Chapter 8

Hidden-Surface Removal

When you look at a 3-D scene, you see only the surfaces that are nearest to the eye, while other surfaces behind them are obscured (provided the items are opaque). Drawing speed can be improved by drawing only the items that are visible to the eye in the final scene. *Hidden-surface removal* is the process of determining in advance which surfaces are not visible in the final scene, in order to prevent them from being drawn. This chapter describes how to do hidden-surface removal.

- Section 8.1, “z-buffering,” tells you how to remove hidden surfaces by drawing only the surface closest to the eye.
- Section 8.2, “Using z-buffer Features for Special Applications,” tells you how to use other techniques for determining visible surfaces.
- Section 8.3, “Stenciling,” tell you how to create stencils that allow you to update drawings selectively.
- Section 8.4, “Eliminating Backfacing Polygons,” tells you how to remove hidden surfaces by drawing only the polygons that face the viewer.
- Section 8.5, “Alpha Comparison,” tells you how to use special alpha hardware to indicate transparency.

Two basic methods of hidden-surface removal are discussed in this chapter. One method of determining surface visibility is to use a *z*-buffer to keep track of which item is closest to the eye at each pixel. Another method is to disable drawing for polygons that face away from the viewer. Backfacing polygon removal is not as general as *z*-buffering, but *z*-buffering may be slower than backface removal on some systems.

8.1 *z*-buffering

The *z-buffer* is a bitplane, associated with a framebuffer, that stores the distance from the near clipping plane to each pixel in the window. In *z*-buffer mode, the *z* coordinate (distance to the eye) of the incoming (next to be drawn) pixel is compared to the *z* coordinate of the geometry already drawn at that pixel. If the incoming *z* value shows that the new geometry is closer to the eye than the existing geometry, the values of the old pixel and of the old *z* value, which are stored in the color framebuffer and the *z*-buffer, are replaced by the new ones.

The calculation is performed on a per-pixel basis, because it is possible to have a set consisting of as few as three polygons, each of which is overlapped by another in the set, as shown in Figure 8-1.

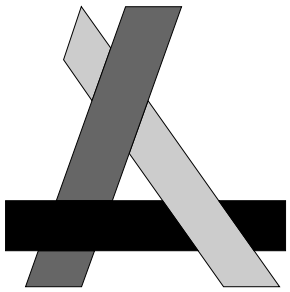

Figure 8-1 Overlapping Polygons

Not all systems support z-buffering. Use `getgdesc(GD_BITS_NORM_ZBUFFER)` to return the z-buffer availability. The draw mode must be set to `NORMALDRAW` to use z-buffering.

On most systems, the *z*-buffer is a hardware option. The size of the *z*-buffer can be from 24 bits to 32 bits per screen pixel, depending on the system type. The *z* value is signed on all systems except IRIS-4D / GT / GTX systems.

RealityEngine systems provide software support for selecting the size of the hardware z-buffer to be 0 or 32 bits and support for multisampled z-buffering, which is described in Chapter 15. See *zbsize*(3G) for more information.

The IRIS Indigo Entry system has a software *z*-buffer, with 32 bit *z*-buffer memory allocated on a per window basis, rather than per screen.

A 24-bit hardware *z*-buffer is optional on XS and XS24 systems and is standard on Elan systems. On Elan systems, the low bit of the 24-bit *z* buffer is reserved for fast clears, so that bit should be ignored when reading data back from the *z*-buffer. Elan systems also allocate bits from the *z*-buffer for stencil operations, so the resolution of the *z*-buffer is decreased when performing stenciling.

By default, *z*-buffering is turned off. To set up *z*-buffering, you enable *z*-buffer mode, then write the maximum *z* value to every location in the *z*-buffer, using the following commands:

```
zbuffer(TRUE);/* Enable z-buffering */
zclear();    /* Write the maximum z value to the z-buffer */
```

Before the system draws anything, it compares the *z* value of each incoming pixel to the *z*-buffer value for that pixel. If the *z* value of the incoming pixel is smaller than the value in the *z*-buffer, the pixel is colored, and that pixel's *z*-buffer value is set to the new *z* value. If the incoming pixel's *z* value is greater than the corresponding *z*-buffer value, the pixel is not drawn. The values in the *z*-buffer thus always represent the distance to the item that is currently closest to the eye.

The color value stored in the bitplanes represents the color of that item. Use `getzbuffer()` to determine whether or not *z*-buffering is enabled; TRUE means *z*-buffering is enabled and FALSE means it is not enabled.

Another consideration when using *z*-buffering is that the *znear* and *zfar* values in the call to `perspective()` have a profound effect on the resolution of the *z*-buffer's comparison facility. The *z*-buffer contains a finite number of integers, each with a limited range of values that can be used to compare against the *z* value of the incoming pixel. You can enhance the resolution of the *z*-buffer by setting the near and far values as close together as possible. With a smaller range of total values, more bits of precision are available. It is particularly important to move the near clipping plane as far from the eye as possible.

