REAL SHADOWS

1a

1b

1c

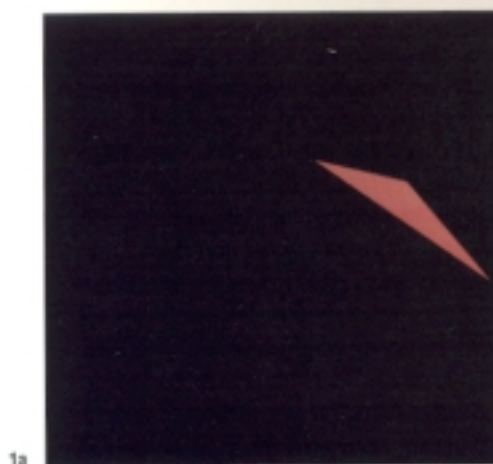**S**hadows in a three-dimensional scene are a powerful cue to understanding the shape of an object and its relation to other objects in the scene. Historically, shadow creation technique has been limited as shadows could only be cast onto a single flat surface, and objects could not shadow themselves. With some clever use of the stencil planes and the z-buffer on the IRIS VGX graphics hardware, it is possible to generate correct shadows, cast on any number and shape of objects, and do it in real-time for fairly complex scenes.

A number of techniques have been devised for calculating and displaying shadows, ray-tracing being only one of them. The method we are using here is called "shadow volumes." First proposed by Frank Crow in 1977[1], it works like this: Consider a triangle lit by a single point light source (Figure 1a). The volume of space behind the triangle in the shape of a truncated pyramid extending off to infinity is in shadow (Figure 1b). If you place a surface behind the triangle (Figure 1c), the part that lies in that volume will be in shadow. To calculate the value of that part of the surface, consider just the polygonal faces which bound the shadow volume.

For the shadow volume cast by a triangle, there are three faces. View them as if they were a semi-transparent shell which intersects the other objects in the scene, and look at the scene from your desired point of view. At every pixel in the scene, as you travel along the path from your viewpoint to the object visible at that pixel, count the number of times you cross through one of these semi-transparent surfaces going into shadow, and subtract the number of times you cross going out of shadow (Figure 1d). When you finally hit the object, if you have gone into shadow more times than you have gone out, then that pixel of the object is in shadow (Figure 1e).

REAL TIME

In the VGX implementation of the algorithm, we let the stencil planes do the counting. The stencil planes are additional bitplanes on the VGX which can be used to limit drawing to certain areas of the screen on a pixel-by-pixel basis. Like the z-buffer, the stencil plane values are set at all the pixels covered when a line or polygon is drawn to the screen. Also like the z-buffer, a test can be done on the stencil value at each pixel — greater than, less than, equal to some value, and so on — to decide whether that pixel gets drawn. But unlike the z-buffer, the stencil plane values can be made to increment or decrement at each pixel covered by a line or polygon. That feature lets us do the counting we need to make this method successful.

First, we draw the entire scene from the viewers perspective to provide proper values into the z-buffer. Then, after clearing the stencil planes to zero, we draw the faces of the shadow volume as polygons, using the "increment" stencil operation on faces which are going into shadow, and "decrement" on faces going out. We do a z-buffer comparison when drawing these faces to correctly calculate where objects in the scene hit the shadow volumes, but we do not change the values already in the z-buffer or do any new drawing into the color planes.

