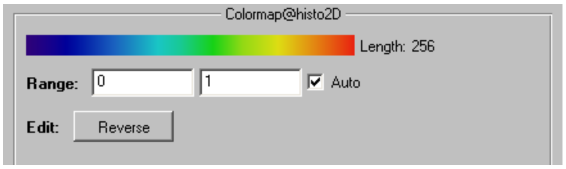


Figure 10.17: Editable colormap parameters.

10.10.2.7 Editing (analytical) curve parameters

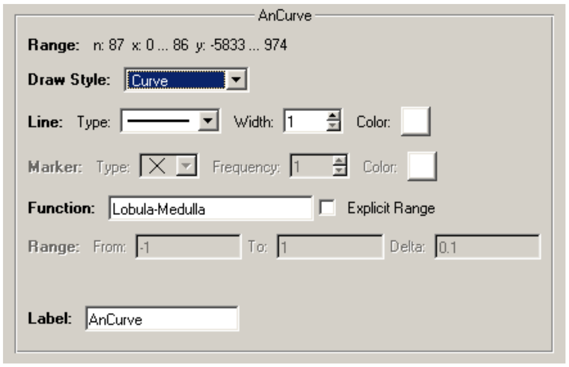

AnCurve
Range: n: 87 x: 0 ... 86 y: -5833 ... 974
Draw Style: Curve
Line: Type: Width: 1 Color:
Marker: Type: Frequency: 1 Color:
Function: Lobula-Medulla Explicit Range
Range: From: -1 To: 1 Delta: 0.1
Label: AnCurve

Figure 10.18: Editable (analytical) curve parameters.

Besides the more or less self-describing parameters of both curve and analytical curve (AnCurve) parameters there are a few things to mention regarding AnCurves only. The *Function* text widget contains the formula which is computed on every plot update. The default formula is *x* which produces a transversal line where x runs from -1 to 1. It is possible to use the name of another curve as a variable in the function. In this case the x-values of the first curve in the formula are taken to evaluate the y-values of all variables in use. These y-values are then used for the computation. You can restrict the range of the computation by activating the *Explicit Range* toggle and entering the appropriate range values. There are a lot of built-in functions such as sin, cos, tan, sqrt, exp, ... which can be used in formulas.

10.10.3 Working with plot objects

Furthermore you can copy or delete plot objects. e.g., it may be useful for comparison purposes to copy a curve before the data of the original curve will be changed. To do so you have to select the object in the list and then choose the *Edit->Copy* pulldown menu. After that choose *Paste* or *Append* in the menu and the object will be inserted at the current position (= position of the selected object) resp. the object will be placed behind the selected object.

With the *New* pulldown menu you can create new objects which will be inserted at the appropriate positions in the list of objects.

10.10.4 Printing

The *File* pulldown menu of the main plot window provides a *Snapshot* item where you can send the plot directly to the default printer or save it as an image file. It is also possible to generate a vectorbased PostScript file of the plot by using the `print` command (see below).

10.10.5 Saving data

The *Save Data...* item under the *File* menu lets you save the data of all curves in a file. The file dialog presents a list of formats suitable for saving the data.

10.10.5.1 Data formats

The *Amira-Plot-Format* is a propriatary format which should be used if you want to plot the data later on within the amira plot facilities. Use the *Gnuplot-Format* if you plan to use Gnuplot to plot the data outside of amira. If you need to import the plot data into a spreadsheet program like Microsofts's Excel use the *CSV-Format* (CSV = Comma Separated Values). The *Amira-Spreadsheet-Format* can be loaded and processed by Amira.

10.10.6 Saving the plot state

If you changed your plot setup a lot you can save this setup in a similar fashion like the *Save Network* functionality. I.e. first save the amira network and then save the plot state into a second file. To resume an amira session invoke amira and load the two script files (1. amira network, 2. Plot state). Note, that the size and the position of a plot window is saved automatically if it is opened by an amira module and the Save Network menu item is invoked.

Commands

In the amira environment Plot Tool commands have the following structure

`$thePlot` *command* [*parameters*]

or if the command applies to a plot object:

`$thePlot` *objectname command* [*parameters*]

The following general commands are available:

`getSize`
Returns the size of the plot window.

`setSize <width> <height>`
Sets the size of the plot window.

`getPosition`
Returns the size of the plot window.

`setPosition <x> <y>`
Sets the position of the plot window relative to the upper left corner of the screen.

`setBackgroundColor <r> <g> <b>`
This command sets the color of the background to a specific value. The color may be specified either as a triple of integer RGB values in the range 0...255, as a triple of rational RGB values in the range 0.0...1.0, or simply as plain text, e.g., white, where the list of allowed color names is defined in /usr/lib/X11/rgb.txt.

`hide`
Hides the plot window.

`show`
Shows the plot window if it is hidden.

`getObjects`
Displays a list of all plot objects currently in use.

`update`
Processes the plot object and updates the display.

`snapshot <filename>`
Takes a snapshot and saves it under the given name. The suffix of the filename determines the raster format used. Available formats are: TIFF (`.tif,.tiff`), SGI-RGB (`.rgb,.sgi,.bw`), JPEG (`.jpg,.jpeg`), PNM (`.pgm,.ppm`), BMP (`.bmp`), PNG (`.png`), and Encapsulated PostScript (`.eps`)

`print [options] <filename>`
Prints the plot window into a PostScript file (vector based) with the following options:
`-a4`: generates a4 landscape output (Default).
`-a5`: generates a5 portrait output.
`-auto`: switches autoscaling on (Default).
`-bw`: generates black and white output.
`-color`: generates color output (Default).
`-eps`: generates encapsulated PostScript.
`-fillbg`: fills the background according to the plot window.
`-frame`: draws a frame around the plot.
`-landscape`: prints in landscape format.
`-noauto`: switches autoscaling off.

`-portrait`: prints in portrait format.
`-windowsize`: generates output of window size.

`saveData <filename>`
Saves the data of all data based plot objects in a propriatary format.

`saveState <filename>`
Saves the current plot state into a script file useful to restore the plot setup in a future amira session.

`load <filename>`
Loads data from the filename and stores it in a curve plot object.

The following commands apply to plot objects:

`getMinMax`
Returns the minimum and maximum values of objects of type: *Curve, Cartesian Axis, Polaraxis*.

`setMinMax <minX> <maxX> <minY> <maxY>`
Sets the range of *Cartesian Axis, Polaraxis.*

`getArea`
Returns the plotting area of *PlotArea* objects.

`setArea <lowerleftX> <lowerleftY> <upperrightX> <upperrightY>`
Sets the plotting area of a *PlotArea* object.

`getXValues`
Returns the x-values of a *Curve*.

`getYValues`
Returns the y-values of a *Curve*.

`set[X|Y|Phi|R]TickValues <1.  tick> <2.  tick> ...  <n.  tick>`
Sets the position of the ticks along an axis. It is available for objects of type: Cartesian Axis, Polaraxis.

`set[X|Y|Phi|R]TickLabels <1.  label> <2.  label> ...  <n.  label>`
Sets the labels of the ticks along an axis. It is available for objects of type: Cartesian Axis, Polaraxis. If there are fewer labels than ticks the remaining ticks will be unlabelled.

10.11 Segmentation Editor

The *Segmentation Editor* is a tool for interactively segmenting 3D image data. Image segmentation is the process of dividing an image into different subregions (also called segments). In the bio-medical context these segments can be for example different organs or tissue types. In Amira segmentation is done by first selecting voxels and then assigning these voxels to a particular material. The labels are stored in a Label Field. The *Segmentation Editor* lets you edit such a label field. From the final label field polygonal surfaces can be reconstructed using the SurfaceGen module. The documentaton of the *Segmentation Editor* is separated into the following parts:

- Overview of the segmentation editor
- Manipulating the material list
- Working in 4 viewer mode
- Edit buttons
- Segmentation tools
- Selection filters
- Label filters
- Key bindings

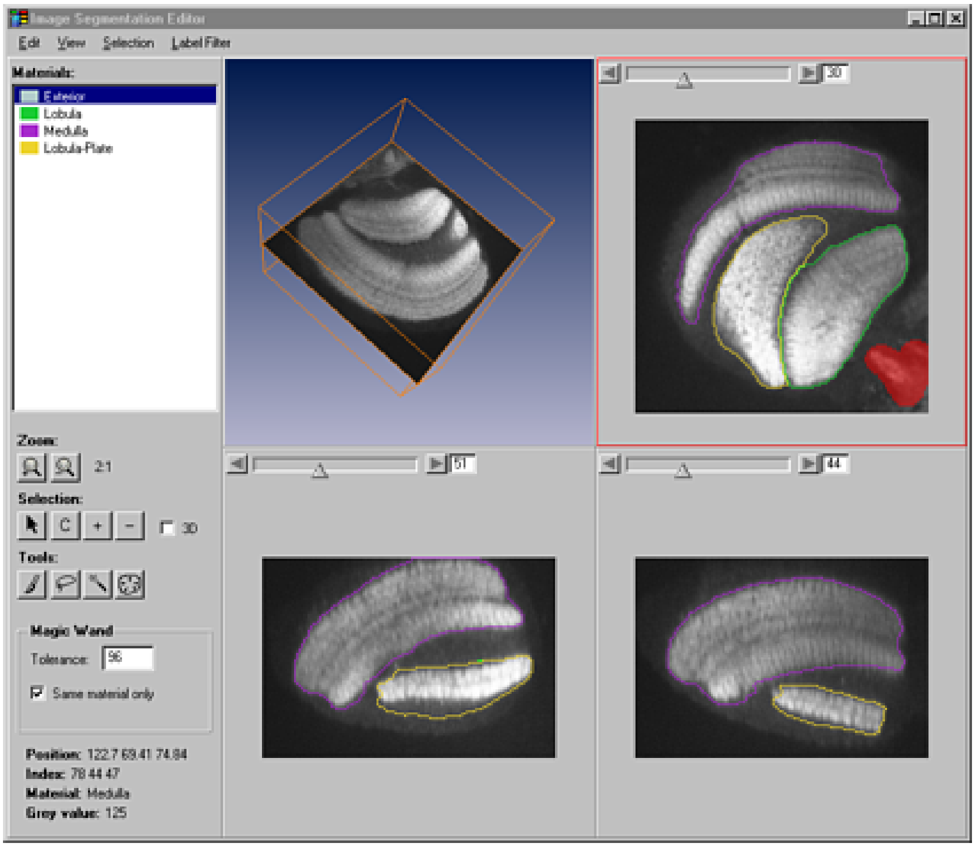
10.11.1 Overview of the segmentation editor

In order to activate the segmentation editor push the pencil-shaped edit button of a selected *Label Field*. Then an instance of the editor will be created and a new window as shown in Figure 10.19 pops up. The major parts of this new window are:

- **Material List:** In the upper left corner of the editor window a list of materials is presented. Here you can add, remove, and rename materials, and you can select the current material.
- **Tool box:** Below the material list several tools for interactive manipulation of the segmentation can be selected. Depending on the currently selected tool additional widgets show up in the options frame.
- **Info Area:** Below the tool box some basic information like the current cursor position or the material under the cursor is displayed.
- **Menu Bar:** From the menu bar additional tools and filters can be accessed. Menu entries with dots (...) behind the name open an additional dialog window.
- **Image Viewer(s):** The biggest part of the window is covered by one or four image viewers, displaying the labels and the current selection in differently oriented slices.

Basic principle of interactive segmentation

The basic idea of interaction in the segmentation editor is a to first *select* some voxels and then to *assign* them to the *active material*. The simplest way of selecting pixels is to simply draw with the mouse when the *Brush* or the *Lasso* tool is active (see below for details). Selected pixels are displayed in red.



Materials:
- Exterior
- Lobula
- Medulla
- Lobula-Plate

2:1

☐ 3D

Tolerance: 96

☑ Same material only

Position: 122.7 69.41 74.84
Index: 78 44 47
Material: Medulla
Grey value: 125

Figure 10.19: The Image Segmentation Editor.

To add selected pixels to the active material, click the ”+” button. The active material is the material, which is currently selected in the material list. New materials can be added using the right-mouse popup menu in the material list. Every pixel can only be assigned to one material.

Note that even with the advanced tools provided in Amira, image segmentation can be a time-consuming process! Due to limited main-memory and for performance reasons, there is only a limited undo space for 2D interaction and most 3D operations cannot be undone. Therefore it is highly recommended to *frequently save the label field* during the process of segmentation.

To get started it might be a good idea, to choose the image segmentation demo from the Users Guide's demo section and start by modifying an existing label field.

The menu bar

The menu bar of the segmentation editor window has four submenus. The buttons of the *Selection* and *Labels* are described in separate sections below.

- **Edit:** This menu provides an *undo* entry that undoes the last operation. Successive invocation of *undo* is possible, allowing you to undo several operations. Note that for memory and performance reasons, 3D operations can not be undone. Therefore it is highly recommended to frequently save the label field.
  The *Data window...* entry brings up an additional dialog, in which you can control the data window and the opacity of the selection.
- **View:** This menu lets you change the *Layout* of the image viewer area. You can choose between either one single viewer, or four viewers (three slice viewers, one 3D viewer). Details on these two arrangements are described below.
  In the single viewer layout, you can choose the *Orientation* of the viewer.
  The *Info line* toggle lets you switch off the on-line information displayed in the lower left part of the window.
- **Selection:** This menu provides a number of filters and operators, which modify the current selection. They are described below. Menu entries with dots (...) behind their name open an additional dialog.
- **LabelFilter:** These filters and operators directly modify the label field, and not the selection.

10.11.2 Manipulating the material list

In the material list the current material can be selected with the left mouse button. The current material is the material which is assigned to selected voxels if the "+" button is pressed. A special behavior is obtained if one of the *Shift*, *Control*, or *Alt* keys on the keyboard is held down, while selecting the material:

- If *Shift* is held down, all voxels which are already assigned to the selected material are added to the current selection (either in the current slice or in the whole volume, depending on the value of the 3D toggle described below).
- If *Control* is held down, all voxels assigned to the selected material will be deselected (either in the current slice or in the whole volume, depending on the value of the 3D toggle described below).
- Holding down the *Alt* key does the same as *Shift* except that the selection is cleared before new voxels are selected.

Additional options of the material list can be accessed via a popup menu, which shows up when the right mouse button is pressed on a particular material entry. The following menu entries are provided:

- **Draw Style:** A submenu offers different styles how pixels belonging to a particular material are rendered in the image viewer. The possible appearances are:

    - *Invisible:* The material is not visible.
    - *Contour:* The segments are enclosed by lines.
    - *Hatched:* The segments are enclosed by lines and shown hatched.
    - *Dotted:* The segments are enclosed by lines and filled with dots.
    - *Light Dots:* The segments are enclosed by lines and filled with less dots.
- **Locate:** This command sets all viewers to the slice with the largest region of this material. This is especially useful to navigate in large data sets with many materials.
- **Delete:** If you choose *Delete* the material you have clicked on will be deleted. Voxels belonging to that material will be assigned to the first material in the list (typically the background).
- **Rename:** With this option you can change the name of a material. A text cursor appears and lets you edit the name as easily as any other text field. If the new name is equal to the name of an existing material you are asked if the these two materials should be merged. It is not possible to have to different materials with the same name.
- **Edit color:** This command brings up a color editor in which you can easily change the color of this material.
- **Lock/Unlock Material:** You can lock a material with this option. A key will be displayed on the right hand side of each locked material. If a material is locked, no voxels can be subtracted from this region, neither explicitly, nor implicitly by adding them to another material.
- **New Material:** By selecting this option from the popup menu a new entry is added to the material list with a randomly chosen color and a default name, e.g., ”NewMaterial”. These properties can be changed as described above.

10.11.3 Working in 4 viewer mode

Initially, in the central part of the image segmentation editor the current slice with colored contours surrounding the different segments is shown. Near the top border you find a slider to control which slice of the 3D volume is currently displayed. The orientation of the slice can be changed via the menu entry *View Orientation* in the menu bar. In medical notation XZ denotes an axial slice, XZ a frontal slice, and YZ a sagittal slice.

In addition to the standard single viewer layout Amira also provides a *4 viewer* display. You can switch between the two layouts using the *View Layout* menu. If the *4 viewer* mode is active, three 2D viewers with different orientations and an additional 3D viewer are displayed. The orientation menu is disabled in this mode. You can arbitrarily switch between the viewers when segmenting images. One of the viewers is the active viewer. It is decorated with a red frame. A viewer can be made the active viewer by clicking on its title bar (where the slice slider is located), or into the viewer itself. In the latter case be careful when clicking into the image in 3D mode. If the *Pick* tool or the *Magic Wand* tool is active the resulting selection operation may take some time to compute. All 2D operations like the ”+” or ”-” are always performed in the active viewer, i.e., the slice and orientation of the active viewer will be considered.

The 3D viewer

In the *4 viewer* mode the upper-left viewer is a standard Amira 3D viewer. All modules will display in this viewer too (depending on the modules' viewer masks, as usual). In addition there is some special 3D support for segmentation, which is controlled by the buttons in the line just above the 3D viewer and by an additional dialog.

By default the current selection will be displayed in the 3D viewer as a point cloud. After each change of the current selection (i.e. after each brush stroke), the display will be updated. On smaller computers or for very large data sets, this may become too slow. In this case the *Auto* toggle can be switched off. Then the 3D scene will only be updated after explicitly hitting the *Update* button.

The *Draw* button allows to modify the selection in 3D. After hitting the button a curve can be drawn in the 3D viewer (similar to the lasso tool). Then all 3D points which ly inside the surrounded region will be cropped away (i.e. deselected). If the *Shift* key is pressed when starting to draw the contour the surrounded region will be selected. If the *Shift* and *Control* keys are pressed simultaneously when starting to draw the contour the inverse of the surrounded region will be selected. The draw mode can also be activated by pressing the *d* key in the viewer window.

Using the zoom buttons

The zoom buttons control the size of the image. An info line on the right hand side of the buttons shows the current zoom factor. For example a zoom of 2:1 means that 2 pixels on the screen correspond to one pixel of the original data set (magnification), while 1:4 would mean, that four pixel of the data set correspond to one pixel on the screen. If the pixel-size is not the same in all dimensions, the zoom factor belongs to the horizontal direction only, to maintain the correct aspect ratio of the voxels.

10.11.4 Edit buttons

The basic principle of the segmentation editor is to first select voxels and then to assign selected voxels to the current material. This assignment is done using the ”+” button. Besides the ”+” button there are some other buttons which can be used to perform edit operations. These are the following:

- **3D toggle:** If the 3D toggle is activated other edit operations or tools operate in 3D mode. For example, the *Clear* button clears the whole 3D selection, not only the current slice. Currently, in 3D mode no undo is available. Therefore, be careful when activating the 3D toggle. Usually, it is advisable to disable the toggle again after completing a particular 3D operation.
- **Picker:** This tool can be used to mark a connected area belonging to one material. It is described in more detail in the tools section.
- **Clear (C):** This button lets you clear the current selection. If the 3D toggle is activated the selection is cleared in all slices.
- **Replace (R):** This button tries to replace a selected region. Assume you want to modify the contours of some region. As long as the new contours fully enclose the previous ones this can be easily achieved using the ”+” button. However, if the new region is smaller things are

more complicated. The replace button looks for connected regions under the current selection belonging to the current material. Pixels which belong to such a region but which are not selected are automatically assigned to some other neighbouring material. Selected pixels are assigned to the current material. In this way a replacement is performed.

- **Add (+):** This button adds all selected voxels to the material currently selected in the material list. Voxels assigned to a locked material are not affected. If the *Shift* key is held down while clicking the button, the voxels remain selected. Otherwise the selection is cleared afterwards.
- **Subtract (-):** This button lets you subtract all selected voxels belonging to the current material from that material, provided the current material is not locked. The pixels are automatically assigned to some neighbouring material which is interpreted as a local background.

10.11.5 Segmentation tools

Several segmentation tools are provided allowing you to select areas in the current slice for subsequent edit operations. A tool is activated by clicking on its icon with the left mouse button. On default, selected areas are drawn in transparent red color. The opacity of the red color can be modified using a dialog under the *Edit Data Window...* menu. Alternatively, selected areas can be visualized using the same draw styles as ordinary labels. The different draw styles are listed under the *View Selection* submenu. However, in most cases the default transparent display is best suited, because then selected areas cannot be mistaken for labeled regions.

Once a segmentation tool is activated certain actions can be performed in one of the viewer windows using the mouse. The operation of most of the tools can be modified by pressing the *Ctrl* or the *Shift* key. *Ctrl* typically deselects voxels instead of selecting them, while *Shift* adds voxels to the selection instead of replacing it. The description of the individual tools mentions more details.

Pick & Move

This tools does two things. First, it lets you select a connected region assigned to one particular material by clicking onto an image voxel with the left mouse button. If the *select all* toggle is active all voxels belonging to the same material as the clicked voxel will be selected, either in 3D or in the current slice only, depending on the value of the 3D toggle. If the *Ctrl* key is held down voxels will be deselected.

If the mouse is over an already selected pixel the mouse cursor takes on the shape of a hand. Then the tool lets you translate the current selection. By pressing down the *Shift* key the current selection can also be rotated. However, note that this might give strange results for pixels with an aspect ratio different from one (for example in frontal or sagittal slices in a label field with stacked coordinates).

Brush

If this tool is activated, you can select regions by painting voxels with the left mouse held down. The size of the brush can be modified via a slider in the tool's control panel. For common smaller sizes dedicated push buttons are provided. Note that the size is specified in screen pixels, not in image pixels. If the center of an image pixel is covered by the brush that pixel is selected. If the *Ctrl* key is held down or if the middle mouse button is used, pixels are deselected instead of being selected.

The right mouse button is a flood fill tool. For example, you may surround a region in the image by drawing along its border and then click into the middle of the currently unselected interior part, to fill it. Again the *Ctrl* key inverts the operation, i.e., it deselects connected parts of the selection.

Lasso

The *Lasso* lets you define an area by generating a closed contour curve. Usually, you draw such a curve freehand with the the left mouse button pressed, but it is also possible to fence an area with line segments by pressing **Alt**, and clicking successively to points with the left mouse button (holding down the *Alt* key all the time). Successive points will be connected with straight line segments. To finish interaction release the *Alt* key and click a last time. The contour is then closed and filled automatically.

When the *auto trace* option is enabled (this is the default) the line segments are automatically fitted to image edges. Click with the left mouse button to define a starting point. Dragging the mouse after releasing the button a contour curve automatically fitted to the closest image edge is drawn. Further mouse clicks mark fixed points on the contour. Contour parts drawn before the latest mouse click remain in place. Close the contour by clicking with the middle mouse button.

The underlying algorithm called *intelligent scissors* works by finding the shortest path in a cost matrix which is computed from the image's gradient image. If the option *trace edges* is switched off then the image itself is interpreted as a cost matrix. Disabling this option is useful for images where object boundaries already appear as isolated bright lines.

Magic Wand

This tool performs a so-called region growing either in 2D or in 3D, depending on whether the 3D toggle is activated or not. Clicking with the left mouse button on a voxel selects the largest connected area that contains the voxel itself and all voxels with gray values lying inside a user-defined range. The range can be specified via two spin boxes shown in the tool's control panel. The values of the spin boxes either define absolute gray values or values relative to the gray value of the seed pixel, depending on whether the *absolute values* toggle is activated or not. If necessary, the range will be automatically modified so that it the contains the seed voxel. It is possible to modify the range even after the seed voxel has been selected. For convenience, the lower threshold can be quickly modified by clicking the middle mouse button (or shift-clicking the right mouse button) and moving the mouse horizontally. Likewise, the upper threshold can be quickly modified by clicking the right mouse button and moving the mouse horizontally (*virtual slider mode*). However, note that in 3D mode the selection is not updated on-the-fly while modifying the thresholds. Instead, it is updated once after releasing the mouse.

The *same material only* toggle restricts the search to voxels which are assigned to the same material as the seed voxel. For example, if the seed voxel belongs to *Exterior*, voxels which are already assigned to different materials will not be selected. This is useful to restrict the selection to certain parts of the image. Paths out of a region of interest might be obstructed using a blocker region.

If the *fill interior* toggle is set, holes inside the selected region will be filled automatically. This has the same effect as pressing the *f* key afterwards, which also fills the current selection. Automatic filling is only done in 2D mode, but not in 3D mode.

Finally, the *draw limit line* button lets you specify additional barriers for 2D region growing. After pressing this button you can draw arbitrary polylines in the viewer window. Voxels covered by such a limit line will not be considered for region growing. Instead of pressing the limit line button you can also press the *Ctrl* key to temporarily enter the limit line mode. Existing limit lines can be deleted by clicking on them with the *Ctrl* key being pressed.

Note, that all seed points together with the corresponding range values and limit lines are stored in the parameter section of the label field. This makes it possible to easily correct a selection at a later time.

PropagatingContour

This tool is a fast Active Contour. From a number of specified seed points a boundary front evolves. It computes the position of the front at all times up to the stop time. A slider and a text box allow the user select the appropriate front after pressing the initialization button. There are three buttons: *Menu* pops up the parameter menu, *Clear* removes the current seed points (hot-key *T*), and *Init* starts the computation (hot-key *B*).

Blowtool

By clicking with the left mouse button on a voxel and dragging the mouse without releasing the button an initially circle-formed contour blows up. The greater the distance to the initial position of the mouse click gets the more the contour grows. The contour is designed to grow in areas with homogeneous grey values and to stop where grey values change abruptly, i.e. at image edges.

The tolerance slider in the *option panel* controls how sharp the image edge has to be in order to stop the contour from growing at each contour point. The smaller the tolerance the sharper the image edge has to be. If the tolerance is large, the contour will even stop at weak edges. After releasing the mouse button the area within the contour will be marked.

Before the computation the image is smoothed using a Gaussian smoothing filter. You can change the width of the filter within ranges of 1 (no smoothing) to 8 (very broad smoothing). The default value is 4.

Crosshair

The crosshair tool is only active in 4 viewer mode. It simply displays a crosshair in all three orthogonal viewers. The colors denote the different directions (red=x-axis, green=y-axis, blue=z-axis). Clicking into one of the viewers moves the crosshair and simultaneously updates the two other viewers so that the crosshair's center is visible in all three viewers.

10.11.6 Selection Filters

Under the *Selection* button of the menu bar several filters for modifying the current selection are provided. These include simple morphological operations, smoothing filters, and interpolation tools.

- **Grow:** This filter performs a morphological dilatation of the current selection, i.e., the selection is made bigger by one pixel in every direction. The filter can be applied to the current slice only (short cut is *Ctrl* +), to all slices (orientation is defined by the active viewer), or to the whole volume.

- **Shrink:** This filter performs a morphological erosion of the current selection, i.e., the selection is made smaller by one pixel in every direction. The filter can be applied to the current slice only (short cut is *Ctrl* -), to all slices (orientation is defined by the active viewer), or to the whole volume.

- **Fill:** This filter fills the current selection, i.e., it closes all holes in it. The filter can be applied to the current slice only (short cut is key *F*), or to all slices (orientation is defined by the active viewer).

- **Smooth:** This filter smooths the current selection. Smoothing works by applying a Gauss filter to the binary selection image and then reselecting all pixels with an intensity bigger than 0.5. Usually the filter is applied to a selection computed by the threshold filter or by the magic wand tool. The short cut for this filter is *Ctrl M*.

- **Invert:** This filter inverts the current selection, i.e., selected voxels are deselected, while unselected voxels are selected. The filter can be applied in the current slice only (short cut is key *I*), or in the whole volume.

- **Active Contour...:** This filter belongs to a family of powerful tools for semi-automatic segmentation called Active Contour. From a user defined initial selection the contour propagates automatically according to 'forces' within the image. Use the STOP button of the amira main window to stop the process when the desired selection has been computed.

- **Snakes...:** This filter automatically fits the contour of a selected region to the edges of the image. Edges are regions with a high contrast.

  An additional dialog is opened, which allows you to copy the selection from the current slice into the next or previous slice (up and down arrows). After copying the snakes algorithm tries to adjust the contour to the grey values in the new slice.

  Clicking on *Adjust* runs the same algorithm on the current selection in the current slice. This is especially useful to readjust after a manual correction.

  If *Extrapolate* is switched on, the selection is not only copied to the next slice but extrapolated by taking into account selections in the two previous slices.

  If *3DInfo* is switched on, a modified snakes algorithm is used which gives better results in the case of relatively equal voxel sizes in all three directions.

Relax smooths the contour of the current selection.

- **Threshold...:** Select voxels by threshold segmentation. In the additional dialog, a minimal and a maximal image intensity can be specified. By clicking *Apply*, all voxels falling into the interval are selected. If the *current material only* option is activated, only voxels assigned to the current material are selected. If the *3D* option is active, the operation is performed in the whole volume, otherwise it is done in the current slice of the active viewer.
- **Make Ellipse:** Pressing this button causes the selected area to be replaced by an ellipse that fits the original selection as good as possible. The filter operates in the current slice of the active viewer.
- **Interpol:** This button interpolates the selection between all slices where areas have been selected manually. Pressing the button again after performing corrections in certain slices will interpolate the selections between all slices where changes have been made. For example, suppose you have selected an object in slice 1 and in slice 10. Pressing the *Interpolate* button automatically computes a selection in slices 2...9. You may then modify the selection in slice 5, e.g., using the *Brush* tool. Pressing *Interpolate* again now interpolates the selection in slices 2...4 and 6...9. The short cut for this filter is *Ctrl I*.
- **Wrap:** This filter also interpolates the selection. However, a different algorithm based on scattered data interpolation with radial basis functions is applied. The filter always operates in 3D. In contrast to the previous filter it is possible to start from slices with different orientations, e.g., after selecting slices in each spatial direction (xy, xz, yz) this tool can be used to compute an interpolating selection which wraps up the object whose skeleton is given by the slices.

  Note: Initially no two successive slices with the same orientation must be labeled. Otherwise, an error message is displayed. The tool may take a few seconds to complete its operation. The short cut for this filter is *Ctrl W*.

10.11.6.1 Active Contours

Active contour models are powerful tools for semi-automatic segmentation. An initial contour moves automatically according to image forces such that it separates two different materials.

The movement is governed by parameters that the user must adjust to suit his image data.

There exist two different Active Contour tools. The ”Propagating Contour” tool computes the shape of the contour at all times up to the *Stop Time* value. The contour starts evolving from one or more seed points and stops at edges found in the image. After the computation the user can choose any time parameter in the range the separates the materials best.

The Active Contour filter takes any initial shape (which may only be a single seed point) and expands this shape until a material boundary is found, where the contour stops. Additionally, this tool takes into account the curvature to generate smoother shapes and avoid leaking through small wholes.

The velocity of the active contour is computed from the original image data as well as from the gradient image. While the image data provides the image intensity values, the gradient image contains the edge information. In the *View* menu of the Segmentation Editor, the user can choose a gradient image. By default, the Segmentation Editor computes a gradient image from the currently chosen gray image. Alternatively, the user can provide the gradient image and select it in the *View* menu.

If the active contour does not advance, you have to change the parameters. In particular, lower the values for edge and image sensitivity.

Note that often the results can be improved by preprocessing the image data, for instance by pre-smoothing. Furthermore, better results are achieved for isotropic data, i.e., large slice distances can be disadvantageous.

Ports

StopTime

The *Stop Time* limits the movement of the active contour to a certain range. If the region has not been fully segmented, this parameter must be increased.

Edge Sensitivity

The *Edge Sensitivity* value governs the dependence on the the gradient image,i.e., when the contour stops. If this value is too small, the algorithm might run over edges. If it is too large, there might be no movement at all. If the contour does not move at all, this value should be decreased.

Image Intensity

The advancement of the contour can also depend on the image intensity (grey value). This parameter controls how strong the dependence is. The speed function is multiplied by a Gaussian function. The width of this Gaussian is the reciprocal of the value of this port (such that a value of 0 means no influence at all) and the mean intensity value if computed as the average over all seed point or over the average of the selection, respectively. Hence, the more different the image intensity of a voxel is from the starting points or region, the slower the progress is.

Curvature

The *curvature* parameter controls the regularity of the contour. A value close to 1 leads to a smooth contour. Set to 1, the algorithm works as a filter that refines an existing selection in order to snap to edges in the image. If this parameter is set to 0, the algorithm is merely expanding until stopped by an edge without enforcing any smoothness. Setting the curvature to a value different from 1 can be used to make a small seed selection expand until stopped by an edge. The *curvature* regularization is not available for the propagating contour tool.

Attractor

The *attractor* force parameter specifies the strength of the edge attraction. It allows the tool to be used as a filter that refines an initial approximation because it attracts the contour to the nearest edges in the image. Mostly, a value of 1 works best. To switch off the edge attraction, set the value to 0.

Practical hints:

- In all cases, apply these tools to smoothed data only. No smoothing is built in to allow the user to choose the smoothing filter. For instance, Gaussian smoothing is an appropriate preprocessing.
- It is recommendable to use the fast *PropagatingContour Tool* first and then refine the selection with the *ActiveContour Filter*.
- In cases where not all the targeted region is selected by the active contour or it leaks out before, the user can proceed step by step. First select a partial region and then segment the remaining region starting from another seed point or selection.
- At the beginning, certainly a bit of probing is necessary until you find the best set of parameters for your kind of data.

10.11.7 Label Filters

Under the *Labels* button of the menu bar several filters for modifying the current labelling are provided. These filters directly operate on the labels. They do not take the current selection into account.

- **Fill holes:** This filter removes islands of arbitrary size in the *current material*, i.e., all pixels completely surrounded by the current material are also assigned to this material (short cut is *Shift-H* for the current slice). Suppose you have segmented a CT image using thresholding as provided by the LabelVoxel module. You then want to assign the dark marrow of the bones to the material *Bone* as well. In order to do this, select *Bone* in the material list so that it becomes the current material and then call the *fill holes* filter. The filter can be applied to the current slice in the active viewer, or to all slices.

- **Remove Islands...:** This filter removes islands in or between regions, depending on the *size* and *percentage* values that are set in the filters dialog. An island is a connected area of voxels containing a number of voxels less than or equal to the *size* value specified in the dialog.

  If an island is encountered the percentages of the surrounding regions with respect to the total number of voxels adjacent to the island are calculated and compared with the *percentage* threshold. The island is assigned to the region with the highest percentage value greater than or equal to *percentage*. If no region exceeds the *percentage* threshold, the island remains untouched. Note that the *percentage* value must be between 0.0 and 1.0.

  With the *mode* buttons you can control whether the filter works in the current *slice*, in *all slices* or in the 3D volume. The difference between the two latter cases is, that in 3D mode the filter searches for true 3D connected regions. Note that a long thin structure, like a blood vessel can be a very small region in each slice, but has a rather large volume in 3D.

This filter is invoked by pressing the *Remove* button. Pressing the *Select* button performs the same computations as before. However, the islands are not yet removed but are only selected. This is useful in order to preview the effect of the filter and to find optimal size and percentage values.

- **Smooth...:** This is a modified Gauss filter, which smoothes the region boundaries, and therefore slightly changes the labeling. In *3D* mode, additionally probabilities of voxels belonging to regions are computed and assigned to voxels. The probability is one for voxels in the interior of a region and decreases towards the region boundary. Voxels near region boundaries are also assigned probabilities with respect to neighboring regions. The probability information is used in other modules - in particular in the SurfaceGen module - in order to make region boundaries appear smooth, otherwise these may be displayed as zigzag lines.

  The *size* option allows you to enter the size of the filter mask which is a square with *size* x *size* pixels. The smoothing effect increases with the mask size, but computation time is higher for bigger masks.

  The *2D* button lets you start the smoothing operation on the present slice, the *3D* button lets you start a 3D smoothing on the whole data volume.

- **Remove Skin...:** This filter is only active if the Tcl-variable `giRemoveSkinFilter` is set to 1. Its purpose is to remove thin prolate regions of muscle or fat (skin) on the boundary to exterior.

  The filter removes the currently chosen material on the boundary to exterior. No other material than muscle or fat can be removed.

  The *Layers* option is for entering the number of layers (in units of pixels) outside of exterior (i.e. inside the body) where the filter looks for possible appearances of the chosen material. The larger the number of layers the more of the skin region will be removed.

  The filter can be applied to the current slice in the active viewer, or to all slices.

10.11.8 Key bindings

Important tools of the image segmentation editor can be accessed via hot keys. These hot keys are summarized below:

- **Changing the current slice**

  Space or **CursorDown** - Go to next slice
  Backspace or **CursorUp** - Go to previous slice
  PageDown - Go forth over the next five slices
  PageUp - Go back over the previous five slices
  Home - Go to the first slice
  End - Go to the last slice

  By holding down the *Shift* key while pressing one of these keys the current selection is copied to the target slice. By holding down the *Control* key, the user can move only between slices that contain a selection.

- **Selection**

  A or **+** - Add the selection to the current material
  S or **-** - Subtract the selection from the current material
  R - Replace current material under the selection
  C - Clear the selection
  E - Extrapolate the selection
  F - Fill the selection
  I - Invert the selection
  Ctrl-+ - Grow selection in current slice
  Ctrl– - Shrink selection in current slice
  Ctrl-M - Smooth selection in current slice
  Ctrl-I - Interpolate selection between multiple slices
  Ctrl-W - Wrap selection using radial basis functions
  Shift-S - Save label field to disk
  RETURN - Redraw .

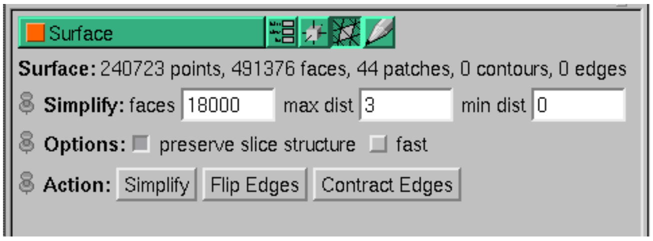
- **Tools**

  1 - Brush tool
  2 - Lasso tool
  3 - Magic Wand tool (region growing)
  4 - Blow tool
  5 - Crosshair tool
  6 - Pick, move, and rotate tool

- **Others U** - Undo
  Q - Toggle 3D button
  V - Toggle 1 and 4 viewer mode
  . - Select next material in the list
  , - Select previous material in the list
  Z - Decrease zoom factor
  Shift-Z - Increase zoom factor
  D - Toggle draw styles for all materials
  Shift-D - Toggle draw style for region under cursor

10.12 Simplification Editor

This editor can be used to reduce the number of triangles of a surface. In particular, the output of the SurfaceGen module has to be processed in this way before a tetrahedral patient model can be generated. Surface simplification is done by means of an edge collapsing algorithm. Edges of the original surface are sucessively reduced to points. The shape of the original surface is preserved by minimizing a certain error criterium. Special care is taken to prevent the triangles of the simplified surface from intersecting each other. However, in some cases intersections can still occur. Therefore, the resulting surface should be checked for intersections using the surface editor.



Surface: 240723 points, 491376 faces, 44 patches, 0 contours, 0 edges

Simplify: faces 18000 max dist 3 min dist 0

Options: preserve slice structure fast

Figure 10.20: User interface of surface simplifi cation editor.

Ports

Simplify This port provides three text fields for controlling some parameters of the simplification process. Field *faces* determines the desired number of triangles of the simplified surface. You may simplify the surface in multiple steps. However, somewhat more accurate results are obtained if the simplification is performed in a single step. When the simplification process is finished a check is made whether the surface contains duplicated triangles. Such triangles are removed automatically. Therefore the total number of triangles of the reduced surface will usually be somewhat less than the value specified in *faces*.

The second field *max dist* defines a maximum edge length for the triangles of the simplified surface. Usually, the default value of 3 cm should be suitable, but you can try to modify this value if the reduced surface still contains intersections.

The third field *min dist* defines a minimum edge length. Shorter edges are contracted if button *Contract Edges* is pressed (see below).

Options If the toggle *preserve slice structure* is set, edges of *exterior* triangles of the surface are treated in a special way, so that the slice structure of the original voxel grid is preserved.

If toggle *fast* is set, a less extensive intersection test strategy is selected. Surface simplification will run faster, but there is a higher probability that intersecting triangles will occur (see explanation of command *setIntersectionTestStrategy* below).

Action Button *Simplify* starts surface simplification. The simplification process will take several minutes. You may interrupt it by hitting the stop button of the progress bar.

Button *Flip Edges* automatically flips some edges of a surface if this improves the triangle quality. The Quality is defined as the ratio of the circumscribed circle and the inscribed circle of a triangle. The smaller this ratio the higher the quality. Again, duplicated triangles are removed automatically after this operation.

Button *Contract Edges* contracts all edges shorter than the value defined at field *min dist* (see above).

Commands

`Simplifier setRadiusRatio <value>`
This command defines a quality threshold for the edge flipping algorithm. Edges are flipped only if the radius ratio of a triangle exceeds the given value. By default a radius ratio of 20 is used.

`Simplifier smooth <nSteps> <lambda>`
This commands shifts the vertices of the surface so that it gets smoother. The value for *lambda* should be in the range of 0...1. This option is experimental and should not be used for routine work.