This sample program, *zbuffer.c*, draws three rectangular boxes that tumble through one another while the whole scene rotates. While the left mouse button is up, the scene is drawn without *z*-buffering; when you press it, *z*-buffering is enabled. If you run the program with an argument—by entering **`zbuffer 5`**, for example, there is a short delay before drawing each polygon. The left mouse button still controls the *z*-buffering, but it is easier to see what is happening because you can see each polygon drawn one at a time.

```
#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>

float v[8][3] = {
    {-1.0, -1.0, -1.0},
    {-1.0, -1.0, 1.0},
    {-1.0, 1.0, 1.0},
    {-1.0, 1.0, -1.0},
    { 1.0, -1.0, -1.0},
    { 1.0, -1.0, 1.0},
    { 1.0, 1.0, 1.0},
    { 1.0, 1.0, -1.0},
};
unsigned int delaycount;

void delay()
{
    if (delaycount)
        sleep(delaycount);
}
void drawcube()
{
    color(RED);
    bgnpolygon();
        v3f(v[0]);
        v3f(v[1]);
        v3f(v[2]);
        v3f(v[3]);
    endpolygon();
    delay();
    color(GREEN);
    bgnpolygon();
        v3f(v[0]);
        v3f(v[4]);
        v3f(v[5]);
        v3f(v[1]);
    endpolygon();
```

```
    delay();
    color(BLUE);
    bgnpolygon();
        v3f(v[4]);
        v3f(v[7]);
        v3f(v[6]);
        v3f(v[5]);
    endpolygon();
    delay();
    color(YELLOW);
    bgnpolygon();
        v3f(v[3]);
        v3f(v[7]);
        v3f(v[6]);
        v3f(v[2]);
    endpolygon();
    delay();
    color(MAGENTA);
    bgnpolygon();
        v3f(v[5]);
        v3f(v[1]);
        v3f(v[2]);
        v3f(v[6]);
    endpolygon();
    delay();
    color(CYAN);
    bgnpolygon();
        v3f(v[0]);
        v3f(v[4]);
        v3f(v[7]);
        v3f(v[3]);
    endpolygon();
}

main(argc, argv)
int argc;
char *argv[];
{
    Angle xrot, yrot, zrot;
    short val;

    xrot = yrot = zrot = 0;
    if (argc == 1)
        delaycount = 0;
    else
        delaycount = 1;
```

```
if (getgdesc(GD_BITS_NORM_ZBUFFER) == 0) {
    fprintf(stderr, "Z-buffer not available on this machine\n");
    return 1;
}
prefsize(400, 400);
winopen("zbuffer");
if (delaycount == 0)
    doublebuffer();
gconfig();
mmode(MVIEWING);
ortho(-4.0, 4.0, -4.0, 4.0, -4.0, 4.0);
qdevice(ESCKEY);
qdevice(LEFTMOUSE);  /* don't want window manager to act on clicks */

while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
    pushmatrix();
        rotate(xrot, 'x');
        rotate(yrot, 'y');
        rotate(zrot, 'z');
        color(BLACK);
        clear();
        if (getbutton(LEFTMOUSE)) {
            zbuffer(TRUE);
            zclear();
        }
        else
            zbuffer(FALSE);
        pushmatrix();
            scale(1.2, 1.2, 1.2);
            translate(0.3, 0.2, 0.2);
            drawcube();
        popmatrix();
        pushmatrix();
            rotate(450 + zrot, 'x');
            rotate(300 - xrot, 'y');
            scale(1.8, 0.8, 0.8);
            drawcube();
        popmatrix();
        pushmatrix();
            rotate(500 + yrot, 'z');
            rotate(-zrot, 'x');
            translate(-0.3, -0.2, 0.6);
            scale(1.4, 1.2, 0.7);
            drawcube();
        popmatrix();
    popmatrix();
```

```
        if (delaycount == 0)
            swapbuffers();
        xrot += 11;
        yrot += 15;
        if (xrot + yrot > 3500)
            zrot += 23;
        if (xrot > 3600)
            xrot -= 3600;
        if (yrot > 3600)
            yrot -= 3600;
        if (zrot > 3600)
            zrot -= 3600;
    }
    gexit();
    return 0;
}
```

The part of the program that turns on the *z*-buffering is the two subroutines:

```
zbuffer(TRUE);
zclear();
```

First, `zbuffer(TRUE)` enables *z*-buffer comparisons to be made before each write, then `zclear()` sets all the *z* values to the largest possible value for pixels in the viewport. In this example, `zbuffer(TRUE)` is called for every frame; however, this is normally not necessary because a typical program turns it on at the beginning. The code is written as it is because the left mouse button can come up at any time, in which case *z*-buffering is turned off.

