Realtime Machine Vision Perception and Prediction

James Bruce
Advisors: Manuela Veloso and Tucker Balch
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
jbruce@cs.cmu.edu

May 2000

0.1 Introduction

0.1.1 Motivation

The need for sensing in truly autonomous robotics is ubiquitous. Among the various sensors that can be applied, one of the most powerful and inexpensive is through machine vision. Color-based region segmentation, where objects in the world are identified by specific color (but not necessarily uniquely), has proved popular in robotics and automation, because color coding is a relatively unobtrusive modification of the environment. With the coding, balls, goals, obstacles, other robots, and even people can be detected. Information can be calculated such as egocentric angle to the object, rudimentary shape characteristics, and with some domain knowledge, distance to the object. While this type of environment has become popular in robotics domains, no general vision library has been developed to fill the need for an inexpensive software-only solution, with performance that can keep up with camera frame rates. Thus this serious need exists in realtime[1] perception and prediction tasks for autonomous systems, yet no library has fully met this need.

0.1.2 Problem

Although popular as a sensor due to low hardware cost, vision has proved difficult due to high processing requirements and a large input stream to sift through in order to generate perceptual information for higher levels in the system. Thus the problem is that of mapping an input video stream to a perceptually more salient representation for other parts of the agent. The representation popular in hardware and domain-specific approaches to this problem is to segment the video stream into colored regions (representing all or part of a colored object). This is the representation we also choose, both for consistency, and because this representation has proved successful [2].

0.1.3 Contributions

It is unclear what general prediction methods can be designed to satisfy the wide variety of information an autonomous agent might perceive, or what behaviors may be necessary to carry out a task. Thus we devoted

[1]We define "realtime" as full frame processing at camera rates (at or above 30Hz.), with bounded processing time guarantees.

our efforts toward developing a vision system that provided all the common information that popular prediction and behavior methods need in order to operate efficiently. This may appear at first to be a given, yet many previous systems have been designed without thought of how they would be used. The result has been a failure of general high performance vision systems to replace ad-hoc solutions created for specific applications. It is our view that taking care to enumerate the uses and demands of many behavioral and prediction methods will fill that gap. Thus our Color Machine Vision Library was developed, or CMVision for short. It is the fastest freely available visual tracking system to our knowledge, has been successfully applied in three robotics systems already, and is poised to serve many more as it has been released to the robotics community under a free non-commercial license. In addition to high performance, the system also has predictable runtime properties, making it a suitable design for soft or hard realtime vision. We address only the former here, mainly out of need and availability, although the points important for the latter will be outlined.

The rest of this report is organized into an evaluation of related work, an exploration for approaches to solving parts of the problem we have identified, motivation for the choices we made, and a presentation of the system we developed. Several existing applications of it are described, and it is evaluated in terms of its progress toward the outlined goals. Finally, documentation for the current software distribution of what we developed is provided.

0.1.4 Related Work

We set out to match the performance of special purpose hardware approaches to machine vision, and to approach the problem from the point of view of what was needed by higher layers, rather than only by what is easy to perform.

During our work we became aware of three similar software systems. The first, XVision, was developed at Yale from 1993 to 1997 [13]. This system began as an edge tracker, but had been extended with additional tracking methods such as the colored region tracking in CMVision. We were happy to find however, that our system offered between 10 and 30 times the performance of XVision in color region tracking. Thus while it did implement a similar feature set, practically it was out of the range of realtime perception and prediction at camera speeds.

Recently two other available libraries have come to our attention, the Vis-

Lib library and the ACTS tracking system, both from ActivMedia robotics [16]. Vislib is another implementation of a subset of XVision's capabilities, but does not appear to be a significant performance increase over it [14]. ACTS apparently has similar performance to CMVision, and is similar in the information it provides. It was made available in early 2000, several months after we first publicly outlined our approach at AAAI-99. It is not freely available, and is not available in source form, making objective comparisons difficult. Both these systems are also limited to the Brooktree BTV848-based frame capture devices, while CMVision natively supports the video for Linux I and II standards [18], and can easily be extended for other platforms or drivers due to source availability and logical separation of vision processing and image capture in the source code.

0.2 Approaches

Here we examine possible segmentation approaches common in machine vision, and evaluate their characteristics and applicability in fitting the goals we wish to meet. Three approaches that often been used are color segmentation, edge or boundary detection, and texture based segmentation. For each, common implementations will also be described.

0.2.1 Color Segmentation

By far the most popular approach in real time machine vision processing has been color segmentation. It is currently popular due to the relative ease of defining special colors as markers or object labels for a domain, and has proved simpler than other methods such as the use of geometric patterns or barcodes. Among the many approaches taken in color segmentation, the most popular employ single-pixel classification into discrete classes. Among these, linear and constant thresholding are the most popular. Other alternatives include nearest neighbor classification and probability histograms.

Linear color thresholding works by partitioning the color space with linear boundaries (e.g. planes in 3-dimensional spaces). A particular pixel is then classified according to which partition it lies in. This method is convenient for learning systems such as neural networks (ANNs), or multivariate decision trees (MDTs) [3].

A second approach is to use nearest neighbor classification. Typically several hundred pre-classified exemplars are employed, each having a unique location in the color space and an associated classification. To classify a new pixel, a list of the K nearest exemplars are found, then the pixel is classified according to the largest proportion of classifications of the neighbors [4]. Both linear thresholding and nearest neighbor classification provide good results in terms of classification accuracy, but do not provide realtime performance using off-the-shelf hardware.

Another approach is to use a set of constant thresholds defining a color class as a rectangular block in the color space [2]. This approach offers good performance, but is unable to take advantage of potential dependencies between the color space dimensions. A variant of the constant thresholding has been implemented in the hardware vision device made by Newton Laboratories [17]. Their product provides color tracking data at realtime rates, but is potentially more expensive than software-only approaches on general purpose hardware.

A final related approach is to store a discretized version of the entire joint probability distribution [12]. So, for example, to check whether a particular pixel is a member of the color class, its individual color components are used as indices to a multi-dimensional array. When the location is looked up in the array the returned value indicates probability of membership. This technique enables a modeling of arbitrary distribution volumes and membership can be checked with reasonable efficiency. The approach also enables the user to represent unusual membership volumes (e.g. cones or ellipsoids) and thus capture dependencies between the dimensions of the color space. The primary drawback to this approach is high memory cost — for speed the entire probability matrix must be present in memory.

0.2.2 Edge Detection/Segmentation

Edge detection uses relative contrast in nearby pixels to determine boundaries in an image. Although popular in traditional machine vision and robotics, it has been difficult to run this type of processing at real time rates without specialized hardware. This is due to the usual representation requiring a neighborhood of pixels in order to generate a value. Color segmentation operates on individual pixels, and so does not incur the overhead of processing areas of pixels.