`Simplifier setIntersectionTestStrategy <mode>`
This command lets you choose between different intersection test strategies. *mode* is a three-digit number, for example 100. The digits specifiy the strategy used for testing modified triangles against existing edges, for testing modified edges against existing triangles, and for testing planar intersections. The allowed values are 0-3 for the first, 0-2 for the second, and 0-1 for the last digit. Larger values correspond to more extensive tests which of course also require more computing time. By default a value of 211 is used. Modes 111 or 101 are faster but more likely to produce intersecting triangles.

`Simplifier setFactError <value>`
In surface simplification the maximal edge length and the maximal error (estimated distance between original and simplified surface) can be controlled separately. The maximal error is computed by multiplying the maximal edge length by a certain factor *factError*. The value of *factError* can be set using command *setFactError*. The default value is 1.

`Simplifier getFactError`
This command returns the current value of *factError*.

10.13 Surface Editor

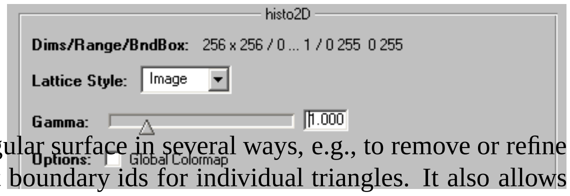

histo2D
Dims/Range/BndBox: 256 x 256 / 0 ... 1 / 0 255 0 255
Lattice Style: Image
Gamma: 1.000
Options: Global Colormap

Figure 10.16: Editable lattice parameters

The surface editor allows you to modify a triangular surface in several ways, e.g., to remove or refine triangles, to flip edges or move points, or to set boundary ids for individual triangles. It also allows you to check a surface for intersecting triangles or for falsely oriented patches.

In contrast to other editors in amira the surface editor places its GUI controls directly into the main 3D viewer window. This makes interactive surface editing very comfortable. In particular, the editor provides a central menu bar as well as several tool bars. The user interface of the editor is depicted in Figure 10.21. Note that, only one surface editor can be active at a time. Activating a second editor causes the first one to be closed.

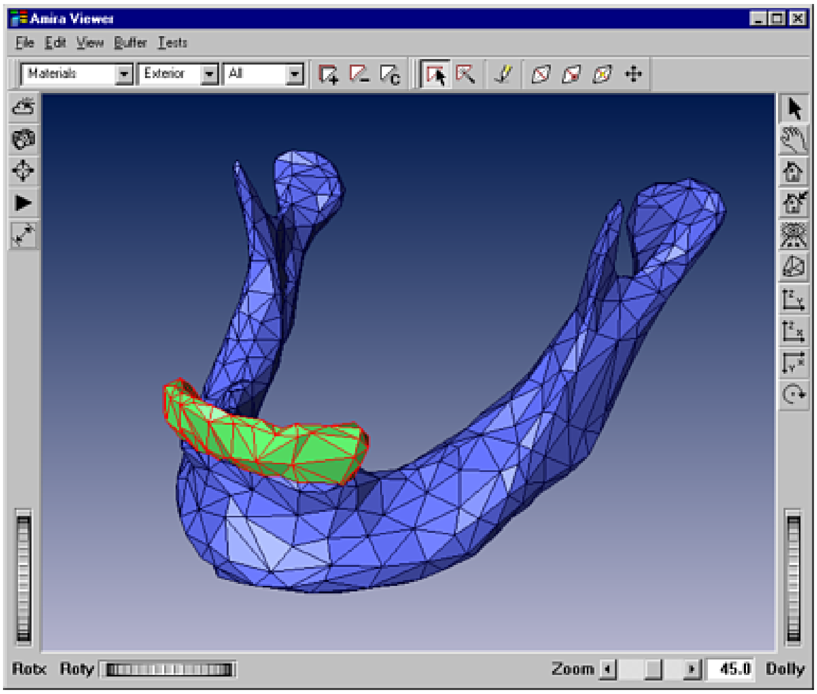


Figure 10.21: User interface of the surface editor.

The surface editor works in conjunction with a SurfaceView module. If such a module is not already connected to the surface, an instance of it will be created automatically when the editor is activated. However, most settings of the *SurfaceView* module can be controlled directly via menu entries of the editor itself. Therefore there is no need to have the *SurfaceView* module permanently selected.

A basic concept of the surface editor are the notions of *visible* and *highlighted triangles* respectively. Highlighted triangles are drawn in red wireframe. Many tools operate not on the surface as a whole, but only on highlighted triangles. For example, if you want to refine parts of the surface you'll first have to highlight the involved triangles before choosing *Refine faces* from the *Edit* menu. Besides the highlight buffer there is another buffer determining which triangles will actually be drawn. This allows you to cut away parts of a complex surface. Highlighted triangles can be added to the visible buffer or removed from it, just as in an ordinary *SurfaceView* module. Note that hiding a triangle doesn't mean to delete it.

Triangles can be highlighted in a number of ways. On the one hand there are interactive *tools* allowing you to select triangles using some kind of mouse interaction. For example, triangles can be picked individually or a contour can be drawn in the 3D viewer. Buttons representing the different mouse tools are listed in a tool bar. Clicking on a button causes the corresponding tool to be activated. On the other hand triangles can be highlighted using automatic *selectors*. Selectors are listed in the leftmost combo box of the surface editor's selector tool bar. There are selectors for highlighting triangles depending on their inner and outer region, for highlighting triangles with a certain boundary id or belonging to certain patch, and many more.

In the following different elements of the surface editor will be described in detail, namely

- menu entries
- selectors including surface tests
- mouse tools for selecting triangles and editing the surface

10.13.1 Menu Entries

This section describes the entries of the surface editor's main menu which is displayed inside the 3D viewer window while the editor is active.

10.13.1.1 File Menu

- *Save*: Saves the surface under the same file name as it has been saved under before. This option allows you to quickly save intermediate results during complex edit tasks.
- *Save As...*: Pops up the file dialog and saves the surface under a user-chosen file name.
- *Close*: Terminates the surface editor.

10.13.1.2 Edit Menu

- *Undo*: Undoes the last edit operation (edge flip, edge collapse, edge bisection, vertex movement). Selection and highlight changes can be undone as well. The size of the undo buffer is limited to up to 100 entries.
- *Delete highlighted faces*: Permanently removes the highlighted triangles from the surface.
- *Remove coplanar faces*: Permanently removes coplanar triangles, i.e., triangles with the same three vertices, from the surface. The inner region and outer region id of the remaining triangle is updated appropriately.
- *Recompute connectivity*: Recomputes the surface's connectivity information, i.e., contours and branching points. Connectivity information is optional. If it is present, contours can be displayed using the ContourView module. In addition, *recompute connectivity* causes unused points of the surface to be deleted.
- *Refine faces*: Subdivides all highlighted triangles by bisecting their edges. Each refined triangle is replaced by four new ones. In addition, adjacent non-highlighted triangles are split in two in order to avoid dangling nodes.
- *Flip edges...*: Pops up a dialog allowing you to flip certain edges inside the highlighted part of the surface. The edge flips are performed in order to improve the aspect ratio of the involved triangles. An edge flip is only attempted if the aspect ratio of one of the two involved triangles is worse than a user-specified value.
- *Reorder patches...*: Pops up a dialog allowing you to rearrange the internal order of the surface patches. Two different patches can be selected and then swapped. This operation is useful for certain applications where two different surfaces are required to have the same patch structure.
- *Set boundary ids...*: Pops up a dialog allowing you to edit the surface's boundary ids. The dialog permits you to define new boundary ids or to change existing ones (including the preferred colors). Highlighted triangles can be assigned a new boundary id using the dialog's *set* button.
- *Magic wand settings...*: This menu entry pops up a dialog alowing you to change parameters of the magic wand tool. Details for that are described below.

10.13.1.3 View Menu

This menu allows you to change certain settings of the SurfaceView module used by the surface editor. The settings affect the draw style and the color mode used for rendering the surface. Possible draw styles are *outlined*, *shaded*, *lines*, *points*, and *transparent*. Possible color modes are *normal*, *mixed*, *twisted*, and *boundary ids*. In the first three modes colors are chosen according to the material ids assigned to the surface's patches. In the last mode colors are chosen according to the triangles' boundary ids.

10.13.1.4 Buffer Menu

This menu contains several entries for modifying the editor's highlight and view buffers. The buffers determine whether a triangle is highlighted or whether it is visible in a normal fashion. Both buffers are independent from each other. They can be stored in an internal backup buffer using the menu entries *copy highlights* and *copy buffer* respectively. Once one of the buffers has been copied it can be restored using *paste highlights* or *paste buffer*.

10.13.1.5 Tests Menu

This menu lists certain test operations which can be performed in order to check the quality and consistency of the surface. Internally, the tests are considered as *selectors* because they cause certain triangles of the surface to be highlighted. Therefore, the test are described in detail in the selectors section.

10.13.2 Selectors

This section describes tools for automatically highlighting certain parts of the surface. The selectors are listed in a combo box on the very left just beneath the main menu. Most selectors provide some additional controls which are displayed right beside this combo box.

- *Materials*: Just as in a SurfaceView module this selector allows you to highlight parts of the surface separating two particular regions from each other. The two regions are specified in two additional combo boxes which are shown once the selector is activated.

- *Boundary ids*: Highlights all triangles with a particular boundary id. The boundary ids defined in the surface are listed in a separate combo box. Boundary ids can be modified using the *Set boundary ids...* option of the *Edit* menu.

- *Patches*: Highlights all triangles belonging to a particular patch of the surface. The different patches can be selected using a slider displayed right beside the selector combo box. A patch is a group of triangles separating the same two regions and having the same boundary id.

- *Intersections*: Performs an intersection test and highlights intersecting triangles. The *compute* button actually initiates the test, while the *back* and *forward* buttons allow you to cycle through the list of intersecting faces. The total number of intersections is printed in the console window. Intersections can be repaired using the edit tools described below (edge flip, edge collapse, edge bisection, vertex movement).

- *Orientation*: Performs an orientation test and highlights falsely oriented triangles. You should have removed all intersections before applying this check. For simple configurations the wrong triangles are removed automatically. The number of inconsistently oriented triangles and the number of removed triangles are printed in the console window. If the automatic method could not remove all incorrect orientations, you can proceed as in the case of intersections and try to repair them manually. **Important:** A prerequisite for the orientation test is that the outer triangles of the surface are assigned to material *Exterior*. If the surface does not contain such a material or if the assignment to *Exterior* is not correct the test will falsely report orientation errors.

- *Aspect Ratio*, *Dihedral Angle*, and *Tetra Quality* sort all surface triangles according to different quality measures. The aspect ratio is the ratio of the radii of the circumcircle and the incircle of a triangle. The largest aspect ratio should be below 20 (better 10). The *dihedral angle* is the angle between two adjacent triangles at their common edge. The smallest dihedral angle should be above 5 degrees (better 10). *Tetra quality* may be useful if you want to create a tetrahedral volume mesh from the surface. For each surface triangle the aspect ratios of the tetrahedra which will probably be created for that triangle are calculated. The largest tetrahedral aspect ratio should be below 50 (better 25).

  The worst triangle according to the selected quality measure is displayed, and the worst quality is printed in the console window. Using the *back* and *forward* buttons right beside the selector combo box you can cycle through all other triangles and edit the surface if necessary.

10.13.3 Tools

This section describes interactive tools for selecting triangles and for performing simple edit operations. Only one such tool can be active at a time. The active tool determines which effect left mouse button clicks have. For each tool there is a separate button contained in the editor's tool bar. Independently from the active mouse tool, edges and triangles can be picked using the middle mouse button. This causes the ids of the picked triangles, edges, and points to be printed on the screen.

- *Pick tool [P]*: A single triangle can be highlighted using a simple mouse click or unhighlighted using a shift-click. If in addition the Ctrl key is held down a group of neighbouring triangles will be highlighted or unhighlighted (with the shift key held down too).
- *Magic wand [M]*: Allows to select a connected group of triangles. Unless the shift key is held down the highlight buffer will be cleared before new triangles are highlighted. Control-click causes all selected triangles to be unhighlighted. The behaviour of the magic wand tool can be modified using the dialog *Magic wand settings...* which can be activated from the *Edit* menu. This dialog lets you define what triangles are considered to be a connected group. In particular, a crease angle smaller than 180 degrees can be chosen. In this case only triangles with roughly the same direction as the clicked triangle will be highlighted. Regardless of these settings a connected group of highlighted triangles will always be unhighlighted if a highlighted triangle is Ctrl-clicked.
- *Draw tool [D]*: This tool allows you to highlight triangles by drawing a contour in the viewer window. Usually the left mouse button must be held down while the contour is drawn. However, you may hold down the Alt key and release the left mouse button. In this case straight line segments can be defined. Currently the tool selects all triangles inside the contour, i.e., hidden triangles are highlighted too. However, with the magic wand tool backward facing parts of the surface can be easily deselected again using a Ctrl-click.

- *Flip tool [F]*: Flips an edge of the surface. Only edges with two adjacent triangles can be flipped, but no boundary edges. Flipping the edge once again restores the original state.

- *Collapse tool [C]*: Contracts an edge, i.e., moves one vertex of an edge onto the other. For non-boundary edges the operation will reduce the number of triangles by two. The vertex of the edge located more closely to the mouse position will be retained.

- *Bisect tool [B]*: Subdivides an edge of the surface. A new vertex is inserted at the edge midpoint. Each triangle adjacent to the edge is bisected into two triangles.

- *Translate tool [T]*: Allows to pick a vertex of the surface and to translate it. At the picked vertex a point dragger will be shown. The dragger can then be picked and translated. Alternatively another point on the surface can be shift-clicked while the point dragger is shown. This moves the dragger to the new position.

10.14 Transform Editor

This editor allows you to add a transformation to a data set or modify an existing one. A transformation may be a translation, rotation and scaling or a combination thereof. The transformation can be edited interactively in the 3D viewer using different Open Inventor draggers.

The *Transform Editor* also lets you enter transformations numerically. This can be done by pressing the *Dialog...* button which pops up the dialog shown in Figure 10.22.

The dialog provides text fields to specifiy the translation, rotation, and scaling of the object individually. Instead of specifying absolute values the object can also be translated, rotated, or scaled incrementally. This is done by activating the tab panels *Relative Local* or *Relative Global* instead of the default *Absolute*. The local and global tabs differ in the way how the translation, rotation, or scaling is applied, namely after an existing transformation (local) or before it (global). An alternative way to modify the transformation of a data object is to use the Tcl commands provided by SpatialData objects.

Display modules connected to a transformed data set will take the transformation into account automatically. Also many, but not all (!), compute modules interprete transformations. If you find a module that does not operate as expected for a transformed data obejct, try to apply the tranformation first using the *Apply transform* button.

Most data file formats do not support transformations. Therefore, transformations are stored in amira network scripts only. Keep this in mind when working with transformed data objects. It is always possible to query and reapply a transformation using the Tcl commands `getTransform` and `set-transform`.

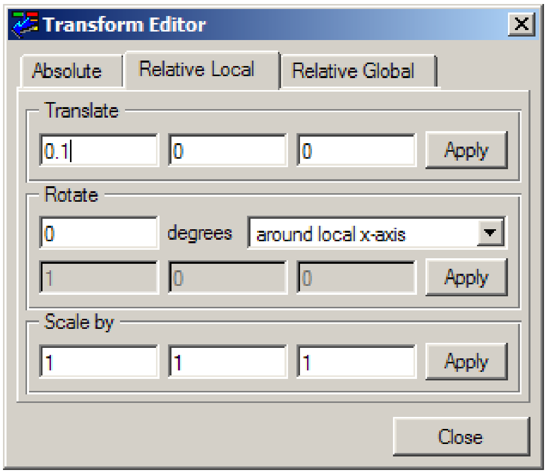
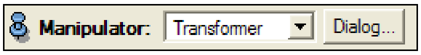


Figure 10.22: Dialog opened by the transform editor.

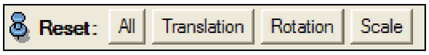
Ports

Manipulator



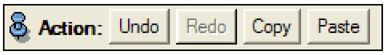
Lets you choose the Open Inventor dragger used to define the transformation in 3D. Make sure to switch the viewer to interaction mode when using the draggers (press the arrow button in the upper right corner of the viewer, or toggle between viewing mode and interaction mode using the ESC key). Details of how to interact with the draggers can be found in the Open Inventor documentation. The *Dialog...* button pops up the transfrom dialog described above.

Reset



This port allows you to reset the transformation matrix, or its translational, rotational, or scaling components.

Action



This port allows you to undo the last change of the transformation matrix, or to redo the last undone operation. In addition, transformations can be copied into an internal buffer, and afterwards pasted into the editor again. This provides a convienient way to copy a transformation from one object to another.

If the editor is invoked for an object derived from VertexSet, a button *Apply Transform* is shown. This button allows you to apply the transformation to the vertices of the data object, and to reset the object’s transformation matrix. This operation does not change the visual appearance of the object, but it is required for example to export the transformed object into a file.

Part III

amira Programmer's Manual

Chapter 11

Introduction

The developer version of amira allows you to add new components such as file read or write routines, modules for visualizing data or modules for processing data. New module classes and new data classes can be defined as subclasses of existing ones.

Note that it is not possible (or possible only to a very limited extent) to change or modify existing modules or parts of amira's graphical user interface.

In the following sections we

- present an overview of the amira developer version,
- discuss the system requirements for the different platforms,
- outline the structure of the amira file tree,
- show how to compile the demo package in a quick start tutorial,
- provide additional hints on compiling and debugging,
- and mention how to upgrade and maintain existing code.

11.1 Overview of the amira Developer Version

The amira developer version is a superset of the ordinary amira binary version. In addition to the files contained in the binary version, the developer version essentially provides all C++ header files needed to compile custom extensions.

11.1.1 Packages and Shared Objects

amira is an object-oriented software system. Besides the core components like the graphical user interface or the 3D viewers, it contains a large number of data objects, modules, readers and writers.

Data objects and modules are C++ classes, readers and writers are C++ functions.

Instead of being compiled into a single static executable, these components are grouped into *packages*. A package is a shared object (usually called .so or .sl on Unix or .dll on Windows) which can be dynamically loaded by amira at run time when needed. This concept has two advantages. On the one hand, the program remains small since only those packages are loaded which are actually needed by the user. On the other hand it provides almost unlimited extensibility since new packages can be added any time without recompiling the main program.

Therefore, in order to add custom components to the amira developer version, new packages or shared objects must be created and compiled. A package may contain an arbitrary number of modules and it is left up to the developer whether he wants to organize his modules into several packages or just in one.

11.1.2 Package Resource Files

Along with each package a *resource file* is stored. This file contains information about the components being defined in a particular package. When amira starts, it first scans the resource files of all available packages and thus knows about all the components which may be used at run-time.

The resource files of the standard amira packages are located under `share/resources` in the directory where amira is installed. Details about registering read and write routines or different kinds of modules in a resource file are provided in Chapters 13 and 14.

11.1.3 The Local amira Directory

Usually amira will be installed by the system administrator at a location where ordinary users are not allowed to create or modify files. Therefore it is recommended that every user creates new packages in his own personal *local amira directory*. The local amira directory has essentially the same structure as the directory where amira is installed. A new local amira directory can most easily be created by using the *Development Wizard*, a special-purpose dialog box described in detail in Section 12.

Once a local amira directory has been set up, resource files located in it will also be scanned by amira when started. In this way new components can be added or existing ones redefined.

11.1.4 External Libraries

amira is based on a number of industry standard libraries. The most important ones are *Open Inventor*, *OpenGL*, *Qt*, and *Tcl*.

amira's 3D graphics is based on OpenGL and Open Inventor. OpenGL is the industry standard for professional 3D graphics. Open Inventor is a C++ library using OpenGL which provides an object-oriented scene description layer. Writing new visualization modules for amira essentially means creating an Open Inventor scene from the input data. If you already have code doing this, it will be straightforward to turn it into an amira module. While the Open Inventor headers are included in the amira developer version, OpenGL must already be installed on your system.

Qt is a platform-independent C++ library for building graphical user interfaces (GUIs). amira is built with Qt. However, the user interface elements used in standard amira modules are encapsulated by special amira classes called *ports*. Therefore you can develop your own modules without knowing Qt and without having Qt installed (Qt headers are not included in the amira developer version). You only need Qt if you plan to add completely new user interface components such as special purpose dialogs. Note also, that in this case you need to install the correct version of the Qt header files. amira 3.1 is linked against Qt 3.2.0.

Finally, Tcl is a C library providing an extensible scripting language used by amira. All required header files are included in the developer version. amira programmers usually need not know details of the Tcl API but merely derive their code from existing examples.

11.2 System Requirements

In order to develop new amira components as described in this document you need the developer version of amira (called *amiraDev*) as well as a C++ development environment. C++ compilers, however, are generally not compatible, therefore the compilers and compiler versions listed below should be used. Other compiler versions may work too, but this is not guaranteed. In particular, it is not possible to use the GNU gcc compiler except on Linux.

On all Unix platforms the GNU make utility (`gmake`) is needed in order to use the GNUmakefiles provided with *amiraDev*. `gmake` is included in the `bin` subdirectory of *amiraDev*. To proceed you either should include this directory in your path, or create a link in a directory already listed in your path, e.g., in */usr/bin*.

In the following text more specific system requirements are listed for each platform. More general hardware requirements such as installed memory or special graphics adapters are listed in the amira*User's Guide*. On all systems an OpenGL library together with the OpenGL header files must be installed.

11.2.1 SGI IRIX

Operating system: IRIX 6.5.15 or higher
Compiler: MIPSpro C/C++ version 7.3.1.2 or higher

Use `CC -version` to find out the version of the MIPSpro compiler.

11.2.2 HP-UX

Operating system: HP-UX 11.00
Compiler: ANSI C/C++ version 3.27 or higher (`aCC`)

Note that the HP standard C++ compiler (`CC`) will not work. Use `aCC -V` to find out the version of the ANSI compiler. Versions earlier than 1.21 may produce errors, especially when compiling optimized code.

11.2.3 Sun Solaris

Operating system: SunOS 5.8 (Solaris 8) or higher
Compiler: Sun WorkShop 6 update 2 or higher (C++ 5.3)

Note that older versions of the compiler will not work. Use `CC -V` to find out the version of the WorkShop compiler.

11.2.4 Linux

Operating system: RedHat 8.0
Compiler: GNU gcc 3.2.x

Linux distributions other than RedHat 8.0 which are also based on glibc 2.2 might work too, but this is not guaranteed. In order to get amira running on other Linux distributions there is a ”patch” provided on `http://www.amiravis.com`, which contains the correct versions of all required system libraries. This patch also works for amiraDev, provided you are using the correct compiler.

11.2.5 Windows

Operating system: Windows 98/ME/2000/XP
Compiler: Microsoft Visual Studio C++ 6.0, Service Pack 4 or higher

The latest Service Pack for Microsoft Visual Studio can be downloaded from `http://msdn.microsoft.com/vstudio/sp`. We recommend to use Windows 2000 or Windows XP.

As an alternative development platform also an amiraDev version for Microsoft Visual Studio Studio .NET 2003 (VC++ 7.1) is provided. You cannot mix this with code compiled using the old Visual Studio because different run-time libraries are required. Note, that the initial version of Microsoft Visual Studio .NET (VC++ 7.0) is not supported by amira.

11.3 Structure of the amira File Tree

Like the ordinary version the developer version of amira is installed in a single directory called the *amira root directory*. This directory contains all the binaries, shared objects, and resource files required to run amira, as well as all the C++ header files required to compile new components. New components themselves are stored independently in a *local amira directory*. Every user may define his/her own local amira directory. The local amira directory has a structure very similar to the amira root directory. In the following two sections the structure of these two directories is described in more detail.

11.3.1 The amira Root Directory

The contents of the amira root directory may differ slightly from platform to platform. For example, on Windows there will be no subdirectory `lib`. Instead, the compiled shared objects are located under `bin/arch-Win32-Optimize`. The following list gives a general overview.

`Amira-3.1/`	*amira root directory, actual name may differ*
`bin/start`	*amira start script (Unix)*
`bin/arch-Win32-Optimize/amiramain.exe`	*Executable (Windows)*
`data/`	*contains amira demo data sets*
`include/`	*contains all required C++ header files*
`lib/arch-*-Optimize/`	*compiled shared objects (Unix)*
`make/`	*make environment for Unix systems*
`share/resources/`	*resource files of all standard packages*
`share/devref/`	*online documentation of amira C++ classes*
`share/doc/`	*online version of amira documentation*

11.3.2 The Local amira Directory

The local amira directory contains the source code and object files of custom modules, the resource files of custom packages, and the compiled custom packages themselves. The packages can be compiled either in a debug version or in an optimized version. The corresponding object files and compiled shared objects reside in different subdirectories called `arch-*-Debug` and `arch-*-Optimize`, respectively. Here the asterisk denotes the particular architecture, e.g., `Win32` for Windows systems or `IRIX` for SGI Unix platforms.

In order to create a new local amira directory the Development Wizard should be used. For details please refer to Section 12. Subdirectories like `AmiraLocal/lib` or `AmiraLocal/share/resources` are created automatically the first time a custom package is compiled. Again, the contents of the local amira directory may differ slightly from platform to platform. For example, on Windows compiled shared objects are located under `bin` instead of `lib`.

`AmiraLocal/`	*e.g., located in the user's home directory*
`AmiraLocal.dsw`	*Visual Studio workspace file (Windows)*

`GNUmakefile`	*global makefile (Unix)*
`src/`	*contains source code of custom packages*
`mypackage/`	*source code of one particular package*
`Package`	*used to generate GNUmakefile and project files*
`GNUmakefile`	*Unix makefile (generated automatically)*
`mypackage.dsp`	*Visual Studio project file (generated automatically)*
`mypackageAPI.h`	*Windows storage-class specification*
`MyModule.h`	*header file of a custom module*
`MyModule.cpp`	*implementation of a custom module*
`MyReader.cpp`	*implementation of a custom read routine*
`...`	*any number of additional components*
`share/resource/mypackage.rc`	*package resource file*
`...`	*any number of additional packages*
`obj/arch-*-Debug/`	*object files on Unix (debug version)*
`obj/arch-*-Optimize/`	*object files on Unix (optimized)*
`lib/arch-*-Debug/`	*compiled shared objects (debug version)*
`lib/arch-*-Optimize/`	*compiled shared objects (optimized)*
`share/resources/`	*package resource files are copied here*

11.4 Quick Start Tutorial

This section contains a short tutorial on how to compile and execute the demo package provided with amiraDev. The demo package contains the source code of the example modules and IO routines described elsewhere in this manual. At this point you should just get a rough idea about the basic process required to develop your own modules and IO routines. Details will be discussed in the following chapters.

For the development of custom amira packages a dedicated directory, the local amira directory, is required. Initially, this directory should be created using the Development Wizard. Lets see how this is done:

- Start amira and choose the Development Wizard from the *Help* menu of the amira main window.
- Make sure that the item *Set local amira directory* is selected in the wizard's dialog window.
- Press the *Next* button.

You can now enter a directory name for the local amira directory. For example you may choose `AmiraLocal` in your home directory. The directory must be different from directory where amira has been installed.

- Enter the name for the local amira directory.

- Select the button *copy demo package*.
- Press the *OK* button.

If the directory did not yet exist, amira asks you if it should be created. The name of the directory is stored in the Windows registry or in the `.AmiraRegistry` file in the Unix home directory, so that the next time amira is started all modules or IO routines defined in this directory will be available.

The next step is to create the Visual Studio project files for Windows or the GNUmakefiles for Unix. These files will be generated from a *Package* file which must be present in each custom package directory. The syntax of the Package file is described in Chapter 12.10. The demo package already contains a Package file, so there is no need to create one here.

- Select *Create build system* on the main page of the Development Wizard.
- Press the *Next* button.
- Choose *all local packages* as target.
- Choose which kind of build system you want to create.
- Press the *OK* button.

The files for the selected build system will now be created automatically. The advantage of the automatic generation is that the include and library paths are always set correctly. Also, any dependencies between local packages are taken into account.

Once the build system has been created you can close the Development Wizard and exit amira. We are now ready to compile the demo package. This is different for each platform:

Windows Visual Studio C++ 6

- Start Visual Studio C++ 6 and load the workspace `AmiraLocal.dsw` from the local amira directory. If your local amira directory is not called `AmiraLocal` the workspace file also has some other name.
- Build the workspace in debug mode by pressing F7 or by choosing *Build* from the *Build* menu.

Windows Visual Studio C++ .NET 2003

- Start Visual Studio C++ .NET 2003 and load the solution file `AmiraLocal.sln` from the local amira directory. If your local amira directory is not called `AmiraLocal` the solution file also has some other name.
- Build all local packages in debug mode by pressing F7 or by choosing *Build Solution* from the *Build* menu.

Unix GNUmakefile system

- Change into the local amira directory in a shell.

- Type in `gmake` to build all local packages is debug mode. If `gmake` is not already installed on your system you can find it in the subdirectory `bin` in the amira root directory. Either add this directory in your path variable or create a link in a directory already listed in your path, e.g., `/usr/bin`.

We are now ready to start amira in order to test the demo package. However, because we have compiled the demo package in debug mode, we need to start amira with the command line option `-debug`. Otherwise, amira would not find the correct DLLs or shared libraries. For convenience, on Windows a link *Amira -debug* is available in the start menu.

In order to check if the demo package has been successfully compiled and can be loaded by amira, you can for example choose the entry *DynamicColormap* from the *Create Data* menu of the amira main window. Then a new colormap object should be created. You can find the source code of this new object in the local amira directory under `src/mypackage/MyDynamicColormap.cpp`. In the same directory there is also the header file for this class.

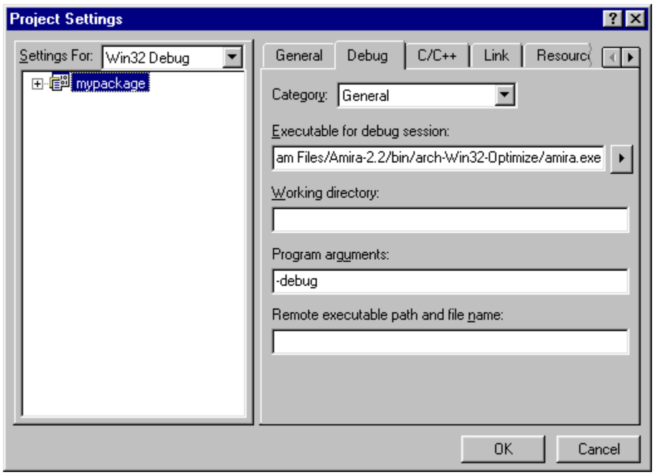
If you want to compile the demo package in release mode, you have to change the active configuration in Visual Studio and recompile the code. On Unix, you have to call `gmake MAKE_CFG=Optimize`. You can also define `MAKE_CFG` as an environment variable.

11.5 Compiling and Debugging

This section provides additional information not covered by the quick start guide on how to compile and debug custom amira packages. You may skip it the first time you are reading this manual. The information will not become relevant until you are actually developing your own code.

It has already been mentioned that the development of custom amira packages should take place in a local amira directory. Initially, such a directory should be created using the Development Wizard described in Chapter 12. The name of the local amira directory is stored in the Windows registry or in the file `.AmiraRegistry` in your Unix home directory. On both Windows and Unix, the name of the local amira directory can be overridden by defining the environment variable AMIRA_LOCAL. This might be useful if you want to switch between different local amira directories. However, in general it is recommended not to set this variable.

For each local package there is a resource file stored in the subdirectory `share/resources` in the local amira directory. This file contains information about all modules and IO routines provided by that package. A local package can be compiled in debug mode suitable for debugging or in release mode with compiler optimization turned on. In the first case the DLLs or shared libraries are stored under `bin/arch-*-Debug` on Windows and `lib/arch-*-Debug` on Unix. In the second case they are stored under `bin/arch-*-Optimize` or `lib/arch-*-Optimize`. Here the '*' indicates the actual architecture name. In the following it will be described how to compile local packages in both modes on the different platforms and how to debug the code using a debugger.



Category: General

Executable for debug session: am Files/Amira-2.2/bin/arch-Win32-Optimize/amira.exe

Working directory:

Program arguments: -debug

Remote executable path and file name:

Figure 11.1: Specifying the name of the executable in Visual Studio.

11.5.1 Windows Visual Studio 6

For Visual Studio 6 Service Pack 4 or higher is required. Code generated with Visual Studio 6 cannot be mixed with code generated with Visual Studio .NET 2003 because different run-time libraries are required.

The workspace and project files for Visual Studio 6 are generated automatically from the amira Package files by the Development Wizard. There should be no need to change the project settings manually.

On default, Visual Studio 6 will compile in debug mode. In order to generate optimized code, you need to change the active configuration. This is done by choosing *Set Active Configuration...* from the *Build* menu.

In order to execute the debug mode version of your local packages you have to start amira with the command line option `-debug`. For convenience, a link *Amira -debug* is provided in the start menu. However, if you want to debug your code you need to start amira from Visual Studio. Thus you need to specify the correct executable in the project settings dialog.

You can bring up the project settings dialog by pressing Alt-F7 or by choosing *Settings...* from the *Build* menu. Select the *Debug* tab. In the field *Executable for debug session* choose the file `bin/arch-Win32-Optimize/amiramain.exe` located in the amira root directory. In the field *Program arguments* type in `-debug` (compare Figure 11.1).

You can now start amira from Visual Studio by pressing F5 or by choosing *Start Debug Go* from the *Build* menu. In order to debug your code you may set breakpoints at arbitrary locations in your code. For example, if you want to debug a read routine, set a breakpoint at the beginning of the routine, execute amira and invoke the read routine by loading some file.

11.5.2 Unix

In order to compile a local package under Unix you need to change into the package directory and execute `gmake` in a shell. The `gmake` utility is provided in the `bin` subdirectory of the amira root directory. Either add this directory to your path or create a link in a directory already listed in your path, e.g., in `/usr/bin`.

The required GNUmakefiles will be generated automatically from the amira Package files by the Development Wizard. There should be no need to edit the GNUmakefiles manually. Depending on the contents of the Package file all source files in a package directory will be compiled, or a subset only. On default all files will be compiled. The Development Wizard will put the name of the amira root directory into the file `GNUmakefile.setroot`. You may overwrite the name by defining the environment variable `AMIRA_ROOT`. For example, this might be useful when working simultaneously with two different amira versions.

By default `gmake` will compile debug code. In order to compile optimized code, invoke `gmake MAKE_CFG=Optimize`. Alternatively, you may set the environment variable `MAKE_CFG` to `Optimize`.

If you have a multi-processor machine you may compile multiple files at once by invoking `gmake` with the option `PARALLELFLAGS=-j<n>`. Here `<n>` denotes the number of compile jobs to be run in parallel. Usually twice the number of processors is a good choice.

If you have compiled debug code, you must invoke amira with the command line option `-debug`. Otherwise, the optimized version will be executed. If no such version exists an error occurs. Instead of specifying `-debug` at the command line you may also set the environment variable `MAKE_CFG` to `Debug`.

In order to run amira in a debugger, invoke the amira start script with one of the following command line options:

- `-cvd` on IRIX
- `-dde` on HP-UX
- `-workshop` on SunOS
- `-gdb` or `-ddd` on Linux

Note that usually you cannot set a breakpoint in a local package right after starting the debugger. This is because the package's shared object file will not be linked to amira until the first module or read or write routine defined in the package is invoked. In order to force the shared object to be loaded without invoking any module or read or write routine, you may use the command `dso open lib<name>.so`, where `<name>` denotes the name of the local package. Once the shared object has been successfully loaded, breakpoints may be set. It depends on the debugger whether these breakpoints are still active when the program is started the next time.

11.6 Maintaining Existing Code

This section is directed to programmers who have already developed custom modules using a previous version of amiraDev. In particular, we describe

- how to upgrade to amiraDev 3.1, and
- how to rename an existing package.

11.6.1 Upgrading to amiraDev 3.1

In amiraDev 3.1 the structure of the local amira directory has been slightly changed. In order to recompile existing packages it is recommended to create a complete copy of the local amira directory and to adapt this copy as described below. amiraDev 3.1 and earlier versions store the path of the local amira directory under different names in the Windows registry or in the Unix `.AmiraRegistry` file. This means that both versions can be used in parallel. The following changes are required to adjust an existing local amira directory so that it can be used with amiraDev 3.1:

- The subdirectory *packages* must be renamed to *src*.
- In each package directory a *Package* file must be created. This file contains the package name, the name of dependent packages, and optionally an explicit list of all source and header files belonging to the package. From the *Package* file a Visual Studio project file or a GNUmakefile for Unix can be generated automatically.
- Instead of modifying and reusing the old Visual Studio project files or GNUmakefiles these files should be created from scratch using the amira development wizard.

Modules, data classes and IO routines developed for amiraDev 3.0 should compile without changes with amiraDev 3.1 (with the possible exception of calls to the C++ standard library, see below). The API of existing classes has been extended in several cases, but no incompatible changes have been introduced. For details about the interface of particular classes please refer to the online reference guide. In addition, the following things have changed:

- *C++ standard library:* On all platforms except of Linux IA64 (which is built on RedHat Advanced Workstation 2.1 with gcc 2.96) and IRIX new-style C++ standard libraries are being used now. In amiraDev 3.0 only on Windows the new-style interface was used.

  In contrast to the old-style headers the new-style headers do not contain the suffix `.h`, e.g., you need to include `iostream` instead of `iostream.h`. In addition, all symbols are defined inside the `std` namespace, i.e., you need to write for example `std::cout` instead of just `cout`, unless you specifically write `using namespace std;` in your code.

  On most platforms the new-style and the old-style C++ standard library cannot be used together. This means that you may need to switch to the new-style interface, if you have used the old-style interface before.

- *Open Inventor and Qt:* amira 3.1 uses Open Inventor 4.0.4 and Qt 3.1.2. As in previous versions the Open Inventor headers are completely provided with amiraDev, but the Qt headers are not. Moreover, instead of the SoWin and SoXt classes (which are not fully platform independent) in amira now the new SoQt interface is used. This means that the amira viewer is now derived from `SoQtExaminerViewer` instead of `SoWinExaminerViewer` or `SoXtExaminerViewer`, respectively. However, because we didn't want the class `HxViewer` to depend on any Qt headers, the actual amira viewer has been renamed to `QxViewer`, while `HxViewer` now is a pure wrapper class. This change should not affect existing code unless platform-dependent methods of the former SoWin or SoXt base classes were used.

11.6.2 Renaming an Existing Package

Sometimes you may want to rename an existing amira package, for example when using an existing package as a template for a new custom package. In order to do so the following changes are required:

- Rename the package directory:
  `AmiraLocal/src/oldname` becomes `AmiraLocal/src/newname`
- Rename the following files in the package directory:
  `oldnameAPI.h` becomes `newnameAPI.h`
  `share/resources/oldname.rc` becomes `share/resources/newname.rc`
- In the package resource file `share/resources/newname.rc` and in the `Package` file replace `oldname` by `newname`.
- In the file `newnameAPI.h` replace `OLDNAME_API` by `NEWNAME_API`.
- In all header and source files of the package, adjust the include directives if necessary, i.e., instead of
  `#include <oldname/SomeClass.h>`
  now write `#include <newname/SomeClass.h>`

All replacements can be performed using an arbitrary text editor. After all files have been modified as necessary a new Visual Studio project file or a new GNUmakefile should be created using the amira development wizard.

Chapter 12

The Development Wizard

The development wizard is a special tool which helps you to set up a local amira directory tree so that you can write custom extensions for amira. In addition, the development wizard can be used to create templates of amira modules or of read or write routines. The details of developing such components are treated in other chapters. At this point we want to give a short overview about the functionality of the development wizard.

In particular, we discuss

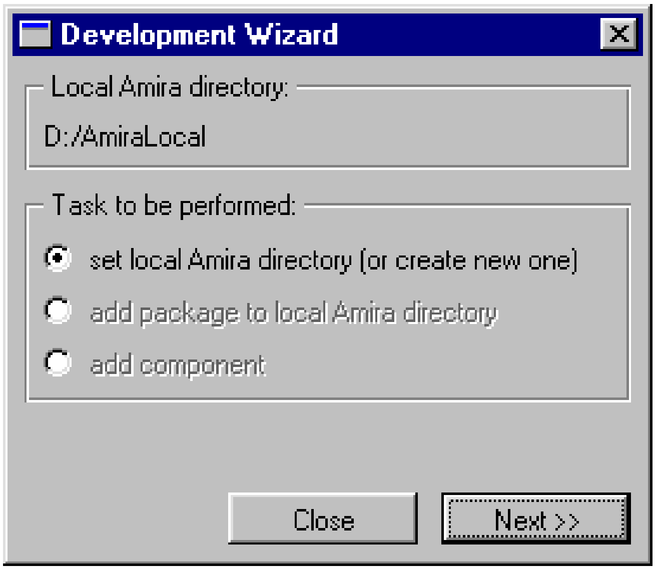
- how to invoke the development wizard
- how to set or create the local amira directory
- how to add a package to the local amira directory
- how to add components to an existing package
- how to create the files for the build system

Finally, a section describing the Package file syntax is provided.

12.1 Starting the Development Wizard

In order to invoke the development wizard, first start amira. Then select *Development Wizard* from the main window's *Help* menu. Note that this menu option will only be available if you are running the developer version of amira.