8.1.1 Controlling *z* Values

Just as `viewport()` controls the scaling of *x* and *y* coordinates, there is a subroutine, `lsetdepth()`, that controls the scaling of *z* coordinates. `lsetdepth()` takes two arguments, corresponding to the near and far clipping planes. By default, *near* is set to the minimum value that can be stored in the *z*-buffer and *far* is set to the maximum value.

These values are system-dependent. Use `getgdesc(GD_ZMIN)` to return the minimum *z* value and `getgdesc(GD_ZMAX)` to return the maximum *z* value, so that you can set *near* and *far* to their minimum and maximum values, regardless of what type of system is used, by calling:

```
lsetdepth(getgdesc(GD_ZMIN), getgdesc(GD_ZMAX));
```

Table 8-1 lists the minimum and maximum *z* values for different models. These are signed values on all systems except IRIS-4D / GT / GTX systems.

System Model	Minimum Value	Maximum Value
IRIS-4D / B or G	0x4000	0x3FFF
IRIS-4D / GT or GTX	0	0x7FFFFF
Personal IRIS, IRIS-4D / VGX, VGXT, SkyWriter, XS, XS24, Elan	0x800000	0x7FFFFF
IRIS Indigo, RealityEngine	0x80000000	0x7FFFFFFF

Table 8-1 Maximum and Minimum *z*-buffer Values

Note: z-buffer hardware is optional on the XS, XS24, and Elan. These systems do not provide a software *z*-buffer in lieu of the hardware *z*-buffer.

8.1.2 Clearing the *z*-buffer and the Bitplanes Simultaneously

A common code sequence in programs that do *z*-buffering is:

```
color(0);
clear();
zclear();
```

This code clears the color bitplanes to zero, then clears the *z*-buffer bitplanes to the maximum value. Unfortunately, it takes a relatively long time, because `clear()` touches each pixel first, then `zclear()` touches each pixel again. In recent hardware implementations, the hardware can, in certain cases, simultaneously clear the color planes and the *z*-buffer planes. Use `czclear()` to do simultaneous clearing of color and *z*-buffer:

```
czclear(long color, long zval)
```

The circumstances and results of using `czclear()` are different on different systems. See the *czclear(3G)* man page for details. `czclear()` clears the bitplanes to *color* and the *z*-buffer to *zval* simultaneously.

Using czclear on IRIS-4D/VGX, VGXT and SkyWriter Systems

IRIS-4D / VGX, VGXT, SkyWriter, and RealityEngine always clear the banks of color and *z* bitplanes sequentially, regardless of the values of *cval* and *zval*.

Using czclear on IRIS-4D/GT/GTX Systems

On IRIS-4D / GT / GTX systems, `czclear()` simultaneously clears the *z*-buffer and bitplanes if circumstances allow it. These systems can perform a simultaneous clear under the following conditions:

- In RGB mode, the 24 least-significant bits of color (red, green, and blue) must be identical to the 24 least-significant bits of *zval*. In the case of RGB mode, it is common to set the background color to black (all zeros). This makes it necessary for you to reverse the orientation of the *z*-buffer near / far clipping values.The following two function calls reverse the *z*-buffer orientation, so that the maximum distance to which all *z* values are initially cleared is 0 instead of ZMAX:

  ```
  lsetdepth(getgdesc(GD_ZMAX), 0x0);
  zfunction(ZF_GEQUAL);
  ```

  At this point, all that has changed is that the system has positioned the viewer so that all *z* compares take place with *near* mapped to a large number and *far* mapped to 0.

- In color map mode, the 12 least-significant bits of color must be identical to the 12 least-significant bits of *zval*. Because the color parameter is an index into the color map index, only the lowest 12 bits are significant.

Using czclear on Personal IRIS, XS, XS24, and Elan Systems

On the Personal IRIS, you can speed up `czclear()` by as much as a factor of four for common values of *zval* if you call `zfunction()` with it, so that one of the conditions in Table 8-2 is met. See Section 8.2.2, “Alternative Comparisons and z-buffer Writemasks,”for information about `zfunction()`.

zval	zfunction
getgdesc(GD_ZMIN)	ZF_GREATER or ZF_GEQUAL
getgdesc(GD_ZMAX)	ZF_LESS or ZF_LEQUAL

Table 8-2 Values of `zfunction()` for Personal IRIS `czclear()`

8.2 Using z-buffer Features for Special Applications

This section discusses special features associated with *z*-buffering. Most of them are rarely used, so this section can be skipped on first reading. Topics include writing directly into the *z*-buffer, using alternate depth comparison functions and sources, and using writemasks for the *z*-buffer.

8.2.1 Drawing into the *z*-buffer

There are certain applications where it is useful to write values directly into the *z*-buffer. `zclear()` is actually a special case of writing into the *z*-buffer, where the values in the *z*-buffer are all set to a particular depth value.

In a flight simulator, for example, suppose that the view on the screen includes an instrument panel surrounding the plane's windshield. If the instrument panel does not change from frame to frame, there is no reason to redraw it, so it might be nice to clear only the portion of the screen and *z*-buffer corresponding to the view out the plane's windshield and redraw only that portion of the window for each frame.