By itself, the edge detection outlines an approach rather than a particular method. This is because there are many definitions as to what constitutes

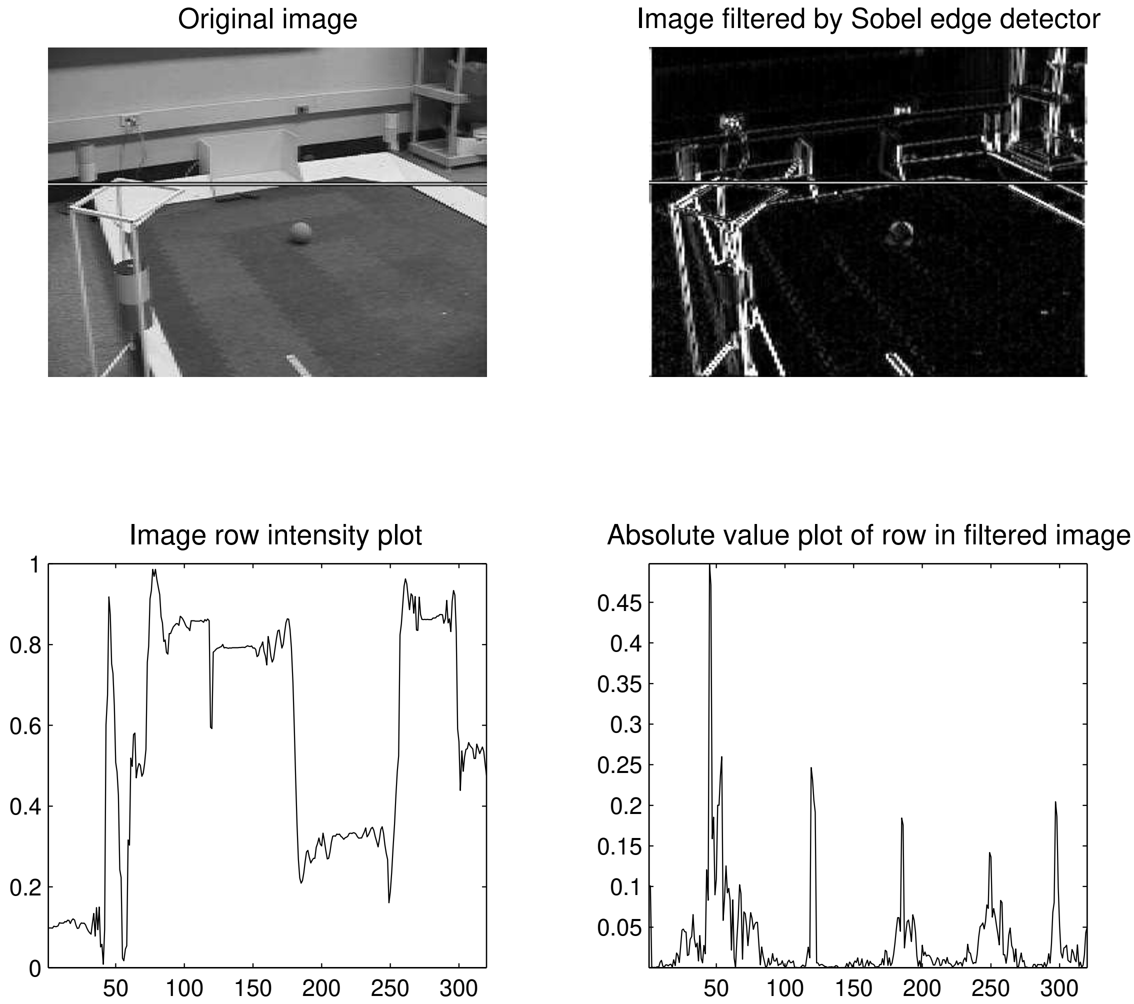

Figure 1: Comparison of original image and the image filtered by a horizontal Sobel operator. The white band in the images indicates the row used for the intensity plots.

an edge value, and thus many representations. The most popular methods generally employ linear shift-invariant filters. These are generally 2D generalizations of 1D signal processing filters and applied through a convolution of the original image[2]. One common such filter is the Sobel operator, which uses a locally weighted difference to compute an edge signal and generally yields good results. The following is a horizontal edge filter used to detect horizontal edges:

$$1/4 \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

The output after applying it to an image can be seen in figure 1. Specifically high values indicate a rising intensity edge and negative values a falling edge. In conjunction with its transpose, the above filter can be used detect edges of any direction. Another alternative is a Laplacian of Gaussian filter[2], which is often approximated by the following 3x3 filter:

$$1/12 \begin{bmatrix} 1 & 2 & 1 \\ 2 & -12 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

This is an omnidirectional filter for detecting edge energy, and thus the absolute value of the filtered image can indicate where boundaries are likely to exist [1].

There are many limitations to segmentation methods based on edges alone, however. Many boundaries contain varying edge sharpness, and regions with no edge at all. The human perceptual system fills such gaps to form completed contours, so these complications are not visible to humans from the original image [7]. A good example would be the vertical pole markers placed around the field in figure 1, which demonstrate strong edge signals in some areas, and little or none in others. Another ambiguity arises in assignment of contours. Often, an edge represents a discontinuity where one nearer object occludes a further one. There is little direct information in the image to indicate which is which however, complicating object detection by requiring what is often complex processing to assign edges and form objects. Because of these limitations, and the relatively high computational cost of edge detection, the system we developed employed color-based vision instead. Once processing power or hardware acceleration advance to the point where both are possible at framerate processing speeds, edge and color segmentation will likely be a good combination, since their processing advantages are complimentary. Color segmentation helps define object areas, while edge detection aids in defining more exact edges and does not rely on prior classification definitions like color thresholds do [1].

0.2.3 Texture Segmentation

Texture image segmentation involves separation of objects based on texture boundaries and texture gradients. Gradient detection involves techniques that can extract perspective transformations of the repeated pattern in order to gauge depth or orientation of surfaces. Texture boundary detection involves the simpler method of assigning objects or regions of objects particular patterns, and separating regions where those patterns abruptly change. Textures are often defined using statistical techniques such as co-occurrence matrices. There are a vast number of overall approaches, and a good introduction to several can be found in Jain [2].

Unfortunately, even the best methods are very slow, limited by strong assumptions, or both. While arguably these methods can encode both color and edge information, and thus are much more powerful or expressive, there will not be extensive use of full texture information in realtime vision without significant algorithmic or processing breakthroughs. Shorter term enhancements to vision systems may be available however, by extracting texture statistics of regions segmented by some other method and employing the measure in a object detection filter. One such feature would be the variance of a color across a colored region.

0.2.4 Object Detection

Object detection, although it can be considered separately, is often composed out of the previous processing steps and treated as a higher level. Color segmentation and edge detection are used to define features which from which objects can be detected. This is the approach commonly taken in machine vision, although there is evidence for much more top-down interaction in humans and animals, where high level knowledge influences relatively early perceptual areas [6]. Such a complicated system is not possible nor necessary at this point however, and a good approximation is to trace the main signal evolution along the perceptual path. This is the bottom-up principle described by Marr [1]. Eventually such complications will have to be taken into account for a more robust system. Although not possible at this point, it is clear that eventually a unified perceptual vision system will employ all the features we identified before: edges, color, texture, and object knowledge. This will require processing hardware and algorithmic advances in for it to become useful at the pace of the world however.