The layout of the development wizard is shown in Figure 12.1. Initially, the wizard informs you about the local amira directory currently being used. If no local amira directory is defined, this is indicated too. Furthermore, the wizard lets you select between four different tasks to be performed. These are


Development Wizard

Local Amira directory:

D:/AmiraLocal

Task to be performed:

(●) set local Amira directory (or create new one)

( ) add package to local Amira directory

( ) add component

Figure 12.1: Initial layout of the amira development wizard.

- setting the local amira directory (or creating a new one)
- adding a new package to the local amira directory
- adding a component to an existing package
- creating the files for the build system

The first option is always available. A new package can only be added if a valid local amira directory has been specified. For the local amira directory to be valid, among others, it must contain a subdirectory called `src`. If at least one package exists in the `src` directory of the local amira directory, a new component, i.e., a module or a read or write routine, can be added to a package. Finally, the last option allows you to create all files required by the build system, i.e., Visual Studio project files or GNUmakefiles for Unix platforms.

12.2 Setting Up the Local amira Directory

The local amira directory contains the source code and the binaries of all custom extensions developed by a user. The name of this directory can be most easily specified using the development wizard (see Figure 12.2). Since potentially every user can write his/her own extensions for amira it is usually recommended that the local amira directory is created in the user's home directory.

If the specified directory does not exist the development wizard asks you whether it should be created.

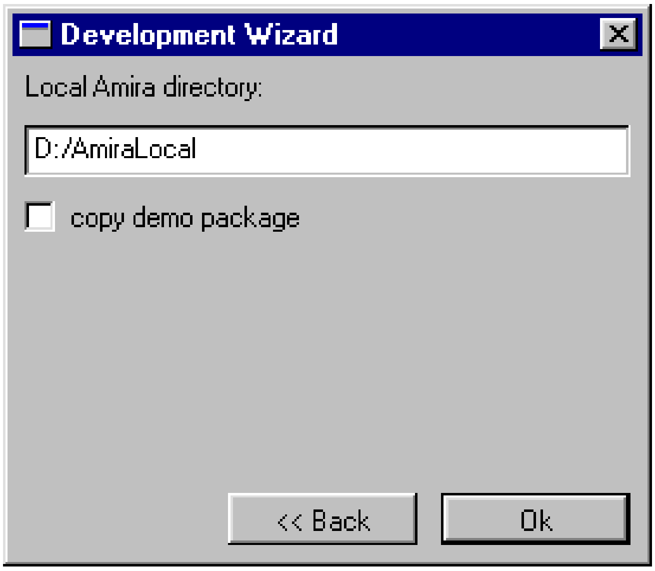


Figure 12.2: Setting the local amira directory.

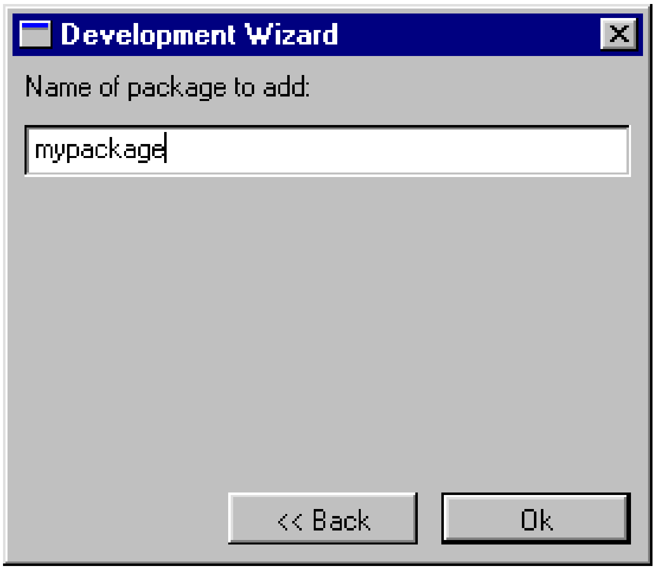
If you confirm, the directory itself together with some subdirectories will be created. You may also specify an existing empty directory in the text field. Then the subdirectories will be created in there.

Finally, you may choose an existing directory which has been created by the development wizard before. In this case a simple check is performed to determine whether the specified directory is valid. If you want to use a directory created with amira 3.0 or an earlier version, please first refer to Section 11.6 (upgrading existing code).

In order to unset the local amira directory you should clear the text field and press *OK*. The directory will not be deleted, but the next time amira is started modules and IO routines defined in the local amira directory will not be available anymore.

Once you have set up the local amira directory, the name of the directory is stored permanently, so the next time amira is started the .rc-files located in the subdirectory `share/resources` of the local amira directory can be read. In this way custom components are made known to amira. On Windows the name of the local amira directory is stored in the Windows registry. On Unix systems it is stored in the file `.AmiraRegistry` in the user's home directory. In both cases, these setting can be overridden by defining the environment variable `AMIRA_LOCAL`.

The development wizard also provides a toggle for copying a demo package to the local amira directory. You will get a warning if this button is activated and an existing local amira directory already containing the demo package has been specified. The demo package is copied to the subdirectory `src/mypackage` in the local amira directory. It contains all read and write routines and modules


Development Wizard

Name of package to add:

mypackage

Figure 12.3: Adding a new package to the local amira directory.

presented as examples in this guide.

12.3 Adding a New Package

All amira components are organized in *packages*. Each package will be compiled into a separate shared object (or DLL file on Windows). Therefore, before any components can be defined at least one package must be created in the local amira directory. In order to do so, choose *add package to local amira directory* on the first page of the wizard and press *Next*. On the next page you can enter the name of the new package (see Figure 12.3).

The name of a package must not contain any white spaces or punctuation characters. When a package is added, a subdirectory of the same name is created under `src` in the local amira directory. In this directory the source code and header files of all the modules and IO routines of the package are stored. In addition in each package directory there must be a *Package* file from which the build system files can be generated.

Initially, a default *Package* file will be copied into a new package directory. This default file adds the most common amira libraries for linking. It also selects all C++ source files in the package directory to be compiled. In order to generate the build system from the *Package* file, please refer to Section 12.9.

In addition to the *Package* file also the files `version.cpp` and `packageAPI.h` will be copied into a new package directory. The first file allows you to put version information into your package, which can later be viewed in the amira system information dialog. The second file contains a macro required for putting symbols in a DLL on Windows. Finally, also an empty file `package.rc` will be copied into `share/resources`. In this file later modules and IO routines will be registered.

12.4 Adding a New Component

If you choose the *add component* option on the first page of the development wizard, you will be asked what kind of component should be added to which package. Remember that the *add component* option will only be available if a valid local amira directory with at least one existing package is found. In particular, templates

- of an ordinary module,
- of a compute module,
- of a read routine,
- or of a write routine

may be created (see Figure 12.4). The option menu in the lower part of the dialog box lets you specify the package to which the component should be added. After you press the *Next* button, you will be asked to enter more specific information about the particular component you want to add. Up to this point no real operation has been performed, i.e., no files have been created or modified.

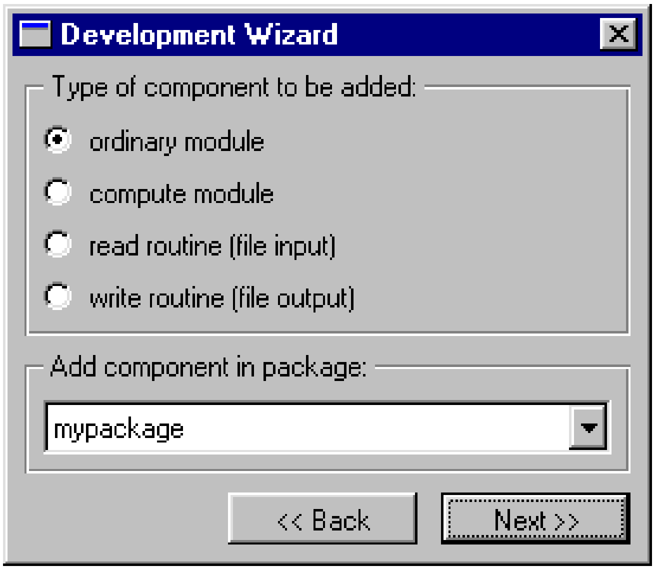
12.5 Adding an Ordinary Module

An ordinary module in amira usually directly visualizes the data object it is attached to. For example, the Isosurface module, the Voltex module, and the SurfaceView module are of this type. Such modules, sometimes also called *display modules*, are represented by yellow icons in the object pool.

In order to create the template for an ordinary module using the development wizard, you must enter the C++ class name of the module, the name to be shown in the pop-up menu of possible input data objects, the C++ class name of possible input data objects, and finally the package where the input class is defined (see Figure 12.5).

Once you press *OK*, two files are created in the package directory, namely a header file and a source code file for the new module. In addition, a new module statement is added to the package resource file located under `share/resources` in the package directory.

After you have added a new module to a package you need to recreate the build system files before you can compile the module. Details are described in Section 12.9.


Development Wizard

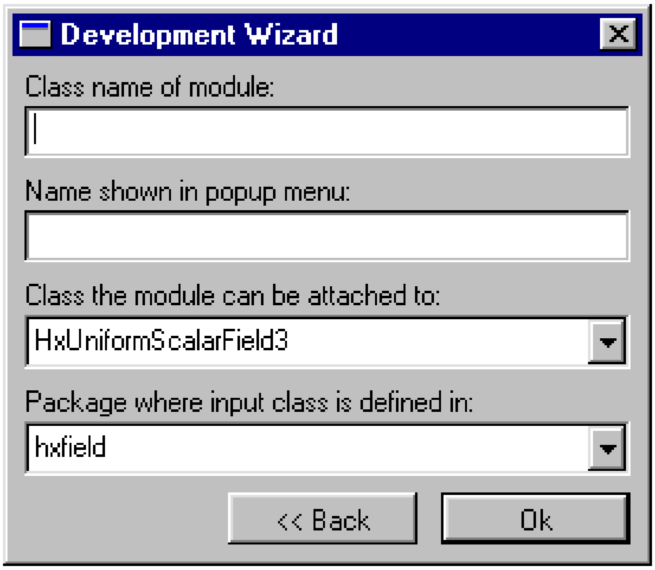
Type of component to be added:

(●) ordinary module
( ) compute module
( ) read routine (file input)
( ) write routine (file output)

Add component in package:

mypackage

Figure 12.4: Adding a new component to an existing package.


Development Wizard

Class name of module:

Name shown in popup menu:

Class the module can be attached to:

HxUniformScalarField3

Package where input class is defined in:

hxfield

Figure 12.5: Creating the template of a custom module.

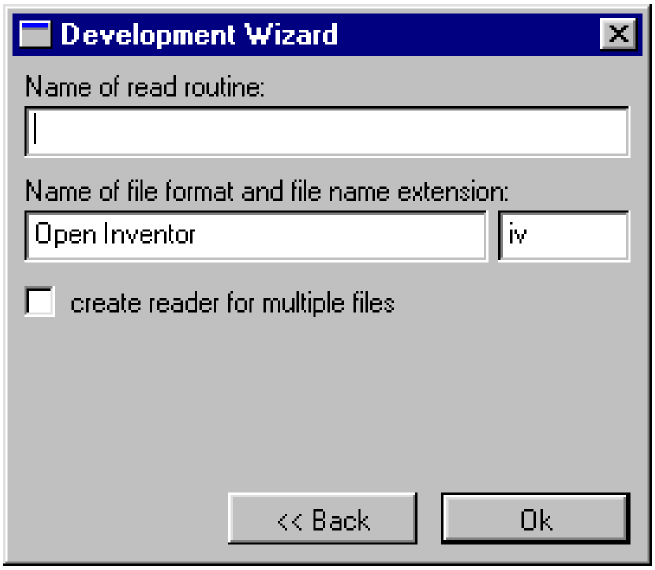

Development Wizard
Name of read routine:
Name of file format and file name extension:
Open Inventor
iv
create reader for multiple files
<< Back
Ok

Figure 12.6: Creating the template of a read routine.

12.6 Adding a Compute Module

A *compute module* in amira usually takes one or more input data objects, performs some kind of computation, and puts back a resulting data object in the object pool. Compute modules are represented by red icons in the object pool.

The only difference between an ordinary module and a compute module is that the former is directly derived from `HxModule` while the latter is derived from `HxCompModule`. When creating a template for a compute module using the development wizard, the same input fields must be filled in as for an ordinary module. The meaning of these input fields is described in Section 12.5.

12.7 Adding a Read Routine

As will be explained in more detail in Section 13.2, read routines are global C++ functions used to create one or more amira data objects from the contents of a file stored in a certain file format. To create the template of a new read routine, first the name of the routine must be specified (see Figure 12.6). The name must be a valid C++ name. It must not contain blanks or any other special characters.

Moreover, the name of the file format and the preferred file name extension must be specified. The extension will be used by the file browser in order to identify the file format. The format name will be displayed next to any matching file.

Finally, a toggle can be set in order to create the template of a read routine supporting the input of multiple files. Such a routine will have a slightly different signature. It allows you to create a single data object from multiple input files. For example, multiple 2D image files can be combined in a single 3D image volume. Details are provided in Section 13.2.3.

After you press *OK* a new file `<name>.cpp` will be created in the package directory, where `<name>` denotes the name of the read routine. In addition, the read routine will be registered in the package resource file. Some file formats can be identified by a unique file header, not just by the file name extension. In such a case you may want to modify the resource file entry as described in Section 13.2.1.

Remember, that after you have added a new read routine to a package you need to recreate the build system files before you can compile it. Details are described in Section 12.9.

12.8 Adding a Write Routine

A write routine is a global C++ function which takes a pointer to some data object and writes the data to a file in a certain file format. The details are explained in Section 13.3. In order to create the template of a new write routine, the name of the routine must first be specified (see Figure 12.7). The name must be a valid C++ name. It must not contain blanks or any other special characters.

In addition, the name of the file format and the preferred file name extension must be specified. Before saving a data object, both the name and the extension will be displayed in the file format menu of the amira file browser.

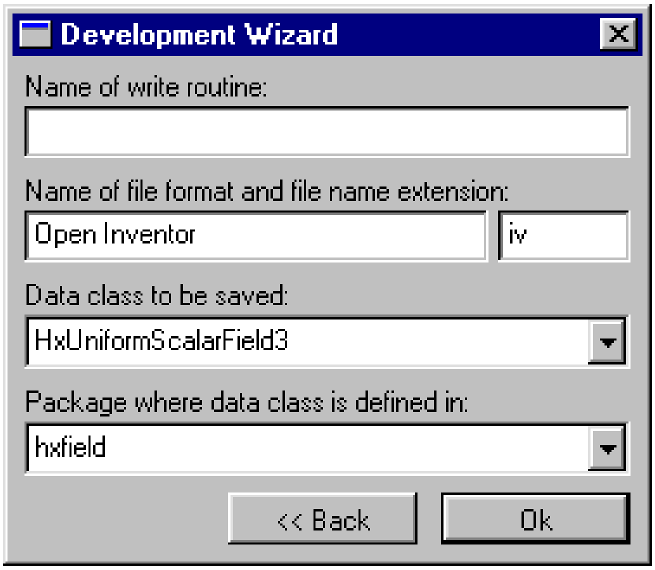
Finally, the C++ class name of the data object to be saved must be chosen as well as the package this class is defined in. Some important data objects such as a `HxUniformScalarField3` or a `HxSurface` are already listed in the corresponding combo box. However, any other class, including custom classes, may be specified here. Instead of the name of a data class, even the name of an interface class such as `HxLattice3` may be used (see Section 15.2.1).

After you press *OK*, a new file `<name>.cpp` will be created in the package directory, where `<name>` denotes the name of the write routine. In addition, the write routine will be registered in the package resource file.

Remember, that after you have added a new write routine to a package you need to recreate the build system files before you can compile it. Details are described in Section 12.9.

12.9 Creating the Build System Files

Before you can actually compile your own packages you need to create project files for Visual Studio on Windows or GNUmakefiles for Unix. These files contain information such as the source code files to be compiled, or the correct include and library paths. Since it is not trivial to set up and edit these


Development Wizard

Name of write routine:

Name of file format and file name extension: Open Inventor | iv

Data class to be saved: HxUniformScalarField3

Package where data class is defined in: hxfield

Figure 12.7: Creating the template of a write routine.

files manually, amira provides a mechanism to create them automatically. In order to do this, a so-called Package file must exist in each package. The Package files contains the name of the package and a list of dependent packages. It may also contain additional tags to customize the build process. The syntax of the package file is described in Section 12.10. However, usually there is no need to modify the default Package file created by the development wizard.

While the automatic generation of the build system files is a very helpful feature it also means that you better do not modify the resulting project or GNUmakefiles manually, because they might be easily overwritten by amira.

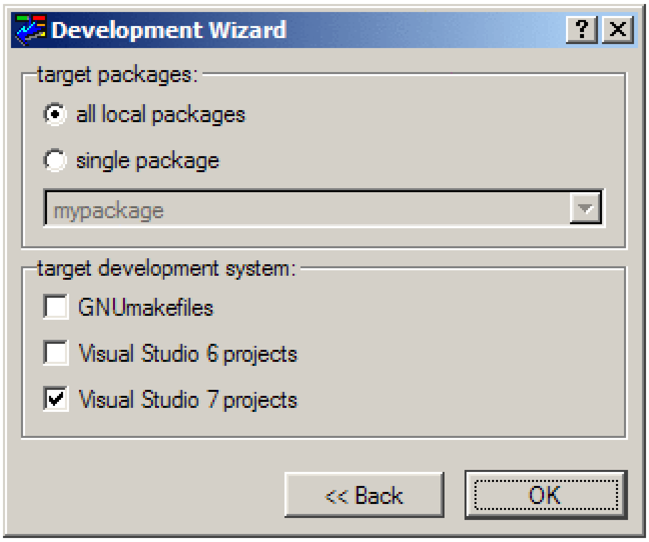
If you select *Create build system* on the main page of the development wizard and then press the *Next* button, the controls shown in Figure 12.8 will be activated. You can choose if you want to create the build system files for all local packages or just for a particular one. Depending on the selected build system the following files will be created:

GNUmakefiles

`share/src/mypackage/GNUmakefile`: A GNUmakefile for building *mypackage*. If *all local packages* is selected such a file will be created in every subdirectory containing a *Package file*.

In order to compile all local packages at once, you can type *gmake* in the local amira directory. In this directory there is a standard GNUmakefile which calls *gmake* in all package directories.

Visual Studio 6


Development Wizard

target packages:

● all local packages

○ single package

mypackage

target development system:

☐ GNUmakefiles

☐ Visual Studio 6 projects

☑ Visual Studio 7 projects

Figure 12.8: Creating the build system fi les.

`AmiraLocal.dsw`: A workspace containing projects for all local packages. This file will only be written if *all local packages* is selected in the development wizard.

`allAmiraLocal.dsp`: A project file which depends on all other projects. Choose this project as active project in Visual Studio if you want to compile all local packages.

`docAmiraLocal.dsp`: A project for creating the documentation for all local packages. This file will only be written if *all local packages* is selected in the development wizard.

`share/src/mypackage/mypackage.dsp`: A project for building *mypackage*. If *all local packages* is selected such a file will be created in every subdirectory containing a Package file.

Visual Studio .NET 2003

`AmiraLocal.sln`: A solution file containing projects for all local packages. This file will only be written if *all local packages* is selected in the development wizard.

`allAmiraLocal.vcproj`: A project file which depends on all other projects. Building this projects causes all other local packages to be build.

`docAmiraLocal.vcproj`: A project for creating the documentation for all local packages. This file will only be written if *all local packages* is selected in the development wizard.

`share/src/mypackage/mypackage.vcproj`: A project for building *mypackage*. If *all local packages* is selected such a file will be created in every subdirectory containing a Package file.

The syntax of the *Package* file is described in the following section.

12.10 The Package File Syntax

The Package file contains information about a local package. From this file Visual Studio project files or GNUmakefiles can be generated. The Package file is a Tcl file. It defines a set of Tcl variables indicating things like the name of the package, dependent libraries, or additional files to be copied when building the package. The default Package file created by the Development Wizard looks as follows:

```
set PACKAGE {mypackage}

set LIBS {
    hxplot hxtime hxsurface hxcolor hxfield
    Amira amiramesh mclib oiv tcl
}

set SHARE {
    share/resources/mypackage.rc
}
```

In most cases the default file works well and need not to be modified. However, in order to accomplish special tasks the default values of the variables can be changed or additional variables can be defined. Here is a detailed list describing the meaning of the different variables:

PACKAGE

The variable `PACKAGE` indicates the name of the package. This should be the same as the name of the package directory. The package name must not contain any characters other than letters or digits.

LIBS

Lists all libraries the package depends on. On default, the most common amira packages are inserted here. You can modify this list as needed. For example, if you want to link against a library called *foo.lib* on Windows or *libfoo.so* on Unix, you should add *foo* to `LIBS`.

In addition to a real library name you may use the following aliases in the `LIBS` variable:

`oiv` - for the Open Inventor libraries
`tcl` - for the Tcl library
`opengl` - for the OpenGL library
`qt` - for the Qt library (not included in amiraDev)

If you want to link against a library only on a particular platform, you can set a dedicated variable `LIBS-arch`, where `arch` denotes the platform. You may further distinguish between the debug and release version of the code. Here is an example:

```
set LIBS {mclib amiramesh schedule hxz qt oiv opengl tcl}
set LIBS-Unix {hxgfxinit}
set LIBS-Win32 {hxgfxinit}
set LIBS-Win32-Debug {msvcrtd mpr}
set LIBS-Win32-Optimize {msvcrt mpr}
```

SHARE

Lists all files which should be copied from the package directory into the local amira directory. On default, only the package resource file will be copied. However, you may add additional files here if necessary. Instead of explicit file names you may use wildcards. These will be resolved using the standard Tcl command `glob`. For example, if you have some demo scripts in your package you could copy them in the following way:

```
set SHARE {
    share/resources/mypackage.rc
    share/demo/mydemos/*.hx
}
```

As for the `LIBS` variable you may append an arch string here, i.e., `SHARE-arch`. The files then will only be copied on the specified platforms.

INCLUDES

This variable may contain a list of additional include paths. These paths are used by the compiler to locate header files. On default, the include path is set to $AMIRA_ROOT/include, $AMIRA_ROOT/include/oiv, $AMIRA_LOCAL/src, and the local package directory.

COPY

This may contain a list of files which are copied from a location other than the local package directory. You need to specify the name of the target file followed by the name of the destination file relative to the local amira directory. For example, you may want to copy certain data files from some archive into the amira directory. This can be achieved in the following way.

```
set COPY {
    D:/depot/data/28523763.dcm data/test
    D:/depot/data/28578320.dcm data/test
    D:/depot/data/28590591.dcm data/test
}
```

As for the `LIBS` variable you may append an arch string here, i.e., `COPY-arch`. The files then will only be copied on the specified platforms. A common application is to copy external libraries required on a particular platform into the amira directory.

SRC

This variable specifies the source code files to be compiled for the package. The default value of this variable is

```
set SRC {*.cpp *.c}
```

This means, that on default all .cpp and .c files in the local package directory will be compiled. Sometimes you may want to replace this default by an explicit list of source files.

Again, you may append an arch string to the `SRC` variable, so that certain files will only be compiled on a particular platform.

INCSRC

This variable specifies the header files to be included into the package project file. The default value of this variable is

```
set INCSRC {*.h *.hpp}
```

This means, that on default all .h and .hpp files in the local package directory will be considered.

Again, you may append an arch string to the `INCSRC` variable, so that certain header files will only be considered on a particular platform.

Chapter 13

File I/O

This chapter describes how user-defined read and write routines can be added to amira. The purpose of custom read and write routines is to add support for file formats not available in amira.

First, some general hints on file formats are given. Then we discuss how read routines are expected to look in amira. Write routines are treated subsequently. Finally, the AmiraMesh API is discussed. Using this API, file I/O for new custom data objects can be implemented rather easily.

13.1 On file formats

Before going into detail, let us clarify some general concepts. In amira, all data loaded into the system are encapsulated in C++ *data classes*. Chapter 15 provides more information about the standard data classes. For example, there is a class to represent tetrahedral grids (*HxTetraGrid*), a separate one for scalar fields defined on tetrahedral grids (*HxTetraScalarField*), and another one for 3D image data (*HxUniformScalarField3*). Every instance of a data class is represented by a green icon in the amira object pool.

The way in which data are stored in a disk file is called a file format. Although there is a relationship between data classes and file formats, these are two different things. It is especially important to understand that there is no one-to-one correspondence between them.

Typically, a specific data class (like 3D image data) can be stored in many different file formats (3D TIFF, DICOM, a set of 2D JPEG files, and so on). On the other hand, a specific file format does not necessarily correspond to exactly one data class. For example, a data file in Fluent UNS format can either contain hexahedral grids (*HxHexaGrid*) or tetrahedral grids (*HxTetraGrid*).

Note that there is also no one-to-one correspondence between the instance of a data class (a green icon in amira) and the instance of a file format (the actual file). Often multiple files correspond to a single data object, for example 2D images forming a single 3D image volume. On the other hand, a single file can contain the data of multiple data objects. For example, an AVS UCD file can contain a tetrahedral grid as well as multiple scalar fields defined on it.

Finally, note that information may get lost when saving a data object to a file in a specific format. For example, when saving a 3D image volume to a set of 2D JPEG images, the bounding box information will be lost. Likewise, there are user-defined parameters or attributes in amira that cannot be encoded in most standard file formats. On the other hand a file reader often does not interpret all information provided by a specific file format.

13.2 Read Routines

As already mentioned in the previous section, a read routine is a C++ function that reads a disk file, interprets the data, creates an instance of an amira data class, and fills that instance with the data read from the file.

In order to write a read routine, obviously two things are needed, namely a specification of the file format to be read as well as the information which of amira's data classes is able to represent the data and how this class is used. More information about the standard amira data classes is given in Chapter 15. The C++ interface of these classes is described in the online reference documentation.

A read routine may either be a static member function of a class or a global function. In addition to the function itself, an entry in the package resource file is needed. In this way amira is informed about the existence of the read routine and about the type of files that can be handled by the reader.

In the following discussion the implementation of a user-defined read routine will be illustrated by two concrete examples, namely a simple read routine for scalar fields and a read routine for surfaces and surface fields. Some more details about read routines will be discussed subsequently.

13.2.1 A Reader for Scalar Fields

In this section we present a simple read routine designed for reading image volumes, i.e., 3D scalar fields, from a very simple file format, which we have invented for this example. The file format is called PPM3D (because it is similar to the ppm 2D image format). The PPM3D format will be an ASCII file format containing a header, three integer numbers specifying the size of the 3D image volume, and the pixel data as integer numbers in the range 0 to 255. An example file could look like this:

```
# PPM3D
4 4 3
43 44 213 9 23 234 3 3 3 44 213 9 23 234 36 63
44 213 9 23 234 35 3 5 44 213 9 23 234 31 13 12
44 213 9 23 234 35 3 5 44 213 9 23 234 31 13 12
```

The full source code of the read routine is contained in the demo package provided with the amira developer version. In order to follow the example below, first create a local amira directory using the Development Wizard. Be sure that the toggle *copy demo package* is activated, as described in Section 12.2. The read routine can then be found in the local amira directory under `packages/mypackage/readppm3d.cpp`.

Let us first take a look at the commented source code of the reader. Some general remarks follow below.

```
/////////////////////////////////////////////////////////////////
//
// Sample read routine for the PPM3D file format
//
/////////////////////////////////////////////////////////////////

#include <Amira/HxMessage.h>
#include <hxfield/HxUniformScalarField3.h>
#include <mypackage/mypackageAPI.h>

MYPACKAGE_API int readppm3d(const char* filename)
{
    FILE* f = fopen(filename, "r"); // open the file

    if (!f) {
        theMsg->ioError(filename);
        return 0; // indicate error
    }

    // Skip header (first line). We could do some checking here:
    char buf[80];
    fgets(buf, 80, f);

    // Read size of volume:
    int dims[3];
    dims[0] = dims[1] = dims[2] = 0;
    fscanf(f, "%d %d %d", &dims[0], &dims[1], &dims[2]);

    // Do some consistency checking.
    if (dims[0]*dims[1]*dims[2] <= 0) {
        theMsg->printf("Error in file %s.", filename);
        fclose(f);
        return 0;
    }

    // Now create 3D image data object. The constructor takes
    // the dimensions and the primary data type. In this case
    // we create a field containing unsigned bytes (8 bit).
    HxUniformScalarField3* field =
        new HxUniformScalarField3(dims, McPrimType::mc_uint8);

    // The HxUniformScalarField3 stores its data in a member
    // variable called lattice. We know that the data is unsigned
    // 8 bit because we specified this in the constructor.
    unsigned char* data =
        (unsigned char*) field->lattice.dataPtr();

    // Now we must read dims[0]*dims[1]*dims[2] data values
    for (int i=0; i<dims[0]*dims[1]*dims[2]; i++) {
        int val=0;
        fscanf(f,"%d",&val);
        data[i] = (unsigned char) val;
    }

    // We are done with reading, close the file.
    fclose(f);

    // Register the data object to make it visible in the
    // object pool. The name for the new object is automatically
    // generated from the filename.
    HxData::registerData(field, filename);

    return 1; // Indicate success
}
```

The source file starts with some includes. First, the file *HxMessage.h* is included. This header file provides the global pointer *theMsg* which allows us to print out text messages in the amira console window. In our read routine we use *theMsg* to print out error messages if a read error occurred.

Next, the header file containing the declaration of the data class to be created is included, i.e., *HxUniformScalarField3.h*. As a general rule, every class in amira is declared in a separate header file. The name of the header file is identical to the name of the C++ class.

Finally, the file *mypackageAPI.h* is included. This file provides import and export storage-class specifiers for Windows systems. These are encoded in the macro `MYPACKAGE_API`. On Unix systems this macro is empty and can be omitted.

The read routine itself takes one argument, the name of the data file to be read. It should return 1 on success, or 0 if an error occurred and no data object could be created. The body of the read routine is rather straightforward. The file is opened for reading. The size of the image volume is read. A new data object of type *HxUniformScalarField3* is created and the rest of the data is written into the data object. Finally, the file is closed again and the data object is put into the object pool by calling `HxData::registerData`. In principle, all read routines look like this example. Of course, the type of data object being created and the way that this object is initialized may differ.

In order to make the new read routine known to amira, an entry must be added to the package resource file, i.e., to the file `mypackage/share/resources/mypackage.rc`. In our case this entry looks as follows:

```
dataFile -name "PPM3D Demo Format"  \
    -header "PPM3D"                 \
    -load "readppm3d"               \
    -package "mypackage"
```

The `dataFile` command registers a new file format called *PPM3D Demo Format*. The option `-header` specifies a regular expression which is used for automatic file format detection. If the first 64 bytes of a file match this expression, the file will be automatically loaded using this read routine. Of course, some data formats do not have a unique file header. In this case, the format may also be detected from a standard file name extension. Such an extension may be specified using the `-ext` option of the `dataFile` command. Multiple extensions can be specified as a comma-separated list. The actual C++ name of the read routine is specified via `-load`. Finally, the package containing the read routine must be specified using the `-package` option.

If you have compiled the example in the mypackage demo package, you can try to load the demo file `mypackage/data/test.ppm3d`. As you will see, the file browser automatically detects the file format and displays *PPM3D Demo Format* in its file list.

13.2.2 A Reader for Surfaces and Surface Fields

Now that you know what a read routine looks like in principle, let us consider a more complex example. In this section we discuss a read routine which creates more than one data object. In particular, we want to read a triangular surface mesh from a file. In addition to the surface description, the file may also contain data values for each vertex of the surface. Data defined on a surface mesh are represented by separate classes in amira. Therefore, the reader must first create a data object representing the surface only. Then appropriate data objects must be created for each surface field.

Again, the file format is quite simple and has been invented for the purpose of this example. We call it the *Trimesh* format. It is a simple ASCII format without any header. The first line contains the number of points and the number of triangles. Then the x-, y-, and z-coordinates of the points are listed. This section is followed by triangle specifications consisting of three point indices for each triangle, point count starts at one. The next section is for vertex data, starting with a line that contains an arbitrary number of integers. Each integer indicates that there is a data field with a certain number of variables defined on the surface's vertices, e.g., 1 for a scalar field or 3 for a vector field. The data values for each vertex follow in separate lines. Here is a small example containing a single scalar surface field:

```
4 2
0.0 0.0 0.0
1.0 0.0 0.0
0.0 1.0 0.0
1.0 1.0 0.0
1 2 4
1 4 3
1
0.0
0.0
1.0
1.0
```

You can find the full source code of the reader in the local amira directory under `packages/mypackage/readtrimesh.cpp`. Remember that the demo package must have been copied into the local amira directory before compiling. For details, refer to Section 12.2. Let us now look at the complete read routine before discussing the details:

```
/////////////////////////////////////////////////////////////////
//
// Read routine for the Trimesh file format
//
/////////////////////////////////////////////////////////////////

#include <McStringTokenizer.h>
#include <Amira/HxMessage.h>
#include <hxsurface/HxSurface.h>
#include <hxsurface/HxSurfaceField.h>
#include <mypackage/mypackageAPI.h>
```

```
MYPACKAGE_API int readtrimesh(const char* filename)
{
    FILE* fp = fopen(filename, "r");

    if (!fp) {
        theMsg->ioError(filename);
        return 0;
    }

    // 1. Read the surface itself

    char buffer[256];
    fgets(buffer,256,fp); // read first line

    int i, j, k, nPoints=0, nTriangles=0;
    // Get number of points and triangles
    sscanf(buffer, "%d %d", &nPoints, &nTriangles);

    if (nPoints<0 || nTriangles<0) {
        theMsg->printf("Illegal number of points or triangles.");
        fclose(fp);
        return 0;
    }

    HxSurface* surface = new HxSurface; // create new surface
    surface->addMaterial("Inside",0); // add some materials
    surface->addMaterial("Outside",1);

    HxSurface::Patch* patch = new HxSurface::Patch;
    surface->patches.append(patch); // add patch to surface
    patch->innerRegion = 0;
    patch->outerRegion = 1;

    surface->points.resize(nPoints);
    surface->triangles.resize(nTriangles);

    for (i=0; i<nPoints; i++) { // read point coordinates
        McVec3f& p = surface->points[i];
        fgets(buffer,256,fp);
        sscanf(buffer, "%g %g %g", &p[0], &p[1], &p[2]);
    }

    for (i=0; i<nTriangles; i++) { // read triangles
        int idx[3];
        fgets(buffer,256,fp);
        sscanf(buffer, "%d %d %d", &idx[0], &idx[1], &idx[2]);

        Surface::Triangle& tri = surface->triangles[i];
        tri.points[0] = idx[0]-1; // indices should start at zero
        tri.points[1] = idx[1]-1;
        tri.points[2] = idx[2]-1;
        tri.patch = 0;
    }
    // Add all triangles to the patch
    patch->triangles.resize(nTriangles);
    for (i=0; i<nTriangles; i++)
        patch->triangles[i] = i;

    // Add surface to object pool
    HxData::registerData(surface,filename);

    // 2. Check if file also contains data fields

    fgets(buffer,256,fp);
    McStringTokenizer tk(buffer);
    McDArray<HxSurfaceField*> fields;

    while (tk.hasMoreTokens()) { // are there any numbers here ?
        int n = atoi(tk.nextToken());
        // Create field with desired number of components
        HxSurfaceField* field = HxSurfaceField::create(surface,
            HxSurfaceField::OnNodes, n);
        fields.append( field );
    }

    if (fields.size()) {
        // Read data values for all fields
        for (i=0; i<nPoints; i++) {
            fgets(buffer,256,fp);
            tk = buffer;
            for (j=0; j<fields.size(); j++) {
                int n = fields[j]->nDataVar();
                float* v = &fields[j]->dataPtr()[i*n];
                for (k=0; k<n; k++)
                    v[k] = atof(tk.nextToken());
            }
        }

        // Add all fields to object pool
        for (i=0; i<fields.size(); i++) {
            HxData::registerData(fields[i], NULL);
            fields[i]->composeLabel(surface->getName(),"data");
        }
    }

    fclose(fp); // close file and return ok
    return 1;
}
```

The first part of the read routine is very similar to the PPM3D reader outlined in the previous section. Required header files are included, the file is opened, the number of points and triangles are read, and a consistency check is performed.

Then an amira surface object of type *HxSurface* is created. The class *HxSurface* has been devised to represent an arbitrary set of triangles. The triangles are organized into *patches*. A patch can be thought of as the boundary between two volumetric regions, an “inner” and an “outer” region. Therefore, for each patch an inner region and an outer region should be defined. In our case, all triangles will be inserted into a single patch. After this patch has been created and initialized, the number of points and triangles is set, i.e., the dynamic arrays *points* and *triangles* are resized appropriately.

Next, the point coordinates and the triangles are read. Each triangle is defined by the three points it consists of. The point indices start at one in the file but should start at zero in the *HxSurface* class. Therefore all indices are decremented by one. Once all triangles have been read, they are inserted into the patch we have created before. The surface is now fully initialized and can be added to the object pool by calling *HxData::registerData*.

The second part of the read routine is reading the data values. First, we check how many data fields are defined and how many data variables each field has. In order to parse this information, we use the utility class *McStringTokenizer*. This class returns blank-separated parts of a string one after the other. For more information about this and other utility classes refer to the online reference documentation of the amira developer version.

For each group of data variables, a corresponding surface field is created. The fields are temporarily stored in the dynamic array *fields*. Instead of directly calling the constructor of the class *HxSurfaceField*, the static method *HxSurfaceField::create* is used. This method checks the number of data variables and automatically creates an instance of a subclass such as *HxSurfaceScalarField* or *HxSurfaceVectorField*, if this is possible. In principle, surface fields may store data on a per-node or a per-triangle basis. Here we are dealing with vertex data, so we specify the encoding to be *HxSurfaceField::OnNodes* in *HxSurfaceField::create*.

Finally, the data values are read into the surface fields created before. Afterwards, all the fields are added to the object pool by calling *HxData::registerData* again. In order to define a useful name for the surface fields, we call the method *composeLabel*. This method takes a reference name, in this case the name of the surface, and replaces the suffix by some other string, in this case ”data”. amira automatically modifies the name so that it is unique. Therefore we can perform the same replacement for all surface fields.

Like any other read routine, our *Trimesh* reader must be registered in the package resource file before it can be used. This is done by the following statement in `mypackage/share/resources/mypackage.rc`:

```
dataFile -name "Trimesh Demo Format"    \
    -ext "trimesh,tm"                   \
    -load "readtrimesh"                 \
    -package "mypackage"
```

Most of the options of the `dataFile` command have already been explained in the previous section. However, in contrast to the PPM3D format, the *Trimesh* format cannot be identified by its file header. Therefore, we use the `-ext` option to tell amira that all files with file name extensions *trimesh* or *tm* should be opened using the *Trimesh* reader.

13.2.3 More About Read Routines

The basic structure of a read routine should be clear from the examples presented in the previous two sections. Nevertheless, there are a few more things that might be of interest in some situations. These will be discussed in the following.

Reading Multiple Images At Once

The amira file browser allows you to select multiple files at once. Usually, all these files are opened one after the other by first determining the file format and then calling the appropriate read routine. However, in some cases the data of a single amira data object are distributed among multiple files. The most prominent example is 3D images where every slice is stored in a separate 2D image file. In order to be able to create a full 3D image, the file names of all the individual 2D images must be available to a read routine. To facilitate this, read routines in amira can have two different signatures. Besides the ordinary form

```
int myreader(const char* filename);
```

read routines can also be defined as follows:

```
int myreader(int n, const char** filenames);
```

In both cases exactly the same `dataFile` command can be used in the package resource file. amira automatically detects whether a read routine takes a single file name as an argument or multiple ones. In the latter case, the read routine is called with the names of all files selected in the file browser, provided all these files have the same file format (if multiple files with different formats are selected, the read routine for each format is called with the matching files only). You can create the template of a multiple files read routine by selecting the toggle *create reader for multiple files* in the Development Wizard (see Section 12.7).

The Load Command

The current state of the amira network with all its data objects and modules can be stored in a script file. When executed, the script should restore the network again. Of course, this is a difficult task especially if data objects have been modified since they have been loaded from files. However, even if this is not the case, amira must know how to reload the data later on.

For this purpose a special parameter called *LoadCmd* should be defined for the data object. This parameter should contain a Tcl command sequence which restores the data object on execution. Usually, the load command is simply set to `load <filename>` when calling `HxData::registerData`. However, this approach fails if the format of the file cannot be detected automatically, or if multiple data objects are created from a single file, e.g., as in our *Trimesh* example.

In such cases the load command should be set manually. In case of the *Trimesh* reader, this could be done by adding the following lines of code at the very end of the routine just before the method's returning point:

```
McString loadCmd;
loadCmd.printf("set TMPIO [load -trimesh %s]\n"
               "lindex $TMPIO 0", filename);
surface->setLoadCmd(loadCmd,1);

for (int i=0; i<fields.size(); i++) {
    loadCmd.printf("lindex $TMPIO %d", i+1);
    fields[i]->setLoadCmd(loadCmd,1);
}
```

This code requires some explanation. The file is loaded and all data objects are created when the first line of the load command is executed. Note that we specified the `-trimesh` option as an argument of `load`. This ensures that the *Trimesh* reader will always be used. The format of the file to be loaded will not be determined automatically. The Tcl command `load` returns a list with the names of all data objects which have been created. This list is stored in the variable TMPIO. Later the names of the individual objects can be obtained by extracting the corresponding elements from this list. This is done using the Tcl command `lindex`.