To do this, set the current "color" to the value returned by the call to `getgdesc(GD_ZMAX)` on your system. Use `zdraw()` to write this value into the *z*-buffer, then draw the polygon(s) representing the windshield. When the outside view is drawn, it is always masked by the plane's windshield frame and instrument panel (which is closer to the eye). Thus an extremely complex instrument panel is possible, because it needs to be drawn only once.

When `zdraw()` is TRUE, `zbuffer()` must be set to FALSE, otherwise both try to alter the *z*-buffer contents simultaneously. In color map mode, the value written to the *z*-buffer by `zdraw()` is the index of the color specified. For example, `color(2)` writes 2s to the *z*-buffer.

`zdraw()` is similar to `frontbuffer()` and `backbuffer()` in that it permits writing into the *z*-buffer bank. Normally, if you are writing into the *z*-buffer, you do not want to write into the front buffer or back buffer at the same time. Usually, drawing into the *z*-buffer should be bracketed by subroutines that first set `backbuffer(FALSE)` and then `backbuffer(TRUE)`, assuming that the program is in double buffer mode. In single buffer mode, `frontbuffer()` normally has no effect. However, if you call `frontbuffer(FALSE)`, a flag is set so that when `zdraw()` is TRUE, the front buffer (which is the only buffer in

single buffer mode) is not written. If `zdraw()` is FALSE, `frontbuffer(FALSE)` has no effect.

On Elan, XS, and XS24 systems, do not use `zdraw()` when drawing into a clipped window, or when using read/modify/write operations such as antialiasing or `logicop()`.

This sample program, *zdraw.c*, illustrates the `zdraw()` masking technique. It draws a spinning cube, seen through square cut-outs in a green wall.

```
#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>
#define RGB_BLACK 0x000000
#define RGB_GREEN 0x00ff00
#define HOLESIZE 32
#define HOLESEP (HOLESIZE/2)

float v[8][3] = {
    {-1.0, -1.0, -1.0},
    {-1.0, -1.0, 1.0},
    {-1.0, 1.0, 1.0},
    {-1.0, 1.0, -1.0},
    { 1.0, -1.0, -1.0},
    { 1.0, -1.0, 1.0},
    { 1.0, 1.0, 1.0},
    { 1.0, 1.0, -1.0},
};

int face[6][4] = {
    {0, 1, 2, 3},
    {3, 2, 6, 7},
    {7, 6, 5, 4},
    {4, 5, 1, 0},
    {1, 2, 6, 5},
    {0, 4, 7, 3},
};

unsigned long facecolor[6] = {
    0xff0000,    /* blue */
    0x0000ff,    /* red */
    0x00ffff,    /* yellow */
    0xffff00,    /* cyan */
    0xff00ff,    /* magenta */
    0xffffff,    /* white */
};
```

```
void drawcube()
{
    int i;
    for (i = 0; i < 6; i++) {
        cpack(facecolor[i]);
        bgnpolygon();
            v3f(v[face[i][0]]);
            v3f(v[face[i][1]]);
            v3f(v[face[i][2]]);
            v3f(v[face[i][3]]);
        endpolygon();
    }
}
main(argc, argv)
int argc;
char *argv[];
{
    int i, j;
    Angle xang, yang;
    short val;
    Boolean use_geom;
    unsigned long holez[HOLESIZE*HOLESIZE];

    if (getgdesc(GD_BITS_NORM_DBL_RED) == 0) {
        fprintf(stderr, "Double buffered RGB not available on this machine\n");
    return 1;
    }
    if (getgdesc(GD_BITS_NORM_ZBUFFER) == 0) {
        fprintf(stderr, "Z-buffer not available on this machine\n");
        return 1;
    }
    if (getgdesc(GD_ZDRAW_GEOM) == 0 && getgdesc(GD_ZDRAW_PIXELS) == 0) {
        fprintf(stderr, "Z-buffer drawing not available on this machine\n");
        return 1;
    }
    prefsize(400, 400);
    winopen("zdraw");
    RGBmode();
    doublebuffer();
    gconfig();
    qdevice(ESCKEY);
    mmode(MVIEWING);
    ortho(-2.0, 2.0, -2.0, 2.0, -2.0, 2.0);
    zbuffer(TRUE);
    zclear();
    use_geom = getgdesc(GD_ZDRAW_GEOM) == 1;
```

```
if (!use_geom) {
    holez[0] = getgdesc(GD_ZMAX);
    for (i = 1; i < HOLESIZE*HOLESIZE; i++)
        holez[i] = holez[0];
}

/* draw the green wall once */
cpack(RGB_GREEN);
frontbuffer(TRUE);
pushmatrix();
    translate(0.0, 0.0, 1.9);
    rectf(-2.00, -2.00, 2.00, 2.00);
popmatrix();
frontbuffer(FALSE);
xang = yang = 0;
while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
/* create the holes in the green wall */
zbuffer(FALSE);
zdraw(TRUE);
backbuffer(FALSE);
if (use_geom) {
    ortho2(-0.5, 399.5, -0.5, 399.5);
    cpack(getgdesc(GD_ZMAX));
}
for (i = 100; i <= 300; i += 50) {
    for (j = 100; j <= 300; j += 50) {
    if (use_geom)
        rectf(i, j, i + HOLESIZE - 1, j + HOLESIZE - 1);
    else
        lrectwrite(i, j, i +HOLESIZE - 1, j + HOLESIZE - 1, holez);
    }
}
if (use_geom)
    ortho(-2.0, 2.0, -2.0, 2.0, -2.0, 2.0);
zdraw(FALSE);
backbuffer(TRUE);
zbuffer(TRUE);

/* z-buffered clear to background color and depth */
cpack(RGB_BLACK);
pushmatrix();
    translate(0.0, 0.0, -1.9);
    rectf(-2.00, -2.00, 2.00, 2.00);
popmatrix();
```

```
    /* draw the outside scene */
    pushmatrix();
        rotate(xang, 'x');
        rotate(yang, 'y');
        drawcube();
    popmatrix();
    swapbuffers();

    /* update the rotation angles for next time through */
    xang += 11;
    yang += 17;
    if (xang > 3600)
        xang -= 3600;
    if (yang > 3600)
        yang -= 3600;
    }
    gexit();
    return 0;
}
```