0.2.5 Summary

The three most common approaches to segmentation use colors, edges, or textures, or possibly some combination of them. Given the abilities and limitations, and the available processing power in a software implementation in software, we decided to use color segmentation. Although limited in its power, it has proved fast enough to satisfy realtime requirements at camera frame rates, without hardware acceleration. It has proved robust enough to work well in several autonomous robotics applications, and is much simpler than a hybrid approach that uses edge or texture processing in addition to color.

0.3 System Requirements

The following listing identifies the specific requirements of various domains we would like to be able to support through our system. From these we extract the core requirements that a vision and tracking system should attempt to meet, for several types of consumers of a vision systems output data. Then we asses how well our current system can achieve these goals.

1. *Behavior Based/Reactive Agents* require low latency sensing, so that the internal state after processing will accurately model the current external state in the world. This is due to the assumption that sensor values can be mapped to actions, commonly without predictive world models. Such agents are robust in the case of lost data or dropped frames, and do not necessarily require a correspondence mapping of objects in successive frames (interframe tracking). Finally, realtime performance is not necessary, but is a desirable enhancement.

2. *Active Tracking* of visual field items requires interframe tracking or object correspondence, in order to establish a location history for an object over time. For calculating object velocity and acceleration, low

noise in the vision system's report of locations is a crucial aspect of performance. Latency is normally not an issue, but realtime performance and a low tolerance for lost data are.

3. *Markov Prediction Methods* have proven popular for describing general tracking and prediction tasks where transitions of internal state are either difficult to observe or fully hidden. Performance is highly dependent on the ability to observe accurate interframe tracking data at high rates. Latency and missing data can be handled, but saliency and performance will degrade with large amounts of either.

4. *History Based Prediction Methods* log position data to yield long term statistical position or action distributions. Low latency and realtime performance are not crucial, and lost data recovery is usually unnecessary due to the ability to accommodate lost frames by not logging the data in the distribution. Interframe object tracking is not needed if only positional data is desired, although more complicated logging features may require the tracking just to generate values to be added to the distribution.

5. *Geometric Tracking/Prediction Methods* generate future position estimates using evolution functions, such as Newton's law with predictive function approximators, regression techniques, splines or other interpolation techniques. These methods generally have short term saliency, the latency and performance are important. Interframe tracking is important for accurately modeling the immediate past, as is low positional noise so that predictions or estimations can be accurate. Geometric methods can generally deal with mild amounts of missing data, so recovery is not usually needed when frames are dropped.

The core requirements are thus as follows, and evaluations of CMVision's applicability to each is discussed.

1. *Low Latency.* Because CMVision operates at camera frame rates on single processor systems, with bounded computation, latency can be less than one frame more than the camera. With high amounts of computing power, and relatively low resolutions, practical applications with latencies of less than 5ms from the time of image capture have been achieved.

2. *Interframe Object Tracking.* Our system does not currently address this directly, although it has been used as the basis for systems that do. All of the robotics applications discussed later in this report have some capacity to peform this type of tracking. This is an item slated for future inclusion into the base library, but currently is relegated to the later levels of processing.

3. *Lost Data Recovery.* Again, since tracking is not internal to CMVision, this is not currently implemented within the general low level library. Preliminary results from the RoboCup F180 tracking system (discussed later), indicate that dropped frames, aborted processing, and the much more likely temporary occlusion of objects can be handled well by relatively simple means. The F180 tracking system uses linear interpolation of missing data points after the object reappears to satisfy statistics and prediction methods that cannot handle missing data internally (such as per-frame iterative procedures).

4. *Realtime Performance.* In addition to the performance described above, the processing requirements internal to CMVision are all liner except for one, region merging, which is a approximately linear for all practical purposes. None require exponential or power running times in their worst case, so accurate running time estimates and bounds can be achieved. CMVision can fail if it overruns pre-allocated internal buffer space, but always aborts with a failure code in all such cases. Together these properties allow hard realtime performance for operating systems which support it, and soft realtime in the rest.

0.3.1 Color Spaces

The color space refers to the multidimensional space the describes the color at each discrete point, or pixel, in an image. The intensity of a black and white image is a segment of single dimensional space, where the value varies from its lowest black value to its highest at white. Color spaces generally occupy three spaces, although can be projected into more or fewer to yield other color representations. The common RGB color space consists of a triplet of red, green, and blue intensity values. Thus each color in the representation lies in a cube with black at the corner (0,0,0), and pure white at the value (1.0,1.0,1.0). Here we will describe the different color spaces we considered for our library, including RGB, a projection or RGB we call fractional YRGB, and the YUV color space used by the NTSC and PAL video standards, among other places.

In our choice of appropriate color spaces, we needed to balance what the hardware provides with what would be amenable to our threshold representation, and what seems to provide the best performance in practice. At first we considered RGB, which is a common format for image display and manipulation, and is provided directly by most video capture hardware. It's main problem lies in the intensity value of light and shadows being spread across all three parameters. This makes it difficult to separate intensity variance from color variance with a rectangular, axis aligned threshold. More complex threshold shapes alleviate this problem, but that was not possible in our implementation. An equally powerful technique is to find another color space or projection of one that is more appropriate to describe using rectangular thresholds.

This limitation lead us to explore a software transformed RGB color space we called fractional RGB. It involves separating the RGB color into four channels, intensity, red, green, blue. The color channels in this case are normalized by the intensity, and thus are fractions calculated using the following definition:

$$Y' = \tfrac{(R+G+B)}{3} \tag{1}$$
$$R' = \tfrac{R}{Y'} \tag{2}$$
$$G' = \tfrac{G}{Y'} \tag{3}$$
$$B' = \tfrac{B}{Y'} \tag{4}$$

The main drawback of this approach is of course the need to perform several integer divides or floating point multiplications per pixel. It did however prove to be a robust space for describing the colors with axis-aligned threshold cubes. It proves useful where RGB is the only available color space, and the extra processing power is available.