Using Dialog Boxes in a Read Routine

In some cases a file cannot be read successfully unless certain parameters are interactively specified by the user. Usually this means that a special-purpose dialog must be popped up within the read routine. This is done, for example, when raw data are read in amira. In order to write your own dialogs, you must use Qt, a platform-independent toolkit for designing graphical user interfaces. Qt is not included with the developer version of amira. However, once you have it installed on your system, you can easily use it to create custom dialogs in amira.

If you don't have Qt or if you don't want to use it, you may consider implementing your read routine within an ordinary module. Although this somewhat breaks amira's data import concept, it will work too, of course. You then can utilize ordinary ports to let the user specify required import parameters.

13.3 Write Routines

Like read routines, write routines in amira are C++ functions, either global ones or static member functions of an arbitrary class. In the following discussion we present write routines for the same two formats for which reader codes have been explained in the previous section. First, a writer for scalar fields will be discussed, then a writer for surfaces and surface fields.

13.3.1 A Writer for Scalar Fields

In this section we explain how to implement a routine for writing 3D images, i.e., instances of the class *HxUniformScalarField3*, to a file using the PPM3D format introduced in Section 13.2.1. The writer is even simpler than the reader. Again, the source code is contained in the demo package of the amira developer version. Once you have created a local amira directory using the Development Wizard and copied the demo package into that directory, you will find the write routine in the local amira directory under `packages/mypackage/writeppm3d.cpp`. Here it is:

```
/////////////////////////////////////////////////////////////////
//
// Sample write routine for the PPM3D file format
//
/////////////////////////////////////////////////////////////////

#include <Amira/HxMessage.h>
#include <hxfield/HxUniformScalarField3.h>
#include <mypackage/mypackageAPI.h>

MYPACKAGE_API
int writeppm3d(HxUniformScalarField3* field, const char* filename)
{
    // For the moment we only want to support byte data
    if (field->primType() != McPrimType::mc_uint8) {
        theMsg->printf("This format only supports byte data.");
        return 0; // indicate error
    }

    FILE* f = fopen(filename, "w"); // open the file

    if (!f) {
        theMsg->ioError(filename);
        return 0; // indicate error
    }

    // Write header:
    fprintf(f, "# PPM3D\n");

    // Write fields dimensions:
    const int* dims = field->lattice.dims();
    fprintf(f, "%d %d %d\n", dims[0], dims[1], dims[2]);

    // Write dims[0]*dims[1]*dims[2] data values:
    unsigned char* data =
        (unsigned char*) field->lattice.dataPtr();

    for (int i=0; i<dims[0]*dims[1]*dims[2]; i++) {
        fprintf(f, "%d ", data[i]);
        if (i%20 == 19) // do some formatting
            fprintf(f,"\n");
    }

    // Close the file.
    fclose(f);

    return 1; // indicate success
}
```

At the beginning, the same header files are included as in the reader. *HxMessage.h* provides the global pointer *theMsg* which allows us to print out text messages in the amira console window. *HxUniformScalarField3.h* contains the declaration of the data class to be written to the file. Finally, *mypack-*

ageAPI.h provides import and export storage-class specifiers for Windows systems. These are encoded in the macro `MYPACKAGE_API`. On Unix systems, this macro is empty and can be omitted.

The signature of a write routine differs from that of a read routine. It takes two arguments, namely a pointer to the data object to be written to a file, as well as the name of the file. Before a write routine is called, amira always checks if the specified file already exists. If this is the case, the user is asked if the existing file should be overwritten. Therefore, such a check need not to be coded again in each write routine. Like a read routine, a write routine should return 1 on success, or 0 if an error occurred and the data object could not be saved.

The body of the write routine is almost self-explanatory. At the beginning, a check is made whether the 3D image really consists of byte data. In general, the type of data values of such an image can be 8-bit bytes, 16-bit shorts, 32-bit integers, floats, or doubles. If the image does contain bytes, a file is opened and the image contents are written into it. However, note that the data object also contains information which cannot be stored using our simple PPM3D file format. First of all, this applies to the bounding box of the image volume, i.e., the position of the center of the first and the last voxel in world coordinates. Also, all parameters of the object (defined in the member variable *parameters* of type *HxParamBundle*) will be lost if the image is written into a PPM3D file and read again.

Like a read routine, a write routine must be registered in the package resource file, i.e., in `mypackage/share/resources/mypackage.rc`. This is done by the following statement:

```
dataFile -name "PPM3D Demo Format"    \
    -save "writeppm3d"                \
    -type "HxUniformScalarField3"     \
    -package "mypackage"
```

The option `-save` specifies the name of the write routine. The option `-type` specifies the C++ class name of the data objects which can be saved using this format. Note that an export format may be registered for multiple C++ objects of different type. In this case multiple `-type` options should be specified. However, for each type there must be a separate write routine with a different signature (polymorphism). For example, if we additionally want to register the PPM3D format for objects of type *HxStackedScalarField3*, we must additionally implement the following routine:

```
int writeppm3d(HxStackedScalarField3* field, const char* fname);
```

Besides the standard data classes, there are so-called *interface classes* that may be specified with the `-type` option. For example, in this way it is possible to implement a generic writer for n-component regular 3D fields. Such data is encapsulated by the interface *HxLattice3*. For more information about interfaces, refer to Chapter 15.

At this point you may try to compile and execute the write routine by following the instructions given in Section 11.5 (Compiling and Debugging).

13.3.2 A Writer for Surfaces and Surface Fields

For the sake of completeness, a writer for the *Trimesh* format introduced in Section 13.2.2 is described in this section. Remember that the *Trimesh* format is suitable for storing a triangular mesh as well as an arbitrary number of data values defined on the vertices of the surface. In amira, surfaces and data fields defined on surfaces are represented by different objects. This also has some implications when designing a write routine.

In our example we actually implement two different write routines, one for the surface and one for the surface field. If the user selects the surface and exports it using the `Trimesh` writer, the surface mesh as well as all attached data fields will be written to file. On the other hand, if the user selects a particular surface field, the corresponding surface and just the selected field will be written.

The source code of the writer can be found in the local amira directory under `packages/mypackage/writetrimesh.cpp`. Remember that the demo package must be copied into the local amira directory before compiling. For details refer to Section 12.2. Again, let us start by looking at the code:

```
/////////////////////////////////////////////////////////////////
//
// Write routine for the Trimesh file format
//
/////////////////////////////////////////////////////////////////

#include <Amira/HxMessage.h>
#include <hxsurface/HxSurface.h>
#include <hxsurface/HxSurfaceField.h>
#include <mypackage/mypackageAPI.h>

static
int writetrimesh(HxSurface* surface,
    McDArray<HxSurfaceField*> fields, const char* filename)
{
    FILE *f = fopen(filename, "w");

    if (!f) {
        theMsg->ioError(filename);
        return 0;
    }

    int i,j,k;
    McDArray<McVec3f>& points = surface->points;
    McDArray<Surface::Triangle>& triangles = surface->triangles;

    // Write number of points and number of triangles
    fprintf(f, "%d %d\n", points.size(), triangles.size());

    // Write point coordinates
    for (i=0; i<points.size(); i++) {
        McVec3f& v = points[i];
        fprintf(f, "%g %g %g\n", v[0], v[1], v[2]);
    }

    // Write point indices of all triangles
    for (i=0; i<triangles.size(); i++) {
        int* idx = triangles[i].points;
        fprintf(f, "%d %d %d\n", idx[0]+1, idx[1]+1, idx[2]+1);
    }

    // If there are data fields write them out too.
    if (fields.size()) {
        for (j=0; j<fields.size(); j++)
            fprintf(f, "%d ", fields[j]->nDataVar());
        fprintf(f, "\n");

        for (i=0; i<points.size(); i++) {
            for (j=0; j<fields.size(); j++) {
                int n = fields[j]->nDataVar();
                float* v = &fields[j]->dataPtr()[i*n];
                for (k=0; k<n; k++)
                    fprintf(f, "%g ", v[k]);
            }
            fprintf(f, "\n");
        }
    }

    fclose(f); // done
    return 1;
}

MYPACKAGE_API
int writetrimesh(HxSurface* surface, const char* filename)
{
    // Temporary array of surface data fields
    McDArray<HxSurfaceField*> fields;

    // Check if there are data fields attached to surface
    for (int i=0; i<surface->downStreamConnections.size(); i++) {
        HxSurfaceField* field =
            (HxSurfaceField*) surface->downStreamConnections[i];
        if (field->isOfType(HxSurfaceField::getClassTypeId()) &&
            field->getEncoding() == HxSurfaceField::OnNodes)
            fields.append(field);
    }

    // Write surface and all attached data fields
    return writetrimesh(surface, fields, filename);
}

MYPACKAGE_API
int writetrimesh(HxSurfaceField* field, const char* filename)
{
    // Check if data is defined on nodes
    if (field->getEncoding() != HxSurfaceField::OnNodes) {
        theMsg->printf("Data must be defined on nodes.");
        return 0;
    }

    // Store pointer to field in dynamic array
    McDArray<HxSurfaceField*> fields;
    fields.append(field);

    // Write surface and this data field
    return writetrimesh(field->surface(), fields, filename);
}
```

In the upper part of the code, first a static utility method is defined which takes three arguments: a pointer to a surface, a dynamic array of pointers to surface fields, and a file name. This is the function that actually writes the data to a file. Once you have understood the *Trimesh* reader presented in Section 13.2.2, it should be no problem to follow the writer code too.

In the lower part of the code, two write routines mentioned above are defined, one for surfaces and the other one for surface fields. Since these routines are to be exported for external use, we need to apply the package macro `MYPACKAGE_API`, at least on Windows.

Let us now look more closely at the surface writer. This routine first collects all surface fields attached to the surface in a dynamic array. This is done by scanning `surface->downStreamConnections` which provides a list of all objects attached to the surface. The class type of each object is checked using the method `isOfType`. This sort of dynamic type-checking is the same as in Open Inventor. If a surface field has been found and if it contains data defined on its nodes, it is appended to the temporary array `fields`. The surface itself, as well as the collected fields, are then written to file by calling the utility method defined in the upper part of the writer code.

The second write routine, the one adapted to surface fields, is simpler. Here a dynamic array of fields is used too, but this array is filled with data representing the original surface field only. Once this has been done, the same utility method can be called as in the first case.

Although actually two write routines have been defined, only one entry in the package resource file is required. This entry looks as follows (see `mypackage/share/resources/mypackage.rc`):

```
dataFile -name "Trimesh Demo Format"    \
    -ext "trimesh"                      \
    -type "HxSurface"                   \
    -type "HxSurfaceField"              \
    -save "writetrimesh"                \
    -package "mypackage"
```

In order to compile and execute the write, please follow the instructions given in Section 11.5 (Compiling and Debugging).

13.4 The AmiraMesh API

Besides many standard file formats, amira also provides its own native format called AmiraMesh. The AmiraMesh file format is very flexible. It can be used to save many different data objects including image data, finite-element grids, and solution data defined on such grids. Among other features it supports ASCII or binary data encoding, data compression, and storage of arbitrary parameters. The format itself is described in more detail in the reference section of the users guide. In this section we want to discuss how to save custom data objects in AmiraMesh format. For this purpose a special C++ utility class called `AmiraMesh` is provided. Using this class, reading and writing AmiraMesh files becomes very easy.

Below we will first provide an overview of the AmiraMesh API. After that, we present two simple examples. In the first one we show how colormaps are written in AmiraMesh format. In the second one we show how such colormaps are read back again.

13.4.1 Overview

The AmiraMesh API consists of a single C++ class. This class is called `AmiraMesh` as is the file format itself. It is defined in the header file `include/amiramesh/AmiraMesh.h` located in the amira root directory. The class is designed to completely represent the information stored in an AmiraMesh file in memory. When reading a file first an instance of an `AmiraMesh` class is created. This instance can then be interpreted and the data contained in it can be copied into a matching amira data object. Likewise, when writing a file, first an instance of an `AmiraMesh` class is created and initialized with all required information. Then this instance is written to file simply by calling a member method.

If you look at the header file or at the `AmiraMesh` class documentation, you will notice that there are four public member variables called `parameters`, `locationList`, `dataList`, and `field-List`. These variables completely store the information contained in a file. The first variable is of type `HxParamBundle`. Like in an amira data object, it is used to store an arbitrary hierarchy of parameters. The other three member variables are dynamic arrays of pointers to locally defined classes. The most important local classes are `Location` and `Data`, which are stored in `locationList` and `dataList`, respectively.

A `Location` defines the name of a single- or multi-dimensional array. It does not store any data by itself. This is done by a `Data` class. Every `Data` class must refer to some `Location`. For example, when writing a tetrahedral grid, we may define two different one-dimensional locations, one called *Nodes* and the other one called *Tetrahedra*. On the nodes we define a `Data` instance for storing the x-, y-, and z-coordinates of the nodes. Likewise, on the tetrahedra we define a `Data` instance for storing the indices of the four points of a tetrahedron.

As stated in the `AmiraMesh` class documentation, the *Data* class can take a pointer to some already existing block of memory. In this way it is prevented that all data must be copied before it is written to file. In order to write compressed data, the member method `setCodec` has to be called. Currently, two different compression schemes are supported. The first one, called *HxByteRLE,* implements simple run-length encoding on a per-byte basis. The second one, called *HxZip*, uses a more sophisticated compression technique provided by the external *zlib* library. In any case, the data will be automatically uncompressed when reading an AmiraMesh file.

It should be pointed out that the AmiraMesh file format itself merely provides a method for storing arbitrary data organized in single- or multi-dimensional arrays in a file. It does not specify anything about the semantics of the data. Therefore, when reading an AmiraMesh file it is not clear what kind of data object should be created from it. To facilitate file I/O of custom data objects, the actual contents of an AmiraMesh file are indicated by a special parameter called *ContentType*. For each such type, a special read routine is registered. Like an ordinary read routine, an AmiraMesh reader is a global function or a static member method of a C++ class. It has the following signature:

```
int readMyAmiraMesh(AmiraMesh* m, const char* filename);
```

This method is called whenever the *ContentType* parameter matches the the one the read method is registered for. The reader should create an amira data object from the contents of the AmiraMesh class. The filename can be used to define the name of the resulting data object. In order to register an AmiraMesh read routine, a statement similar to the following one must be put into the package resource file:

```
amiramesh -ContentType "MyType" \
    -load "readMyAmiraMesh" \
    -package "mypackage"
```

13.4.2 Writing an AmiraMesh File

As a concrete example, in this section we want to show how a colormap is written in AmiraMesh format. In particular, we consider colormaps of type `HxColormap256`, consisting of N discrete RGBA tuples. Like most other write methods, the AmiraMesh writer is a global C++ function. Let us first look at the code before discussing the details.

```
HXCOLOR_API
int writeAmiraMesh(HxColormap256* map, const char* filename)
{
    float minmax[2];
    minmax[0] = map->minCoord();
    minmax[1] = map->maxCoord();
    int size = map->getLength();

    AmiraMesh m;
    m.parameters = map->parameters;
    m.parameters.set("MinMax", 2, minmax);
    m.parameters.set("ContentType", "Colormap");

    AmiraMesh::Location* loc =
        new AmiraMesh::Location("Lattice", 1, &size);
    m.insert(loc);

    AmiraMesh::Data* data = new AmiraMesh::Data("Data", loc,
        McPrimType::mc_float, 4, (void*) map->getDataPtr());
    m.insert(data);

    if ( !m.write(filename,1) ) {
        theMsg->ioError(filename);
        return 0;
    }

    setLoadCmd(filename);
    return 1;
}
```

In the first part of the routine a variable `m` of type `AmiraMesh` is defined. The parameters of the colormap are copied into `m`. In addition, two more parameters are set. The first one, called *MinMax*, describes the coordinate range of the colormap. The second one indicates the content type of the AmiraMesh file. This parameter ensures that the colormap can be read back again by a matching AmiraMesh read routine (see Section 13.4.3).

Before the RGBA data values can be stored, a `Location` of the right size must be created and inserted into the `AmiraMesh` class. Afterwards, an instance of a `Data` class is created and inserted. The constructor of the `Data` class takes a pointer to the `Location` as an argument. Moreover, a pointer to the RGBA data values is specified. Each RGBA tuple consists of four numbers of type float.

13.4.3 Reading an AmiraMesh File

In the previous section we presented a simple AmiraMesh write routine for colormaps. We now want to read back such files again. For this reason we define a static AmiraMesh read function in class `HxColormap256`. Of course, a global C++ function could be used as well. The read function is registered in the package resource file `hxcolor.rc` in the following way:

```
amiramesh -ContentType "Colormap" \
    -load "HxColormap256::readAmiraMesh" \
    -package "hxcolor"
```

This statement indicates that the static member method `readAmiraMesh` of the class `HxColormap256` defined in package `hxcolor` should be called if the AmiraMesh file contains a parameter *ContentType* equal to `Colormap`. The source code of the read routine looks as follows:

```
int HxColormap256::readAmiraMesh(AmiraMesh* m,
    const char* filename)
{
    for (int i=0; i<m->dataList.size(); i++) {
        AmiraMesh::Data* data = m->dataList[i];
        if (data->location()->nDim() != 1)
            continue;

        if (data->dim()<3 || data->dim()>4)
            continue;

        if (data->primType() != McPrimType::mc_uint8 &&
            data->primType() != McPrimType::mc_float)
            continue;

        int dim = data->dim();
        int size = data->location()->dims()[0];

        HxColormap256* colormap = new HxColormap256(size);
        colormap->parameters = m->parameters;

        switch (data->primType()) {
        case McPrimType::mc_uint8: {
            unsigned char* src =
                (unsigned char*) data->dataPtr();
            for (int k=0; k<size; k++, src+=dim) {
                float a = (dim>3) ? (src[3])/255.0 : 1;
                colormap->setRGBA(k, src[0]/255., src[1]/255.,
                    src[2]/255., a);
            } } break;

        case McPrimType::mc_float: {
            float* src = (float*) data->dataPtr();
            for (int k=0; k<size; k++, src+=dim) {
                float a = (dim>3) ? src[3] : 1;
                colormap->setRGBA(k, src[0], src[1], src[2], a);
            } } break;
        }

        float minmax[2] = { 0,1 };
        m->parameters.findReal("MinMax", 2, minmax);
        colormap->setMinMax(minmax[0], minmax[1]);

        HxData::registerData(colormap, filename);
        return 1;
    }

    return 0;
}
```

Compared to the write routine, the read routine is a little bit more complex since some consistency checks are performed. First, the member `dataList` of the `AmiraMesh` structure is searched for a one-dimensional array containing vectors of three or four elements of type byte or float. This array should contain the RGB or RGBA values of the colormap. If a matching `Data` structure is found, a new instance of type `HxColormap256` is created. The parameters are copied from the `AmiraMesh` class into the new colormap. Afterwards, the actual color values are copied. Although the write routine only exports RGBA tuples of type float, the read routine also supports byte data. For this reason two different cases are distinguished in a switch statement. If the file only contains 3-component data, the opacity value of each colormap entry is set to 1. Finally, the coordinate range of the colormap is set by evaluating the 2-component parameter *MinMax*, and the new colormap is added to the object pool by calling `HxData::registerData`.

Chapter 14

Writing Modules

Besides the data classes, modules are the core of amira. They contain the actual algorithms for visualization and data processing. Modules are instances of C++ classes derived from the common base class `HxModule`.

There are two major groups of modules: compute modules and display modules. The first group usually performs some sort of operation on input data, creates some resulting data object, and deposits the latter in the object pool. In contrast, display modules usually directly visualize their input data. In this chapter both types of modules will be covered in separate sections. For each case a concrete example will be presented and discussed in detail.

In addition, we also discuss the amira Plot API in this chapter. This API makes it possible to create simple line plots or bar charts within a module.

14.1 A Compute Module

As already mentioned *compute modules* usually take one or more input data objects and calculate a new resulting data object from these. The resulting data object is deposited in the object pool. Compute modules are represented by red icons in the object pool. They are derived from the base class `HxCompModule`.

In order to learn how to implement a new compute module, we will take a look at a concrete example. In particular, we want to write a compute module which performs a threshold operation on a 3D image, i.e., on an input object of type *HxUniformScalarField3*. The module produces another 3D image as output. In the resulting image, all voxels with a value below a user-specified minimum value or above a maximum value should be set to zero.

For easier understanding we start with a very simple and limited version of the module. Then we iteratively improve the code. In particular, we proceed in three steps:

- Version 1: merely scans the input image, does not yet produce a result
- Version 2: creates an output object as result, uses the progress bar
- Version 3: adds a *DoIt* button, overwrites the existing result if possible

You can find the source code of all three versions in the demo package provided with the amira developer version, i.e., under `packages/mypackage` in the local amira directory. For each version there are two files: a header file called `MyComputeThresholdN.h` and a source code file called `MyComputeThresholdN.cpp` (where `N` is either 1, 2, or 3). Since the names are different, you can compile and execute all three versions in parallel.

In order to create a new local amira directory, please follow the instructions given in Section 12.2. In order to compile the demo package, please refer to Section 11.5 (Compiling and Debugging).

14.1.1 Version 1: Skeleton of a Compute Module

The first version of our module does not yet produce any output. It simply scans the input image and prints the number of voxels above and below the threshold.

Like most other modules, our compute module consists of a header file containing the class declaration as well as a source file containing the actual code (or the class definition). Let us look at the header file `MyComputeThreshold1.h` first:

```
/////////////////////////////////////////////////////////////////
//
// Example of a compute module (version 1)
//
/////////////////////////////////////////////////////////////////
#ifndef MY_COMPUTE_THRESHOLD_H
#define MY_COMPUTE_THRESHOLD_H

#include <Amira/HxCompModule.h>
#include <Amira/HxPortFloatTextN.h>
#include <mypackage/mypackageAPI.h>

class MYPACKAGE_API MyComputeThreshold1 : public HxCompModule
{
    // This macro is required for all modules and data objects
    HX_HEADER(MyComputeThreshold1);

  public:
    // Every module must have a default constructor.
    MyComputeThreshold1();

    // This virtual method will be called when the port changes.
    virtual void compute();

    // A port providing float text input fields.
    HxPortFloatTextN portRange;
};
```

```
#endif
```

As usual in C++ code, the file starts with a define statement that prevents the contents of the file from being included multiple times. Then three header files are included. `HxCompModule.h` contains the definition of the base class of our compute module. The next file, `HxPortFloatTextN.h`, contains the definition of a *port* we want to use in our class.

A port represents an input parameter of a module. In our case we use a port of type `HxPortFloat-TextN`. This port provides one or more text fields where the user can enter floating point numbers. The required text fields and labels are created automatically within the port constructor. As a programmer you simply put some ports into your module, specifying their types and labels, and do not have to bother creating a user interface for it.

Following `HxPortFloatTextN.h`, the package header file `mypackageAPI.h` is included. This file provides import and export storage-class specifiers for Windows systems. These are encoded in the macro `MYPACKAGE_API`. A class declared without this macro will not be accessible from outside the DLL it is defined in. On Unix systems the macro is empty and can be omitted.

In the rest of the header file nothing more is done than deriving a new class from `HxCompModule` and defining two member functions, namely the constructor and an overloaded virtual method called `compute`. The `compute` method is called when the module has been created and whenever a change of state occurs on one of the module's input data objects or ports. In fact, a connection to an input data object is also established by a port, as we shall see later on. In this example we just declare one port in our class, specifically an instance of type `HxPortFloatTextN`.

The corresponding source file looks like this:

```
/////////////////////////////////////////////////////////////////
//
// Example of a compute module (version 1)
//
/////////////////////////////////////////////////////////////////

#include <Amira/HxMessage.h>
#include <hxfield/HxUniformScalarField3.h>
#include <mypackage/MyComputeThreshold1.h>

HX_INIT_CLASS(MyComputeThreshold1,HxCompModule) // required macro

MyComputeThreshold1::MyComputeThreshold1() :
    HxCompModule(HxUniformScalarField3::getClassTypeId()),
    portRange(this,"range",2) // we want to have two float fields
{
}

void MyComputeThreshold1::compute()
{
    // Access the input data object. The member portData, which
    // is of type HxConnection, is inherited from HxModule.
    HxUniformScalarField3* field =
        (HxUniformScalarField3*) portData.source();

    // Check whether the input port is connected
    if (!field) return;

    // Get the input parameters from the user interface:
    float minValue = portRange.getValue(0);
    float maxValue = portRange.getValue(1);

    // Access size of data volume:
    const int* dims = field->lattice.dims();

    // Now loop through the whole field and count the pixels.
    int belowCnt=0, aboveCnt=0;
    for (int k=0; k<dims[2]; k++) {
        for (int j=0; j<dims[1]; j++) {
            for (int i=0; i<dims[0]; i++) {
                // This function returns the value at the specific
                // grid node. It implicitly casts the result
                // to float if necessary.
                float value = field->evalReg(i,j,k);
                if (value<minValue)
                    belowCnt++;
                else if (value>maxValue)
                    aboveCnt++;
            }
        }
    }

    // Finally print the result.
    theMsg->printf("%d voxels < %g, %d voxels > %g\n",
        belowCnt, minValue, aboveCnt, maxValue);
}
```

Following the include statements and the obligatory `HX_INIT_CLASS` macro, the constructor is defined. The usual C++ syntax must be used in order to call the constructors of the base class and the class members. The constructor of the base class `HxCompModule` takes the class type of the input data object to which this module can be connected. amira uses a special run-time type information system that is independent of the rtti feature provided by the newer ANSI C++ compilers.

The second method we have to implement is the `compute` method. We first retrieve a pointer to our input data object through a member called `portData`. This port is inherited from the base class `HxModule`, i.e., every module has this member. The port is of type `HxConnection` and it is represented as a blue line in the user interface (if connected). The rest of the compute method is rather straightforward. The way the actual data are accessed and how the computation is performed, of course, is highly specific to the input data class and the task the module performs. In this case we simply loop over all voxels of the input image and count the number of voxels below the minimum value and above the maximum value. In order to access a voxel's value, we use the *evalReg* method. This method is provided by any scalar field with regular coordinates, i.e., by any instance of class

HxRegScalarField3. Regardless of the primitive data type of the field, the result will always be cast to float.

Once you have compiled the mypackage demo package, you can load the file lobus.hm from amira 's data/tutorials directory and attach the module to it. Try to type in different threshold values, or use different input data sets. Instructions for compiling local packages are provided in Section 11.5 (Compiling and Debugging).

14.1.2 Version 2: Creating a Result Object

Now that we have a first working version of the module, we can add more functionality. First, we want to create a real output data object. Then we further want to improve the module by using amira's progress bar and by providing better default values for the range port. The header file of our module will not be affected by all these changes. We merely need to add some code in the source file MyComputeThreshold2.cpp.

Let us start with the output data object. In the compute method just before the for-loop, we insert the following statements:

```
// Create output with same primitive data type as input:
HxUniformScalarField3* output =
    new HxUniformScalarField3(dims, field->primType());

// Output shall have same bounding box as input:
output->coords()->setBoundingBox(field->bbox());
```

This creates a new instance of type HxUniformScalarField3 with the same dimensions and the same primitive data type as the input data object. Since the output has the same bounding box, i.e., the same voxel size as the input, we copy the bounding box. Note that this approach will only work for fields with uniform coordinates. For other regular coordinate types such as stacked or curvilinear coordinates, we refer to Section 15.2.

After the output object has been created, its voxel values are not yet initialized. This is done in the inner part of the nested for-loops. The method set, used for this purpose, automatically performs a cast from float to the primitive data type of the output field. In summary, the inner part of the for-loop now looks as follows:

```
float value = field->evalReg(i,j,k);
float newValue = 0;

if (value<minValue)
    belowCnt++;
else if (value>maxValue)
    aboveCnt++;
else newValue = value;

output->set(i,j,k,newValue);
```

Creating a new data object using the `new` operator will not automatically make it appear in the object pool. Instead, we must explicitly register it. In a compute module this can be done by calling the method `setResult`:

```
    setResult(output); // register result
```

This method adds a data object to the object pool if it is not already present there. In addition, it connects the object's *master* port to the compute module itself. Like any other connection, this link will be represented by a blue line in the object pool. The master port of a data object may be connected to a compute module or to an editor. Such a master connection indicates that the data object is controlled by an 'upstream' component, i.e., that its contents may be overridden by the object it is connected to.

Now that we have created an output object, let us address the progress bar. Although for the test data set `lobus.am` our threshold operation does not take very long, it is good practice to indicate that the application is busy when computations are performed that could take long time on large input data. Even better is to show a progress bar, which is not difficult. Before the time-consuming part of the compute routine, i.e., before the nested for-loops, we add the following line:

```
    // Turn \amira into busy state, don't activate Stop button.
    theWorkArea->startWorkingNoStop("Computing threshold");
```

We use the global instance `theWorkArea` of class `HxWorkArea` here. The corresponding header file must be included at the beginning of the source file. The method turns the application into the 'busy' state and displays a working message in the status line. As opposed to the method `start-Working`, this variant does not activate the stop button. See Section 17.2 for details. When the computation is done, we must call

```
    theWorkArea->stopWorking(); // stop progress bar
```

in order to switch off the 'busy' state again. Inside the nested for-loops we update the progress bar just before a new 2D slice is processed. This is done by the following line of code:

```
        // Set progress bar, the argument ranges between 0 and 1.
        theWorkArea->setProgressValue((float)(k+1)/dims[2]);
```

The value of `(float)(k+1)/dims[2]` progressively increases from zero to one during computation. Note that you should not call `setProgressValue` in the inner of the three loops. Each call involves an update of the graphical user interface and therefore is relatively expensive. It is perfectly okay to update the progress bar several hundred times during a computation, but not several hundred thousand times.

Another slight improvement we have incorporated into the second version of our compute module concerns the `range` port. In the constructor we have set new initial values for the minimum and maximum fields. While both values are 0 by default, we now set them to 30 and 200, respectively:

```
    // Set default value for the range port:
    portRange.setValue(0,30);  // min value is 30
    portRange.setValue(1,200); // max value is 200
```

You may now test this second version of the compute module by loading the test data set `lobus.am` from amira's `data/tutorials` directory. Attach the `ComputeThreshold2` module to it. To better appreciate the progress bar, try to resample the input data, for example to 512x512x100, and connect the compute module to the resampled data set. However, be sure that you have enough main memory installed on your system.

14.1.3 Version 3: Reusing the Result Object

Testing the first two versions of our module, we saw that the module's compute method is triggered automatically when the module is created and whenever the range port is changed. Each time a new result output data object is created. This quickly fills up the computer's main memory as well as amira's graphical user interface. Therefore, we now change this behavior: A new result object is to be created only the first time. Whenever the range port is changed afterwards, the existing result object should be overridden. In order to achieve this, we modify the middle part of the compute method in the following way:

```
    // Check if there is a result which we can reuse.
    HxUniformScalarField3* output =
        (HxUniformScalarField3*) getResult();

    // Check for proper type.
    if (output && !output->isOfType(
            HxUniformScalarField3::getClassTypeId() ))
        output = 0;

    // Check if size and primType still match the current input:
    if (output) {
        const int* outdims = output->lattice.dims();
        if (dims[0]!=outdims[0] ||dims[1]!=outdims[1] ||
            dims[2]!=outdims[2] ||
            field->primType() != output->primType())
            output=0;
    }

    // If necessary, create a new result data set.
    if (!output) {
        output = new HxUniformScalarField3(dims,
            field->primType());
        output->composeLabel(field->getName(),"masked");
    }
```

The `getResult` method checks whether there is a data set whose master port is connected to the compute module. This typically is the object set by a previous call to `setResult`. However, it also may be any other object. Therefore, a run-time type check must be performed by calling the `isOfType` member method of the output object. If the output object is not of type `HxUniform-ScalarField3`, the variable `output` will be set to null. Then a check is made whether the output object has the same dimensions and the same primitive data type as the input object. If this test fails, `output` will also be set to null. At the end, a new result object will only be created if no result yet exists or if the existing result does not match the input. It is possible to interactively try different range values without creating a bunch of new results.

However, when one of the numbers of the range port is changed, computation starts immediately. Sometimes this may be desired, but in this case we prefer to add a *DoIt* button as present in many other compute modules. The user must explicitly push this button in order to start computation. In order to use the *DoIt* button, the following line of code must be added in the public section of the module's header file:

```
    // Start computation when this button is clicked.
    HxPortDoIt portDoIt;
```

Of course, the corresponding include file `Amira/HxPortDoIt.h` must be included as well. Like for the other port, we must initialize `portDoIt` in the constructor of our module in the source file:

```
MyComputeThreshold3::MyComputeThreshold3() :
    HxCompModule(HxUniformScalarField3::getClassTypeId()),
    portRange(this,"range",2), // we want to have two float fields
    portDoIt(this,"action")
{
    ...

    // Set text of doIt button
    portDoIt.setLabel(0,"DoIt");
}
```

To achieve the desired behavior we finally change our compute method so that it immediately returns unless the *DoIt* button was pressed. This can be done by adding the following piece of code at the beginning of the compute method:

```
    // Check whether doIt button was hit
    if (!portDoIt.wasHit()) return;
```

With these changes, the module is already quite usable. Try to attach the final version of the module to some data set, press *DoIt*, change the range and press *DoIt* again. Attach an *OrthoSlice* module to the result while experimenting with the range (use the histogram mapping in the *OrthoSlice* in order to see small changes). Try to detach the connection between the result and the module and press *DoIt* again.

Finally, some remarks on performance. Although it is probably not critical in this simple example, performance typically becomes an issue in real-world applications. In the inner-most loop, calling the methods `field-> evalReg` and `output-> set` is convenient but rather expensive. For example, if the input consists of bytes like in `lobus.am`, these methods involve a cast from `unsigned char` to `float` and back to `unsigned char`.

The performance can be improved by writing code which explicitly handles a particular primitive data type. A pointer to the actual data values of a *HxUniformScalarField3* can be obtained by calling `field-> lattice.dataPtr()`. The value returned by this method is of type `void*`. It must be explicitly cast to the data type the field actually belongs to. The voxel values itself are arranged without any padding. This means that the index of voxel (i, j, k) is given by `(k*dims[1]+j)*dims[0]+i`, where `dims[0]` and `dims[1]` denote the number of voxels in the x and y directions, respectively.

14.2 A Display Module

Our next example is a module which displays some geometry in amira's 3D viewer. The module takes a surface model as input and draws a little cube at every vertex that belongs to *n* triangles, where *n* is a user-adjustable parameter.

From the previous section we already know the basic idea: We derive a new class from the base class `HxModule`. Since this time our module does not produce a new data set we directly use `HxModule` as base class instead of `HxCompModule`. As input the module should accept data of class `HxSurface`. We need one additional port allowing the user to specify the parameter *n*. As in the previous section we develop different versions of our module, thereby introducing new concepts step by step:

- Version 1:
  creates an Open Inventor scene graph and displays it in the viewer
- Version 2:
  adds a colormap port, provides a parse method for Tcl commands
- Version 3:
  implements a new display mode, dynamically shows or hides a port

You can find the source code of all three versions of the module in the demo package provided with the amira developer version, i.e., under `packages/mypackage` in the local amira directory. For each version there are two files, a header file called `MyDisplayVerticesN.h` and a source code file called `MyDisplayVerticesN.cpp` (where `N` is either 1, 2, or 3). Since the names are different you can compile and execute all three version in parallel.

In order to create a new local amira directory, please follow the instructions given in Section 12.2. In order to compile the demo package, please refer to Section 11.5 (Compiling and Debugging).

14.2.1 Version 1: Displaying Geometry

The first version of our module, called *MyDisplayVertices1*, merely detects the vertices of interest and displays them using little cubes. In order to understand the code, we first need to look more closely at the class `HxSurface`. As we can see in the reference documentation, a surface essentially contains an array of 3D points and an array of triangles. Each triangle has three indices pointing into the list of points. In order to count the triangles per vertex, we simply walk through the list of triangles and increment a counter for each vertex.

Once we have detected all interesting vertices, we are going to display them using small cubes. This is done by creating an Open Inventor scene graph. If you want to learn more about Open Inventor, you probably should look at *The Inventor Mentor*, an excellent book about Open Inventor published by Addison-Wesley. In brief, an Open Inventor scene graph is a tree-like structure of C++ objects which describes a 3D scene. Our scene is quite simple. It consists of one *separator node* containing several cubes, i.e., instances of class `SoCube`. Since an `SoCube` is always located at the origin, we put an additional node of type `SoTranslation` right before each `SoCube`. We adjust the size of the cubes so that each side is 0.01 times the length of the diagonal of the bounding box of the input surface.

After this short overview we now look at the header file of the module. It is called `MyDisplayVertices1.h`:

```
/////////////////////////////////////////////////////////////////
//
// Example of a display module
//
/////////////////////////////////////////////////////////////////
#ifndef MY_DISPLAY_VERTICES_H
#define MY_DISPLAY_VERTICES_H

#include <McHandle.h> // smart pointer template class
#include <Amira/HxModule.h>
#include <Amira/HxPortIntSlider.h>
#include <mypackage/mypackageAPI.h>

#include <Inventor/nodes/SoSeparator.h>

class MYPACKAGE_API MyDisplayVertices1 : public HxModule
{
    HX_HEADER(MyDisplayVertices1);

  public:
    // Constructor.
    MyDisplayVertices1();

    // Destructor.
    ~MyDisplayVertices1();

    // Input parameter.
    HxPortIntSlider portNumTriangles;

    // This is called when an input port changes.
    virtual void compute();

  protected:
    McHandle<SoSeparator> scene;
};

#endif
```

The header file can be understood quite easily. First some other header files are included. Then the new module is declared as a child class of `HxModule`. As usual, the macros `MYPACKAGE_API` and `HX_HEADER` are obligatory. Our module implements a default constructor, a destructor, and a compute method. In addition, it has a port of type `HxPortIntSlider` which allows the user to specify the number of triangles of the vertices to be displayed.

A pointer to the actual Open Inventor scene is stored in the member variable `scene` of type `McHandle<SoSeparator>`. A `McHandle` is a so-called *smart pointer*. It can be used like an ordinary C pointer. However, each time a value is assigned to it, the reference counter of the referenced object is automatically increased or decreased. This is done by calling the `ref` or `unref` method of the object. If the reference counter becomes zero or less, the object is deleted automatically. We recommend using smart pointers instead of C pointers because they are safer.

The actual implementation of the module is contained in the file `MyComputeThreshold1.cpp`. This file looks as follows:

```
/////////////////////////////////////////////////////////////////
//
// Example of a compute module (version 1)
//
/////////////////////////////////////////////////////////////////

#include <Amira/HxMessage.h>
#include <hxsurface/HxSurface.h>
#include <mypackage/MyDisplayVertices1.h>

#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoTranslation.h>

HX_INIT_CLASS(MyDisplayVertices1,HxModule)

MyDisplayVertices1::MyDisplayVertices1() :
    HxModule(HxSurface::getClassTypeId()),
    portNumTriangles(this,"numTriangles")
{
    portNumTriangles.setMinMax(1,12);
    portNumTriangles.setValue(6);
    scene = new SoSeparator;
}

MyDisplayVertices1::~MyDisplayVertices1()
{
    hideGeom(scene);
}

void MyDisplayVertices1::compute()
```

```
{
    int i;

    // Access input object (portData is inherited from HxModule):
    HxSurface* surface = (HxSurface*) portData.source();

    if (!surface) { // Check if input object is available
        hideGeom(scene);
        return;
    }

    // Get value from input port, query size of surface:
    int numTriPerVertex = portNumTriangles.getValue();
    int nVertices = surface->points.size();
    int nTriangles = surface->triangles.size();

    // We need a triangle counter for every vertex:
    McDArray<unsigned short> triCount(nVertices);
    triCount.fill(0);

    // Loop over all triangles and increase vertex counters:
    for (i=0; i<nTriangles; i++)
        for (int j=0; j<3; j++)
            triCount[surface->triangles[i].points[j]]++;

    // Now create the scene graph...
    // First remove all previous childs:
    scene->removeAllChildren();

    // Cube size should be 1% of the bounding box diagonal:
    float size = surface->getBoundingBoxSize().length() * 0.01;

    // Pointer to coordinates cast from McVec3f to SbVec3f.
    SbVec3f* p = (SbVec3f*) surface->points.dataPtr();

    SbVec3f q(0,0,0); // position of last point
    int count = 0; // vertex counter

    for (i=0; i<nVertices; i++) {
        if (triCount[i] == numTriPerVertex) {
            SoTranslation* trans = new SoTranslation;
            trans->translation.setValue(p[i]-q);

            SoCube* cube = new SoCube;
            cube->width = cube->height = cube->depth = size;

            scene->addChild(trans);
            scene->addChild(cube);

            count++;
            q=p[i];
        }
    }
    theMsg->printf("Found %d vertices belonging to %d triangles",
        count, numTriPerVertex);

    showGeom(scene); // finally show scene in viewer
}
```

A lot of things are happening here. Let us point out some of these in more detail now. The constructor initializes the base class with the type returned by `HxSurface::getClassTypeId`. This ensures that the module can only be attached to data objects of type `HxSurface`. The constructor also initializes the member variable `portNumTriangles`. The range of the slider is set from 1 to 12. The initial value is set to 6. Finally, a new Open Inventor separator nodes is created and stored in `scene`.