This program is written in RGB mode, because in color map mode, every color, including the "color" drawn into the *z*-buffer, is masked to 12 bits.

The idea behind the program is that a green wall is drawn nearer the eye than anything else. This sets all the *z*-buffer values so they record data near the eye. In the main loop, 25 holes are drilled into the wall by setting the *z*-buffer values in the squares to indicate that the surface is far away. Then a black background is drawn farther from the eye than any part of the cube. Finally, the cube is drawn, and it is visible only through the holes.

8.2.2 Alternative Comparisons and *z*-buffer Writemasks

In the default *z*-buffer mode, the *z* coordinate of the incoming pixel is compared to the *z* coordinate of the geometry already drawn at that pixel. If the incoming *z* value shows that the new geometry is closer to the eye than the old one, the values of the old pixel and of the old *z* value are replaced by the new ones. Thus the new value is compared to the old, and if it is less than the old, the old quantities are replaced.

It is possible to change the comparison function from "less-than" to another type of decision, to achieve a different effect. To change the comparison function, use `zfunction()`.

The *z* comparison functions are:

`ZF_NEVER`	Never overwrite the source pixel value.
`ZF_LESS`	Overwrite the source pixel value if the *z* value of the source pixel is less than the *z* value of the destination pixel.
`ZF_EQUAL`	Overwrite the source pixel value if the *z* value of the source pixel is equal to the *z* value of the destination pixel.
`ZF_LEQUAL`	Overwrite the source pixel value if the *z* value of the source pixel is less than or equal to the *z* value of the destination pixel (default).
`ZF_GREATER`	Overwrite the source pixel value if the *z* value of the source pixel is greater than the *z* value of the destination pixel.
`ZF_NOTEQUAL`	Overwrite the source pixel value if the *z* value of the source pixel is not equal to the *z* value of the destination pixel.
`ZF_GEQUAL`	Overwrite the source pixel value if the *z* value of the source pixel is greater than or equal to the *z* value of the destination pixel.
`ZF_ALWAYS`	Always overwrite the source pixel value regardless of value of the destination pixel.

You can also control writing into the *z*-buffer with `zwritemask()`. This might be useful for using a very complicated background into which a few items are to be drawn and moved quickly. Setting `zwritemask()` to zero locks the background information in, and prevents its modification; the new items are drawn or not depending on the depth comparison.

8.3 Stenciling

IRIS-4D / VGX, SkyWriter, and RealityEngine systems support an additional *z*-buffer-like test that uses a different algorithm from the one described previously in this chapter. This test uses the flexible frame buffer configuration on these systems to allocate *stencil planes*.

Stenciling, like *z*-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using GL drawing primitives, then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, like decals, outlining, and constructive solid geometry rendering.

The `stencil()` statement controls testing of stencil bitplanes before the system writes to the frame buffer:

```
void stencil(long enable, unsigned long ref, long func,
             unsigned long mask, long fail, long pass, long zpass);
```

The first argument to `stencil()` enables or disables stenciling. To enable stenciling, call `stencil()` with *enable* set to TRUE. If you call `stencil()` with *enable* set to FALSE, the following parameters are ignored and stencil testing is not performed.

When stenciling is enabled, the system tests the defined stencil bitplanes for each pixel against a programmed reference value before writing to that pixel. Based on the contents of the stencil bitplanes and the programmed tests defined by the `stencil()` statement, the system then conditionally modifies the pixel's contents (both for the color bitplanes and for the *z*-buffer) by a programmed value, as defined in the call to `stencil()`.