The final color space we tried was the YUV format, which consists of an intensity (Y) value, and two chrominance (color) values (U,V). It is used in video standards due to its closer match with human perception of color, and since it is the raw form of video, it is provided directly by most analog video capture devices. Since intensity is separated into its own separate parameter, the main cause of correlations between the color component values

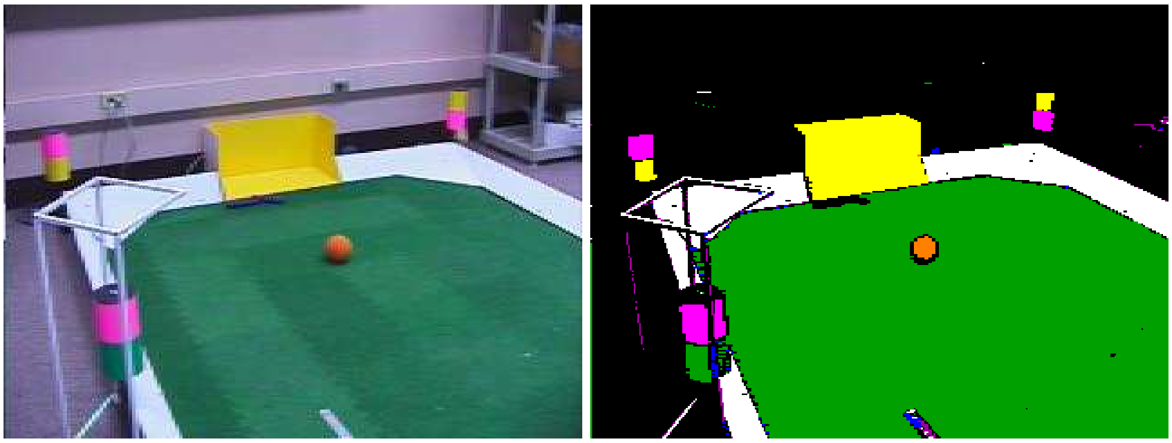

Figure 2: The original video image on the left, and the image with colors classified by predefined YUV thresholds.

has been removed, and thus is a better representation for the rectangular thresholds. This is because the implementation requires axis aligned sides for the thresholds, which cannot model interactions among the component color values. Thus YUV proved to be robust in general, fast since it was provided directly by hardware, and a good match for required assumptions of component independence in our implementation. An example YUV histogram, with a threshold shown outlining a target yellow color is given in figure 3

One color space we have not tried with our library is HSI, or hue, saturation, intensity. In its specification, hue is the angle on the color wheel, or dominant spectral frequency, saturation is the amount of color vs. neutral gray, and I is the intensity. Although easy for humans to reason in (hence its use in color pickers in painting programs), it offers little or no advantage over YUV, and introduces numerical complications and instabilities. Complications come from the angle wrapping around from 360 to 0, requiring thresholding operations work on a modular number values. More seriously, at low saturation values (black, gray, or white), the hue value becomes numerically unstable, making thresholds to describe these common colors unwieldy, and other calculations difficult[9]. Finally, HSI can be approximated by computing a polar coordinate version of the UV values in YUV. Since YUV is available directly from the hardware, it is simplest just to threshold in the pre-existing YUV space, thus avoiding the numerical problems HSI poses.

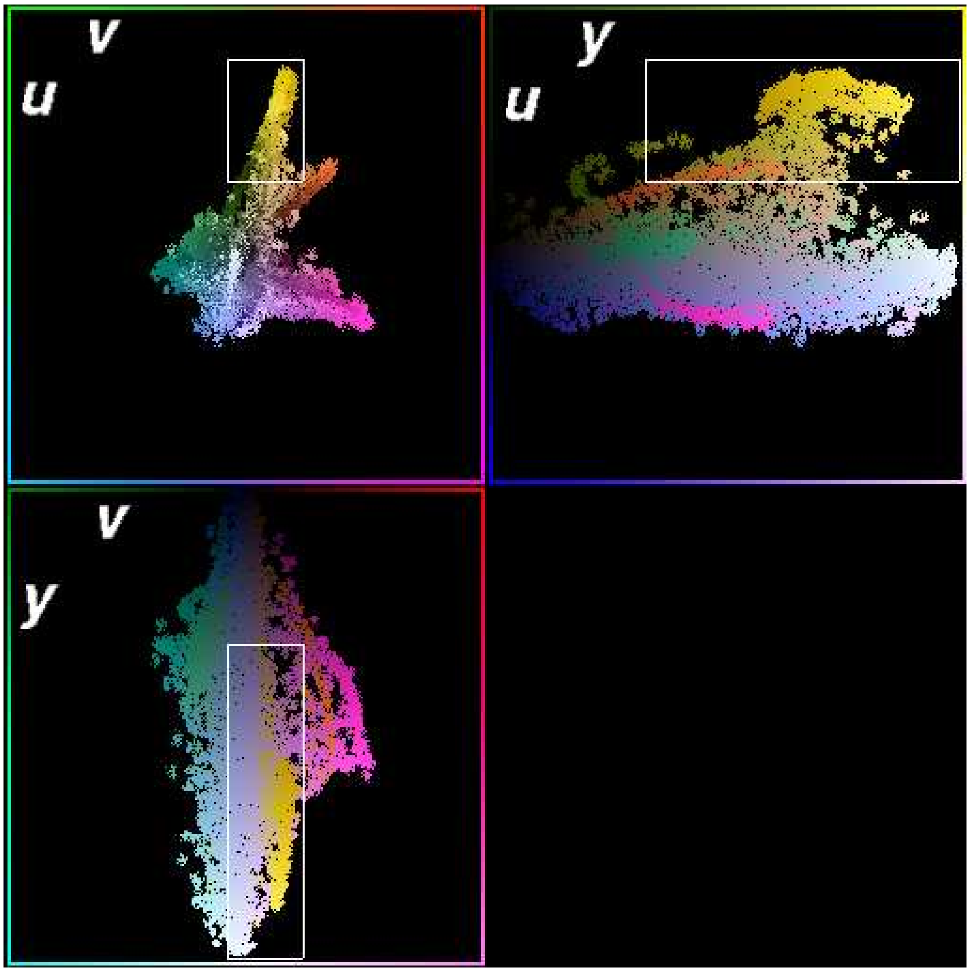

Figure 3: A YUV histogram of a RoboCup soccer field, pictured in figure 2, with a threshold defined for a yellow target color.

0.4 CMVision: The Color Machine Vision Library

0.4.1 Thresholding

The thresholding method described here can be used with general multidimensional color spaces that have discrete component color levels, but the following discussion will describe only the YUV color space, since generalization of this example will be clear. In our approach, each color class is initially specified as a set of six threshold values: two for each dimension in the color space, after the transformation if one is being used. The mechanism used for thresholding is an important efficiency consideration because the thresholding operation must be repeated for each color at each pixel in the image. One way to check if a pixel is a member of a particular color class is to use a set of comparisons similar to

```
if ((Y >= Ylowerthresh)
   AND (Y <= Yupperthresh)
   AND (U >= Ulowerthresh)
   AND (U <= Uupperthresh)
   AND (V >= Vlowerthresh)
   AND (V <= Vupperthresh))
   pixel_color = color_class;
```

to determine if a pixel with values `Y, U, V` should be grouped in the color class. Unfortunately this approach is rather inefficient because, once compiled, it could require as many as 6 conditional branches to determine membership in one color class for each pixel. This can be especially inefficient on pipelined processors with speculative instruction execution.