If we draw the objects in our scene the first time with just the emitted and ambient components of their surface material — those components that would be visible in shadow — then all that remains is to draw the entire scene again, adding the diffuse and specular material components at those pixels which are not in shadow. We can constrain our drawing to those areas by setting a stencil condition to succeed only at pixels where the stencil planes have a zero value — that is, where we have gone out of shadow as many times as we have gone into the shadow.
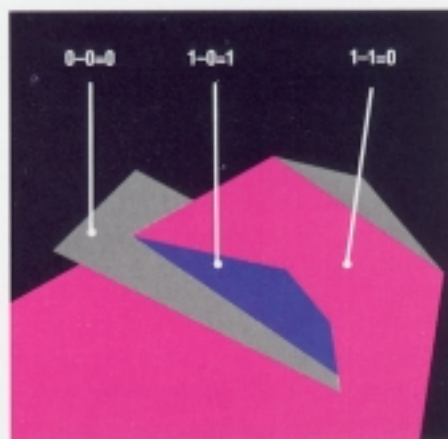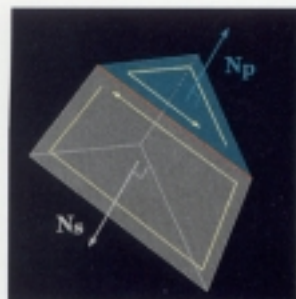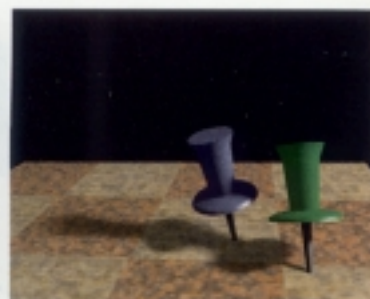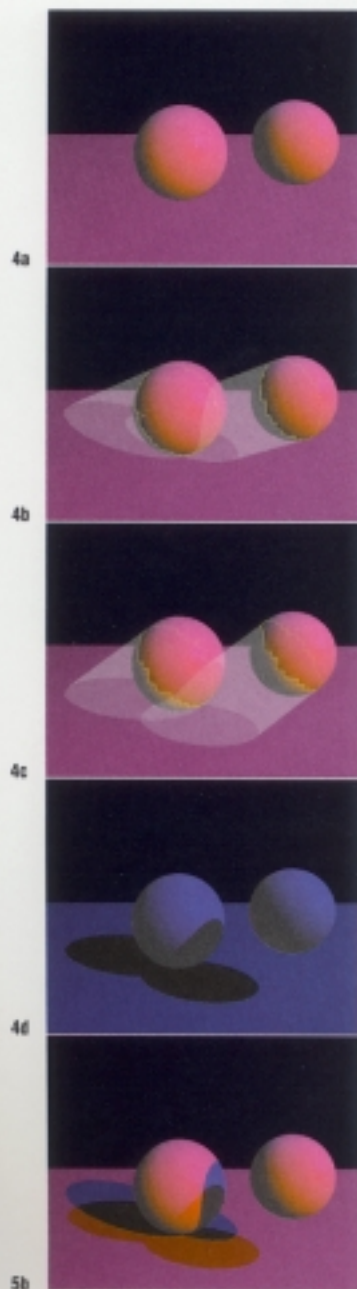

1d

1e

BY TIM HEIDMANN

2

3

6

4a

4b

4c

4d

5b

If you now repeat the process of clearing the stencil planes — drawing shadow volume faces and adding highlight material components with a light source of a different color in a different position — you can build up the correct result for a scene containing any number of lights (Figures 4a, 4b, 4c, 4d, 5b)!

Now that we have explained the basics of the idea, let's explore some more details about the technique:

How do you calculate the vertices of the faces bounding the shadow volumes? To accomplish this, for every polygon edge in every object casting shadows, you need to project a line from the light source through the vertices on either end of the edge for some arbitrary distance to come up with two more vertices, which complete the quadrilateral face of a shadow volume. Although the volumes should actually extend to infinity, initially we need to pick a length which will extend through every object potentially in shadow, but still lets us draw the resulting quadrilateral.

How do you determine whether a shadow volume face is going into shadow or going out? To manage this, it is important to pay close attention to how polygons are drawn. Assume all polygons of the objects in your scene are drawn in a counter clockwise direction, so each polygon normal (Np) faces outward from the object (Figure 2). When a polygon is facing the light source, if the shadow volume face obtained by projecting an edge is drawn in the direction opposite that of the direction of the original edge, the normal of that face (Ns) will point outward from the shadow volume. When the scene is viewed from the eye's position, front-facing poly-

gons will be going into shadow, and back-facing polygons will be going out of the shadow. When the original polygon is facing away from the light source, these directions are reversed.

Although the shadow volumes method works if we project every single edge of every polygon in the scene, all we really need to do is consider the faces formed by projecting the silhouette edges of an object. A silhouette edge can be defined as an edge bounding only one polygon, or an edge shared by two polygons, one of which is facing toward the light and the other facing away (Figure 3).