The destructor contains only one call, `hideGeom(scene)`. This causes the Open Inventor scene to be removed from all viewers (provided it is visible). The scene itself is deleted automatically when the destructor of `McHandle` is called.

The actual computation is performed in the `compute` method. The method returns immediately if no input surface is present. If an input surface exists, the numbers of triangles per point are counted. For this purpose a dynamic array `triCount` is defined. The array provides a counter for each vertex. Initially it is filled with zeros. The counters are increased in a loop over the vertices of all triangles.

In the second part of the compute method the Open Inventor scene graph is created. First, all previous children of `scene` are removed. Then the length of the diagonal of the input surface is determined. The size of the cubes will be set proportional to this length. For convenience the pointer to the coordinates of the surface is stored in a local variable `p`. Actually the coordinates are of type `McVec3f`. However, this class is fully compatible with the Open Inventor vector class `SbVec3f`. Therefore the pointer to the coordinates can be cast as shown in the code.

After everything has been set up, every element of the array `triCount` is checked in a for-loop. If the value of an element matches the selected number of triangles per vertex, two new Inventor nodes of type `SoTranslation` and `SoCube` are created, initialized, and inserted into `scene`. Since the `SoTranslation` also affects all subsequent translation nodes we must remember the position of the last point in `q` and subtract this position from the one of the current point. Alternatively, we could have encapsulated the `SoTranslation` and the `SoCube` in an additional `SoSeparator` node. However, this would have resulted in a more complex scene graph. At the very end of the compute method, the new scene graph is made visible in the viewer by calling `showGeom`. This method automatically checks if a node has already been visible. Therefore it may be called multiple times with the same argument.

The module is registered in the usual way in the package resource file, i.e., in `mypackage/share/resource/mypackage.rc`. Once you have compiled the demo package, you may test the module by loading the surface `mypackage/data/test.surf` located in the local amira directory.

14.2.2 Version 2: Adding Color and a Parse Method

In this section we want to add two more features to our module. First, we want to use a colormap port which allows us to specify the color of the cubes. Second, we want to add a parse method which allows us to specify additional Tcl commands for the module.

A colormap port is used to establish a connection to a colormap, i.e., to a class of type `HxColormap`. It is derived from `HxConnection` but, in contrast to the base class, it provides a graphical user interface showing the contents of the colormap and letting the user change its coordinate range. If no colormap is connected to the port, a default color is displayed. The default color can be edited by the user by double-clicking the color bar.

In order to provide our module with a colormap port, we must insert the following line into the module's header file:

```
    HxPortColormap portColormap;
```

Of course, we must also include the header file of the class `HxPortColormap`. This file is located in package `hxcolor`. Note that the order in which ports are displayed on the screen depends on the order in which the ports are declared in the header file. If we declare `portColormap` before `portNumTriangles`, the colormap port will be displayed before the integer slider.

In the compute method of our module we add the following piece of code just after the previous children of the scene graph have been removed:

```
    SoMaterial* material = new SoMaterial;
    material->diffuseColor =
        portColormap.getColor(numTriPerVertex);
    scene->addChild(material);
```

With these lines we insert a material node right before all the translation and cube nodes into the separator. The material node causes the cubes to be displayed in a certain color. We call the `getColor` method of the colormap port in order to determine this color. If the port is not connected to a colormap, this method simply returns the default color. However, if it is connected, the color is taken from the colormap. As an argument we specify `numTriPerVertex`, the number of triangles of the selected vertices. Depending on the value of `portNumTriangles`, the cubes therefore will be displayed in different colors. Of course, this requires that the range of the colormap extend from something like 1 to 10 or 12.

Besides the colormap port, we also want to add a Tcl command interface to our module. This is done by overloading the virtual method `parse` of `HxModule`. We therefore insert the following line into the module's class declaration:

```
virtual int parse(Tcl_Interp* t, int argc, char **argv);
```

In a parse method special commands can be defined which allow us to control the module in a more sophisticated way. A typical application is to set special parameters which should not be represented by a separate port in the user interface. As an example, we want to provide a method which allows us to change the size of the cubes. In the initial version of the module the cubes were adjusted so that each side was 0.01 times the length of the diagonal of the bounding box of the input surface. The value of the scale factor shall now be stored in the member variable `scale`. In order to set and get this variable, two Tcl commands `setScale` and `getScale` shall be provided. The implementation of the parse method looks as follows:

```
int
MyDisplayVertices2::parse(Tcl_Interp* t, int argc, char **argv)
{
    if (argc < 2) return TCL_OK;
    char *cmd = argv[1];

    if (CMD("setScale")) {
        ASSERTARG(3);
        scale = atof(argv[2]);
        fire(); // ensures that cubes will be updated immediately
    }
    else if (CMD("getScale")) {
        Tcl_VaSetResult(t, "%g", scale);
    }
    else return HxModule::parse(t,argc,argv);

    return TCL_OK;
}
```

Commands are defined in a sequence of if-else statements. For each command, the macro `CMD` should be used. At the end of the if-else sequence the parse method of the base class should be called. Note that after a command is issued, the compute method of the module will not be called automatically by default. This is in contrast to interactive changes of ports. However, we may explicitly call `fire` in a command like shown above. In this case the size of the cubes then will be adjusted immediately. You may test the parse method by loading the file `mypackage/data/test.surf`, attaching `DisplayVertices2` to it, and then typing something like `DisplayVertices2 setScale 0.03` into the amira console window.

14.2.3 Version 3: Adding an Update Method

Besides a compute method, modules may also define a *update method*. This method is called just before the compute method and also whenever a module is selected. In the update method, the user-interface of the module can be configured, i.e., ports can be shown or hidden dynamically if this is required, the sensitivity of ports can be adjusted, or the number of entries of an option menu can be modified dynamically.

In order to illustrate how an update method might work, we implement an alternate display mode in our module. In this mode all vertices of a surface should be displayed, not only the ones with a certain number of neighboring triangles. In this second mode the slider `portNumTriangles` is not meaningful anymore. We therefore hide it by defining an appropriate update method. The following lines are added in the header file `MyDisplayVertices3.h`:

```
    // Mode: 0=selected vertices, 1=all vertices
    HxPortRadioBox portMode;

    // Shows or hides required ports.
    virtual void update();
```

The new radio box port lets the user switch between the two display modes. Like the compute method, the update method takes no arguments and also has no return value.

If you look into the source code file `MyDisplayVertices3.cpp` you will notice that the radio box port is initialized in the constructor of the module and that the text labels are set properly. The update method itself is quite simple:

```
void MyDisplayVertices3::update()
{
    if (portMode.getValue()==0)
        portNumTriangles.show();
    else portNumTriangles.hide();
}
```

The slider `portNumTriangles` is shown or hidden depending on the value of the radio box port. Note that before the update method is called, all ports are marked to be shown. Therefore you must hide them every time `update` is called. For example, the `show` and `hide` calls should not be encapsulated by an if statement which checks if some input port is new.

In order to support the new all-vertices display style, we slightly modify the way the Open Inventor scene graph is created. Instead of a single `SoMaterial` node, we insert a new one whenever the color of a cube needs to be changed, i.e., whenever the number of triangles of a vertex differs from the previous one. The new for-loop looks as follows:

```
    int lastNumTriPerVertex = -1;
    int allVertices = portMode.getValue();

    for (i=0; i<nVertices; i++) {
        if (allVertices || triCount[i]==numTriPerVertex) {

            if (triCount[i]!=lastNumTriPerVertex) {
                SoMaterial* material = new SoMaterial;
                material->diffuseColor =
                    portColormap.getColor(triCount[i]);
                scene->addChild(material);
                lastNumTriPerVertex = triCount[i];
            }
            SoTranslation* trans = new SoTranslation;
            trans->translation.setValue(p[i]-q);

            SoCube* cube = new SoCube;
            cube->width = cube->height = cube->depth = size;

            scene->addChild(trans);
            scene->addChild(cube);

            count++;
            q=p[i];
        }
    }
```

Again, you can test the module by loading the file `mypackage/data/test.surf` and attaching `DisplayVertices3` to it. If you connect the *physics.icol* colormap to the colormap port, adjust the colormap range to 1...9, and select the all-vertices display style, you should get an image similar to the one shown in Figure 14.1.

14.3 A Module With Plot Output

In some cases you may want to show a simple 2D plot in an amira module, for example a histogram or some bar chart. To facilitate this task amira provides a special-purpose *Plot API* which can be used in any amira object, regardless of whether it is a compute module or a display module.

The class `PzEasyPlot` provides the necessary methods to open a plot window and to draw in that window. In the following, we illustrate how to use this class, again by means of an example. In particular, we are going to write a module which plots the number of voxels per slice for all materials defined in a label field. A label field usually represents the results of an image segmentation operation. For each voxel there is a label indicating which material the voxel belongs to. In a separate section further features of the Plot API will be described.

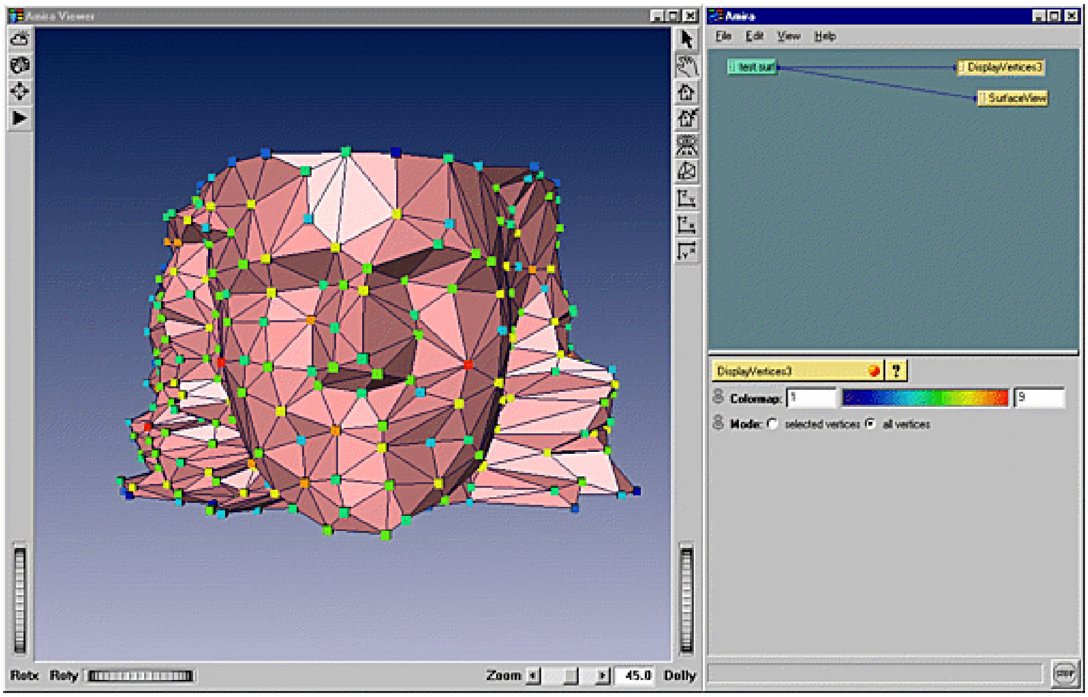
14.3.1 A Simple Plot Example

In this section we show how to plot some simple curves using the class `PzEasyPlot`. As mentioned above, the curves represent the number of voxels per slice for the materials of a label field. For this purpose we define a new module called `MyPlotAreaPerSlice`.

Like the other examples, this module is contained in the amira demo package. In order to check out the demo package, you must create a local amira directory as described in Section 12.2. In order to compile the demo package, please refer to Section 11.5 (Compiling and Debugging).

Let us first look at the header file `MyPlotAreaPerSlice.h`:

```
/////////////////////////////////////////////////////////////////
//
```



Colormap: 1 9
Mode: ( ) selected vertices (•) all vertices

Figure 14.1: The demo module *DisplayVertices3* displays little cubes at the vertices of a surface. The cubes are colored according to the number of neighboring triangles.

```
// Example of a plot module (header file)
//
/////////////////////////////////////////////////////////////////
#ifndef MY_PLOT_AREA_PER_SLICE_H
#define MY_PLOT_AREA_PER_SLICE_H

#include <Amira/HxModule.h>
#include <Amira/HxPortButtonList.h>
#include <hxplot/PzEasyPlot.h> // simple plot window
#include <mypackage/mypackageAPI.h>

class MYPACKAGE_API MyPlotAreaPerSlice : public HxModule
{
    HX_HEADER(MyPlotAreaPerSlice);

  public:
    // Constructor.
    MyPlotAreaPerSlice();

    // Shows the plot window.
    HxPortButtonList portAction;

    // Performs the actual computation.
    virtual void compute();

  protected:
    McHandle<PzEasyPlot> plot;
};

#endif
```

The class declaration is very simple. The module is derived directly from `HxModule`. It provides a constructor, a compute method, and a port of type `HxPortButtonList`. In fact, we will only use a single push button in order to let the user pop up the plot window. The plot window class `PzEasyPlot` itself is referenced by a smart pointer, i.e., by a variable of type `McHandle<PzEasyPlot>`. We have already used smart pointers in Section 14.2.1, for details see there.

Now let us take a look at the source file `MyPlotAreaPerSlice.cpp`:

```
/////////////////////////////////////////////////////////////////
//
// Example of a plot module (source code)
//
/////////////////////////////////////////////////////////////////

#include <Amira/HxWorkArea.h>
#include <hxfield/HxLabelLattice3.h>
#include <mypackage/MyPlotAreaPerSlice.h>

HX_INIT_CLASS(MyPlotAreaPerSlice,HxModule)

MyPlotAreaPerSlice::MyPlotAreaPerSlice() :
    HxModule(HxLabelLattice3::getClassTypeId()),
    portAction(this,"action",1)
{
    portAction.setLabel(0,"Show Plot");
    plot = new PzEasyPlot("Area per slice");
    plot->autoUpdate(0);
}

void MyPlotAreaPerSlice::compute()
{
    HxLabelLattice3* lattice = (HxLabelLattice3*)
        portData.source(HxLabelLattice3::getClassTypeId());

    // Check if valid input is available.
    if (!lattice) {
        plot->hide();
        return;
    }

    // Return if plot window is invisible and show button
    // wasn’t hit
    if (!plot->isVisible() && !portAction.isNew())
        return;

    theWorkArea->busy(); // activate busy cursor

    int i,k,n;
    const int* dims = lattice->dims();
    unsigned char* data = lattice->getLabels();
    int nMaterials = lattice->materials()->nBundles();

    // One counter per material and slice
    McDArray< McDArray<float> > count(nMaterials);

    for (n=0; n<nMaterials; n++) {
        count[n].resize(dims[2]);
        count[n].fill(0);
    }

    // Count number of voxels per material and slice
    for (k=0; k<dims[2]; k++) {
        for (i=0; i<dims[1]*dims[0]; i++) {
            int label = data[k*dims[0]*dims[1]+i];
            if (label<nMaterials)
                count[label][k]++;
        }
    }

    plot->remData(); // remove old curves

    for (n=0; n<nMaterials; n++) // add new curves
        plot->putData(lattice->materials()->bundle(n)->name(),
            dims[2], count[n].dataPtr());
    plot->update(); // refresh display
    plot->show(); // show or raise plot window

    theWorkArea->notBusy(); // deactivate busy cursor
}
```

In the constructor the base class `HxModule` is initialized with the class type ID of the class `HxLabelLattice3`. This class is not a data class derived from `HxData` but a so-called *interface*. Interfaces are used to provide a common API for objects not directly related by inheritance. In our case, `MyPlotAreaPerSlice` can be connected to any data object providing a `HxLabelLattice3` interface. This might be a `HxUniformLabelField3` but also a `HxStackedLabelField3` or something else.

Also in the constructor, a new plot window of type `PzEasyPlot` is created and stored in `plot`. Then the method `plot->autoUpdate(0)` is called. This means that we must explicitly call the `update` method of `PzEasyPlot` after the contents of the plot window are changed. Auto-update should be disabled when more than one curve is being changed at once.

As usual, the actual work is performed by the compute method. First, we retrieve a pointer to the label lattice. Since we want to use an interface instead of a data object itself, we must specify the class type ID of the interface as an argument of the `source` method of `portData`. Otherwise we would get a pointer to the object providing the interface, but we can't be sure about the type of this object.

The method returns if no label lattice is present or if the plot window is not visible and the show button has not been pressed. Otherwise, the contents of the plot window are recomputed from scratch. For this purpose a dynamic array of arrays called `count` is defined. The array provides a counter for each material and for each slice of the label lattice. Initially all counters are set to zero. Afterwards, they are incremented while the voxels are traversed in a nested for-loop.

The actual initialization of the plot window happens subsequently. First, old curves are removed by calling `plot->remData`. Then, for each material, a new curve is added by calling `plot->putData`. Afterwards, `plot->update` is called. If we had not disabled 'auto update' in the constructor, the plot window would have been updated automatically in each call of `putData`. The `putData` method creates a curve with the given name and sets the values. If a curve of the given name exists, the old values are overridden. The method returns a pointer to the curve which in turn can be used to set attributes for the curve individually (see below). Finally, the plot window is popped up and the 'busy' cursor we have activated before is switched off again.

To test the module, first compile the demo package. For instructions, see Section 11.5 (Compiling and Debugging). Then load the file `data/tutorials/lobus.labels.am` from the amira root directory. Attach `PlotAreaPerSlice` to it and press the show button. You then should get a result like that shown in Figure 14.2.

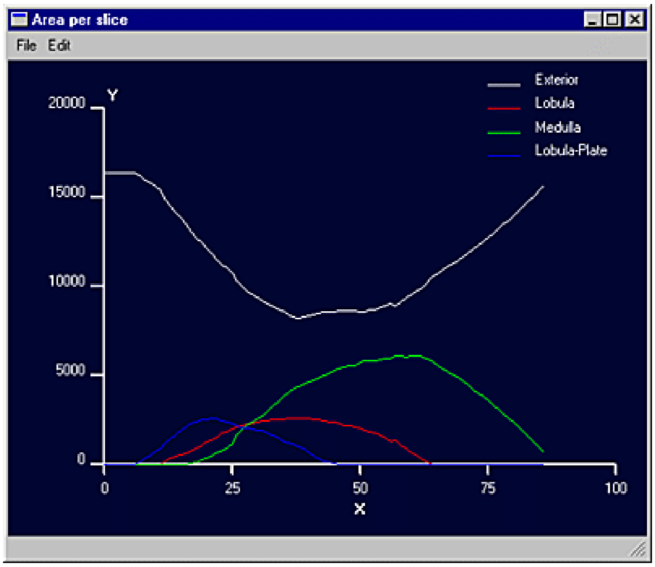


Figure 14.2: Plot produced by sample module *PlotAreaPerSlice*.

14.3.2 Additional Features of the Plot API

The 'pointer to curve' objects returned by the `putData` call can be used to access the curve directly, i.e., to manipulate its attributes. The most important attributes of curve objects are:

- Color, represented by a RGB values between 0 and 1. Can be set by calling:
  `curve->setAttr("color", r, g, b);`
- Line width, represented by an integer number. Can be set by calling:
  `curve->setAttr("linewidth", linewidth);`
- Line type, represented by an integer number. Available line types are 0=no line, 1=line, 2=dashed, 3=dash-dotted, and 4=dotted. Can be set by calling:
  `curve->setAttr("linetype", type);`
- Curve type, represented by an integer number. Available curve types are 0=line curve, 1=histogram, 2=marked line, 3=marker. Can be set by calling:
  `curve->setAttr("curvetype", type);`

For each attribute corresponding `getAttr` methods are available.

In order to access the axis of the 'easy plot' window, you must call

```
    PzAxis* axis = plot->getTheAxis();
```

Don't forget to include the corresponding header file `PzAxis.h`.

The color, line width, and line type attributes of the curves apply to axes as well. Besides this, there are some more methods to change the appearance of axes:

```
// Set the range of the axes
float xmin = 0.0, xmax = 1.0;
float ymin = 0.0, ymax = 1.0;
axis->setMinMax(xmin, xmax, ymin, ymax);

// Set the label of an axis
axis->setLabel(0, "X Axis");
axis->setLabel(1, "Y Axis");
```

If you are not satisfied with the size of the plot window and you don't want to change it using the mouse every time, just call `setSize` right after creating the plot window:

```
plot->setSize(width, height);
```

As you would expect, the methods `getMinMax`, `getLabel` and `getSize` are also available with the same parameter list as their `set` counterparts.

Finally, it is also possible to have a legend or a grid in the plot. In this case more arguments must be specified in the constructor of `PzEasyPlot`:

```
int withLegend = 1;
int withGrid = 0;
plot = new PzEasyPlot("Area per slice",
   withLegend, withGrid);
```

Like the axis, the legend and the grid are internally represented by separate objects of type `PzLegend` and `PzGrid`. You can access these objects by calling the methods `getTheLegend` and `getTheGrid`. Details about the member methods of these objects are listed in the class reference documentation.

Chapter 15

Data Classes

This chapter provides an overview of the structure of amira data classes. Important classes are discussed in more detail. In particular, the following topics will be covered:

- an introduction to data classes, including the hierarchy of data classes
- data on regular grids, e.g., 3D images with uniform or stacked coords
- tetrahedral grids, including data fields defined on such grids
- hexahedral grids, including data fields defined on such grids
- other issues related to data classes, including transparent data access

15.1 Introduction

A profound knowledge of the amira data objects is essential to developers. Data objects occur as input of write routines and almost all modules, and as output of read routines and compute modules. In the previous chapters we already encountered several examples of amira data objects such as 3D image data (represented by the class `HxUniformScalarField3`), triangular surfaces (represented by the class `HxSurface`), or colormaps (represented by the class `HxColormap`). Like modules, data objects are instances of C++ classes. All data objects are derived from the common base class `HxData`. Data objects are represented by green icons in amira's object pool.

In the following let us first present an overview of the hierarchy of data classes. Afterwards, we will discuss some of the general concepts behind it.

15.1.1 The Hierarchy of Data Classes

The hierarchy of amira data classes roughly looks as follows (derived classes are indented, auxiliary base classes are ignored):

`HxData` *base class of all data objects*

`HxSpreadSheet` *spreadsheet containing an arbitrary number of rows and columns*

`HxColormap` *base class of colormaps*

`HxColormap256` *colormap consisting of discrete RGBA tuples*

`HxCameraPath` *base class of camera paths*

`HxKeyframeCameraPath` *camera path based on interpolated key-frames*

`HxSpatialData` *data objects embedded in 3D space*

`HxIvData` *encapsulates an Open Inventor scene graph*

`HxField3` *base class representing fields in 3D space*

`HxScalarField3` *scalar field (1 component)*

`HxRegScalarField3` *scalar field with regular coordinates*

`HxUniformScalarField3` *scalar field with uniform coordinates*

`HxUniformLabelField3` *material labels with uniform coordinates*

`HxStackedScalarField3` *scalar field with stacked coordinates*

`HxStackedLabelField3` *material labels with stacked coordinates*

`HxAnnaScalarField3` *scalar field defined by an analytic expression*

`HxTetraScalarField3` *scalar field defined on a tetrahedral grid*

`HxHexaScalarField3` *scalar field defined on a hexahedral grid*

`HxVectorField3` *vector field (3 components)*

`HxRegVectorField3` *vector field with regular coordinates*

`HxUniformVectorField3` *vector field with uniform coordinates*

`HxEdgeElemVectorField3` *vector field defined by Whitney elements*

`HxAnnaVectorField3` *vector field defined by an analytic expression*

`HxTetraVectorField3` *vector field defined on a tetrahedral grid*

`HxHexaVectorField3` *vector field defined on a hexahedral grid*

`HxComplexScalarField3` *complex-valued scalar field (2 components)*

`HxRegComplexScalarField3` *complex scalar field with regular coordinates*

`HxUniformComplexScalarField3` *complex scalar field w/ uniform coords*

`HxTetraComplexScalarField3` *complex scalar field defined on a tetra grid*

`HxHexaComplexScalarField3` *complex scalar field defined on a hexa grid*

`HxComplexVectorField3` *complex-valued vector field (6 components)*

`HxRegComplexVectorField3` *complex vector field with regular coordinates*

`HxUniformComplexVectorField3` *complex vector field w/ uniform coords*

`HxEdgeElemComplexVectorField3` *complex vector field w/ Whitney elements*

`HxTetraComplexVectorField3` *complex field defined on a tetrahedral grid*

`HxHexaComplexVectorField3` *complex field defined on a hexahedral grid*

`HxColorField3` *color field consisting of RGBA-tuples*

`HxRegColorField3` *color field with regular coordinates*

`HxUniformColorField3` *color field with uniform coordinates*

`HxRegField3` *other n-component field with regular coordinates*

`HxTetraField3` *other n-component field defined on a tetrahedral grid*

`HxHexaField3` *other n-component field defined on a hexahedral grid*

`HxVertexSet` *data objects providing a set of discrete vertices*
`HxSurface` *represents a triangular surface*
`HxTetraGrid` *represents a tetrahedral grid*
`HxHexaGrid` *represents a hexahedral grid*
`HxLineSet` *represents a set of line segments with vertex data*
`HxLandmarkSet` *represents one or multiple sets of corresponding landmarks*
`HxCluster` *represents a set of vertices with associated data values*
`HxSurfaceField` *base class for fields defined on triangular surfaces*
`HxSurfaceScalarField` *scalar field defined on a surface (1 component)*
`HxSurfaceVectorField` *vector field defined on a surface (3 components)*
`HxSurfaceComplexScalarField` *complex scalar surface field (2 components)*
`HxSurfaceComplexVectorField` *complex vector surface field (6 components)*
`HxSurfaceField` *other n-component field defined on a surface*

Note that you can find an in-depth description of every class in the online reference documentation. This description has been generated automatically from the commented amira header files by a tool called DOC++. You may view it by pointing an external web browser such as Internet Explorer or Netscape Navigator to the file `share/doc++/index.html` contained in the amira root directory. The reference documentation not only covers data objects but all classes provided with the amira developer version. As you already know, these classes are arranged in packages. For example, all data classes derived from `HxField3` are located in package `hxfield`, and all classes related to triangular surfaces are located in package `hxsurface`.

15.1.2 Remarks About the Class Hierarchy

All data classes are derived from the base class `HxData`. This class in turn is derived from `HxObject`, the base class of all objects that can be put into the amira object pool. The class `HxData` adds support for reading and writing data objects, and it provides the variable `parameters` of type `HxParamBundle`. This variable can be used to annotate a data object by an arbitrary number of nested parameters. The parameters of any data object can be edited interactively using the parameter editor described in the User's Guide.

We observe that the majority of data classes are derived from `HxSpatialData`. This is the base class of all data objects which are embedded in 3D space as opposed for example to colormaps. `HxSpatialData` adds support for user-defined affine transformations, i.e., translations, rotations, and scaling. For details refer to Section 15.5.2. It also provides the virtual method `getBoundingBox` which is redefined by all derived classes. Two important child classes of `HxSpatialData` are `HxField3` and `HxVertexSet`.

`HxVertexSet` is the base class of all data objects that are defined on an unstructured set of vertices in 3D space, like surfaces or tetrahedral grids. The class provides methods to apply a user-defined affine transformation to all vertices of the object, or modify the point coordinates in some other way.

`HxField3` is the base class of data fields defined on a 3D-domain, like 3D scalar fields or 3D vector fields. HxField3 defines an efficient procedural interface to evaluate the field at an arbitrary 3D point within the domain, independent of whether the latter is a regular grid, a tetrahedral grid, or something else. The procedural interface is described in more detail in Section 15.5.1.

Looking at the inheritance hierarchy again, we observe that a high level distinction is made between fields returning a different number of data values. For example, all 3D scalar fields are derived from a common base class HxScalarField3, and all 3D vector fields are derived from a common base class HxVectorField3. The reason for this structure is that many modules depend on the data dimensionality of a field only, not on the internal representation of the data. For example, a module for visualizing a flow field by means of particle tracing can be written to accept any object of type HxVectorField3 as input. It then automatically operates on all derived vector fields, regardless of the type of grid they are defined on.

On the other hand, it is often useful to treat the number of data variables of a field as a dynamic quantity and to distinguish between the type of grid a field is defined on. For example, we may wish to have a common base class of fields defined on a regular grid and derived classes for regular scalar or vector fields. Since this structure and the one sketched above are very hard to incorporate into a common class graph, even if multiple inheritance were used, another concept has been chosen in amira, namely *interfaces*. Interfaces were first introduced by the Java programming language. They allow the programmer to take advantage of common properties of classes that are not related by inheritance.

In amira interfaces can be implemented as class members, or as additional base classes. In the first case a data class *contains* an interface class, while in the second case it is *derived* from HxInterface. Important interface classes are HxLattice3, HxTetraData, and HxHexaData, which are members of fields defined on regular, tetrahedral, and hexahedral grids, respectively. Another example is HxLabelLattice3, which is a member of HxUniformLabelField3, as well as HxStacked-LabelField3. In Section 14.3.1 we have already presented an example of how to use this interface in order to write a module which operates on any label field, regardless of the actual coordinate type.

15.2 Data on Regular Grids

Fields defined on a regular grid occur in many different applications. For example, 3D image volumes fall into this category. The term ‘regular’ means that the nodes of the grid are arranged as a regular 3D array, i.e., every node can be addressed by an index triple (i,j,k). A regular field can be characterized by three major properties: the coordinate type, the number of data components, and the primitive component data type (for example short or float).

In the class hierarchy a major distinction is made between the number of data components of a field. For example, there is a class HxRegScalarField3 representing (one-component) scalar fields defined on a regular grid. This class is derived from the general base class HxScalarField3. Similar classes exist for (three-component) vector fields, complex scalar field, and complex vector fields defined on regular grids. Fields not falling into one of these categories, i.e., fields defined on regular grids with a different number of data components, are represented by the class HxRegField3 which is directly derived from `HxField3`. Moreover, there are separate subclasses for the most relevant combinations of the number of data components and the coordinates type, like `HxStacked-ScalarField3` or `HxUniformVectorField3`. All regular data classes provide a member variable `lattice` of type `HxLattice3`. This variable is an *interface*. It can be used to access data fields with a different number of components in a transparent way.

Below we first discuss the lattice interface in more detail. We then present an overview of all supported coordinate types. Afterwards, two more types of data fields defined on regular coordinates are discussed, namely label fields and color fields.

Note that all these fields can be evaluated without regard to the actual coordinate type or the primitive data type by means of amira's procedural interface for 3D fields (see Section 15.5.1).

15.2.1 The Lattice Interface

The actual data of any regular 3D field is stored in a member variable `lattice` of type `HxLattice3`. This variable essentially represents a dynamic 3D array of n-component vectors. The number of vector components as well as the primitive data type are subject to change, i.e., a data object of type `HxLattice3` can be re-initialized to hold a different number of components of different primitive data type. However, a lattice contained in an object of type `HxRegScalarField3` always consists of 1-component vectors, while a lattice contained in an object of type `HxRegVectorField3` always consists of 3-component vectors. In addition, the coordinates of the field are stored in a separate coordinate object that is also referenced by the lattice.

Accessing the Data

To learn what kind of methods are provided by the lattice class, please refer to the online reference documentation or directly inspect the header file `HxLattice3.h` located in package `hxfield`. At this point, we just present a short example which shows how the dimensionality of the lattice, the number of data components, and the primitive data type can be queried. The primitive data type is encoded by the class `McPrimType` defined in package `mclib`. In particular, the following six data types are supported by amira:

- `McPrimType::mc_uint8` (8-bit unsigned bytes)
- `McPrimType::mc_int16` (16-bit signed shorts)
- `McPrimType::mc_uint16` (16-bit unsigned shorts)
- `McPrimType::mc_int32` (32-bit signed integers)
- `McPrimType::mc_float` (32-bit floats)
- `McPrimType::mc_double` (64-bit doubles)

Regardless of the actual type of the lattice data values, the pointer to the data array is returned as `void*`. The return value must be explicitly cast to a pointer of the correct type. This is illustrated in the following example where we compute the maximum value of all data components of a lattice. Note that the data values are stored one after another without any padding. The first index runs fastest.

```
    HxLattice3& lattice = field->lattice;
    const int* dims = lattice.dims();
    int nDataVar = lattice.nDataVar();

    switch (lattice.primType()) {
    case McPrimType::mc_uint8: {
        unsigned char* data = (unsigned char*) lattice.dataPtr();
        unsigned char* max = data[0];
        for (int k=0; k<dims[2]; k++)
            for (int j=0; j<dims[1]; j++)
                for (int i=0; i<dims[0]; i++)
                    for (int n=0; n<nDataVar; n++) {
                        int idx =
                            nDataVar*((k*dims[1]+j)*dims[0]+i)+n;
                        if (data[idx]>max)
                            max = data[idx];
                    }
        theMsg->printf("Max value is %d", max);
        } break;

    case McPrimType::mc_int16: {
        short* data = (short*) lattice.dataPtr();
        short* max = data[0];
        for (int k=0; k<dims[2]; k++)
            for (int j=0; j<dims[1]; j++)
                for (int i=0; i<dims[0]; i++)
                    for (int n=0; n<nDataVar; n++) {
                        int idx =
                            nDataVar*((k*dims[1]+j)*dims[0]+i)+n;
                        if (data[idx]>max)
                            max = data[idx];
                    }
        theMsg->printf("Max value is %d", max);
        } break;

    ...

    }
```

As a tip, note that the processing of different primitive data types can often be simplified by defining appropriate template functions locally. In the case of our example, such a template function may look like this:

```
template<class T>
void getmax(T* data, const int* dims, int nDataVar)
{
    T* max = data[0];
    for (int k=0; k<dims[2]; k++)
```

```
        for (int j=0; j<dims[1]; j++)
            for (int i=0; i<dims[0]; i++)
                for (int n=0; n<nDataVar; n++) {
                    int idx =
                        nDataVar*((k*dims[1]+j)*dims[0]+i)+n;
                    if (data[idx]>max)
                        max = data[idx];
                }
    theMsg->printf("Max value is %d", max);
}
```

Using this template function, the above switch statement looks as follows:

```
    switch (lattice.primType()) {
    case McPrimType::mc_uint8:
        getmax((unsigned char*)lattice.dataPtr(),dims,nDataVar);
        break;
    case McPrimType::mc_int16:
        getmax((short*)lattice.dataPtr(),dims,nDataVar);
        break;

    ...

    }
```

Though less efficient, another possibility for handling different primitive data types is to use one of the methods `eval`, `set`, `getData`, or `putData`. These methods always involve a cast to `float` if the primitive data type of the field requires it.

Accessing the Lattice Interface

Imagine you want to write a module which operates on any kind of regular field, i.e., on objects of type `HxRegScalarField3`, `HxRegVectorField3`, and so on. One way to achieve this would be to configure the input port of the module so that it can be connected to all possible regular field input objects. This can be done by calling the method `portData.addType()` in the module's constructor multiple times with the required class type IDs. In addition, all input types must be listed in the package resource file. This can be done by specifying a blank-separated list of types as the argument of the `-primary` option of the `module` command. In the compute method of the module, the actual type of the input must be queried, then the input pointer must be cast to the required type before a pointer to the lattice member of the object can be stored.

Of course, this approach is very tedious. A much simpler approach is to make use of the fact that the lattice member of a regular field is an interface. Instead of the name of a real data class, the class type ID of `HxLattice3` may be used to specify to what kind of input object a module may be connected to. In fact, if this is done, any data object providing the lattice interface will be considered as a valid input. In order to access the lattice interface of the input object, the following statement must be used in the module's compute method (also check Section 14.3.1 for an example of how to deal with interfaces):

```
    HxLattice3* lattice = (HxLattice3*)
        portData.source(HxLattice3::getClassTypeId());
```

Creating a Field From an Existing Lattice

When working with lattices, we may want to deposit a new lattice in the object pool, for example as the result of a compute module. However, since `HxLattice3` is not an amira data class, this is not possible. Instead we must create a suitable field object which the lattice is a member of. For this purpose the class `HxLattice3` provides a static method `create` which creates a regular field and puts an existing lattice into it. If the lattice contains one data component, a scalar field will be created; if it contains three components, a vector field will be created, and so on. The resulting field may then be used as the result of a compute module. Note that the lattice must not be deleted once it has been put into a field object. The concept is illustrated by the following example:

```
    HxLattice3* lattice = new HxLattice3(dims, nDataVar,
        primType, otherLattice->coords()->duplicate());

    ...

    HxField3* field = HxLattice3::create(lattice);
    theObjectPool->addObject(field);
```

15.2.2 Regular Coordinate Types

Currently four different coordinate types are supported for regular fields, namely uniform coordinates, stacked coordinates, rectilinear coordinates, and curvilinear coordinates. The coordinate types are distinguished by way of the enumeration data type `HxCoordType`. The coordinates themselves are stored in a separate utility class of type `HxCoord3` which is referenced by the lattice member of a regular field. For each coordinate type there is a corresponding subclass of `HxCoord3`.

As already mentioned in the introduction, for some important cases there are special subclasses of a regular field dedicated to a particular coordinate type. Examples are `HxStacked-ScalarField3` (derived from `HxRegScalarField3`) or `HxUniformVectorField3`) (derived from `HxRegVectorField3`). If such special classes do not exist, the regular base class should be used instead. In this case the coordinate type must be checked dynamically and the pointer to the coordinate object has to be down-cast explicitly before it can be used. This is illustrated in the following example:

```
    HxCoord3* coord = field->lattice.coords();

    if (coord->coordType() == c_rectilinear) {
        HxRectilinearCoord3* rectcoord =
            (HxRectilinearCoord3*) coord;
        ...
    }
```

Uniform Coordinates

Uniform coordinates are the simplest form of regular coordinates. All grid cells are axis-aligned and of equal size. In order to compute the position of a particular grid node, it is sufficient to know the number of cells in each direction as well as the bounding box of the grid.

Uniform coordinates are represented by the class `HxUniformCoord3`. This class provides a method `bbox` which returns a pointer to an array of six floats describing the bounding box of the grid. The six numbers represent the minimum x-value, the maximum x-value, the minimum y-value, the maximum y-value, the minimum z-value, and the maximum z-value in that order. Note, that the values refer to grid nodes, i.e., to the corner of a grid cell or to the center of a voxel. In order to compute the width of a voxel, you should use code like this:

```
    const int* dims = uniformcoords->dims();
    const float* bbox = uniformcoords->bbox();
    float width = (dims[0]>1) ? (bbox[1]-bbox[0])/(dims[0]-1):0;
```

Stacked Coordinates

Stacked coordinates are used to describe a stack of uniform 2D slices with variable slice distance. They are represented by the class `HxStackedCoord3`. This class provides a method `bboxXY` which returns a pointer to an array of four floats describing the bounding box of a 2D slice. In addition, the method `coordZ` returns a pointer to an array containing the z-coordinate of each 2D slice.

Rectilinear Coordinates

Same as for uniform or stacked coordinates, in the case of rectilinear coordinates the grid cells are aligned to the axes, but the grid spacing may vary from cell to cell in each direction. Rectilinear coordinates are represented by the class `HxRectilinearCoord3`. This class provides three methods, `coordX`, `coordY`, and `coordZ`, returning pointers to the arrays of x-, y-, and z-coordinates, respectively.

Curvilinear Coordinates

In the case of curvilinear coordinates, the position of each grid node is stored explicitly as a 3D vector of floats. A single grid cell need not to be axis-aligned anymore. An example of a 2D curvilinear grid is shown in Figure 15.1.

Curvilinear coordinates are represented by the class `HxCurvilinearCoord3`. This class provides a method `pos` which can be used to query the position of a grid node indicated by an index triple (i,j,k). Alternatively, a pointer to the coordinate values may be obtained by calling the method `coords`. The coordinate vectors are stored one after another without padding and with index i running fastest. Here is an example:

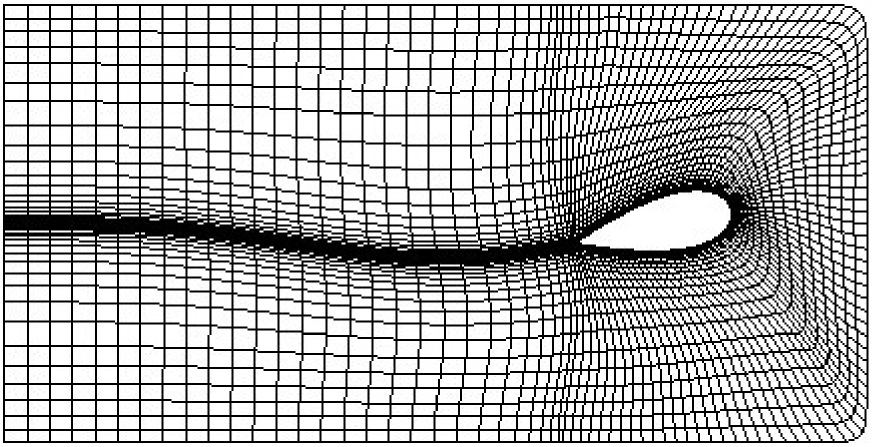


Figure 15.1: Example of a 2D grid with curvilinear coordinates.

```
const int* dims = curvilinearcoords->dims();
const float* coords = curvilinearcoords->coords();

// Position of grid node (i,j,k)
float x = coords[3*((k*dims[1]+j)*dims[0]+i)];
float y = coords[3*((k*dims[1]+j)*dims[0]+i)+1];
float z = coords[3*((k*dims[1]+j)*dims[0]+i)+2];
```

15.2.3 Label Fields and the Label Lattice Interface

Label fields are used to store the results of an image segmentation process. Essentially, at each voxel a number is stored indicating which material the voxel belongs to. Consequently, label fields can be considered scalar fields. In fact, currently there are two different types of label fields, one for uniform coordinates (represented by class `HxUniformLabelField3` derived from `HxUniformScalarField3`) and one for stacked coordinates (represented by class `HxStackedLabelField3` derived from class `HxStackedScalarField3`). Since the two types are not derived from a common base class, a special-purpose interface called `HxLabelLattice3` is provided. In fact, this interface is in turn derived from `HxLattice3`. It replaces the standard lattice variable of ordinary regular fields (see Section 15.2.1).