Instead, our implementation uses a boolean valued decomposition of the multidimensional threshold. Such a region can be represented as the product of three functions, one along each of the axes in the space (Figure 4). The decomposed representation is stored in arrays, with one array element for each value of a color component. Thus class membership can be computed as the bitwise `AND` of the elements of each array indicated by the color component values:

```
pixel_in_class = YClass[Y]
             AND UClass[U]
             AND VClass[V];
```

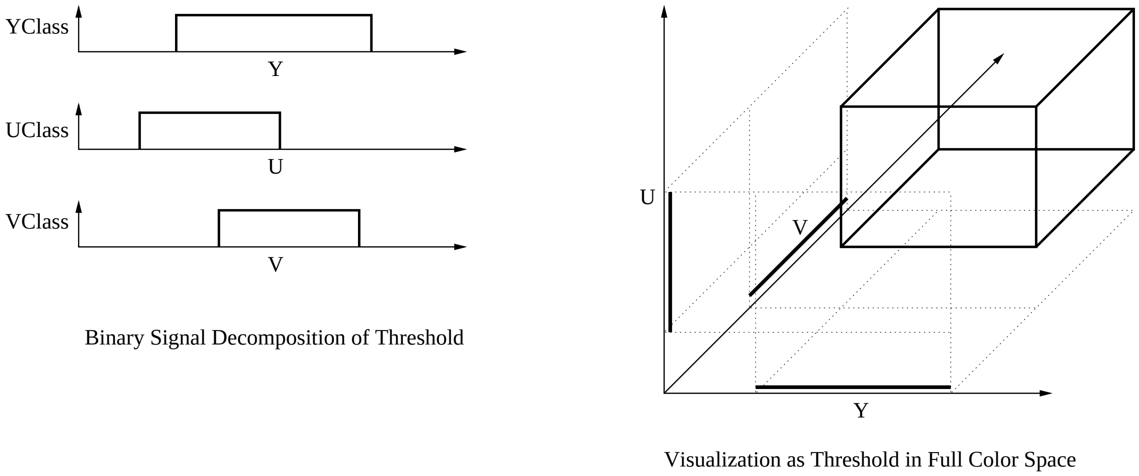

Figure 4: A three-dimensional region of the color space for classification is represented as a combination of three binary functions.

The resulting boolean value of `pixel_in_class` indicates whether the pixel belongs to the class or not. This approach allows the system to scale linearly with the number of pixels and color space dimensions, and can be implemented as a few array lookups per pixel. The operation is much faster than the naive approach because the the bitwise `AND` is a significantly lower cost operation than an integer compare on most modern processors.

To illustrate the approach, consider the following example. Suppose we discretized the YUV color space to 10 levels in each each dimension. So "orange," for example might be represented by assigning the following values to the elements of each array:

```
YClass[] = {0,1,1,1,1,1,1,1,1,1};
UClass[] = {0,0,0,0,0,0,0,1,1,1};
VClass[] = {0,0,0,0,0,0,0,1,1,1};
```

Thus, to check if a pixel with color values `(1,8,9)` is a member of the color class "orange" all we need to do is evaluate the expression `YClass[1] AND UClass[8] AND VClass[9]`, which in this case would resolve to 1, or `true` indicating that color is in the class "orange."

One of the most significant advantages of our approach is that it can determine a pixel's membership in multiple color classes *simultaneously*. By exploiting parallelism in the bit-wise `AND` operation for integers we can determine membership in several classes at once. As an example, suppose the region of the color space occupied by "blue" pixels were represented as follows:

```
YClass[] = {0,1,1,1,1,1,1,1,1,1};
UClass[] = {1,1,1,0,0,0,0,0,0,0};
VClass[] = {0,0,0,1,1,1,0,0,0,0};
```

Rather than build a separate set of arrays for each color, we can combine the arrays using each bit position an array element to represent the corresponding values for each color. So, for example if each element in an array were a two-bit integer, we could combine the "orange" and "blue" representations as follows:

```
YClass[] = {00,11,11,11,11,11,11,11,11,11};
UClass[] = {01,01,01,00,00,00,00,10,10,10};
VClass[] = {00,00,00,01,01,01,00,10,10,10};
```

Where the first (high-order) bit in each element is used to represent "orange" and the second bit is used to represent "blue." Thus we can check whether `(1,8,9)` is in one of the two classes by evaluating the single expression `YClass[1] AND UClass[8] AND VClass[9]`. The result is `10`, indicating the color is in the "orange" class but not "blue."

In our implementation, each array element is a 32-bit integer. It is therefore possible to evaluate membership in 32 distinct color classes at once with two `AND` operations. In contrast, the naive comparison approach could require 32×6, or up to 192 comparisons for the same operation. Additionally, due to the small size of the color class representation, the algorithm can take advantage of memory caching effects.

0.4.2 Connected Regions

After the various color samples have been classified, connected regions are formed by examining the classified samples. This is typically an expensive operation that can severely impact realtime performance. Our connected components merging procedure is implemented in two stages for efficiency reasons.

The first stage is to compute a run length encoded (RLE) version for the classified image. In many robotic vision applications significant changes in

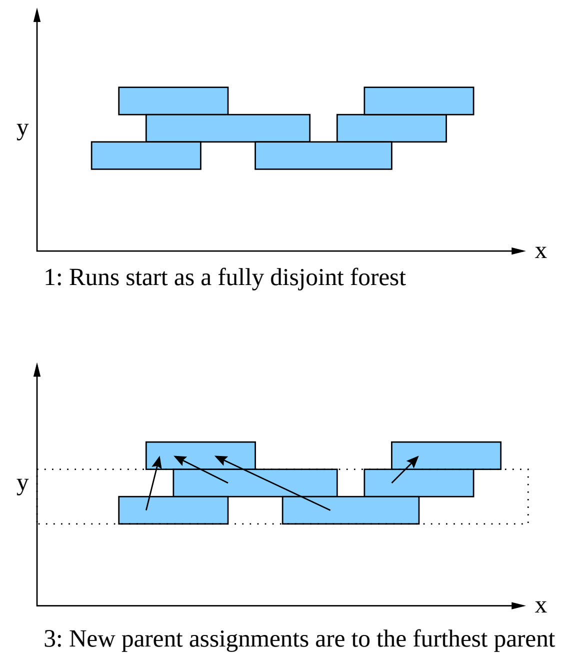

Figure 5: An example of how regions are grouped after run length encoding.

adjacent image pixels are relatively infrequent. By grouping similar adjacent pixels as a single "run" we have an opportunity for efficiency because subsequent users of the data can operate on entire runs rather than individual pixels. There is also the practical benefit that region merging need now only look for vertical connectivity, because the horizontal components are merged in the transformation to the RLE image.

The merging method employs a tree-based *union find* with path compression. This offers performance that is not only good in practice but also provides a hard algorithmic bound that is for all practical purposes linear [5]. The merging is performed in place on the classified RLE image. This is because each run contains a field with all the necessary information; an identifier indicating a run's parent element (the upper leftmost member of the region). Initially, each run labels itself as its parent, resulting in a completely disjoint forest. The merging procedure scans adjacent rows and merges runs which are of the same color class and overlap under four-connectedness. This results in a disjoint forest where the each run's parent pointer points upward toward the region's global parent. Thus a second pass is needed to compress all of the paths so that each run is labeled with its the actual parent. Now each set of runs pointing to a single parent uniquely identifies a connected region. The process is illustrated in Figure 5).