On the current VGX hardware, stencil values are clamped from 0 to 255. If we draw a going-out-of-shadow face before the corresponding going-into-shadow face, we could hit the lower limit. To avoid this, either draw all going-into-shadow faces first, or take the simpler approach of using some positive number, such as 128, as the "zero" stencil value. This allows shadow faces to be drawn in any order with some range for the count to go high or low.

Figure 6 was created from thirty-two separate images in only a few seconds on the VGX hardware. By drawing a shadowed scene several times, each time moving the light source a small amount and accumulating the images in the soft shadows, we achieve accurate soft shadows. The technique described is just a starting point for a wide variety of applications.

Tim Heidmann is Silicon Graphics' Manager of Technology for Creative Applications.

[1]"Shadow Algorithms for Computer Graphics," by Frank Crow, *Computer Graphics*, Vol. 11, No.3, Proceedings of SIGGRAPH 1977, July 1977.

```
/* Example Code Segment for Drawing Accurate Shadows */

/* Set up. */
. . .
if (getgdesc(GD_BITS_STENCIL) == 0) exit(1);        /* This requires stencil hardware.          */
stensize(getgdesc(GD_BITS_STENCIL));                /* Use all the stencil bitplanes.           */
gconfig();
zbuffer(TRUE);                                      /* z comparison always active.              */
. . .

/* Draw scene w/ ambient light, setting z-buffer. */
for (i=0; i<nLights; i++)                           /* Turn on all the lights for               */
  lmbind(LIGHT0+i, light[i].index);                 /* ambient light value calculation.         */
zwritemask(0xffffffff);                             /* We do want to set z values,              */
czclear(0x00000000, getgdesc(GD_ZMAX));             /* using the full z range.                  */
blendfunction(BF_ONE, BF_ZERO);                     /* No alpha blending.                       */
DrawObjects(matl_emitambient);                      /* Draw all objects using only ambient      */
for (i=0; i<nLights; i++)                           /* and emitted material components.         */
  lmbind(LIGHT0+i, 0);                              /* Turn off all the lights when done.       */

/* Accumulate effects of each light in the scene in turn. */
for (i=0; i<nLights; i++) {
  /* Draw faces of shadow volumes, setting stencil planes.                                      */
  lmbind(LIGHT0, light[i].index);                   /* Turn on just this light.                 */
  sclear(MYZERO);                                   /* Reset the stencil values to ''zero''     */
  zwritemask(0x0);                                  /* We aren't changing any z values          */
  wmpack(0x0);                                      /* or any color values.                     */
  for (iEdge=0; iEdge<nEdges; iEdge++) {            /* For each edge in the scene,              */
    v3= . . .; v4= . . .; ingoing= . . .;           /* project shadow face from this edge.      */
    stencil(TRUE, 0, SF_ALWAYS, 0xff,               /* Write stencil unconditionally,           */
      ST_KEEP, ST_KEEP,                             /* do nothing if zbuffer test fails,        */
      ingoing ? ST_INCR : ST_DECR)                  /* otherwise, add or subtract one.          */
    bgnpolygon(); . . .                             /* Draw the shadow volume face.             */
  }

  /* Add highlights for this light, constrained by stencil mask to unshadowed areas. */
  zclear(getgdesc(GD_ZMAX));                         /* Reset the z-buffer contents.             */
  zwritemask(0xffffffff);                            /* Enable writing to the z- buffer and      */
  wmpack(0xffffffff);                                /* the color planes. Set up to add          */
  blendfunction(BF_ONE, BF_ONE);                     /* highlight color to existing values.      */
  stencil(TRUE, MYZERO, SF_EQUAL, 0xff,              /* Constrain drawing to ''zero'' areas      */
    ST_KEEP, ST_KEEP, ST_KEEP);                      /* of stencil. Don't change stencil.        */
  DrawObjects(matl_diffspec);                        /* Draw lit part of object material.        */
  lmbind(LIGHT0, 0);                                 /* Leave this light off for next pass.      */
}
. . .
```