Once you enable stenciling, the system tests the color and *z*-buffer bitplanes for each pixel. The results of the tests are determined by a reference value, passed through the argument *ref*, the value in the stencil bitplanes, and a stencil function that operates on them both.

This function, passed as the argument *func*, can be one of the following:

SF_NEVER	Do not perform the specified stencil update (passed as the value of *fail*, *pass*, and *zpass*) regardless of the results of the comparison.
SF_LESS	Perform the specified stencil update (specified as the value of *fail*, *pass*, and *zpass*) if *ref* is less than the value in the stencil planes.
SF_EQUAL	Perform the specified stencil update (specified as the value of *fail*, *pass*, and *zpass*) if *ref* is equal to the value in the stencil planes.
SF_LEQUAL	Perform the specified stencil update (specified as the value of *fail*, *pass*, and *zpass*) if *ref* is less than or equal to the value in the stencil planes.
SF_GREATER	Perform the specified stencil update (specified as the value of *fail*, *pass*, and *zpass*) if *ref* is greater than the value in the stencil planes.
SF_NOTEQUAL	Perform the specified stencil update (specified as the value of *fail*, *pass*, and *zpass*) if *ref* is not equal to the value in the stencil planes.
SF_GEQUAL	Perform the specified stencil update (specified as the value of *fail*, *pass*, and *zpass*) if *ref* is greater than or equal to the value in the stencil planes.
SF_ALWAYS	Perform the specified stencil update (specified as the value of *fail*, *pass*, and *zpass*) regardless of the results of the comparison.

The *mask* argument to stencil defines which stencil bitplanes are significant during the comparison operation. Use this argument to ignore individual planes you do not want to use in the stencil test.

When stencil performs its test as defined by *func*, it returns one of three possible values:

fail	The stencil test (defined in the call to stencil) fails.
pass	The stencil test passes, but the *z*-buffer test fails.
zpass	The stencil test passes, and the *z*-buffer test passes.

These three possible values are reflected as the arguments *fail*, *pass*, and *zpass* (the last three arguments passed to stencil). If the *z*-buffer is not enabled, only

fail and *pass* are considered. These arguments define the operation to be performed based on the results of the stencil test. The system performs one of the functions defined by the value of *fail*, *pass*, and *zpass* passed to `stencil()`.

`ST_KEEP`	Keep the value currently in the bitplanes (no change).
`ST_ZERO`	Replace the contents of the pixel with zeros.
`ST_REPLACE`	Replace the contents of the pixel with the value of *ref*.
`ST_INCR`	Add 1 to the contents of the pixel. This is clamped to the maximum value of the pixel at that location.
`ST_DECR`	Subtract 1 from the contents of the pixel. This is clamped to 0.
`ST_INVERT`	Invert all bits in that pixel.

Based on the results of the test, the system performs the function that applies to the conditions.

A sample stencil command follows, with a description of its results.

```
stencil(TRUE, 220, SF_EQUAL, 0xff, ST_REPLACE, ST_KEEP, ST_KEEP);
```

In this example, the system compares 220 against the contents of the stencil planes. Because *mask* is 0xff, all eight planes are valid in this comparison. The test is to see whether the stencil planes are exactly equal to *ref*, which is 220. If the test fails— that is, if the contents of the stencil planes do not equal *ref*— the system replaces them with the value of *ref* (220). Both *pass* and *zpass* are set to `ST_KEEP`, which means that there is no change to the pixels or to the *z*-buffer if the test passes. If the *z*-buffer is enabled, color and depth are drawn only in the *zpass* case (meaning that both color and *z*-buffer planes pass the test). If the *z*-buffer is not enabled, *zpass* is ignored and only the *pass* function is performed.

stensize

Use `stensize()` to define the bitplanes you wish to use as the stencil:

```
void stensize(long planes)
```

You can define up to 8 stencil planes. Systems without the optional alpha bitplanes allocate the stencil bitplanes from the least-significant planes of the *z*-buffer. Use `getgdesc()` to determine whether your system has alpha bitplanes. Once you have allocated a number of bitplanes for use as a stencil,

these planes can be used to store information that is later used by the `stencil()` statement.

sclear

Use `sclear()` to set the value of every pixel in the stencil buffer:

```
void sclear(unsigned long sval)
```

Pass the desired value to `sclear()` as *sval*. The clearing operation is limited by the current `viewport()` and `scrmask()` statements in effect, and is masked by the current `swritemask()`.

swritemask

Use `swritemask()` to specify which of the stencil bitplanes can be modified by `sclear()` and normal stencil operation:

```
void swritemask(unsigned long mask)
```

The next sample program, *stencil.c*, uses stenciling to render the outline of an object in an arbitrary image. Because the bitmap of an image takes a lot of space, the code mimics drawing the image by actually drawing polygons.