0.4.3 Extracting Region Information

In the next step we extract region information from the merged RLE map. The bounding box, centroid, and size of the region are calculated incrementally in a single pass over the forest data structure. Because the algorithm is passing over the image a run at a time, and not processing a region at a time, the region labels are renumbered so that each region label is the index of a region structure in the region table. This facilitates a significantly faster lookup. A number of other statistics could also be gathered from the data structure, including the convex hull and edge points which could be useful for geometric model fitting.

After the statistics have been calculated, the regions are separated based on color into separate threaded linked lists in the region table. Finally, they are sorted by size so that high level processing algorithms can deal with the larger (and presumably more important) blobs and ignore relatively smaller ones which are most often the result of noise.

0.4.4 Density-Based Region Merging

In the final layer before data is passed back up to the client application, a top-down merging heuristic is applied that helps eliminate some of the errors generated in the bottom up region generation. The problem addressed here is best introduced with an example. If a detected region were to have a single line of misidentified pixels transecting it, the lower levels of the vision system would identify it as two separate regions rather than a single one. Thus a minimal change in the initial input can yield vastly differing results.

One solution in this case is to employ a sort of grouping heuristic, where similar objects near each other are considered a single object rather than distinct ones. Since the region statistics include both the area and the bounding box, a density measure can be obtained. The merging heuristic is operationalized as merging pairs of regions, which if merged would have a density is above a threshold set individually for each color. Thus the amount of "grouping force" can be varied depending on what is appropriate for objects of a particular color. In the example above, the area separating the two regions is small, so the density would still be high when the regions are merged, thus it is likely that they would be above the threshold and would be grouped together as a individual region.

0.5 Applications

0.5.1 RoboCup Sony Quadruped League

The vision for the Sony Quadruped employed most of the features of the CMVision library, except that the hardware provided the YUV thresholding capability natively. The system processed images captured by the robot's camera to report the locations of various objects of interest relative to the robot's current location. In the RoboCup soccer domain these include the ball, 6 unique location markers, two goals, teammates, and opponents. The features of the approach, as presented below, are:

1. **Image capture/classification:** images are captured in YUV color space, and each pixel is classified in hardware by predetermined color thresholds for up to 8 colors.

2. **Region segmenting:** pixels of each color are grouped together into connected regions.

3. **Region merging:** colored regions are merged together based on satisfaction of a minimum density for the merged region set for each color.

4. **Object filtering:** false positives are filtered out via specific geometric filters, and a confidence value is calculated for each object.

5. **Distance and transformation:** the angle and distance to detected objects are calculated relative to the image plane, and then mapped into ego-centric coordinates relative to the robot.

The onboard camera provides 88x60 images in the YUV space at about 15Hz. These are passed through a hardware color classifier to perform color classification in realtime based on learned thresholds.

When captured by the camera, each pixel's color is described as a 3-tuple of 8 bit values in YUV space. The color classifier then determines which color classes the pixel is a member of, based on a rectangular threshold for each class in the two chrominance dimensions (U,V). These thresholds can be set independently for every 8 values of intensity (Y). An example of the results of classification is provided in Figure 6.

The final result of the color classification is a new image indicating color class membership rather than the raw captured camera colors. From this point the standard CMVision processing can follow, although extensions specific to the robot were employed to enhance the system's performance in its specific environment.

The next two stages follow the standard model described above, where the run length encoding is applied, and the union find computation joins the connected color regions. Next, region information is extracted from the merged RLE map. The bounding box, centroid, and size of each region are calculated incrementally in a single pass over the forest data structure.

After the statistics have been calculated, the regions are separated by color into separate threaded linked lists in the region table. Finally, they are sorted by size so that later processing steps can deal with the larger (and presumably more important) blobs, and ignore relatively smaller ones which are most often the result of noise.

The next step attempts to deal with one of the shortcomings of object detection via connected color regions. Due to partial occlusion, specular highlights, or shadows, it is often the case that a single object is broken into a few separate but nearby regions. A single row of pixels not in the same color class as the rest of the object is enough to break connectivity, even

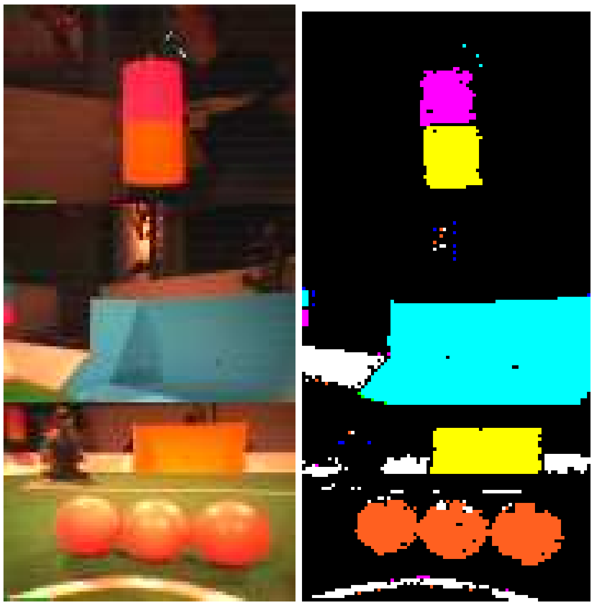

Figure 6: An example of our image classification on the Sony robots is on the right, along with the corresponding original video image on the left. The images are a composite of objects: a position marker (top), a goal area (middle) and three soccer balls (bottom).

though the object may occupy many rows. In order to correct for cases when nearby regions are not connected but should be considered so, a density based merging scheme was employed. Density is represented here as the ratio of the number of pixels of the color class in the connected region to the overall area of its bounding box. By this measurement heuristic, two regions that have a small separation relative to their sizes will likely be merged, since they would tend to have relatively high density.

The next step is to finally calculate the location of the various objects given the colored regions. Various top down and geometric object filters are applied in each case to limit the occurrence of false positives, as well as serving the basis for confidence values.

For the ball, it is determined as the largest orange blob below the horizon. The confidence value is calculated as the error ratio of the density of the detected region and the actual density of a perfect circle. The distance is estimated as the distance required for a circular object to occupy the same area as the observed region. The field markers are detected as pink regions with green, cyan, or yellow regions nearby. The confidence is set as the error ratio of the difference between the squared distance between the centers of the regions and the area of each region (since they are ideally adjacent square patches, these two should be equal).

The colored blob on the marker indicates position along the length of the field. The relative elevation of the pink and colored regions disambiguates which side of the field the marker is on given the assumption that the robot's head is not upside-down. Thus the marker represented by a pair of regions can be uniquely determined. In case of multiple pairs which are determined to be the same marker, the one of maximal confidence is chosen. The distance to the marker is estimated from the distance between the centers of the two regions, since they are of known size.