The primitive data type of a label field is always `McPrimType::mc_uint8`, i.e., bytes. In addition to the standard lattice interface, the label lattice interface also provides access to the label field's materials. Materials are stored in a special parameter subdirectory of the underlying data object. While discussing the plot API, we already encountered an example of how to interpret the materials of a label field (see Section 14.3.1). Note that whenever a new label is introduced, a new entry should also be put into the material list. Existing materials are marked so that they can not be removed from the material list (this would corrupt the labeling). In order to remove obsolete materials, call the method `removeEmptyMaterials` of `HxLabelLattice3`.

In addition to the labels, special weights can be stored in a label lattice. These weights are used to achieve sub-voxel accuracy when reconstructing 3D surfaces from the segmentation results. A pointer to the weights can be obtained by calling `getWeights` or `getWeights2` of the label lattice. For more details about `HxLabelLattice3`, please refer to the online class documentation.

15.2.4 Color Fields

Color fields are yet another type of regular fields. They consist of 4-component RGBA-byte-tuples and are represented by the class `HxRegColorField3` derived from `HxColorField3`. The latter class is closely related to `HxScalarField3` or `HxVectorField3`, see the overview on data class inheritance presented in Section 15.1.1. For color fields with uniform coordinates there is a special subclass `HxUniformColorField3`. Like any other regular fields, color fields provide a member `lattice` which can be used to access the data in a transparent way.

15.3 Unstructured Tetrahedral Data

Another important type of data refers to fields defined on unstructured tetrahedral grids. Such grids are often used in finite element simulations (FEM). In amira, tetrahedral grids and data fields defined on such grids are implemented by two different classes or groups of classes and are also distinguished in the user interface by different icons. The reason is that by separating grid and data there is no need for replicating the grid in case many fields are defined on the same grid, a case that occurs frequently in practice.

In the following two sections, we introduce the grid class `HxTetraGrid` before discussing the corresponding field classes and the interface `HxTetraData`.

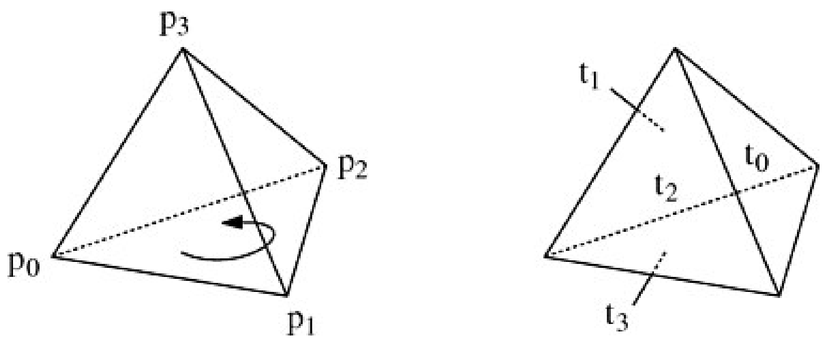


Figure 15.2: Numbering of points in a tetrahedron with positive volume (left). Numbering of the corresponding triangles (right).

15.3.1 Tetrahedral Grids

Tetrahedral grids in amira are implemented by the class `HxTetraGrid` and its base class `Tetra-Grid`. Looking at the reference documentation of `TetraGrid` we observe that a tetrahedral grid essentially consists of a number of dynamic arrays such as `points`, `tetras`, or `materialIds`.

- The `points` array is a list of all 3D points contained in the grid. A single point is stored as an element of type `McVec3f`. This class has the same layout as the Open Inventor class `SbVec3f`. Thus, a pointer to `McVec3f` can be cast to a pointer to `SbVec3f` and vice versa.
- The `tetras` array describes the actual tetrahedra. For each tetrahedron the indices of the four points and the indices of the four triangles it consists of are stored. The numbering of the points and triangles is shown in Figure 15.2. In particular, the fourth point is located above of the triangle defined by the first three points. Triangle number *i* is located opposite to point number *i*.
- The `materialIds` array contains 8-bit labels that assign a ’material’ identifier to every tetra-hedron. For example, this is used in tetrahedral grids generated from segmented image data to distinguish between different image segments corresponding to different material components of physical objects represented by the (3D) image data Like in the case of label fields or surfaces, the set of possible material values is stored in the parameter list of the grid data object.

The three arrays, `points`, `tetras`, and `materialIds`, must be provided by the ’user’. The triangles of the grid are stored in an additional array called `triangles`. This array can be constructed automatically by calling the member method `createTriangles2`. This method computes the triangles from scratch and sets the triangle indices of all tetrahedra defined in `tetras`.

The `triangles` array also provides a way for accessing neighboring tetrahedra. Among other in-formation (see reference documentation) stored for each triangle, the indices of the two tetrahedra it belongs to are available. In the case of boundary triangles, one of these indices is -1. Therefore, in order to get the index of a neighboring tetrahedron you can use the following code:

```
    // Find tetra adjacent to tetra n at face 0:
    int triangle = grid->tetras[n].triangles[0];
    otherTetra = grid->triangles[triangle].tetras[0];
    if (otherTetra == n)
        otherTetra = grid->triangles[triangle].tetras[1];
    if (otherTetra == -1) {
        // No neighboring tetra, boundary face
        ...
    }
```

Note that it is possible to define a grid with duplicated vertices, i.e. with vertices having exactly the same coordinates. This is useful to represent discontinuous data fields. The method *createTriangles2* checks for such duplicated nodes and correctly creates a single triangle between two geometrically adjacent tetrahedra, even if these tetrahedra refer to duplicated points.

Optionally the edges of a grid can be computed in addition to its points triangles, and tetrahedra by calling *createEdges*. The edges are stored in an array called `edges` and another array `edgesPerTetra` is used in order to store the indices of the six edges of a tetrahedron.

Moreover, the class `TetraGrid` provides additional optional arrays, for example to store a dynamic list of the indices of all tetrahedra adjacent to a particular point (`tetrasPerPoint`). This and other information is primarily used for internal purposes, for example to facilitate editing and smoothing of tetrahedral grids.

15.3.2 Data Defined on Tetrahedral Grids

In most applications, you will not only have to deal with a single tetrahedral grid, but also with data fields defined on it, for example scalar fields (e.g. temperature) or vector fields (e.g. flow velocity). amira provides special classes for these data modalities, namely `HxTetraScalarField3`, `HxTetraVectorField3`, `HxTetraComplexScalarField3`, `HxTetraComplexVectorField3`, and `HxTetraField3` (see class hierarchy in Section 15.1.1).

Like in the case of regular data fields, the actual information is stored in a special member variable called *data*, which is of type `HxTetraData`. Like the corresponding member type `HxLattice3` for regular data, `HxTetraData` is an `interface`, i.e., derived from `HxInterface`. It provides transparent access to data fields defined on tetrahedral grids regardless of the actual number of data components of the field. In order to access that interface without knowing the actual type of input object within a module, you may use the following statement:

```
    HxTetraData* data = (HxTetraData*)
        portData.source(HxTetraData::getClassTypeId());
    if (!data) return;
```

Data on tetrahedral grids must always be of type `float`. The data values may be stored in three different ways, indicated by the encoding type as defined in `HxTetraData`:

- PER_TETRA: One data vector is stored for each tetrahedron. The data are assumed to be constant inside the tetrahedron.
- PER_VERTEX: One data vector is stored for each vertex of the grid. The data are interpolated linearly inside a tetrahedron.
- PER_TETRA_VERTEX: Four separate data vectors are stored for each tetrahedron. The data are also interpolated linearly.

This last encoding scheme is useful for modeling discontinuous fields. In order to evaluate a field at an arbitrary location in a transparent way, amira's procedural data interface should be used. This interface is described in Section 15.5.1.

Like `HxLattice3`, the class `HxTetraData` provides a static method `create` which can be used to create a matching data field, e.g., an object of type `HxTetraScalarField3`, from an existing instance of `HxTetraData`. The `HxTetraData` object will not be copied but will be directly put into the field object. Therefore it may not be deleted afterwards. Also see Section 15.2.1.

15.4 Unstructured Hexahedral Data

In an unstructured hexahedral grid the grid cells are defined explicitly by specifying all the points in the cell. This is in contrast to regular hexahedral grids where the grid cells are arranged in a regular 3D array and thus are defined implicitly. The implementation of hexahedral grids is very similar to tetrahedral grids as described in the previous section. There are separate classes for the grid itself and for data fields defined on a hexahedral grid.

In the following two sections we introduce the grid class `HxHexaGrid` before discussing the corresponding field classes and the interface `HxHexaData`.

15.4.1 Hexahedral Grids

Hexahedral grids in amira are implemented by the class `HxHexaGrid` and its base class `HexaGrid`. Looking at the reference documentation of `HexaGrid` we observe that a hexahedral grid essentially consists of a number of dynamic arrays such as `points`, `hexas`, or `materialIds`.

- The `points` array is a list of all 3D points contained in the grid. A single point is stored as an element of type `McVec3f`. This class has the same layout as the Open Inventor class `SbVec3f`. Thus, a pointer to `McVec3f` can be cast to a pointer to `SbVec3f` and vice versa.
- The `hexas` array describes the actual hexahedra. For each hexahedron the indices of the eight points and the indices of the six faces it consists of are stored. The numbering of the points is shown in Figure 15.3. Degenerate cells such as prisms or tetrahedra may be defined by choosing the same index for neighboring points.
- The `materialIds` array contains 8-bit labels, which assign a material identifier to every hexahedron. Like the case of label fields or surfaces, the set of possible material identifiers is stored

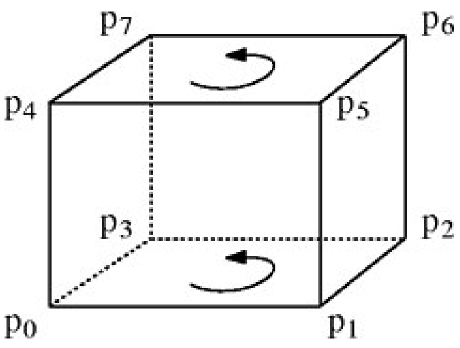


Figure 15.3: Numbering of points in a hexadron with positive volume.

in the parameter list of the grid data object.

The three arrays, `points`, `hexas`, and `materialIds`, must be provided by the 'user'. The faces of the grid are stored in an additional array called `faces`. This array can be constructed automatically by calling the member method `createFaces`. This method computes the faces from scratch and sets the face indices of all hexahedra defined in `hexas`.

Note that, in contrast to tetrahedral grids, in a hexahedral grid degenerate cells are allowed, i.e., cells where neighboring corners in a cell coincide. In this way grids with mixed cell types can be defined. The faces of a hexahedron are stored in a small dynamic array called `faces`. For a degenerate cell this array contains less then six faces.

Also note that, although non-conformal grids are allowed, i.e., grids with hanging nodes on edges and faces, currently the method *createFaces* does not detect the connectivity between neighboring hexahedra sharing less than four points. Thus, faces between such cells are considered to be external cells.

15.4.2 Data Defined on Hexahedral Grids

In most applications, you will not only have to deal with a single hexahedral grid, but also with data fields defined on it, for example scalar fields (e.g. temperature) or vector fields (e.g., flow velocity). amira provides special classes for these data modalities, namely `HxHexaScalarField3`, `HxHexaVectorField3`, `HxHexaComplexScalarField3`, `HxHexaComplexVectorField3`, and `HxHexaField3` (see class hierarchy in Section 15.1.1).

Like for fields defined on tetrahedral grids, the actual information is stored in a special member variable called *data*, which is of type `HxHexaData`. `HxHexaData` is a so-called interface, i.e. derived from `HxInterface`. The *data* variable provides transparent access to data fields defined on hexahedral grids regardless of the actual number of data components the field has. In order to access the interface without knowing the actual type of input object within a module, you may use the following statement:

```
    HxHexaData* data = (HxHexaData*)
        portData.source(HxHexaData::getClassTypeId());
    if (!data) return;
```

Data on hexahedral grids must always be of type `float`. The data values may be stored in three different ways, indicated by the encoding type defined in `HxHexaData`:

- PER_HEXA: One data vector is stored for each hexahedron. The data are assumed to be constant inside the hexahedron.
- PER_VERTEX: One data vector is stored for each vertex of the grid. The data are interpolated trilinearly inside a hexahedron.
- PER_HEXA_VERTEX: Eight separate data vectors are stored for each hexahedron. The data are also interpolated trilinearly.

The last encoding scheme is useful for modeling discontinuous fields. In order to evaluate a field at an arbitrary location in a transparent way, amira's procedural data interface should be used. This interface is described in Section 15.5.1.

Like `HxLattice3`, the class `HxHexaData` provides a static method `create` which can be used to create a matching data field, e.g., an object of type `HxHexaScalarField3`, from an existing instance of `HxHexaData`. The `HxHexaData` object will not be copied but will be directly put into the field object. Therefore it may not be deleted afterwards. Also see Section 15.2.1.

15.5 Other Issues Related to Data Classes

In this section the following topics will be covered:

- amira's procedural interface for evaluating 3D fields
- coordinate systems and transformations of spatial data objects
- defining parameters and materials in data objects

15.5.1 Procedural Interface for 3D Fields

The internal representation of a data field very much depends on whether the field is defined on a regular, tetrahedral, or hexahedral grid. There are even data types such as `HxAnnaScalarField3` or `HxAnnaVectorField3` for fields that are defined by an analytical mathematical expression. To allow for writing a module which operates on any scalar field without having to bother about the particular data representation, a transparent interface is needed. One could think of a function like

```
    float value = field->evaluate(x,y,z);
```

For the sake of efficiency, a slightly different interface is used in amira. Evaluating a field defined on tetrahedral grid at an arbitrary location usually involves a global search to detect the tetrahedron which contains that point. The situation is similar for other grid types. In most algorithms, however, the field is typically evaluated at points not far from each other, e.g., when integrating a field line. To take advantage of this fact, the concept of an abstract `Location` class has been introduced. A `Location` describes a point in 3D space. Depending on the underlying grid, `Location` may keep track of additional information such as the current grid cell number. The `Location` class provides two different search strategies, a global one and a local one. In this way performance can be improved significantly. Here is an example of how to use a `Location` class:

```
float pos[3];
float value;
...
HxLocation3* location = field->createLocation();
if (location->set(pos))
    field->eval(location, &value);
...
if (location->move(pos))
    field->eval(location, &value);
...
delete location;
```

First a location is created by calling the virtual method `createLocation` of the field to be evaluated. The two methods, `location->set(pos)` and `location->move(pos)`, both take an array of three floats as argument, which describe a point in 3D space. The `set` method always performs a global search in order to locate the point. In contrast, `move` first tries to locate the new point using a local search strategy starting from the previous position. You should call `move` when the new position differs only slightly from the previous one. Both `set` and `move` may return 0 in order to indicate that the requested point could not be located, i.e., that it is not contained in any grid cell.

In order to locate the field at a particular location, `field->eval( location, &value)` is called. The result is written to the variable pointed to by the second argument. Internally the `eval` method does two things. First it interpolates the field values, for example, using the values at the corners of the cell the current point is contained in. Secondly, it converts the result to a float value if the field is represented internally by a different primitive data type.

15.5.2 Transformations of Spatial Data Objects

In amira, all data objects which are embedded in 3D space are derived from the class `HxSpatial-Data` defined in the subdirectory `kernel/Amira` (see class hierarchy in Section 15.1.1). On the one hand, this class provides a virtual method `getBoundingBox` which derived classes should redefine. On the other hand, it allows the user to transform the data object using an arbitrary geometric transformation. The transformation is stored in an Open Inventor *SoTransform* node. This node is applied automatically to any display module attached to a transformed data object.

In total there are three different coordinate systems:

- The *world coordinate system* is the system the camera of the 3D viewer is defined in.
- The *table coordinate system* is usually the same as the world coordinate system. However, it might be different if special modules displaying, for example, the geometry of a radiotherapy device is used. These modules should call the method `HxBase::useWorldCoords` with a non-zero argument in their constructor. Later they may then call the method `HxController::setWorldToTableTransform` of the global object `theController`. In this way they can cause all other objects to be transformed simultaneously.
- Finally, the *local coordinate* system is defined by the transformation node stored for objects of type `HxSpatialData`. This transformation can be modified interactively using the transformation editor. Transformations can be shared between multiple data objects using the method `HxBase::setControllingData`. Typically, all display modules attached to a data object will share its transformation matrix, so that the geometry generated by these modules is transformed automatically when the data itself is transformed.

The transformation node of a spatial data object may be accessed using the `SoTransform* HxSpatialData::getTransform()` method, which may return a NULL pointer when the data object is not transformed.

Often it is easier to use `HxSpatialData::getTransform(SbMatrix& matrix)` instead, which returns the current transformation matrix or the identity matrix when there is no transformation. This matrix is to be applied by multiplying it to a vector from the right-hand side. It transforms vectors from the local coordinate system to the table or world coordinate system.

If you want to transform table or world coordinates to local coordinates, use `HxSpatialData::getInverseTransform( SbMatrix& matrix)`. For example, consider the following code which transforms the lower left front corner of object A into the local coordinate system of a second object B:

```
float bbox[6];
SbVec3f originWorld,originB;
SbMatrix matrixA, inverseMatrixB;

// Get origin in local coordinates of A
fieldA->getBoundingBox(bbox);
SbVec3f origin(bbox[0],bbox[1],bbox[2]);

// Transform origin to world coordinates:
fieldA->getTransform(matrixA);
matrixA.multVecMatrix(origin,originWorld);

// Transform origin from world coords to local coords of B
fieldB->getInverseTransform(inverseMatrixB);
inverseMatrixB.multVecMatrix(originWorld,originB);
```

Instead of this two-step approach, the two matrices could also be combined:

```
SbMatrix allInOne = matrixA;
```

```
    allInOne.multRight(inverseMatrixB);

    allInOne.multVecMatrix(origin,originB);
```

Note that the same result is obtained in the following way:

```
    SbMatrix allInOne = inverseMatrixB;
    allInOne.multLeft(matrixA);

    allInOne.multVecMatrix(origin,originB);
```

Since the transformation could contain a translational part, special attention should be paid when directional vectors are transformed. In this case the method `HxSpatialData::getTransformNoTranslation( SbMatrix& matrix)` should be used.

15.5.3 Parameters and Materials

For every data object an arbitrary number of attributes or parameters can be defined. The parameters are stored in a member variable `parameters` of type `HxParamBundle`. The header file of the class `HxParamBundle` is located in the subdirectory `kernel/amiramesh`.

`HxParamBundle` is derived from the base class `HxParamBase`. Another class derived from `HxParamBase` is `HxParameter`. This class is used to actually store a parameter value. A parameter value may be a string or an n-component vector of any primitive data type supported in amira (byte, short, int, float, or double). The bundle class `HxParamBundle` may hold an arbitrary number of `HxParamBase` objects, i.e., parameters or other bundles. In this way parameters may be ordered hierarchically.

Many data objects such as label fields, surfaces, or unstructured finite element grids make use of the concept of a `material` list. Material parameters are stored in a special sub-bundle of the object's parameter bundle called *Materials*. In order to access all material parameters of such an object, the following code may be used:

```
  HxParamBundle* materials = field->parameters.materials();
  int nMaterials = materials->nBundles();

  for (int i=0; i<nMaterials; i++) {
      HxParamBundle* material = materials->bundle(i);
      const char* name = material->name();
      theMsg->printf("Material[%d] = %s\n", name);
  }
```

The class `HxParamBundle` provides several methods for looking up parameter values. All these *find*-methods return 0 if the requested parameter could not be found. For example, in order to retrieve the value of a one-component floating point parameter called *Transparency*, the following code may be used:

```
  float transparency = 0;
  if (!material->findReal("Transparency",transparency))
      theMsg->printf("Transparency not defined, using default");
```

In order to add a new parameter or to overwrite the value of an existing one, you may use one of several different *set*-methods, for example:

```
  material->set("Transparency",transparency);
```

Many modules check whether a color is associated to a particular 'material' in the material list of a data object. If this is not the case, the color or some other value is looked up in the global material database amira provides. This database is represented by the class `HxMatDatabase` defined in `kernel/Amira`. It can be accessed via the global pointer `theDatabase`. Like an ordinary data object, the database has a member variable `parameters` of type `HxParamBundle` in order to store parameters and materials. In addition, it provides some convenience methods, for example `getColor(const char* name)`, which returns the color of a material, defining a new one if the material is not yet contained in the database.

Chapter 16

Documentation of Modules in amiraDev

amiraDev allows the user to write the documentation for his own modules and integrate it into the user's guide. For this, in the package directory a subdirectory named `doc` must be created, e.g.,

`$AMIRA_LOCAL/src/mypackage/doc`.

The documentation must be written in amira's native documentation style. The syntax is borrowed from the *Latex* text processing language. Documentation files for new modules can easily be created by the Tcl command `createDocFile`. To create a documentation template for `MyModule`, create such a module and type

```
MyModule createDocFile
```

in the amira console. This will generate a template for the documentation file as well as snapshots of all ports. Copy these files to

`$AMIRA_LOCAL/src/mypackage/doc`.

The file `MyModule.doc` already provides the skeleton for the module description and includes the port snapshots.

The command `createSnapshots` only creates the snapshots of the module ports. This is useful when the ports have changed and their snapshots must be updated in the user's guide.

16.1 The documentation file

In this section the basic elements of a documentation file are presented. A typical documentation file looks as follows:

```
  \begin{hxmodule}{MyModule}
  This command indicates the begin of a description file.
  MyModule is the module name.

  \begin{hxdescription}
  This block contains a general module description.
  All beginning blocks must have an end.
  \end{hxdescription}

  \begin{hxconnections}
  \hxlabel{MyModule_data}
  \hxport{Data}{\tt [required]}\\
  Here the required master connection is described.
  The hxlabel command sets a label such that this
  connection can be referenced in the documentation.

  \end{hxconnections}

  \begin{hxports}
  \hxlabel{MyModule_portDoIt}
  \hxport{PortDoIt}\\
  \hximage{MyModule_portDoIt}\\
  Here the snapshot image of PortDoIt is placed in the
  documentation. Below the snapshot, the port is described.

  Anywhere in the documentation a label can be referenced:
  \link{MyModule_data}{Text for reference}

  \end{hxports}
  \end{hxmodule}
```

This file contains the documentation of a module. It starts with

```
\begin{hxmodule}{name}
```

This statement specifies that the doc file contains the documentation of a module. The entry `name` will be automatically included in the list of modules in the amira online help. Other documentation types are available too:

```
\begin{hxmodule2}{name}{short description to ap-
pear in the list of modules}
```

```
\begin{hxfileformat}{name}
\begin{hxfilefromats2}{name}{short description to ap-
pear in the list of file formats}
\begin{hxdata}{name}
\begin{hxdata2}{name}{short description to appear in the list of data types}
\begin{hxeditor}{name}
\begin{hxeditor}{name}{short description to ap-
pear in the list of editors}
\begin{hxcomponent}{name}
\begin{hxextension}{title of an extension}
\begin{hxdemo}{title of a demo}
```

The file always must be closed by the corresponding `end` command. `hxcomponent` refers to a port which can be used in an amira module. `hxextension` refers to an extension, i.e., to a set of one or more packages which can be considered as a logical entity. Extensions are not listed in a reference section, but are included directly in the main page of the online help.

More commands

Other formats allow to format and structure the documentation:

```
\begin{itemize}
\item This is an enumeration.
      Eeach item starts with the key word item.
\end{itemize}

{\bf This will be set in bold face.}

{\it This will be set in italics.}

{\tt This will be set in Courier.}

This is a link to the \link{HxIsosurface}{Isosurface} module.
```

Formulas can be included by means of the text processor *LaTeX*. They must be written in the *Latex* syntax. This requires that *LaTeX* and *Ghostscript* are installed on the user's system. The following environment variables need to be set:

- `DOC2HTML_LATEX` points to the *LaTeX* executable. The default value is `latex` on Unix systems, and `C:/cygwin/bin/tex --fmt latex` on Windows systems.
- `DOC2HTML_DVIPS points` to the dvi-to-postscript converter. The default value is `dvips` on Unix systems and `C:/cygwin/bin/dvips` on Windows systems.

- `DOC2HTML_GS` points to *Ghostscript*. The default is `gs` on Unix systems and `C:/cygwin/bin/gs` on Windows systems.

16.2 Generating the documentation

All documentation files must be converted to HTML files and copied into the user's guide. For this purpose, the program `doc2html` is provided. Run this program from a command shell with the following option:

```
doc2html -a
```

This converts the documentation and copies the HTML files to the appropriate places in the `AMIRA_ROOT/share/doc/usersguide` directory. Call `doc2html` without option to get a complete list of options.

Chapter 17

Miscellaneous

This chapter covers a number of additional issues of interest for the amira developer. In particular, the following topics are included:

- Import of time-dependent data, including the use of `HxPortTime`
- Important global objects, such as `theMessage` and `theWorkArea`
- Save-network issues, making save-network work for custom modules
- Troubleshooting, providing a list of common errors and solutions

17.1 Time-Dependent Data And Animations

This section covers some more advanced topics of amiraDev, namely the handling of dynamic data sets and the implementation of animated compute tasks. Before reading the section you should at least know how to write ordinary IO routines and modules.

17.1.1 Time Series Control Modules

In general, the processing of time-dependent data sets is a challenging task in 3D visualization. Usually not all time steps of a dynamic data series can be loaded at once because of insufficient main memory. Even if all time steps would fit into memory it is usually not a good idea to load every time step as a separate object in amira. This would result in a large number of icons in the object pool. The selection between different time steps would become difficult.

A better solution comprise special-purpose control modules. An example is the time series control module described in the user's guide. This module is created if a time series of data objects each stored in a separate file is imported via the *Load time series...* option of the main window's file menu. Instead of loading all time steps together the control module loads only one time step at once. The current time step can be adjusted via a time slider. When a new time step is selected the data objects associated with the previous one are replaced.

If you want to support a file format where multiple time steps are stored in a common file, you can write a special time series control module for that format. For each format a special control module is needed because seeking for a particular time step inside the file of course is different for each format. For convenience, you may derive a control module for a new format from the class *HxDynamicDataControl* contained in the package *hxtime*. This base class provides a time slider and a virtual method `newTimeStep(int k)` which is called whenever a new new time step is to be loaded. In contrast to the standard time series control module in most other control modules data objects should be created only once. If a new time step is selected existing objects should be updated and reused instead of replacing them by new objects. In this way the burden of disconnecting and reconnecting down-stream objects is avoided.

17.1.2 The Class HxPortTime

In principal, an ordinary float slider (*HxPortFloatSlider*) can be used to adjust the time of a time series control module or of some other time-dependent data object. However, in many cases the special-purpose class *HxPortTime* defined in the package *hxtime* is more appropriate. This class can be used like an ordinary float slider but it provides many additional features. The most prominent one is the possibility to auto-animate the slider. In addition, *HxPortTime* can be connected to a global time object of type HxTime. In this way multiple time-dependent modules can be synchronized. In order to create a global time object, choose *Time* from the main window's *Edit Create* menu.

Another feature of *HxPortTime* is that the class is also an interface, i.e., it is derived from *HxInterface* (compare Section 15.1.2). In this way it is possible to write modules which can be connected to any object containing an instance of *HxPortTime*. An example is the *DisplayTime* module. In order to access the time port of a source object the following C++ dynamic cast construct should be used:

```
HxPortTime* time = dynamic_cast<HxPortTime*>(
    portData.source(HxPortTime::getClassTypeid()));
```

In the previous section we discussed how time-dependent data could be imported using special-purpose control modules. Another alternative is to derive a time-dependent data object from an existing static one. An example of this is the class *MyDynamicColormap* contained in the demo package of amiraDev. Looking at the header file `packages/mypackage/MyDynamicColormap.h` in the local amira directory you notice that this class is essentially an ordinary colormap with an additional time port. Here is the class declaration:

```
class MYPACKAGE_API MyDynamicColormap : public HxColormap
{
    HX_HEADER(MyDynamicColormap);
```

```
  public:
    // Constructor.
    MyDynamicColormap();

    // This will be called when an input port changes.
    virtual void compute();

    // The time slider
    HxPortTime portTime;

    // Implements the colormap
    virtual void getRGBA1(float u, float rgba[4]) const;
};
```

The implementation of the dynamic colormap is very simple too (see the file `MyDynamicColormap.cpp`). First, in the constructor the time slider is initialized:

```
portTime.setMinMax(0,1);
portTime.setIncrement(0.1);
portTime.setDiscrete(0);
portTime.setRealTimeFactor(0.5*0.001);
```

The first line indicates that the slider should go from 0 to 1. The increment set in the next line defines by what amount the time value should be changed if the backward or the forward button of the slider is pressed. The next line unsets the discrete flag. If this flag is on, the slider value always would be an integer multiple of the increment. Finally, the so-called real-time factor is set. Setting this factor to a non-zero value implies that the slider is associated with physical time in animation mode. More precisely, the number of microseconds elapsed since the last animation update is multiplied with the real-time factor. Then the result is added to the current time value.

In order to see the module in action compile the demo package, start amira (use the `-debug` option if you compiled in debug mode), and choose *DynamicColormap* from the main window's *Edit Create* menu. Attach a *DisplayColormap* module to the colormap and change the value of the colormap's time slider. Animate the slider. The speed of the animation can be adjusted by resetting the value of the real-time factor using the Tcl command `DynamicColormap time setRealTimeFactor`.

17.1.3 Animation Via Time-Out Methods

In some cases you might want certain methods to be called in regular intervals without using a time port. There are several ways to do this. First, you could use the Open Inventor class `SbTimerSensor` or related classes. Another possibility would be to use the Qt class `QTimer` (this requires that you install the correct version of the Qt library in addition to amiraDev on your system). However, both methods have the disadvantage that the application can get stuck if too many timer events are emitted at once. In same cases it could even be impossible to press the stop button or some other button for turning off user-defined animation. For this reason amira provides its own way off registering time-out methods. The relevant methods are implemented by the class `HxController`. Suppose, you have written a module with a member method called `timeOut`. If you want this method to be called automatically once in a second, you can use the following statement:

```
    theController->addTimeOutMethod(
this,(HxTimeOutMethod)timeOut,1000);
```

In order to stop the animation again, use

```
    theController->removeTimeOutMethod(
this,(HxTimeOutMethod)timeOut);
```

Instead of using a member method of an amira object class, you can also register an arbitrary static function using the method `addTimeOutFunction` of class `HxController`. The corresponding remove method is called `removeTimeOutFunction`. For more information, see the reference documentation of `HxController`.

The amiraDev demo package contains the module `MyAnimateColormap` which makes use of the above time-out mechanism. The source code of the module again is quite easy to understand. After compiling the demo package, you can attach to module under the name *DoAnimate* to an existing colormap. The colormap then is modified and copied. After pressing the animate toggle of the module the output colormap is shifted automatically at regular intervals. Note, that in this example the `fire` method of the module is used as time-out method. `fire` invokes the modules `compute` method and also updates all down-stream objects.

17.2 Important Global Objects

Beside the base classes of modules and data objects, there are some more classes in the amira kernel that are important to the developer. Many of these classes have exactly one global instance. A short summary of these global objects is presented here. For details, please refer to the online reference documentation by looking at the file `share/programmersguide/index.html` in the amira root directory.

HxMessage: This class corresponds to the amira console window in the lower right part of the screen. There is only one global instance of this class, which can be accessed by `theMsg`. All text output should go to this object. Text can be printed using the function `theMsg->printf("...",...)`, which supports common C-style `printf` syntax. `HxMessage` also provides static methods for popping up error and warning messages or simple question dialogs.

HxObjectPool: This class maintains the list of all currently existing data objects and modules. In the graphical user interface the object pool is represented by the green area in the main window containing the modules' and data objects' icons. There is only one global instance of this class, which can be accessed by the pointer `theObjectPool`.

HxWorkArea: This class displays the ports of selected objects and provides the progress bar and busy-state functionality. Important functions are `startWorking`, `stopWorking`, `wasInterrupted` as well as `busy` and `notBusy`. There is only one global instance of this class, which can be accessed by the pointer `theWorkArea`.

HxFileDialog: This class represents the file browser used for loading and saving data. Normally the developer does not need to use this class since the standard I/O mechanism is completely implemented in the amira kernel. However, for special purpose modules, a separate file browser might be useful. There is a global instance of this class, which can be accessed by the pointer `theFileDialog`.

HxResource: This class maintains the list of all registered file formats and modules as defined in the package resource files. It also provides information about the amira root directory, the local amira directory, the version number, and so on. Normally there is no need for the developer to use this class directly. There is no instance of this class, since all its members are static.

HxViewer: This class represents an amira 3D viewer. There can be multiple instances which are accessed via the method `viewer` of the global object `theController`. Normally you will not not need to use this class. Instead, you should use the member functions `showGeom` and `hideGeom` which every module and data object provides in order to display geometry.

HxController: This class controls all 3D viewers and Open Inventor geometry. In order to access a viewer you may use the following statement:

```
    HxViewer* v0 = theController->viewer(0,0);
```

The first argument indicates the ID number of the viewer to be retrieved. In total there may be up to 16 different viewers. The second argument specifies whether the viewer should be created or not if it does not already exist.

HxColorEditor: amira's color editor. Used, for example, to define the background color of the viewer. In a standard module you should use a port such as `HxPortColorList` or `HxPortColorMap` instead of directly accessing the color editor. There is a global instance of this class, which can be accessed by the pointer `theColorEditor`.

HxHTML: A window used to display HTML files. This class is used for amira's online help. The global instance used for displaying the online user's guide and the online programmer's guide can be accessed by the pointer `theBrowser`.

HxMatDatabase: This class represents amira's global parameter and material database. For example, the database contains default colors for a number of biomedical tissue types such as fat, muscle, and bone. The database can be accessed by the global pointer `theDatabase`. Details about the material database are discussed in Section 15.5.3.

17.3 Save-Network Issues

This section describes the mechanism used in amira to save networks. For most modules this is done transparently for the developer.

The menu command "Save Network" dumps a Tcl script that should reconstruct the current network.

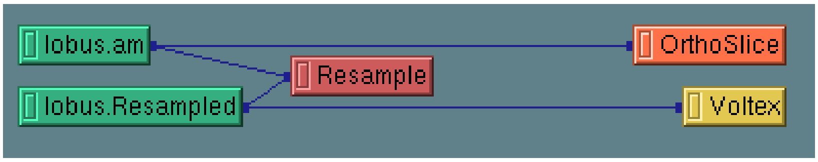


Figure 17.1: When loading this network, the *Resample* module recreates the *lobus.Resampled* data object on the fly.

Essentially this is done by writing a `load ...` command for each data object, a `create ...` command for each module and `setValue ...` commands for each port of a module.

This suffices to reconstruct the network correctly if all information about a module's state is kept in the module's ports only. If this is not the case, e.g., if the developer uses extra member variables that are important for the modules current state, those values are not restored automatically. If you cannot avoid this, you must extend the 'Save Network' functionality of your module. In order to do so, you can override the virtual function `savePorts` so that it writes additional Tcl commands. For example, let us take a look at the `HxArbitraryCut` class, which is the base class e.g., for the `ObliqueSlice` module and which has to save its current slice orientation:

```
void HxArbitraryCut::savePorts(FILE* fp)
{
    HxModule::savePorts(fp);
    ...
    fprintf(fp, "%s setPlane %g %g %g %g %g %g %g %g %g\n",
        getName(),
        origin()[0], origin()[1], origin()[2],
        uVec()[0], uVec()[1], uVec()[2],
        vVec()[0], vVec()[1], vVec()[2]);
}
```

Note that this method requires that `HxArbitraryCut` or some of its parent classes implement the Tcl command `setPlane`. Hints about implementing new Tcl commands are given in Section 14.2.2.

Some remarks about how to generate the load command for data objects are given in Section 13.2.

There is a special optimization for data objects created by computational modules. amira automatically determines whether data objects which are created by other modules are not yet saved and asks the user to do so if necessary. However, in some cases, this may not be desired, since saving the data object consumes disk space and regenerating can sometimes be nearly as fast as loading the object from disk. As an example, consider the network in figure 17.1. In this case the resample module can automatically recompute the `lobus.Resampled` data object when the network is loaded. In order to determine whether a compute module is able to do so, the module must implement the function `int HxResample::canCreateData(HxData* data, McString& createCmd)`. This function is called whenever a network containing newly created data objects is saved and these objects have not yet been saved but are still connected to a compute module. The method should return 1 if the compute module is able to recreate that particular data object. In this case the corresponding Tcl command should be stored in the string `createCmd`. When executed, the Tcl command should return the name of the newly created data object.

Determining whether a compute module can create a data object may be tricky. Typically, it must be assured that in the time between the actual creation of the data object by the computational module and the execution of the save network command neither the parameters nor the input has changed, and that the resulting data object had not been edited.

In order to implement this behavior, most compute modules use a flag that they set when they create a data object and which they clear when the module's `update()` method is called, indicating that some input has changed. In order to check whether the data object was edited, the data object's `touchTime` variable is saved. `touchTime` is increased automatically whenever a module is edited. A typical method could look like this:

```
int HxResample::canCreateData(HxData* data, McString& createCmd)
{
    if (resultTouchTime != data->getTouchTime() ||
        parameterChanged)
            return 0;

    createCmd.printf("%s action hit; %s fire; %s getResult\n",
        getName(), getName(), getName());
    return 1;
}
```

17.4 Troubleshooting

This section describes some frequently occurring problems and some general problem solving approaches related to amira development.

The section is divided into two parts: Problems which may occur when compiling a new package, and problems which may occur when executing it.

17.4.1 Compile-Time Problems

Unknown identifier, strange errors: A very common problem occurring in C++ programming is the omission of necessary include statements. In amira, most classes have their own header file (.h file) containing the class declaration. You must include the class declaration for each class that you are using in your code. When you get strange error messages that you do not understand, check whether all classes used in the neighborhood of the line the compiler complains about have their corresponding include statement.

Unresolved symbols: If the linker complains about unresolved symbols, you probably are missing a library on your link line. The amira development wizard makes sure that the amira kernel library and important system libraries are linked. If you are using amira data classes, you will need to link with the corresponding package library `hxfield`, `hxcolor`, `hxsurface`, and so on. To add libraries to your link line on Unix, edit the `GNUmakefile`, find the line starting with `LIBS += ...`, and append `-l<name>`, where `<name>` is the name of the package you want to add. On Windows, use Visual Studio's project settings dialog. Details are given in Section 11.5.

17.4.2 Run-Time Problems

The module does not show up in the popup menu: If your module did compile, but is not visible in the popup menu of a corresponding data object, there is probably a problem with the resource file. The resource file will be copied from your package's `share/resources` directory to the directory `share/resources` in your local amira directory. Verify that this worked. **Note:** Currently on Windows, the resource files are copied in a post link step. Therefore, if you change the resource file after linking, you must build the package again, e.g., by changing one of the source files.

If the resource file is present, the next step is to check whether it is really parsed. Add a line `echo "hello mypackage"` to the resource file. Verify that the message appears in the amira console when amira starts. If not, probably the environment variable AMIRA_LOCAL is not set correctly.

If the file is parsed, but the module still does not show up, the syntax of the rc file entry might be wrong or you specified a wrong primary data type, so that the module will appear in the menu of a different data class.

There is an entry in the pop-up menu, but the module is not created: Probably something is wrong with the shared library (the .so or .dll file). In the amira console, type `dso verbose 1` and try to create the module again. You will see some error messages, indicating that either the dll is not found, or that it cannot be loaded (and why) or that some symbol is missing. Check whether your building mode (debug/optimize) and execution mode are the same. In particular, if you have compiled debug code you must start amira using the `-debug` command line option (see Section 11.5).