The image is a basic checkerboard on black background. It "jitters" four times and increments the stencil value each time a pixel is hit. This leaves the outlines with stencil values of 0x1, 0x2, and 0x3, while pixels with stencil values of 0x0 and 0x4 are completely outside and inside the object, respectively. The last step is to render a polygon over the entire region, turning on only those pixels that are on the outline.

```
#include <stdio.h>
#include <gl/gl.h>

floatrect0[4][2] = {
    {-0.5, -0.5},
    { 0.5, -0.5},
    { 0.5, 0.5},
    {-0.5, 0.5},
    };
```

```
main()
{
    if (getgdesc(GD_BITS_STENCIL) == 0) {
        fprintf(stderr, "stencil not available on this machine\n");
        return 1;
    }

    prefsize(400, 400);
    winopen("stencil");
    RGBmode();
    stensize(3);
    gconfig();
    mmode(MVIEWING);
    ortho2(-10.0, 10.0, -10.0, 10.0);

    cpack(0);
    clear();
    sclear(0);
    wmpack(0);
    stencil(1, 0x0, SF_ALWAYS, 0x7, ST_INCR, ST_INCR, ST_INCR);
    viewport(0, 398, 0, 398);
    checker();
    viewport(1, 399, 0, 398);
    checker();
    viewport(1, 399, 1, 399);
    checker();
    viewport(0, 398, 1, 399);
    checker();

    stencil(1, 0x0, SF_NOTEQUAL, 0x3, ST_KEEP, ST_KEEP, ST_KEEP);
    wmpack(0xffffffff);
    scale(15.0, 15.0, 0.0);
    bgnpolygon();
        v2f(rect0[0]);
        v2f(rect0[1]);
        v2f(rect0[2]);
        v2f(rect0[3]);
    endpolygon();

    sleep(10);
    gexit();
    return 0;
}
```

```
checker()
{
    int i,j;

    cpack(0x0000ff00);
    pushmatrix();
    translate(-5.0, -5.0, 0.0);
    for (i=0; i<9; i++) {
        for (j=0; j<9; j++) {
            translate(1.0, 0.0, 0.0);
            if ((i^j) & 0x1) {/* checker board */
                bgnpolygon();
                    v2f(rect0[0]);
                    v2f(rect0[1]);
                    v2f(rect0[2]);
                    v2f(rect0[3]);
            endpolygon();
            }
        }
        translate(-9.0, 1.0, 0.0);
    }
    popmatrix();
}
```

8.4 Eliminating Backfacing Polygons

In a scene composed entirely of opaque closed surfaces, *backfacing polygons* (polygons whose face is pointing away from the viewer) are never visible. Eliminating these invisible polygons from the scene has an obvious benefit—it speeds drawing time by not drawing some of the polygons in the scene. This is especially useful on systems such as the IRIS Indigo that have slower *z*-buffer rates.

The idea is that if the polygons making up a surface are all oriented the same way, and if the surface is closed, after transformation, all the polygons on the front have one orientation and those on the back have the opposite orientation. A special mode can be turned on to check whether the transformed polygons are oriented clockwise or counterclockwise, and only those oriented counterclockwise are drawn. The method is not sufficient for all hidden surface removal if the object being drawn is not convex, or if there is more than one object.

A backfacing polygon is defined as a polygon whose vertices appear in clockwise order in screen space. When backfacing polygon removal is turned on, only polygons whose vertices appear in counterclockwise order are displayed, that is, polygons that point toward you. Therefore, the vertices of all polygons should be specified such that they are drawn in counterclockwise order when the front face of the polygon is visible (see Chapter 2 for examples).

Use `backface()` to initiate or terminate backfacing polygon removal. The `backface()` utility is used to improve the performance of programs that represent solid shapes as collections of polygons. The vertices of the polygons on the side of the solid facing away from the viewer are in clockwise order and are not drawn. `backface()` takes a single argument. TRUE enables backfacing polygon elimination, and FALSE (the default) disables it.

Use `frontface()` to initiate or terminate frontfacing polygon removal. The `frontface()` utility is used to display hidden surfaces (backfacing polygons). In this case, the polygons on the side of the solid facing away from the viewer are drawn; those facing the viewer are not drawn. `frontface()` takes a single argument. TRUE enables frontfacing polygon elimination, and FALSE (the default) disables it.

`getbackface()` returns the state of backfacing polygon removal. If backface removal is on, the system draws only those polygons that face the viewer. If backfacing polygon removal is enabled, 1 is returned; otherwise 0 is returned.

8.5 Alpha Comparison

On some systems, you can also use the alpha planes to determine whether to draw pixels by comparing incoming alpha values to a reference constant value. Not all systems support alpha operations. Use `getgdesc(GD_BITS_NORM_SNG_ALPHA)` to test for the availability of alpha planes.

Note: RealityEngine systems feature multisampled alpha comparison, which is described in Chapter 15.