The goals are detected similarly. They are the largest yellow or cyan regions with centers below the horizon. The distance measure is a very coarse approximation based on the angular height of the goal in the camera image, and the merging density is set to a very low value since many occlusions are possible for this large, low lying object. The confidence is estimated based on the difference in comparing the relative width and height in the image to the known ratio of the actual dimensions.

The final objects detected are opponents and teammates. Due to the multiple complicated markers present on each robot, no distance or confidence was estimated, but regions were presented in raw form as a list of patches.

Figure 7: The first minnow robot tracking a soccer ball.

These simply indicate the possible presence of an opponent or teammate.

Finally, the vision system must transform from image coordinates to egocentric coordinates. The system performed well in practice; it had a good detection rate and was robust to the unmodeled noise experienced in a competition due to competitors and crowds. The distance metrics and confidence values were also useful in this noisy environment.

For a more detailed description of the other components of the system, see Veloso et. al[8].

0.5.2 Minnow Autonomous Robot

The second application of the vision system is the minnow autonomous robot project. Minnow's goal is to investigate teams consisting of simple, reliable, and inexpensive robots, cooperating to perform complex tasks. The platform is a Cye robot[10], with local control through a single board computer with a camera and video capture card. The computer has an AMD K-6 350 processor, and runs CMVision and the TeamBots[11] robot architecture on top of Linux operating system.

The vision system is the primary sensor, and processes 160x120 images at 30Hz. It currently tracks a ball, obstacles, and a goal. Tracking of other possible features in the RoboCup soccer domain is currently being explored. CMVision is linked to the Java TeamBots architecture using a JNI API, which is included in the TeamBots distribution.

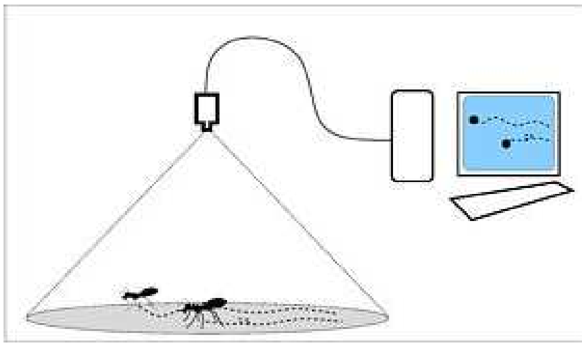

Figure 8: General design of automated biological tracking system.

0.5.3 RoboCup F180 Small Robot League

The next robotics application, which is currently being developed, is a realtime tracking system for RoboCup's F180 Small Robot League. In this domain, 18cm maximum diameter robots play 5 on 5 competitive soccer on a field the size of a ping-pong table. Control is usually offboard, and global sensors are allowed, which is typically a camera mounted above the field. Robot speeds in excess of 2m/s, and ball speed up to 5m/s are now possible by some teams, making this a challenging domain in which to apply and develop vision based tracking and prediction systems. CMVision is fast enough to track the objects using full frame processing, although extensions are being developed to handle the relatively small segmented regions (often only 2x2 pixels), applying post processing to remove false positives due to noise pixels in the acquired image, and to accurately locate the visible markers and objects with subpixel resolution.

0.5.4 Biological Agent Tracking

The newest application of CMVision is to track biological agents using machine vision as part of an automated logging system to observe behaviors. Currently the system can track fish, and work is underway to track ants even on natural backgrounds. Work in these highly noisy environments with natural colors is indicating enhancements to be made to the system in order to deal with such objects more robustly. Ultimately, if the system proves robust enough to log long term activity, we hope to be able to make statistically sound statements about group behavior in natural organisms. It also offers

Figure 9: The experimental apparatus set up to track fish.

a good test bed for improving the robustness of our visual tracking system for use in less artificial domains.

0.6 Conclusion

We have presented a new system to accelerate low level segmentation and tracking using color machine vision. We evaluated the properties of existing approaches, chose color thresholding as a segmentation method, and YUV or fractional YRGB as robust color spaces for thresholding. We then presented a system that can perform bounded computation, full frame processing at camera frame rates. The system can track up to 32 colors at resolutions up to 640x480 and rates at 30 or 60Hz without specialized hardware. Thus the primary contribution of this system is that it is a software-only approach implemented on general purpose, inexpensive, hardware. This provides a significant advantage over more expensive hardware-only solutions, or other, slower software approaches. The approach is intended primarily to accelerate low level vision for use in realtime applications where hardware acceleration is either too expensive or unavailable.

Building on this lower level, we were able to incorporate geometric object detection, and are working on high performance tracking and prediction layers. Successful applications have been made using the software in the Sony Quadrupeds, the Minnow robots, the RoboCup F180 League, and tracking animals.

In our continuing work we hope to address higher processing levels of tracking and more elaborate prediction models. We are looking at methods to perform automatic online recalibration of thresholds, and detection of more types of visual features such as lines or color transitions. We are also exploring hybrid systems with edge detection or texture processing in conjunction with color vision to provide a more robust system where higher processing power permits.

Bibliography

[1] D. Marr. Vision: A Computational Investigation into the Human Representation and Processing of Visual Information. W. H. reeman and Company, San Francisco, 1982.

[2] R. Jain, R. Kasturi, and B.G. Schunck. *Machine Vision*. McGraw-Hill, 1995.

[3] C.E. Brodley and P.E. Utgoff. Multivariate decision trees. *Machine Learning*, 1995.

[4] T.A. Brown and J Koplowitz. The weighted nearest neighbor rule for class dependent sample sizes. *IEEE Transactions on Information Theory*, pages 617–619, 1979.

[5] R.E. Tarjan. Data structures and network algorithms. *Data Structures and Network Algorithms*, 1983.

[6] T. Watanabe, S. Miyauchi. Roles of Attention and Form in Visual Motion Processing: Phsychophisical and Brain Imaging Studies *High Level Motion Processes: Computational, Neurobiological, and Psychophysical Perspectives*, Ed. Takeo Watanab. MIT Press, Cambridge Massachusetts, 1998, p95-113

[7] M. Singh, D.D. Hoffman, M.K. Albert. Contour Completion and Relative Depth: Petter's rule and support ratio. *Psychological Science*, Vol 10(5), 1999, 423-428.

[8] M. Veloso, E. Winner, S. Lenser, J. Bruce, T. Balch. Vision-Servoed Localization and Behaviors for an Autonomous Quadruped Legged Robot Artificial Intelligence Planning Systems, 2000

[9] The Color FAQ http://www.inforamp.net/ poynton/ColorFAQ.html

[10] The Cye Personal Robot http://www.probotics.com, 2000

[11] The TeamBots Robot Control Architecture http://www.teambots.org, 2000

[12] E.M. Silk. Human detection and recognition for an hors d'oeuvres serving robot. http://www.cs.swarthmore.edu/ silk/robot/, 1999.

[13] A Brief Tour of XVision. http://www.cs.yale.edu/AI/VisionRobotics/XVision/, 1997.

[14] VisLib vision library. http://robots.activmedia.com/vislib/, 1997.