A read or write routine does not work: The procedure for such problems is the same. First check whether the load function is registered. Then verify that your save-file format shows up in the file format list when saving the corresponding data object. For a load method, right click on a filename in the load-file dialog. Choose format and check whether your format appears in the list. If that is the case, you probably have a dll problem. Follow the steps above. If the library can be loaded, but the symbol cannot be found, your method may have either a wrong signature (wrong argument types) or on Windows you might have forgotten the `<PACKAGE>_API` macro. This macro indicates that the routine should be exported by the DLL.

In general, if you have problems with **unresolved** and/or **missing symbols** you should take a look at the symbols in your library. On Unix, type `nm lib/arch-*-Debug/libmypackage.so`. On Windows, type in a command shell: `dumpbin /exports bin/arch-Win32-Debug/mypackage.dll`.

17.4.3 Debugging Problems

Setting breakpoints does not work: Since amira uses shared libraries, the code of an individual package is not loaded even after the program has started. Therefore some debuggers refuse to set breakpoints in such packages or disable previously set breakpoints. To overcome this problem, first create your module and then set the breakpoint. If you want to debug a module's constructor or a read or write routine, of course, this does not work. In these cases, load the library by hand, by typing into the amira console `dso open libmypackage.so` (if your package is called mypackage). Then set the breakpoint and create your module or load your data file.

Chapter 18

Online Class Documentation

The online class documentation is a reference guide consisting of a description of every C++ class that is part of the Amira developer version. The class documentation contains links for example to base classes or to the formatted header files. In order to view it, point a web browser such as Internet Explorer or Netscape Navigator to the file `share/devref/index.html` located in the Amira root directory, i.e., in the directory where Amira is installed. The documentation can also be viewed using the Amira help viewer, but a web browser might be more convenient because it allows you to define bookmarks or to open multiple windows.

Part IV

amiraVR Manual

Chapter 19

amiraVR Configuration

amiraVR is an amira extension providing support for large tiled displays as well as immersive multi-wall displays like *CAVEs* or *Holobenches*. amiraVR supports multi-threaded rendering on multi-pipe machines, head tracking, active and passive stereo modes, advanced 3D user interaction, soft edge blending and many more. Any VRCO *trackd*-compatible tracking system can be used together with amiraVR. Existing amira modules can be directly used in an immersive environment by means of 3D menus. In addition, a simple API is provided, allowing an amiraDev programmer to add display modules with a specific interaction behaviour.

Beginning with version 3.1 amiraVR can also be run on a graphics cluster. The particular requirements and limitations of this cluster version are described in a separate section below.

- amiraVR essentials
- Flat screen configurartions
- Immersive configurations
- Calibrating the tracking system
- The amiraVR cluster version

Working with amiraVR:

- 3D user interaction, including the 3D menu
- Writing amiraVR custom modules

amiraVR Reference:

- Config file reference
- The amiraVR control module

19.1 amiraVR essentials

amiraVR can be configured in many different ways. A particular amiraVR configuration is described in a config file under `$AMIRA_ROOT/share/config` or `$AMIRA_LOCAL/share/config`. The config file contains things like the physical extent of the screens of the display system, information about the X display or the parts of the desktop mapped onto the screens, as well as calibration data for the tracking system.

When amiraVR is installed the amira main window provides an additional menu labelled *Config*. This menu lists all config files found in the config directory. Once a particular configuration is selected, an *amiraVR* module is created (if there not already exists one). Depending on the particular configuration the amiraVR module allows one to to connect to the tracking system, to calibrate the tracking system, and to activate additional options.

The trackd server

amiraVR makes use of the VRCO *trackd* software in order to access the tracking system. However, trackd itself is not part of amiraVR. It has to be purchased and installed separatley. For more information about this product please refer to `www.vrco.com` or `www.tgs.com`.

Before a tracking system can be used in amiraVR the trackd server has to be started. The server connects to the tracking system and provides the actual tracker and controller data in two shared memory segments, which are read by amira. In contrast to previous versions of amiraVR it is *not* necessary anymore to manually link additional libraries like `libtrackdAPI.so` into the amira lib directory. Once the trackd server is running it can be accessed automatically.

Starting amira

After amiraVR has been installed amira can be started as usual. However, in order to activate parallel rendering of screens assigned to different thread groups, amira should be started with the `-mt` command line options. This option enables multi-threading. In order to permanently activate multi-threading, the environment variable `AMIRA_MULTITHREAD` can be set. In order to disable multi-threading when `AMIRA_MULTITHREAD` is set, amira can also be started with the `-st` command line option (single-threaded mode).

Note: On SGI Onyx systems there are known problems with the OpenGL driver related to the use of texture objects in different OpenGL contextes. If you are rendering more than one screen on a single pipe, you should define the environment variable `AMIRA_NO_CONTEXT_SHARING`. This bug will probably be fixed in IRIX 6.5.17 and higher.

19.2 Flat screen confi gurations

A flat screen configuration consists of usually two or more screens forming a bigger 2D virtual graphics window commonly called a *tiled display* or *power wall*. Users can interact with the ordinary 2D mouse, i.e., mouse events in the different windows are translated and interpreted in the 2D virtual window. There is no need for a tracking system in case of a flat screen configuration. For such a configuration the only option provided by the amiraVR control module is a stereo toggle allowing the user to enable or disable stereo viewing. Besides standard OpenGL active stereo modes passive stereo modes can also be configured. In the most simple case this is done by defining two full screen windows on two different channels, one for the left eye view and one for the right eye view. This particular configuration is illustrated in Figure 19.1. Other passive stereo configurations are possible too, e.g., a super-wide tiled passive stereo configuration with four channels (left side left eye, left side right eye, right side left eye, right side right eye), possibly with an overlap region for soft-edge blending.

The main advantage of a flat screen configuration is its ease of use. There is no need for a tracking system. Flat screen configurations are well suited for presentations targeted to a larger audience, e.g., presentations in a seminar room or in a lecture hall.

In the following we describe some common flat screen configurations in more detail, namely a standard-resolution two-projector passive stereo configuration, a super-wide two-projector mono configuration with soft-edge blending, and a tiled 2x2 four-channel monitor configuration. For some cases also hardware solutions are available, namely special-purpose video splitters converting an interlaced active stereo signal into separate non-interlaced left eye and right eye signals for the passive stereo case, or hardware edge-blending units for blended super-wide configurations. However, these hardware solutions are usually expensive and less flexible. Therefore they are often not suitable for temporary demonstrations or experiments.

- A two-channel passive stereo configuration
- A super-wide configuration with soft-edge blending
- A tiled four-channel 2x2 monitor configuration

19.2.1 Example: A two-channel passive stereo configuration

For a single-screen passive stereo projection system two video projectors emitting orthogonally polarized light are required. One projector displays the left eye image, the other one the right eye image. This is illustrated in Figure 19.1. Both images are projected onto a non-depolarizing screen either using front projection or rear projection. Observers wear suitable light-weight polarized glasses so that each eye only sees its own image, but not the image determined for the other eye. Without special-purpose hardware such a passive stereo projection system can be driven in full screen mode even using a very inexpensive low-end dual-head graphics adapter (for example NVidia GeForce2 MX TwinView).

We assume that the dual-head graphics computer is configured so that the left half of the desktop is output on one channel and the right half is output on the other channel. Here is the amiraVR configuration file:

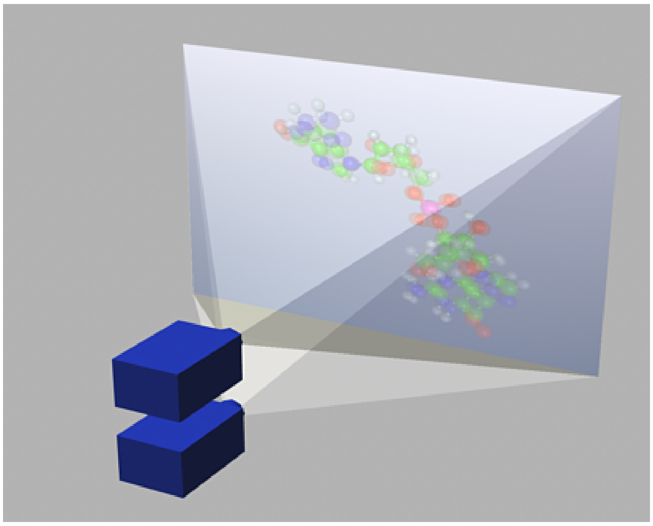


Figure 19.1: Sketch of a single-screen passive stereo projection system.

```
#Inventor V2.1 ascii

Separator {
    SoScreen {
        name                "Left eye view"
        channelOrigin   0 0
        channelSize     0.5 1
        tileOrigin      0 0
        tileSize        1 1
        cameraMode      LEFT_VIEW
    }
    SoScreen {
        name                "Right eye view"
        channelOrigin   0.5 0
        channelSize     0.5 1
        tileOrigin      0 0
        tileSize        1 1
        cameraMode      RIGHT_VIEW
    }
}
```

The fields *channelOrigin* and *channelSize* indicate that the two windows exactly cover the left half and the right half of the desktop. The fields *tileOrigin* and *tileSize* indicate that both screens should display the full viewer window, i.e., there is no tiling at all. Finally, the field *cameraMode* indicates that one screen should display the left eye view and the other the right eye view. Note, that the graphics computer need not to support active stereo for this configuration. In order to change the default stereo parameters *eye offset* and *stereo balance* the following standard amira Tcl command can be used:

```
    viewer 0 setStereo [-b <balance>] <offset>
```

Here `<offset>` denotes the eye offset and `<balance>` denotes the stereo balance, i.e., the location of the zero parallax plane. Depending on the balance value objects appear to be in front of the projection screen or behind it.

19.2.2 Example: A super-wide configuration with soft-edge blending

Super-wide images with two times the XGA- or SXGA-resolution can be displayed using two projectors and a low-end dual-head graphics adapter (for example NVidia GeForce2 MX TwinView). However, often it is very desirable to have an overlap between the two projected images. In the overlap region one image softly fades out from full intensity to black, while the other image fades in from black to full intensity. This technique is called *soft-edge blending* (compare Figure 19.2). With soft-edge blending the border between the two projected images becomes almost invisible. amiraVR is able to generate two partially overlapping images. In addition, the soft-edge can be computed in software. With these features perfect full-screen demos can be presented on a super-wide projection system.

We assume that the dual-head graphics computer is configured so that the left half of the desktop is output on one channel and the right half is output on the other channel. We further assume that there is a 20 percent overlap between the projected images of the two projectors. Here is the amiraVR configuration file:

```
#Inventor V2.1 ascii

Separator {
    SoScreen {
        name            "Left half"
        channelOrigin   0 0
        channelSize     0.5 1
        tileOrigin      0 0
        tileSize        0.6 1
        softEdgeOverlap [ 0, 0.2, 0, 0 ]
        softEdgeGamma   [ 0, 1.2, 0, 0 ]
    }
    SoScreen {
        name            "Right half"
```

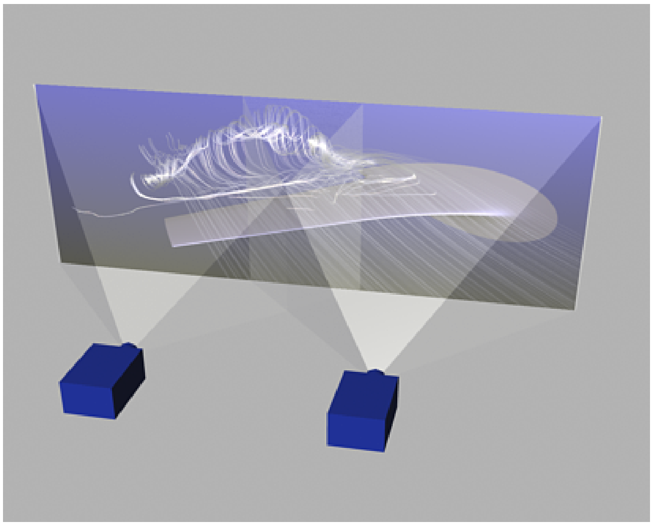


Figure 19.2: Sketch of a super-wide projection system with soft-edge blending.

```
        channelOrigin   0.5 0
        channelSize     0.5 1
        tileOrigin      0.4 0
        tileSize        0.6 1
        softEdgeOverlap [ 0.2, 0, 0, 0 ]
        softEdgeGamma   [ 0, 1.2, 0, 0 ]
    }
}
```

The fields *channelOrigin* and *channelSize* indicate that the two windows exactly cover the left half and the right half of the desktop. The fields *tileOrigin* and *tileSize* indicate that both screens display an area of 0.6 times the width of the full tiled window. The first screen displays the left half of that window, the second screen displays the right half. The field *softEdgeOverlap* specifies the relative width of the soft-edge region at the left, right, bottom, and top border of the screen. For the first screen there is a 20 percent soft-edge region at the right border, for the second screen there is a 20 percent soft-edge region at the left border. Finally, the field *softEdgeGamma* specifies the gamma factor for the soft-edge region. The gamma factor determines how the fading from full intensity to black is done. A gamma factor of one means a linear transition in terms of RGB values. However, since RGB values are usually not mapped linearly to light intensity by the video projector often a decrease of overall light intensity is observed in the overlap region. In order to compensate for this a gamma factor larger than one can be used.

In order to facilitate the initial calibration of a super-wide projection system with soft-edge blending the amiraVR module provides a special-purpose Tcl command *setSoftEdge*. This command is used in the following way:

```
    AmiraVR setSoftEdge <overlap> <gamma>
```

This command automatically creates a two-screen configuration similar to the one above. However, the correct values for *tileSize*, *tileOrigin*, and *softEdgeOverlap* are computed automatically from the relative overlap value. By gradually adapting this value the correct settings for a given but unknown setup of the projection system can be easily found.

19.2.3 Example: A tiled four-channel 2x2 monitor configuration

There are some interesting graphics workstations with two two-channel graphics pipes. One example is the SGI Octance Fuel. With such a computer four different monitors or projectors can be used. For some purposes it is useful to have a single viewer subdivided into 2x2 parts and each part being displayed by a different monitor. Such a configuration can be easily created using amiraVR.

We assume that the graphics computer has two pipes with two channels each. The two pipes are configured as two separate X11 screens, `:0.0` and `:0.1`. The left and right part of each screen is output on the two different channels of the particular pipe. Then the configuration files looks as follows:

```
#Inventor V2.1 ascii

Separator {
    SoScreen {
        name            "Upper left monitor"
        display         ":0.0"
        channelOrigin   0 0
        channelSize     0.5 1
        tileOrigin      0 0.5
        tileSize        0.5 0.5
        threadGroup     0
    }
    SoScreen {
        name            "Upper right monitor"
        display         ":0.0"
        channelOrigin 0.5 0
        channelSize 0.5 1
        tileOrigin      0.5 0.5
        tileSize        0.5 0.5
        threadGroup     0
    }
    SoScreen {
        name            "Lower left monitor"
        display         ":0.1"
        channelOrigin   0 0
        channelSize     0.5 1
        tileOrigin      0 0
        tileSize        0.5 0.5
        threadGroup     1
    }
    SoScreen {
        name            "Lower right monitor"
        display         ":0.1"
        channelOrigin 0.5 1
        channelSize 0.5 1
        tileOrigin      0.5 0
        tileSize        0.5 0.5
        threadGroup     1
    }
}
```

Since the two X11 screens `:0.0` and `:0.1` are driven by two independent graphics pipes it makes sense to perform the rendering on these pipes in parallel. This is done by assigning the corresponding screens two different thread groups. Note, that for all common current graphics architectures it makes no sense to render multiple windows on the same pipe in parallel. Typically, this even implies a significant performance decrease. Therefore, here we use only two thread groups instead of four. In order to actually activate parallel rendering amira must be started with the command line option `-mt`. Alternatively, the environment variable `AMIRA_MULTITHREAD` can be defined.

19.3 Immersive confi gurations

In amiraVR an immersive configuration differs from a flat screen configuration mainly in the way the screens are described in the config file. While for a flat screen configuration it was sufficient to specify which part of a big 2D virtual screen was covered by each screen, for an immersive configuration the true physical coordinates of the screens have to be specified. Knowing the exact spatial arrangements of the screens has the advantage that correct perspective views can be computed for all screens, provided the 3D position of the observer is also known. In particular, the different screens no longer need to be arranged in a plane. Instead, the screens might be arranged perpendicular to each other like in a *CAVE* or on a *Holobench*. In fact, amiraVR supports any other oblique arrangement as well.

Since the 3D position of the observer need to be known in order to compute correct perspective views usually the observer's eye position is tracked in an immersive environment. In amiraVR this so-called *head tracking* can be achieved using a variety of different tracking systems. Instead of accessing the tracking system directly an intermediate software layer is used, namely the *trackd* software of VRCO. *trackd* runs as a server, communicates with the tracking system, and stores the tracking data in a shared-memory areas which then is read by amiraVR. Using a second sensor not only the observer's eye position can be tracked, but also the position of a virtual wand, i.e., a kind of pointing or interaction device to be held in a hand. For large planar screen configurations it sometimes also makes sense to use a virtual wand without head tracking. The images then will always be computed for a fixed observer's eye position. Image flicker due to noise in the tracking data is avoided, which is an advantage for 3D demonstrations in front of a larger audience.

Obviously, immsersive configurations are more difficult to setup than flat screen configurations. Besides defining a suitable amiraVR configuration file, also the tracking system and the *trackd* server have to be properly initialized. Finally, the tracking system has to be calibrated for use with amiraVR. In the following sections we first want to present some example config files for common immersive environments. In particular config files for a single-wall workbench, for a double-wall Holobench, and for a four-sided CAVE shall be discussed. The process of calibrating the tracking system and customizing 3D user interaction is described in subsequent sections.

- A Workbench configuration
- A Holobench configuration
- A CAVE configuration

19.3.1 Example: A Workbench configuration

A single-screen 3D projection system is often called an immersive workbench. The actual projection screen can either be in up-right position, or it can be oriented like the surface of a table. Of course, any other orientation is possible too. There are even devices like the Barco Baron with can be arbitrarily tilted between fully horizontal and fully vertical position. For amiraVR there is no difference between these configurations, as long as the tracking system is calibrated in the right way. Traditionally a workbench is driven by a CRT projector using active stereo. However, today also passive stereo systems based on two LCD or DLP projectors become more common. In the config file below we assume that an active stereo system is used, or that a passive stereo system is connected via an apropriate signal splitter. This means that the workbench can be used with any standard stereo-capable graphics computer. No dual-head or multi-pipe computer is necessary. An example of a single-screen workbench is shown in Figure 19.3.

An amiraVR config file for driving a single-screen immersive workbench with a tracked 3D input device but without head tracking is listed below. Instead of actually tracking the observer's eye position in this case a fixed default camera position is used. This can be useful for demonstrations in front of small or medium-size groups, since it produces less fidget images. In addition, it saves a second sensor for the tracking system.

```
#Inventor V2.1 ascii
```

Figure 19.3: A single-screen 3D projection system.

```
Separator {
    SoScreen {
        name                "Workbench"
        lowerLeft           0 0 0
        lowerRight          170 0 0
        upperRight          170 130 0
        upperLeft           0 130 0
        cameraMode          ACTIVE_STEREO
    }
    SoTracker {
        server              "4147:4148"
        autoConnect         TRUE
        wandTrackerId       0
        headTrackerId       -1
        defaultCameraPosition 85 65 140
        defaultObjectPosition 85 65 0
        referencePoints [ 0 0 0, 170 0 0, 170 130 0, 0 130 0 ]
    }
}
```

In the *SoScreen* section of the config file the physical coordinates of the four corners of the workbench are defined. This can be done using an arbitrary right-handed coordinate system with arbitrary units. In this case the coordinates might be specified in centimeters. The origin of the coordinate system was chosen in the lower left corner of the screen. By default, a full-screen graphics window is opened when activating this configuration. The field *cameraMode* indicates that the graphics window should be opened in active stereo mode by default.

In the *SoTracker* section of the config file the tracking system is described. First the shared memory ids of the trackd server as specified in the *trackd.conf* file are listed. The syntax is *id of controller reader:id of tracker reader*. The *autoConnect* field indicates that a connection to the trackd server should be established automatically as soon as the configuration is activated. *wandTrackerId* denotes the trackd sensor id of the 3D input device. Head tracking is disabled by setting *headTrackerId* to -1. Instead the default camera position set in the line below is used. This position must be specified using the same coordinate system as the screen. In this case the camera is located 140 cm in front of the screen's center. The default object position is the position where the scene is placed by default (or whenever a *view all* request comes from the viewer).

At the end of the config file four reference points are specified, namely the four corners of the screen. Before the configuration can be actually used, the tracking system has to be calibrated (see section 19.4). This is done by placing the input device at the reference points and clicking an input button. Once the tracking system is calibrated, the config file should be written by clicking the *write config* button of the amiraVR control module. The new config file will contain the same information listed above, but in addition it will also contain some calibration data, e.g., the transformation between raw tracker coordinates and screen coordinates.

19.3.2 Example: A Holobench configuration

A Holobench (TM) is a special display system consisting of two screens oriented perpendicular to each other. One screen is oriented vertically, the other one is oriented horizontally like a table. On a good Holobench there is almost no visible border between the two screens. Provided the observer's eye position is known, correct perspective views can be computed so that the displayed scene finally does not appear to have any break. However, in contrast to a single-screen workbench this is only the case for the tracked observer. Other spectators will more or less clearly notice a break.

In order to drive a workbench at least a stereo-capable dual-head graphics computer is required. In principle, two different settings are possible. Either, there is a big desktop and one half of it is output on a first channel and the other one is output on a second channel. Or, there are two independent graphics adapters (pipes) which are configured as two different X11 displays or as one display with two X11 screens. In the config file below we assume, that the desktop is split into two halfs (left and right). An example of how to use a multi-pipe graphics computer is presented in the next section where a 4-sided CAVE configuration is described.

```
#Inventor V2.1 ascii

Separator {
    SoScreen {
        name            "Vertical Screen"
        lowerLeft       0 0 0
        lowerRight      180 0 0
        upperRight      180 110 0
        upperLeft       0 110 0
        channelOrigin   0 0
        channelSize     0.5 1
        cameraMode      ACTIVE_STEREO
    }
    SoScreen {
        name            "Horizontal Screen (rotated)"
        lowerLeft       180 0 0
        lowerRight      0 0 0
        upperRight      0 0 110
        upperLeft       180 0 110
        channelOrigin   0.5 0
        channelSize     0.5 1
        cameraMode      ACTIVE_STEREO
    }
    SoTracker {
        server          "4147:4148"
        autoConnect     TRUE
        wandTrackerId   1
        headTrackerId   0
        leftEyeOffset   6 0 0
        rightEyeOffset  13 0 0
        defaultCameraPosition 90 55 110
        defaultObjectPosition 90 20 20
        referencePoints [ 60 0 110, 120 0 110, 120 0 55, 60 0 55 ]
    }
}
```

In the two *SoScreen* sections of the config file the geometry of the Holobench is described by specifying the physical coordinates of the four corners of each screen. An arbitrary right-handed coordinate system with arbitrary units can be chosen. Here the coordinates are specified in centimeters and the origin was put in the lower left corner of the vertical screen. Notice, that the horizontal screen was rotated by 180 degress with respect to the vertical screen. I.e., instead of the upper left corner the lower right corner of the horizontal screen is located at the origin. This is because the horizontal image of a Holobench is usually projected that way. If the corresponding lower scan-lines of the two images meet at the border between the two screens artifacts due to delayed response of the active shutter glasses are avoided. The fields *channelOrigin* and *channelSize* indicate that the graphics window for the vertcial screen should be opened on the left half of the desktop, while the graphics window for the horizontal screen should be opened on the right half. Both windows are opened in stereo mode by default as specified by *cameraMode*.

In the *SoTracker* section of the config file the tracking system is described. First the shared memory ids of the trackd server as specified in the *trackd.conf* file are listed. The syntax is *id of controller reader:if of tracker reader*. The *autoConnect* field indicates that a connection to the trackd server should be established automatically as soon as the configuration is activated. *wandTrackerId* denotes the trackd sensor id of the 3D input device. *headTrackerId* denotes the trackd sensor id of the head sensor which is usually mounted at the shutter glasses. The fields *leftEyeOffset* and *rightEyeOffset* specify the actual position of the eyes with respect to the head sensor. Standing in front of the Holobench the x-axis points horizontally to the right, the y-axis points upwards, and the z-axis points towards the observer. In this case we assume the head sensor to be mounted on the left side of the glasses since both left and right eye have a positive offset in x-direction. The default camera position and the default object position both are specified in the same coordinate system as the screens. The default camera posistion is only used if the tracking system is disconnected. The default object posistion is the position where the scene is placed by default (or whenever a *view all* request comes from the viewer).

At the end of the config file four reference points are specified, namely four points on the horizontal screen. Before the configuration can be actually used, the tracking system has to be calibrated (see section 19.4). This is done by placing the input device at the reference points and clicking an input button. The reference points were chosen so that they can be easily accessed with the hand. In addition, it is important that the points are well inside the operating range of the tracking system. During calibration the raw coordinates of the wand sensor are displayed, so you can check if these values are reasonable. Once the tracking system is calibrated, the config file should be written by clicking the *write config* button of the amiraVR control module. The new config file will contain the same information listed above, but in addition it will also contain some calibration data, e.g., the transformation between raw tracker coordinates and screen coordinates.

19.3.3 Example: A 4-side CAVE configuration

A CAVE (TM) is a more or less fully immersive VR display system with the shape of a cubical box. Typically the size of the box is something like 3 x 3 x 3 meters. Three, four, five, or even six sides of the box are implemented as projection screens. In this way an observer inside the box can completely dive into a virtual world. In order to compute correct perspective views the eye position of the observer need to be tracked. Other non-tracked observers will perceive distorted images, especially at the edges between the individual walls. A schematic view of a 3-side CAVE is shown in Figure 19.4.

The amiraVR config file listed below was designed for a 4-side CAVE. In order to drive such a system four different images need to be generated, one for the front, bottom, left, and right wall repectively. This can be done for example using a two-pipe SGI Onyx system. Each pipe has two channels. Thus it is able to output two different images. We assume that there is one X server running on the machine. The two pipes are configured as two independent X11 screens, denoted `:0.0` and `:0.1`. On each X11 screen there is a double-wide desktop. The left half and the right half of the two desktops are output by the two different channels of each pipe. For such a setting the amiraVR config file looks as

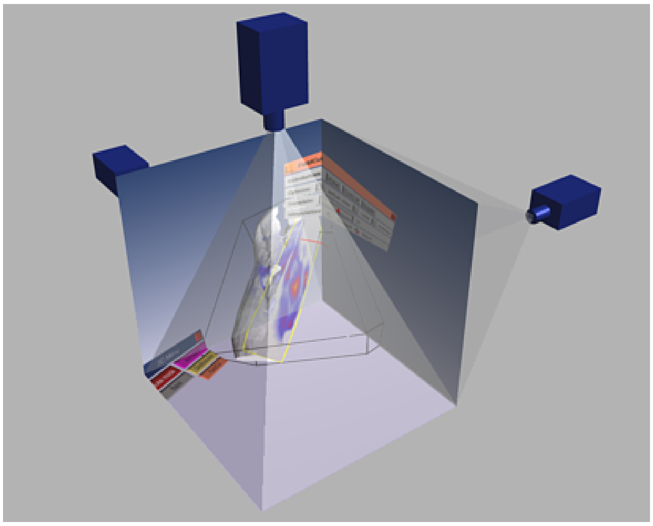


Figure 19.4: Schematic view of a 3-side CAVE system.

follows:

```
#Inventor V2.1 ascii

Separator {
    SoScreen {
        name                "Front"
        display             ":0.0"
        lowerLeft           0 0 0
        lowerRight          300 0 0
        upperRight          300 300 0
        upperLeft           0 300 0
        channelOrigin       0 0
        channelSize         0.5 1
        cameraMode          ACTIVE_STEREO
        threadGroup         0
    }
    SoScreen {
        name                "Bottom"
        display             ":0.0"
        lowerLeft           0 0 300
        lowerRight          300 0 300
        upperRight      300 0 0
        upperLeft       0 0 0
        channelOrigin   0.5 0
        channelSize     0.5 1

        # If the bottom image is rotated try this:
        # lowerLeft       300 0 0
        # lowerRight      0 0 0
        # upperRight      0 0 300
        # upperLeft       300 0 300

        # This modifies the \amira gradient background
        backgroundMode  BG_LOWER
        cameraMode      ACTIVE_STEREO
        threadGroup     0
    }
    SoScreen {
        name            "Left"
        display         ":0.1"
        lowerLeft       0 0 300
        lowerRight      0 0 0
        upperRight      0 300 0
        upperLeft       0 300 300
        channelOrigin   0 0
        channelSize     0.5 1
        cameraMode      ACTIVE_STEREO
        threadGroup     1
    }
    SoScreen {
        name            "Right"
        display         ":0.1"
        lowerLeft       300 0 0
        lowerRight      300 0 300
        upperRight      300 300 300
        upperLeft       300 300 0
        channelOrigin   0.5 0
        channelSize     0.5 1
        cameraMode      ACTIVE_STEREO
        threadGroup     1
    }
    SoTracker {
        server          "4147:4148"
        autoConnect     TRUE
        wandTrackerId   1
        headTrackerId   0
        leftEyeOffset   6 0 0
        rightEyeOffset  13 0 0
        defaultCameraPosition 50 50 100
        defaultObjectPosition 50 50 0
        referencePoints [
            0 150 100, 0 150 100, 100 150 0, 200 150 0, 300 150 100 ]
    }
}
```

19.4 Calibrating the tracking system

This section describes how to calibrate a tracking system for use in amiraVR. Here, calibration essentially means finding the transformation between the raw tracker coordinates and the coordinates in which the geometry of the display system, i.e., the corners of the screens, has been defined. Calibrating the tracking system does not involve changing the trackd config file *trackd.conf* in any way. Instead the calibration data is completely stored in the amiraVR config file. Usually the tracking system need to be calibrated only once. A new calibration is necessary for example if the antenna of an electromagnetic tracking system is moved with respect to the display system, if the screen of a single-wall immersive workbench is tilted, or if a new 3D input device is used where the 3D sensor is oriented in a different way. The only part of the calibration process which one might repeat many a time is picking the wand. Picking the wand means to specify the offset between the actual position of the 3D input device and the visual representation of the wand in amira. Sometimes it is useful not to have any offset, while sometimes an offset provides more pleasent less exhausting working conditions. The wand offset can be quickly adapted even multiple times within a single amiraVR session.

Below there is a complete step-by-step description of the calibration process. Before starting calibration make sure that you have a valid amiraVR config file for your particular display system and that the trackd software and the tracking system itself are setup in the right way. In particular, make sure that the tracker coordinates are reported in a right-handed coordinate system. Many electromagnetic tracking systems cannot distinguish between two opposite hemispheres. In the *trackd.conf* file you have to specify in which hemisphere you want to operate. If the wrong hemisphere is specified a left-handed coordinate system is used, which cannot be correctly calibrated in amira.

1. Make sure that the trackd server is running and that the tracking system is operating. Start amira and choose the appropriate configuration from the main window's *Config* menu.
2. In the amiraVR control module connect to the tracking system if necessary. Activate the *values* toggle in order to display the current 3D coordinates in the upper left corner of the main screen. Make sure that the coordinates are changing if you move the 3D input device and the head sensor (unless head tracking is disabled in the amiraVR config file). Make also sure, that the button

status changes if you press a button of the 3D input device.

Note: Unless you are in calibration mode the coordinates reported by the *values* option are transformed coordinates, i.e., they are in the same coordinate system in which the screens in the config file have been defined.

Hint: If the wand tracker and the head tracker seem to be exchanged modify the fields *wandTrackerId* and *headTrackerId* in the amiraVR config file.

3. Click the *Calibrate* button of the amiraVR control module. You are now in calibration mode. A fixed 2D control grid is displayed on all screens. You are requested to click at the first reference point. In the current version of amiraVR, the control grid is not guaranteed to match the actual reference points. However, you could choose the reference points in the config file to be at the corners of the control grid (one third and two thirds of the screen width in horizontal direction, one half of the screen height in vertical direction). Besides the number of the reference point its coordinates as specified in the config file are displayed in the upper left corner of the main screen.

   Now move the 3D input device at the first reference point and click any button of the device.

   Note: Make sure that the raw tracker coordinates which are displayed in the upper left corner of the main screen are valid at the reference point. Some devices just report zero values if the sensor is outside the operating range of the tracking system. Some other devices report nonsense values, typically with large oscillations. If you don't get stable values at a reference point, choose some other point in the amiraVR config file.

4. Next you are asked to click at second reference point using the 3D input device. Repeat the above procedure until all reference points have been located.

   Note: If not at least three different reference points are specified in the config file, by default the upper left, lower left, and lower right corner of the first screen are used as reference points. You may want to specify other reference points or a larger number of reference points in order to improve accuracy. *The reference points must not lie all on one line*. However, they may well be located in a common plane, e.g., in the plane of the main screen.

5. Next, you need to align the glasses for head tracking. However, if head tracking is disabled in the amiraVR config file, this step is omitted.

   Align the glasses parallel to the x-axis of the screen coordinate system. Usually the x-axis will oriented horizontally with respect to the front screen of the display system. The up-direction of the glasses should be aligned parallel to the y-axis of the screen coordinate system. Usually the y-axis will be oriented vertically with respect to the front screen. Once you have aligned the glasses click any button of the 3D input device.

6. Now camera calibration is done and correct stereoscopic images should be displayed. As a final step you are requested to pick the wand in order to define the wand offset. The wand is being displayed half-way between the current eye position (or the default camera position, if head tracking is disabled) and the default object position. Now move the 3D input device near the origin of the wand (or at any other position) and click any button. The virtual wand remains stick to the 3D input device in exactly that position. You can repeat this last calibration step any

time by clicking on the *pick wand* button of the amiraVR module.

Hint: If you don't get a clear 3D image of the wand make sure left and right eye images are not exchanged. If the eyes are exchanged you get a result which somehow also looks 3D but which isn't comfortable to work with at all. Also make sure that the fields *leftEyeOffset* and *rightEyeOffset* are set correctly in the config file.

7. At this point, the calibration is finished. You may now want to write a new amiraVR config file (or to overwrite the original one) in order to save the calibration data permanently. This can be achieved most easily by pressing the *write config* button of the amiraVR module. Pressing this button causes the amira file browser to be activated. You can accept the name of the previous config file or choose a new one.

   When reloading the new configuration calibrated tracker data will be generated automatically. You can verify the correctness of the calibration process by turning on the *values* toggle and moving the 3D input device to one of the reference points. The reported values should be almost the same as the coordinates of the reference point specified in the config file.

19.5 The amiraVR cluster version

The amiraVR cluster version is an extension allowing amiraVR to be used on a graphics cluster. A graphics cluster consists of multiple computers connected via a network. Each computer controls one or more screens of a multi-wall display system. amiraVR synchronizes the different nodes, and ensures that same scene is rendered simultaneously from different perspectives. The amiraVR cluster version facilitates parallel multi-pipe rendering. It does not speed up ordinary computations by distributing them across multiple nodes of the cluster. Instead, on each node a separate instance of amira is running, with the complete network and all data being duplicated. Changes of amira modules made on the master node will be propagated to the slave nodes and executed synchronously.

Requirements:

- All data files and scripts must have the same absolute file path on all nodes of the cluster. Likewise, amira must be installed at the same location on all nodes. For convenience, it is recommended to run amira from a mounted file system. This ensures that the same scripts and config files will be used on all nodes.
- All nodes of the cluster should be of the same type. If different hardware is used, it is not guaranteed that the cluster runs out-of-sync because of round-off errors. Rendering speed is limited by the slowest node of the cluster.
- All nodes of the graphics cluster must be able to communicate with each other via a TCP/IP connection. A standard 10 Mbit ethernet connection is fast enough, since only synchronization commands are exchanged, rather than images or raw data blocks.
- The graphics cards of the different nodes must be genlocked with a genlock cable. Currently only a small number of graphics cards (such as WildCat or SUN Zulu boards) provide genlocking. Genlocking ensures that the video refresh cycles are synchronized. amiraVR itself only

synchronizes OpenGL buffer swaps.

Limitations:

- The focus of the amiraVR cluster version is on working in a VR environment. Consequently all manipulations performed in VR mode, e.g., via the 3D menu, will be properly synchronized. On the other hand, currently not all manipulations made on the master node via the conventional 2D user interface will be synchronized.
- The general work flow is to first create a network in standard mode on the master node, and then to load this network from file in cluster mode. Currently, you cannot create new modules via an object's popup menu in cluster mode, nor you can interactively change connections via the 2D user interface.
- Some modules such as PlanerLIC or DisplayISL make use of random number generators. Currently, it is not ensured that the resulting numbers are always the same on all nodes. Consequently, different visual results might be obtained on different nodes.
- Any kind of 2D mouse interaction in a 3D viewer window also will not be synchronized in cluster mode. Thus you cannot pick a slice or move an Open Inventor dragger with the 2D mouse. You need to perform such manipulations using a 3D mouse in amiraVR.

Running amiraVR cluster

The following steps are required to run the amiraVR cluster version:

1. Install amiraVR on every node of the graphics cluster under the same absolute path name, or better create a mounted file system with the same absolute path name on every node.
2. Create a standard amiraVR config file describing the geometry of your display system. Then add a field *hostname* in each screen section. This field indicates on which node the screen will be rendered. The config file must be stored on all nodes in the directory `$AMIRA_ROOT/share/config`.
3. Choose one node as the master node. On all other nodes start amira with the command line option `-clusterdaemon`. With this option a little daemon is started, to which the master process can talk to. The daemon automatically starts a slave instance of the real amira if necessary.
4. Start amiraVR on the master node. Then select your cluster configuration from the config menu. Slave instances of amiraVR should now be started on all nodes specifed in the config file.
5. Next you can load any standard amira network script. For example, open the online help browser on the master node and choose one of the standard amiraVR tracking demos. The particular script will be loaded on all slaves as well. Once a script has been loaded all standard amiraVR interaction modes are available in cluster mode too.

Chapter 20

Working with amiraVR

20.1 3D user interaction

amiraVR flat screen configurations, i.e., tiled displays or power walls, can be completely controlled using an ordinary 2D mouse. However, in case of true immersive configurations this isn't feasible anymore. Therefore, several ways of 3D user interaction are provided in amiraVR. The view can be changed using different navigation modes, a 3D menu is provided in order to control modules in 3D, and special objects like slices, selection boxes, or 3D point probes can be directly picked and manipulated using a tracked input device. In addition, a 2D mouse mode is provided allowing the user to control the conventional 2D mouse pointer using the tracked 3D mouse. In the following these features will be described in more detail.

In order to utilize 3D user interaction a tracked 3D input device with at least one button is required. By default amira only interpretes a single button. The most easiest way for making use of additional buttons is to associate Tcl procedures with these buttons. This is described below in a section about Tcl event procedures.

- The 3D menu
- User-defined 3D menu items
- 3D module controls
- Navigation modes
- Tcl event procedures
- The 2D mouse mode

20.1.1 The 3D menu

The amiraVR 3D menu provides buttons allowing the user to bring up a 3D version of the user interface of every object in the object pool. In addition it can contain any number of user-defined buttons. These buttons can be configured most easily using the Tcl interface described in Section 20.2.

On default, the menu also contains a button for activating the 2D mouse mode. This mode lets you control the 2D mouse using a 3D input device. In this way also the standard 2D user interface of amira can be used in a VR environment.

The basic shape of the amiraVR 3D menu is shown in Figure 20.1 on the left hand side. The menu can be manipulated in the following way:

- Initially, after activating or reloading an amiraVR configuration, the 3D menu is hidden. In order to bring up the menu double-click the button of the 3D input device. Alternatively, you may select the 3D menu toggle button of the amiraVR control module.

  By default the menu will be positioned in the upper left corner of the first screen defined in the amiraVR config file. You may change the default position by setting the field *menuPosition* in the tracker section of the amiraVR config file. There is also a field for setting the default orientation of the menu.
- You can move the menu by pointing the wand on the blue header labeled *3D menu*, clicking the main button of the 3D input device, and then moving the device while keeping the button pressed. When the wand is on the blue header a yellow frame is displayed, indicating that this element has focus.

  If you pop up the menu using the 3D menu toggle, the menu position will be reset to its default. In contrast, if you pop up the menu by double-clicking the 3D mouse button, the menu will be shown at its current position.
- In order to hide the menu again, click the red *close button* at the right side of the blue header element. Double-clicking the 3D input device again pops up the menu at its previous position.

Since version 3.1 of amiraVR the 3D menu will also be available as an ordinary 2D sub-menu under the VR menu of the amira main window. The 2D menu has the same structure as the 3D menu, and choosing an item from the 2D menu triggers the same actions as choosing an item from the 3D menu. This feature allows you to prepare and test a 3D menu on an ordinary desktop computer without an attched tracking system.