Use `afunction()` to compare the alpha values of source pixels to the value of *ref*:

```
void afunction(long ref, long func)
```

Depending on the value of *func*, `afunction()` determines whether a pixel is completely transparent and draws the pixel conditional to its transparency. `afunction()` assumes that alpha values are proportional to pixel coverage, which is the case if you are using `pointsmooth()`, `linesmooth()`, or `polysmooth()`.

The `afunction()` call makes the system draw pixels only if their alpha value is not equal to 0. Pixels with 0 alpha are presumed to be completely transparent—according to the conventions of `pointsmooth()`, `linesmooth()`, or `polysmooth()`.

The `afunction()` statement compares source alpha values against a reference value that you include in the `afunction()` call. You also specify a comparison function that determines the conditions under which `afunction()` permits the system to draw pixels.

To make the system avoid drawing invisible pixels, call `afunction()` as follows:

```
afunction(0, AF_NOTEQUAL);
```

To return the system to its default operation, call `afunction()` as follows:

```
afunction(0, AF_ALWAYS);
```

This call causes the alpha hardware to compare the values of all pixels in the normal manner.

The following sample program, *afunction.c*, uses `afunction()` to define the shape of a building and its windows. See Chapter 18 to learn how to use texturing.

```
#include <stdio.h>
#include <gl/gl.h>

float mt0[3][3] = {                     /* mountain 0 coordinates */
    {-15.0, -10.0, -15.0},
    { 10.0, -10.0, -15.0},
    { -5.0, 5.0, -15.0},
};

float mt1[3][3] = {                     /* mountain 1 coordinates */
    {-10.0, -10.0, -17.0},
    { 15.0, -10.0, -17.0},
    { 6.0, 12.0, -17.0},
};

float bldg[4][3] = {                    /* building coordinates */
    {-8.0, -10.0, -12.0},
    { 8.0, -10.0, -12.0},
    { 8.0, 10.0, -12.0},
    {-8.0, 10.0, -12.0},
};

float tbldg[4][2] = {               /* building texture coordinates */
    {0.0, 1.0},
    {1.0, 1.0},
    {1.0, 0.0},
    {0.0, 0.0},
};

/*
* Building texture and texture environment
*/

unsigned long bldgtex[8*4] = {
    0xffffffff, 0xffff0000, 0x00000000, 0x00000000,
    0xffffffff, 0xffff0000, 0x0000ffcf, 0xffcfffcf,
    0xffff0000, 0xffff0000, 0x0000ffcf, 0x0000ffcf,
    0xffffffff, 0xffffffff, 0xffffffcf, 0xffcfffcf,
    0xffff0000, 0xffffffff, 0xffffffcf, 0x0000ffcf,
    0xffffffff, 0xffffffff, 0xffffffcf, 0xffcfffcf,
    0xffff0000, 0xffff0000, 0x0000ffcf, 0x0000ffcf,
    0xffffffff, 0xffff0000, 0x0000ffcf, 0xffcfffcf,
};
```

```
float txlist[] = {TX_MAGFILTER, TX_POINT, TX_NULL};
float tvlist[] = {TV_NULL};

main()
{
    if (getgdesc(GD_BITS_NORM_ZBUFFER) == 0) {
        fprintf(stderr, "Z-buffer not available on this machine\n");
        return 1;
    }
    if (getgdesc(GD_TEXTURE) == 0) {
        fprintf(stderr, "Texture mapping not available on this machine\n");
        return 1;
    }
    if (getgdesc(GD_AFUNCTION) == 0) {
        fprintf(stderr, "afunction not available on this machine\n");
        return 1;
    }

    prefsize(400, 400);
    winopen("afunction");
    RGBmode();
    gconfig();
    mmode(MVIEWING);
    ortho(-20.0, 20.0, -20.0, 20.0, 10.0, 20.0);
    zbuffer(TRUE);
    czclear(0, getgdesc(GD_ZMAX));

 /*
 * Draw 2 mountains
 */
    cpack(0xff3f703f);
    bgnpolygon();
        v3f(mt0[0]);
        v3f(mt0[1]);
        v3f(mt0[2]);
    endpolygon();

    cpack(0xff234f00);
    bgnpolygon();
        v3f(mt1[0]);
        v3f(mt1[1]);
        v3f(mt1[2]);
    endpolygon();
```

```
 /*
 * Draw the building
 */
    texdef2d(1, 2, 8, 8, bldgtex, 0, txlist);
    tevdef(1, 0, tvlist);
    texbind(TX_TEXTURE_0, 1);
    tevbind(TV_ENV0, 1);
    afunction(0, AF_NOTEQUAL);
    cpack(0xffffffff);
    bgnpolygon();
        t2f(tbldg[0]);
        v3f(bldg[0]);
        t2f(tbldg[1]);
        v3f(bldg[1]);
        t2f(tbldg[2]);
        v3f(bldg[2]);
        t2f(tbldg[3]);
        v3f(bldg[3]);
    endpolygon();
    sleep(10);
    gexit();
    return 0;
}
```