[15] ACTS vision library. http://robots.activmedia.com/acts/, 1997.

[16] ActivMedia Robotics. http://www.activrobots.com.

[17] Newton Laboratories. Cognachrome image capture device. http://www.newtonlabs.com, 1999.

[18] Video For Linux II. http://millennium.diads.com/bdirks/v4l2.htm, 2000.

0.7 Appendix

0.7.1 Machine Vision Hardware and Drivers

For our work, we have successfully used several BTV848-based capture cards using the Video4LinuxI driver under 2.2.x series Linux kernels, starting with 2.2.6. We have not had success getting these cards to work with beta Video4LinuxII drivers used 2.2.x or 2.3.x kernels, although success has been reported elsewhere. We expect this will be easier once Video4LinuxII is a more stable API, and the drivers have matured.

We have used the Winnov Videum AV series capture cards successfully under Video4LinuxII, with 2.2.x series kernels starting with 2.2.12. The available driver has not been updated for 2.3.x kernels as of this writing.

Finally, the vision processing library has been compiled and run under Microsoft Windows^{tm} using the available AVI capture facilities. It was compiled under Visual C++ 4.0.

0.7.2 CMVision Library Documentation

Global Types

```
struct yuv422{
  unsigned char y1,u,y2,v;
};
```

A structure for the common YUV422 standard machine color format. Most video capture cards that are capable of capturing YUV support this color format. Each structure describes two adjacent pixels in the image, each with an intensity value (y1,y2), and shared u,v values. The reason the chrominance values are half-sampled is because the NTSC (and PAL) video standards favor high quality intensity at the cost of color signal quality. The balance in the sampling thus accurately reflects the available information from the input video stream. Each component value can range from [0..255].

```
struct rgb{
  unsigned char red,green,blue;
};
```

The RGB color structure, common for computer display systems. It is present in CMVision for output purposes, and for visually identification of color thresholds in test output. Each component ranges from [0..255].

CMVision Public Types

```
struct CMVision::region{
  int color;          // id of the color
  int area;           // occupied area in pixels
  int x1,y1,x2,y2;    // bounding box (x1,y1) - (x2,y2)
  float cen_x,cen_y;  // centroid
  int sum_x,sum_y;    // temporaries for centroid calculation
  region *next;       // next region in list
};
```

This is the region structure returned by CMVision for object feature description. The members are defined as follows:

1. *color* This identifies the color of the region. Each threshold or color class is identified by a value from [1..31]. The number is assigned by its location in the configuration file, with 1 being the first color threshold in the file, and increasing from there.

2. *area* This describes the area in pixels of the connected region, where each pixel was determined to be a member of the color class *color* by the current thresholds.

3. *(x1,y1)-(x2,y2)* This identifies the bounding box of the connected region, with (x1,y1) being the smaller screen coordinates of the bounding box, and (x2,y2) being the larger. The screen coordinate system follow the common "matrix" numbering convention, where (0,0) is the top left of the image, and x increases to the right, and y increases downward.

4. *(cen_x,cen_y)* This is the centroid of the connected region, with equal weighting on every pixel in the region. This is in subpixel resolution screen coordinates.

5. *sum_x,sum_* These are calculation temporaries used in the calculation of centroids, among other possible uses. When passed to the user program no particular value should be assumed, thus the values are undefined.

6. *next* This member is a pointer to the next region structure. Regions are passed back to the user program in the form of a linked list, thus each item indicates the next item in the list using this field.

CMVision Public Methods

```
CMVision();
```

This is the CMVision constructor method. No external initialization is made by this function other than setting internal structure values to guarantee consistent results if member functions are called while in an uninitialized state. User programs should call the "initialize" method before attempting to use the other methods.

```
~CMVision();
```

This is the destructor, which is an alias for the explicit “close” method. No harm is caused by redundant calls to “close”, and user programs are encouraged to call “close” when they are done using CMVision, rather than waiting for the CMVision object to fall out of scope. This alias is provided when such would not be convenient.

```
bool initialize(int nwidth,int nheight);
```

This is the main initialization function, which initializes internally allocates structures for vision processing, at the specified video resolution. If memory initialization is successful, true is returned. Otherwise, false is returned. Before calling region and processing methods, the thresholds should be set explicitly or loaded from an options file.

```
bool loadOptions(char *filename);
```

This is a function for loading an options file. See “colors.txt” in the CMVision distribution for an example of the file format. The options file contains the color class name, visualization color, and the default YUV thresholds for that color class.

```
bool saveOptions(char *filename);
```

This method saves the current color information in a new options file, including any modifications made to the threshold values. This is expected to be used by interactive programs for setting thresholds or automatic adaptive threshold systems before shutdown.

```
void close();
```

This method is the main uninitialization call. It removes all allocated internal state if any is present. May be called redundantly without any ill-effects.

```
bool testClassify(rgb *out,yuv422 *image);
```

This method classifies a video frame, and outputs the representative color (stored in the threshold information file) for in an equally sized RGB output frame. An example of such output can be seen in figure 6. Use of this method is intended primarily for visualization and debugging, and not for normal autonomous operation, thus it is not optimized heavily.

```
bool getThreshold(int color,
       int &y_low,int &y_high,
       int &u_low,int &u_high,
       int &v_low,int &v_high);
```

This method returns the maximum and minimum y,u,v threshold values. As per the machine representation, each value is in the range [0..255]. The color parameter specifies which threshold to retrieve, and ranges within [1..CMV_MAX_COLOR] (currently 32). If the color value is out of range, false is returned and none of the output parameters are set, otherwise they are set to the current values and true is returned.

```
bool setThreshold(int color,
       int y_low,int y_high,
       int u_low,int u_high,
       int v_low,int v_high);
```

This method sets the current threshold values using the same representation as getThreshold. If color is out of range, false is returned and no threshold values are set. Otherwise the current thresholds are set to those in the parameters, and true is returned.

```
char *getColorName(int color);
```

This method returns the human readable identifier for color, as appears in the initialization file. The color parameter specifies which color name to retrieve, and ranges within [1..CMV_MAX_COLOR].

```
bool processFrame(yuv422 *image);
```

This method performs full processing on the passed video image, applying the current thresholds and extracting the connecting regions and their associated statistics. If processing was fully successful, true is returned and region structures may be retrieved with getRegions. On failure, false is returned, and the regions are set in a consistent, but possibly incomplete state. Thus if false is returned, processing was aborted and the regions retrieved with getRegions are accurate, but may be incomplete in the form of missing regions. The most likely cause of failure is due to a full intermediate buffer caused by excessive noise in the image. If required the library can be recompiled with larger internal buffers.

```
int numRegions(int color_id);
```

Returns the number of regions of the specified color that were last processed by a processFrame operation. Color is the color identifier, with the range as before.

```
region *getRegions(int color_id);
```

Returns the head of the regions list for the specified color, as last created in by a processFrame operation. The list is sorted by area, with the largest occurring first, and terminated by a NULL in the region's "next" field.