20.2 User-defi ned 3D menu items

Besides the default buttons any number of user-defined buttons can be added to the amiraVR 3D menu using a Tcl script interface. This is useful for executing certain demos or for invoking other special actions. The script interface essentially consists of a single global Tcl method called *menu* which is provided by amiraVR. Calling the *menu* method with special parameters allows the user to add individual buttons or submenus to the main 3D menu. Each button or submenu has an id. This id can

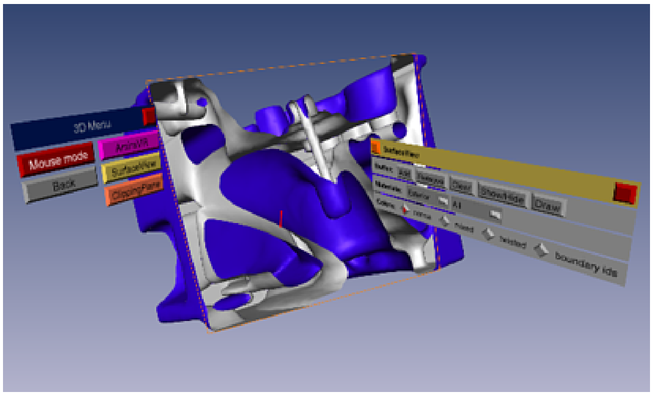


Figure 20.1: amiraVR scene with 3D menu on the left and 3D user interface of the *SurfaceView* module on the right.

be used to modify the particular element afterwards. Whenever a button is pressed a user-defined Tcl procedure is invoked.

An example of how to create a two-level menu hierarchy is shown below. This example is taken from the file *$AMIRA_ROOT/share/demo/tracking/demomenu.hx* contained in the amiraVR demo section. You can execute that script in order to see how the user-defined buttons work.

```
menu reset

menu insertMenu -id 0 -text "Medical"
menu insertMenu -id 1 -text "Flow Dynamics"
menu insertMenu -id 2 -text "Reconstruction"
menu insertMenu -id 3 -text "Multi-Channel"

menu 0 insertItem -id 0 -text "CT Slices" -proc "Menu0"
menu 0 insertItem -id 1 -text "Isosurface" -proc "Menu0"
menu 0 insertItem -id 2 -text "Surface model" -proc "Menu0"
menu 0 insertItem -id 4 -text "Oblique slice" -proc "Menu0"
menu 0 insertItem -id 5 -text "Pseudo-color" -proc "Menu0"

menu 1 insertItem -id 0 -text "Wing" -proc "Menu1"
menu 1 insertItem -id 1 -text "Turbine ISL" -proc "Menu1"
menu 1 insertItem -id 2 -text "Turbine LIC" -proc "Menu1"

menu 2 insertItem -id 0 -text "Slice & Isosurface" -proc "Menu2"
menu 2 insertItem -id 1 -text "Volume Rendering" -proc "Menu2"
menu 2 insertItem -id 2 -text "Simplified Surface" -proc "Menu2"

menu 3 insertItem -id 0 -text "Projection view" -proc "Menu3"
menu 3 insertItem -id 1 -text "Slicing" -proc "Menu3"
menu 3 insertItem -id 2 -text "Isosurface" -proc "Menu3"
menu 3 insertItem -id 3 -text "Volume Rendering" -proc "Menu3"

proc Menu0 { id } {
    global AMIRA_ROOT
    switch $id \
        0 { load $AMIRA_ROOT/share/demo/medical/ctstack.hx } \
        1 { load $AMIRA_ROOT/share/demo/medical/isosurface.hx } \
        2 { load $AMIRA_ROOT/share/demo/medical/surf.hx } \
        3 { load $AMIRA_ROOT/share/demo/medical/tetra.hx } \
        4 { load $AMIRA_ROOT/share/demo/medical/gridcut.hx } \
        5 { load $AMIRA_ROOT/share/demo/medical/pseudocolor.hx } \
        6 { load $AMIRA_ROOT/share/demo/medical/splats.hx }
}

proc Menu1 { id } {
    global AMIRA_ROOT
    switch $id \
        0 { load $AMIRA_ROOT/share/demo/cfd/wing.hx } \
        1 { load $AMIRA_ROOT/share/demo/cfd/turbine-isl.hx } \
        2 { load $AMIRA_ROOT/share/demo/cfd/turbine-lic.hx }
}

proc Menu2 { id } {
    global AMIRA_ROOT
    switch $id \
        0 { load $AMIRA_ROOT/share/demo/recon/recon01.hx } \
        1 { load $AMIRA_ROOT/share/demo/recon/recon05.hx } \
        2 { load $AMIRA_ROOT/share/demo/recon/recon04.hx }
}

proc Menu3 { id } {
    global AMIRA_ROOT
    switch $id \
        0 { load $AMIRA_ROOT/share/demo/multichannel/projectionview.hx } \
        1 { load $AMIRA_ROOT/share/demo/multichannel/slicing.hx } \
        2 { load $AMIRA_ROOT/share/demo/multichannel/isosurfaces.hx }
        3 { load $AMIRA_ROOT/share/demo/multichannel/voltex.hx }
}
```

The example above give a general idea of how to create a user-defined button hierarchy. The *menu* command provides some additional parameters. The general form of the *menu* command is as follows:

```
menu [submenu-id [...]] cmd [options]
```

Here `submenu-id` is the id of any submenu. The root level has no id at all. The first level has one id, the second level has two ids, and so on.

Note: In contrast to previous versions of amiraVR now all user-defined buttons are inserted at the main level of the menu. On default at this level there are already three buttons defined:

- a button for activating the 2D mouse mode (id 1000)
- a button for getting a list of all modules (id 1001)
- a button for getting a list of all data objects (id 1002)

You can remove these default buttons using the command `menu clear`. In order to clear the menu and then reset the default buttons use `menu reset`. The complete list of commands is this:

menu reset
: Removes all buttons and sub-menus from the menu and restores the default layout with entries for mouse mode, modules, and data objects. This command can only be applied at the main level.

menu [...] clear
: Removes all buttons and sub-menus from the specified menu. If applied at the main level also the default buttons (mouse mode, modules, and data) will be removed.

menu [...] count
: Returns the number of entries (action buttons and sub-menu buttons) in the specified menu.

menu [...] idAt *index* Returns the id of the item at position *index*.

menu [...] insertItem [options]
: This command creates a new button and inserts it into the specified menu. The following options are available:

  -id *id*
  : Specifies the id of the new button.

  -index *index*
  : Specifies the position of the new button. If *index* is -1 (which is the default) the new button will be appended at the bottom of the menu.

  -text *text*
  : Specifies the text to be displayed by the new button.

  -fg *”r g b”*
  : Specifies the text or foreground color of the new button.

  -bg *”r g b”*
  : Specifies the background color of the new button.

-proc *proc*

The name of the Tcl procedure to be called when the button is pressed. Either an own procedure without arguments can be used for a button. Alternatively, a procedure with one argument can be used for all buttons in a menu. The argument then is set to the id of the pressed button (see example above).

menu [...] insertMenu [options]

This command creates a new sub-menu button and inserts it into the specified menu. The id of the new button can be used as a sub-menu id for the *menu* command itself. The same options as for *insertItem* can be used. In addition the following option is available:

-type *classtype*

If this option is specified a special sub-menu will be inserted which lists all objects of type *classtype* in the amira object pool. For example, in order to get a list of all modules (like in the default menu) you should use insertMenu with *-type HxModule*.

menu [...] changeItem id -text *text*

Changes the text of an existing button.

menu [...] removeItem id

Removes an action button or sub-menu button from the menu.

menu [...] connectItem id *proc*

Connects a button to a Tcl procedure. The procedure will be called when the button is pressed.

menu [...] disconnectItem id *proc*

Disconnects a button from a Tcl procedure.

20.2.1 3D module controls

The 3D menu provides a button called *Modules*. Clicking on this button brings up a list of all visible modules which currently exist in the amira object pool. Clicking on a module button brings up a 3D version of the modules user interface. This 3D user interface looks very similar to the 2D user interface of the module which is normally shown in the work area part of the amira main window. An example of the 3D user interface of the *SurfaceView* module is shown in Figure 20.1.

The 3D user interface of every module can be moved independently from each other like the main 3D menu itself. Again, this is done by picking the header bar of the module's GUI. The GUI can be removed by clicking on the red *close button* at the right side of the header bar. On the left side an orange *viewer toggle* is displayed. As in the 2D interface a display module can be switched on or off by toggling this button.

In 3D the ports of a module can be manipulated almost in the same way as in 2D. The only remarkable difference is that text input is not possible in 3D. Text fields with numerical values like the range of a colormap port or the data window of an OrthoSlice module still can be changed using a virtual slider. This is done by clicking into the text field with the wand, and then moving the wand upwards or downwards while keeping the button pressed. As a general rule, every active element shows a yellow frame when the wand is pointing on it, i.e., when it has input focus.

20.2.2 Navigation modes

amiraVR supports two different navigation modes. The default mode is *scene manipulation*, i.e., the whole scene can be rotated and translated using the 3D input device. This is done by clicking with the wand into some empty or insensitive region in space, and then moving the wand while keeping the button pressed. In this mode the whole scene remains stick relative to the input device.

The second mode is *fly mode*. This mode can be activated by selecting the second entry from the *Mode* port of the amiraVR control module (either in the 2D or in the 3D user interface). In fly mode you can click the 3D input device at some insensitive point, and then move it in any direction. The vector from the point where the wand was clicked to the current point defines a velocity vector which is used to modify the position of the camera. The longer the vector, i.e., the more the wand is shifted, the faster the camera is moving. In the same way the orientation of the camera is modified by rotating the 3D wand. An incremental rotational change is computed by comparing the current orientation which the orientation at the point where the wand was clicked initially.

20.2.3 Tcl event procedures

On default, amiraVR treats all buttons of a 3D mouse in the same way. This allows you to operate the program even with a one-button mouse. In order to make better use of a 3D input device with multiple buttons, you can define special Tcl procedures which are called when a button is pressed or released. If these procedures return the value 1, this indicates that the event has been handled by the Tcl procedure and that it should not be further processed by amira. In particular, it then will not be send to the Open Inventor scene graph. The following procedures can be defined:

- `vrButton0Press, vrButton1Press, ...`
  These procedures are called when the button with the specified index is pressed. By redefining `vrButton0Press` you can overwrite the default interaction routine.
- `vrButton0Release, vrButton1Release, ...`
  These procedures are called when the button with the specified index is released.
- `vrButton0DblClick, vrButton1DblClick, ...`
  These procedures are called when the button with the specified index is double-clicked. Note, that for the first click of a double-click event an ordinary button press event is generated.

When a 3D mouse button event occurs and no appropriate Tcl event procedure is defined or if this procedure returns 0, the event will be passed to the current amiraVR event handler. The default event handler checks if the 3D menu or some other 3D widget has been clicked. If not, it sends the event to the Open Inventor scene graph. Besides this default event handler there are also other event handlers, namely handlers for managing the different navigation modes (*examine* and *fly*).

If a 3D mouse button event was not handled by the current event handler, finally the following Tcl procedures will be called, provided they are defined:

- `vrButton0PressFallback, vrButton1PressFallback, ...`
- `vrButton0ReleaseFallback, vrButton1ReleaseFallback, ...`
- `vrButton0DblClickFallback, vrButton1DblClickFallback, ...`

Assuming, that the default event handler is active (the one which checks the 3d menu and the Open Inventor scene graph), you can trigger certain actions when the user clicks into empty space. On default, when this happens one of the navigation mode handlers is activated. If you want to disable this feature you can define a dummy `vrButton0PressFallback` procedure which always returns 1.

20.2.4 The 2D mouse mode

The 2D mouse mode is useful for accessing GUI elements which have no direct analogon in 3D. In this mode the position of the ordinary 2D mouse pointer is controlled using the 3D input device.

- In order to activate 2D mouse mode press the red *mouse mode* button of the 3D menu. Once mouse mode is activated the amira main window is popped up automatically.
- You can control the 2D mouse by translating or rotating the 3D input device. Usually, rotating it is more convenient since less movement is required. The position of the 2D mouse is computed by determining the intersection of the virtual wand vector with the screen.
- In order to exit 2D mouse mode, double-click the 3D input device over some insensitive region of the 2D user interface. When leaving 2D mouse mode the main graphics window is raised automatically.

  If the 2D mouse stays in the middle of the screen, of course 3D impression is disturbed. Therefore, usually it is a good idea to move the 2D mouse in a corner before returning to 3D mode.

The two Tcl procedures *AmiraVR_StartMouseMode* and *AmiraVR_StopMouseMode* are called when entering and exiting 2D mouse mode. You can use these methods for example to pop up the amira help browser with a page from which additional demos can be launched. For this, the following definition could be included for example in the file *$AMIRA_ROOT/share/resources/Amira.init*:

```
proc AmiraVR_StartMouseMode { } {
    global AMIRA_ROOT
    load $AMIRA_ROOT/share/usersguide/tracking.html
}
```

In order to activate or deactivate the 2D mouse mode from Tcl, the amiraVR control module provides the command `AmiraVR enableMouseMode <value>`, where `<value>` can be either 0 or 1. In order to check if 2D mouse mode is currently active, you can use `AmiraVR isMouseModeEnabled`.

20.3 Writing amiraVR custom modules

amira can be easily extended by writing new I/O-routines, data types, modules, and other components. Details about this are described in the amira Programmer's Guide.

In order to write custom modules which provide specific interaction features in a VR environment, Open Inventor nodes interpreting events generated by the 3D input device have to be inserted into the scene graph. In particular, the 3D input device generates events of type *SoTrackerEvent* and *SoControllerButtonEvent*. These two event classes have been introduced in Open Inventor 3.0. For more information about these classes please refer to the Open Inventor documentation. amira itself provides an additional class *Hx3DWandBase* which provides some additional information about the virtual 3D wand. Among others, this class also allows to user to highlight the wand in order to provide some visual feedback during interaction.

Below we present the source code of a sample amiraVR module which just displays a number of cubes. The cubes then can be picked and transformed using the 3D wand. The source code of the module is contained in the amiraDev demo package. The particular files are called *MyVRDemo.h* and *MyVRDemo.cpp*. The module can also be directly created from the popup menu of the amiraVR control module.

Here is the listing of the header file:

```
///////////////////////////////////////////////////////////////////////
//
// Illustrates 3D interaction in a VR environment
//
///////////////////////////////////////////////////////////////////////
#ifndef MY_VR_DEMO_H
#define MY_VR_DEMO_H

#include <Inventor/nodes/SoSeparator.h>

#include <McHandle.h>
#include <Amira/HxModule.h>
#include <Amira/HxPortButtonList.h>
#include <mypackage/mypackageAPI.h>

class MYPACKAGE_API MyVRDemo : public HxModule
{
    HX_HEADER(MyVRDemo);

  public:
    MyVRDemo();
    virtual void compute();
    HxPortButtonList portAction;
  protected:
    ˜MyVRDemo();

    McHandle<SoSeparator> scene;
    McHandle<SoSeparator> activeCube;

    bool isMoving;
    SbVec3f refPos;
    SbMatrix refMatrix;
    SbRotation refRotInverse;

    void createScene(SoSeparator* scene);
    SoSeparator* checkCube(const SbVec3f& pos);
    void trackerEvent(SoEventCallback* node);
    void controllerEvent(SoEventCallback* node);

    static void trackerEventCB(void* userData, SoEventCallback* node);
    static void controllerEventCB(void* userData, SoEventCallback* node);
};

#endif
```

Here is the listing of the source file:

```
///////////////////////////////////////////////////////////////////////
//
// Illustrates 3D interaction in a VR environment
//
///////////////////////////////////////////////////////////////////////
#include <stdlib.h>

#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoEventCallback.h>
#include <Inventor/nodes/SoMatrixTransform.h>
#include <Inventor/events/SoTrackerEvent.h>
#include <Inventor/events/SoControllerButtonEvent.h>

#include <Amira/Hx3DWandBase.h>
#include <mypackage/MyVRDemo.h>

HX_INIT_CLASS(MyVRDemo,HxModule)

MyVRDemo::MyVRDemo() :
    HxModule(HxObject::getClassTypeId()),
    portAction(this,"action",1)
```

```
{
    isMoving = 0;
    portAction.setLabel(0,"Reset");
    scene = new SoSeparator;
    createScene(scene);
    showGeom(scene);
}

MyVRDemo::~MyVRDemo()
{
    hideGeom(scene);
}

void MyVRDemo::compute()
{
    if (portAction.isNew() && portAction.getIndex()==0)
        createScene(scene);
}

void MyVRDemo::createScene(SoSeparator* scene)
{
    scene->removeAllChildren();

    SoEventCallback* cb = new SoEventCallback;
    cb->addEventCallback(SoTrackerEvent::getClassTypeId(),
        trackerEventCB, this);
    cb->addEventCallback(SoControllerButtonEvent::getClassTypeId(),
        controllerEventCB, this);
    scene->addChild(cb);

    for (int j=0; j<4; j++) {
        for (int i=0; i<4; i++) {
            SoSeparator* child = new SoSeparator;

            SoMatrixTransform* xform = new SoMatrixTransform;
            SbMatrix M;
            M.setTranslate(SbVec3f(i*3.f,j*3.f,0));
            xform->matrix.setValue(M);

            SoMaterial* mat = new SoMaterial;
            float hue = (rand()%1000)/1000.;
            mat->diffuseColor.setHSVValue(hue,1.f,.8f);

            SoCube* cube = new SoCube;

            child->addChild(xform);
            child->addChild(mat);
            child->addChild(cube);

            scene->addChild(child);
        }
    }
}

SoSeparator* MyVRDemo::checkCube(const SbVec3f& p)
{
    for (int iChild=1; iChild<scene->getNumChildren(); iChild++) {
        SoSeparator* child = (SoSeparator*) scene->getChild(iChild);
        SoMatrixTransform* xform = (SoMatrixTransform*) child->getChild(0);
        const SbMatrix& M = xform->matrix.getValue();

        SbVec3f q;
        M.inverse().multVecMatrix(p,q);
        if (fabs(q[0])<1 && fabs(q[1])<1 && fabs(q[2])<1) {
            if (activeCube.ptr()!=child) {
                if (activeCube) {
                    SoMaterial* mat = (SoMaterial*) activeCube-
>getChild(1);
                    mat->emissiveColor = SbColor(0.f,0.f,0.f);
                }
                SoMaterial* mat = (SoMaterial*) child->getChild(1);
                mat->emissiveColor = mat->diffuseColor[0];
                activeCube = child;
            }
            return activeCube;
        }
    }

    if (activeCube) {
        SoMaterial* mat = (SoMaterial*) activeCube->getChild(1);
        mat->emissiveColor = SbColor(0.f,0.f,0.f);
        activeCube = 0;
    }

    return 0;
}

void MyVRDemo::trackerEventCB(void* userData, SoEventCallback* node)
{
    MyVRDemo* me = (MyVRDemo*) userData;
    me->trackerEvent(node);
}
```

```
void MyVRDemo::trackerEvent(SoEventCallback* node)
{
    SoTrackerEvent* e = (SoTrackerEvent*) node->getEvent();
    Hx3DWandBase* wand = GET_WAND(e);

    if (activeCube && isMoving) {
        if (!wand->getButton(0)) {
            node->setHandled();
            isMoving = 0;
            return;
        }

        SbMatrix T1; T1.setTranslate(-refPos);
        SbMatrix R; R.setRotate(refRotInverse*wand->orientation());
        SbMatrix T2; T2.setTranslate(wand->origin());

        SoMatrixTransform* xform = (SoMatrixTransform*) activeCube-
>getChild(0);
        xform->matrix = SbMatrix(refMatrix*T1*R*T2);

        wand->setHighlight(true);
        node->setHandled();
        return;
    }

    if (checkCube(wand->top()))
        node->setHandled();
}

void MyVRDemo::controllerEventCB(void* userData, SoEventCallback* node)
{
    MyVRDemo* me = (MyVRDemo*) userData;
    me->controllerEvent(node);
}

void MyVRDemo::controllerEvent(SoEventCallback* node)
{
    if (activeCube) {
        SoTrackerEvent* e = (SoTrackerEvent*) node->getEvent();
        Hx3DWandBase* wand = GET_WAND(e);

        if (wand->wasButtonPressed(0)) {
            SoMatrixTransform* xform = (SoMatrixTransform*) activeCube-
>getChild(0);
            refRotInverse = wand->orientation().inverse();
            refPos = wand->origin();
            refMatrix = xform->matrix.getValue();

            wand->setHighlight(true);
            isMoving = 1;
        }

        if (wand->wasButtonReleased(0))
            isMoving = 0;

        node->setHandled();
    }
}
```

Chapter 21

amiraVR Reference

21.1 Config file reference

An amiraVR config file is an Open Inventor file containing one or more SoScreen nodes (one for each screen of the VR system), an optional SoTracker node (containing information about the tracking system), as well as optional Open Inventor nodes (visual representation of the room).

In principal two different configurations are distinguished. A "flat screen" configuration defines a virtual big 2D screen, while a true "immersive" configuration defines multiple non-planar screens. For a flat screen configuration the fields tileOrigin and tileSize (see below) are used, while for an immersive configuration the fields lowerLeft, lowerRight, upperRight, and upperLeft are used. For a tiled configuration ordinary 2D mouse interaction works, while an immersive configuration usually requires a tracking system.

An SoScreen node contains the following fields:

SoSFString name "Vertical screen"
The name of the screen (not used internally)

SoSFVec3f lowerLeft 0 0 0
The coordinates of the lower left corner of the screen. Any right-handed coordinate system can be used. It does not matter in which units the screen corners are defined, because the transformation between the screen coordinates and the tracker coordinates is computed automatically when calibrating the tracking system.

SoSFVec3f lowerRight 180 0 0
The coordinates of the lower right corner of the screen.

SoSFVec3f upperRight 180 110 0
The coordinates of the upper right corner of the screen.

SoSFVec3f upperLeft 0 110 0
The coordinates of the upper left corner of the screen.

SoSFString display ":0.1"

Specifies on which X display the screen should be rendered. On a multi-pipe machine such as an SGI Onyx either different X servers may run on the different pipes, or there may be one X server with multiple screens. Depending on the actual configuration either ":1.0" or ":0.1" should be specifed in order to enable multi-pipe rendering for the second screen. If this field is omitted the window will be opened on the same display as the amira user interface (determined by the value of the DISPLAY environment variable or by the -display command line option).

SoSFString hostname

This field is required if amiraVR shall be run in cluster mode. It specifies on which node of a graphics cluster the particular screen should be rendered. The hostname can either be specified as a name or as a numeric IP address. Special requirements and limitations of the amiraVR cluster version are described in a separate section.

SoSFVec2f channelOrigin 0.0 0.0

Specifies the position of the upper left corner of the graphics window in relative coordinates. (0,0) is the upper left corner of the desktop, (1,1) is the lower right corner of the desktop.

SoSFVec2f channelSize 0.5 1.0

Specifies the size of graphics window in relative coordinates. The size of the whole desktop is (1,1). In order to create a window covering the left half of the desktop, channelOrigina should be (0,0) and channelSize should be (0.5,1).

SoSFVec2f tileOrigin 0.0 0.0

This field is used in case of a 2D tiled display instead of the fields *lowerLeft*,*lowerRight*, *upperRight*, and *upperLeft*. It specifies the origin of a rectangular part of a virtual big screen which should be rendered in the graphics window. The origin can be any point between (0,0) and (1,1). Here (0,0) denotes the LOWER left corner of the virtual big screen, and (1,1) denotes the UPPER right corner.

SoSFVec2f tileSize 1.0 1.0

This field is used in case of a 2D tiled display instead of the fields *lowerLeft*,*lowerRight*, *upperRight*, and *upperLeft*. It specifies the size of a rectangular part of a virtual big screen which should be displayed in the graphics window. The size can be any value between (0,0) and (1,1).

SoSFEnum cameraMode [MONOSCOPIC — LEFT_VIEW — RIGHT_VIEW — ACTIVE_STEREO]

This field specifies the camera mode used for the screen. The default is MONOSCOPIC. The values LEFT_VIEW and RIGHT_VIEW are used for passive stereo applications. If ACTIVE_STEREO is specified, the window is opened in active stereo mode on default.

SoSFEnum backgroundMode [BG_AS_IS — BG_LOWER — BG_UPPER — BG_REVERSE]

This field affects the way how the default amira gradient background is rendered on the screen. If BG_LOWER or BG_UPPER is specified a uniform background with either the lower or upper color of the standard gradient background is used. This is useful for the bottom or ceiling of a CAVE.

SoSFInt32 threadGroup

Screens with different thread groups are rendered in parallel using multiple threads, provided amira has been started with the -mt command line option or the environment variable AMIRA_MULTITHREAD has been set. If the same thread group is used for two different screens, these screens will always be rendered one after the other. Usually, screens being rendered on the same pipe (same display but different channelOrigina or channelSize) should be assigned the same thread group.

SoMFFloat softEdgeOverlap

This field is used for soft-edge blending. It should contain exactly four floating-point values, indicating the size of the soft-edge region on the left, right, bottom, and top border of the screen relative to the total width or height of the screen. For example, in order to specify 25softEdgeOverlap should be [0.0, 0.25, 0.0, 0.0].

SoMFFloat softEdgeGamma
: This fi eld is used for soft-edge gamma correction. It should contain exactly four fbating-point values, indicating the gamma value of the soft-edge region at the left, right, bottom, and top border of the screen. A gamma value of 1 means, that image intensity is linearly faded out to black. For most projectors gamma values bigger than 1 are required to achieve good results. The default gamma value for all borders is 1.2.

An SoTracker node contains information about the tracking system. Usually it is not necessary to defi ne an So-Tracker node for a ”tiled” confi guration, but only for true VR confi gurations with make use of the fi elds lowerLeft, lowerRight, upperRight, and upperLeft of SoScreen (see above).

SoSFString server 4127:4126
: This fi eld specifi es a string used to initialize the connection to the tracking system. In order to connect to a running VRCO trackd server, the string should contain the shared memory key of the trackd controller data, and the shared memory key of the trackd tracker data separated by a colon (”¡controllerKey¿:¡trackerKey¿”). The shared memory keys are defi ned in the trackd.conf fi le.

SoSFBool autoConnect
: If this fi eld is set to TRUE the connection to the tracking system will be established automatically when the confi guration is loaded. The default value is FALSE.

SoSFVec3f defaultCameraPosition 90 50 150
: Defi nes the camera position to be used as long as there is no connection to the tracking system. This position is also used if the head tracker id (below) is -1.

SoSFVec3f defaultObjectPosition 90 50 0
: Defi nes the object position to be used as long as there is no connection to the tracking system. If there is a ”view all” request in amira (e.g., because the space bar of the keyboard was hit), the scene is centered at the default object position. In a CAVE it makes sense to choose the default object position in the center of the front wall.

SoMFVec3f calibration
: The fi eld is used to store calibration data, i.e., data allowing the system to transform raw tracker data into the coordinate system in which the screens have been defi ned. Usually there is no need to set this fi eld manually. Instead, load a confi guration fi le without calibration data, perform calibration in amiraVR, and then export a new confi g fi le.

SoSFRotation rotGlasses
: This fi eld is used for calibrating the orientation of the tracked stereo glasses. Usually there is no need to set this fi eld manually.

SoSFRotation rotWand
: This fi eld is used for calibrating the orientation of the wand tracker. Usually there is no need to set this fi eld manually.

SoSFVec3f leftEyeOffset
: The center of the left eye with respect to the head tracker. For calibration, the glasses must be align parallel to the front screen. In this position the horizontal axis is the x-axis, the vertical axis is the y-axis, and the axis pointing towards the observer is the z-axis. The eye offsets are defi ned with respect to this coordinate

system. The difference between the left eye's x-offset and the right eye's x-offset should be around 6.5 cm (average eye separation).

SoSFVec3f rightEyeOffset
The center of the right eye with respect to the head tracker. Same remarks as for leftEyeOffset apply.

SoMFVec3f referencePoints
Reference points used for calibrating the tracking system. If no reference points are specified, the user is asked for clicking at the upper left, lower left, and lower right corner of the first screen defined in the config file. Note, that at least three different reference points are required and that these points must not lie on a common line. However, the reference points may well lie in a common plane, e.g., in the screen plane itself.

SoSFInt32 wandTrackerId
Specifies the id of the trackd sensor which should be used to control the wand. On default the wand sensor is assumed to have the id 0.

SoSFInt32 headTrackerId
Specifies the if of the trackd sensor which should be used to control the camera (head tracking). On default the head tracker is assumed to have the id 1. If -1 is specified for a head tracker id the default camera position is used instead of actually querying a head sensor.

SoSFInt32 headTrackingEnabled
This field determines if head tracking should be enabled or disabled by default if the configuration file is loaded. The default is TRUE. If the value is set to FALSE head tracking is initially disabled, but can be activated later on by pressing the head tracking toggle of the amiraVR control module. However, headTrackerId must not be set to -1 in this case.

SoSFString wandFile
This is the name of an Open Inventor file which is used as the visual representation of the wand. The origin of the wand should be at (0,0,0). The wand itself should point into the negative z-direction. The wand is scaled so that the length 1 corresponds to 0.16 times the width of the first screen in the config file. The hot spot of the wand should be indicated by an SoInfo node containing a string *"x y z"*. Usually the hot spot will be something like *"0 0 -1"*. The filename can be an absolute or relative path. If it is relative the file is assumed to be in the same directory as the config file itself (namely in *share/config*).

SoSFString highlightWandFile
This is the name of an Open Inventor file which is used as the visual representation of the wand in highlight state. The same remarks as for *wandFile* apply to this field too.

SoSFFloat wandScale
This field specifies a scaling factor applied to the wand geometry. On default, a value of 0.16 times the width of the first screen is assumed. In order to change the length of the wand you can either specify a custom wand file with a hot spot different from *"0 0 -1"*, or you can change the wand scale value.

SoSFVec3f menuPosition
Specifies the default position of the upper left corner of the 3D menu. On default it is placed in the upper left corner of the first screen in the config file. The values must be defined in the same coordinate system as the corners of the screens.

SoSFRotation menuOrientation
Specifies the orientation of the 3d menu. On default the horizontal axis of the menu, i.e., the text direction, is aligned to the x-axis of the VR coordinate system, while the vertical direction is aligned to the y-axis.

This fi eld allows you to change the orientation by appling a rotation the default orientation. The rotation is specifi ed by four numbers. The fi rst three numbers defi ne the axis of rotation, and the fourth numbers defi nes the rotation angle in radians.

SoSFFloat menuSize
This fi eld specifi es the default width of the 3D menu in the same coordinates in which the corners of the screens were defi ned in. You can adjust this value to make the menu smaller or bigger.

SoSFString onLoad
This fi eld defi nes a Tcl command which is executed after the confi guration has been loaded. You can put in additional initialization code here. For example, you may always want to load a particular script *MenuInit.hx* for initializing the 3d menu when a certain confi guration is loaded. You can do this by defi ning *onLoad 'load $AMIRA ROOT/MenuInit.hx"*.

SoSFString onUnload
This fi eld defi nes a Tcl command which is executed before a confi guration is unloaded.

SoSFString onConnect
This fi eld defi nes a Tcl command which is executed after a connection to the tracking system has been established.

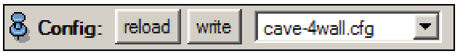
SoSFString onDisconnect
This fi eld defi nes a Tcl command which is executed after the connection to the tracking system was disconnected.

21.2 amiraVR control module

The amiraVR module is the central control module of the amiraVR Virtual Reality Extension. It allows you to activate different screen confi gurations specifi ed in confi g fi les located in `$AMIRA ROOT/share/config` or `$AMIRA_LOCAL/share/config`. For details about supported features such as multi-pipe or multi-thread rendering, soft-edge blending, passive stereo support, or head-tracking please refer to the main amiraVR documentation. In the following we assume that you are already familiar with the basic concepts of amiraVR.

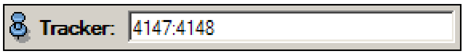
Ports

Confi g



This port provides a list of all confi g fi les found in `$AMIRA ROOT/share/config` and `$AMIRA_LOCAL/share/config`. This is the same as in the *Confi g* menu of the amira main window. Note, that valid confi g fi les are only searched once. It is not possible to add a confi g fi le when amira is running. However, existing confi g fi les may be modifi ed and reloaded using the *reload* button. The *write* button allows you store a modifi ed confi guration into a fi le (for example after the tracking system has been calibrated). If this button is pressed the fi le dialog pops up.

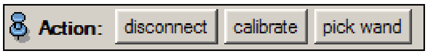
Tracker



Specifies how the connection to the tracking system should be established. In order to connect to a running trackd server the value of this port should be `<id of controller reader>:<id of tracker reader>`, where the two ids are the shared memory ids of the trackd controller reader and the trackd tracker reader as specified in the *trackd.conf* file.

In order to activate a simple tracker emulator you can type in *test* into the text field.

Action

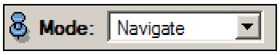


The *connect* button allows you to connect to the tracking system. Once you are connected to the tracking system the button label changes to *disconnect*. Pressing the button then disconnects from the tracking system.

The *calibrate* button allows you to calibrate the tracking system, i.e., to find the transformation between raw tracker coordinates and the screen coordinates used in the config file. For more information about the calibration process please refer to Section 19.4.

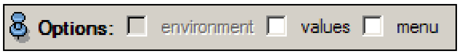
The *pick wand* button allows you to change the offset between the 3D input device and the visual representation of the virtual wand. If you press the button the wand is displayed half-way between the current eye position and the default object position. You can then pick the virtual wand in the way which is most convenient for you.

Mode



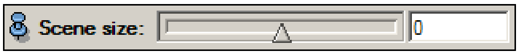
Option menu allowing you to change between ordinary interaction mode (translate and rotate the whole scene using the 3D wand), fly mode (navigate through the scene like an airplane), or 2D mouse mode (control the 2D mouse using the 3D wand). In order to exit 2D mouse mode double-click the 3D wand while the 2D mouse in over some insensitive GUI element.

Options



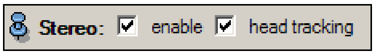
If the *values* toggle is activated the current coordinates of the wand sensor and of the head sensor are displayed in the upper left corner of the screen. The coordinates are transformed into the screen coordinate system which was used in the config file. Only during calibration raw tracker coordinates are displayed.

Adjust size



This slider allows you to change the overall scaling of the scene.

Stereo



This toggle allows you to enable or disable stereo mode. Depending on the camera mode defined in the config file either active stereo or passive stereo will be used. This option is the only one which is provided for flat screen configurations (compare Section 19.2).

The second button labeled allows to temporarily enable or disable *head tracking*. If head tracking is disabled the default camera position specified in the config file will be used instead of the position reported by the head sensor. If no head sensor has been configured or if a flat screen configuration is being used this button is disbaled.

Commands

`setHeadTrackerId <id>`
Changes the head tracker id on-the-fly. Usually the id is defined in the config file. A value of -1 disables the head tracker.

`setWandTrackerId <id>`
Changes the wand tracker id on-the-fly. Usually the id is defined in the config file. A value of -1 disables the wand tracker.

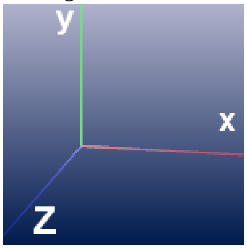
`setSoftEdge <overlap> <gamma>`
Initializes a dual-screen soft-edge configuration with the given overlap percentage `<overlap>`. This is a value between 0 (no overlap) and 1 (full overlap). `<gamma>` defines the gamma value for the soft-edge blending.

`setPanorama <overlap1> [<overlap2> ...]`
Initializes a multi-screen panorama configuration with the given overlap percentages. These are values between 0 (no overlap) and 100 (full overlap). The current configuration must be a flat-screen configuration.

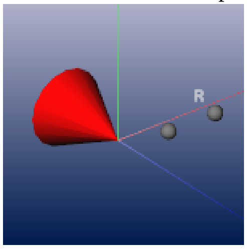
21.3 amiraVR show config module

This module displays a model of the currently loaded AmiraVR configuration. It's intended to give the visual feedback needed during the configuration process and for easier error detection. It shows the projection planes, camera-, sensor- and scene-positions as well as the button state. Therefor the fourth viewer is used.

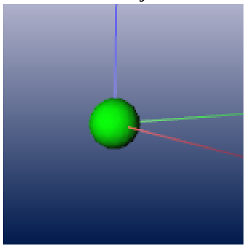
Objects associated with an own local coordinate system are drawed with an axis cross consisting of three colored line segments. X axis is red, y is green and z is drawn with a blue line.



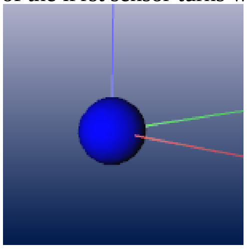
The camera postition is marked with a red cone and two small grey spheres, one for each eye. The eye spheres are translated according to the configured eye offset value. One eye sphere is marked with the letter *R* which means that this is the sphere for the right eye.



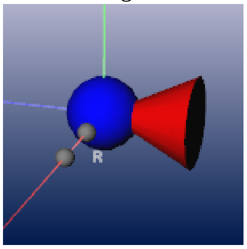
The default object- or scene position is marked with a green sphere.



Blue spheres represent sensors (3D-tracker). One for every sensor. If one of the buttons is pressed the blue sphere of the first sensor turns white until all buttons are released.

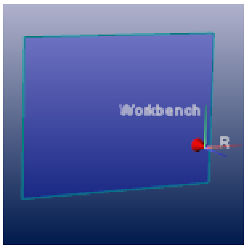


If the camera is bound to one of the 3D-sensors, which means that head tracking is turned on, the red cone representing the camera is glued to the blue sensor sphere and follows its movement. At the moment the head-tracking feature is turned off, the camera model snaps back to the default camera position.

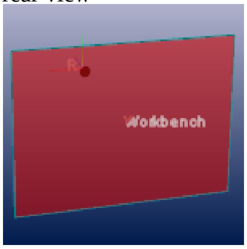


Finally the projection planes are drawn at the positions and dimensions specified in the configuration file. The following two images are showing an environment with a single projection plane. The front side looks blue and a projection plane observed from its back side appears red. Considering the color one can check if the edge points were specified in the right order.

front view



rear view



The user defi ned name of the plane is drawn in it's center. Since projection planes are given as a set of four edge points it is possible to produce a non-planar polygon. This misconfi guration leads to a clearly visible edge crossing the plane.

Connections

Data `[required]`

Connect the amiraVR control module.

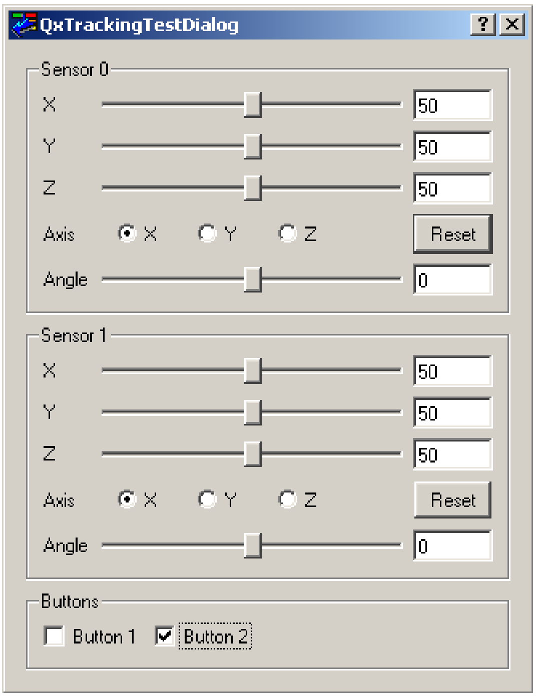
Ports

21.4 Tracker Emulator

The *Tracker Emulator* provides emulation of a 3D positioning device with two spacial sensors and two buttons. To activate it, type the word *test* into the tracker port of the amiraVR control module and press the *connect* button. After that the user interface dialog of the tracker emulator appears.

As mentioned before, the emulator emulates two spacial sensors designated as *sensor0* and *sensor1*. For each of these sensors the position is changeable by three sliders, one for each direction. Alternative and to reach positions of greater distances as the sliders allow one can type the values directly to the entry fi elds to the right of the slider.

The sensors rotation can be changed by selecting one of the main axes as rotation axis and a following adjustment of the rotation angle around this axis using the slider or the entry fi eld. To perform an additional rotation simply



Sensor 0: X 50, Y 50, Z 50, Axis (● X ○ Y ○ Z), Angle 0

Sensor 1: X 50, Y 50, Z 50, Axis (● X ○ Y ○ Z), Angle 0

Buttons: ☐ Button 1 ☑ Button 2

Figure 21.1: The tracker emulators user interface dialog.

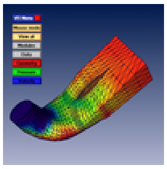
select a different rotation axis and angle. The rotations are performed accumulative. The current rotation doesn't affect the orientation of the main rotation axes. Rotation center is always the current sensor position. To set the sensors rotation to the initial unrotated state, press the *Reset* button.

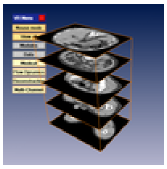
In the buttons section one can toggle the emulated buttons state. If the checkbox is checked, the button is emulated as pressed until its checkbox gets unchecked.

21.5 amiraVR - Tracking module

The tracking module implements head-tracking for immersive virtual reality applications. This means that a correct perspective viewing matrix is computed for an observer wearing stereo glasses with a 3D sensor. The module supports single-pipe multi-channel and true multi-pipe configurations.

The demo script illustrates some basic interaction principles.


tracking demo


demos via 3d menu

Index