Chapter 15

Antialiasing

This chapter describes methods for reducing display artifacts.

- Section 15.1, “Sampling Accuracy,” describes how to smooth scan conversion by enabling primitives to be drawn offset from pixel centers.
- Section 15.3, “One-Pass Antialiasing—the Smooth Primitives,” describes methods for antialiasing RGB images using a pixel-filling technique.
- Section 15.4, “Multipass Antialiasing with the Accumulation Buffer,” describes techniques for quickly generating antialiased points, lines, and polygons.
- Section 15.5, “Antialiasing on RealityEngine Systems,” describes the advanced multiple sampling feature of RealityEngine systems.

15.1 Sampling Accuracy

You may have noticed that lines displayed on a monitor appear to be made of a stairstep series of dots that make the line look jagged. Lines appear jagged because the true mathematical line is approximated by a series of points that are forced to lie on the pixel grid. Except in a few special cases (horizontal, vertical, and 45-degree lines) many of the approximating pixels are not on the mathematical line. Near-horizontal and near-vertical lines appear especially jagged, because their pixel approximations are a sequence of exactly horizontal or vertical segments, each offset one pixel from the next.

The jaggedness that you see is called *aliasing*, and techniques to eliminate or reduce aliasing are called *antialiasing*.

The following program, *jagged.c*, illustrates the aliasing problem.

/* Drag a color map aliased line with the cursor.*/

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

#define WINSIZE 400

Device devs[2] = {MOUSEX,MOUSEY};
float orig[2] = {100.,100.};

main()
{
    short val, vals[2];
    long xorg, yorg;
    float vert[2];

    prefsize(WINSIZE, WINSIZE);
    winopen("jagged");
    mmode(MVIEWING);
    ortho2(-0.5, WINSIZE-0.5, -0.5, WINSIZE-0.5);
    doublebuffer();
    gconfig();
    qdevice(ESCKEY);
    getorigin(&xorg, &yorg);

    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
        color(BLACK);
        clear();
        getdev(2,devs,vals);
        vert[0] = vals[0] - xorg;
        vert[1] = vals[1] - yorg;
        color(WHITE);
        bgnline();
            v2f(orig);
            v2f(vert);
        endline();
        swapbuffers();
    }
    gexit();
    return 0;
}
```

This example draws a line from the point (100,100) to the current cursor position. Move the cursor around, and notice how jagged the line appears,

especially when it is nearly horizontal or nearly vertical. Even at angles far from vertical or horizontal there is some jaggedness, although it is not as noticeable.

The line you see on the screen is aliased because it is composed of discrete pixels, each set either to the color of the line or to the background color. A much better approximation can be developed by considering the exact mathematical line to be the center line of a rectangle of width one, extending the full length of the line. A correct, antialiased sampling of this rectangle onto the pixel grid takes into account the fraction of each pixel that is obscured by the rectangle, rather than simply selecting the pixels most obscured by it. Each pixel is then set to a color between the color of the line and the color of the background, based on the fraction of the pixel that is obscured, or covered, by the line's rectangle.

To correctly sample a point, line, or polygon, the fraction of every pixel covered by the exact projection of the primitive must be computed, and that fraction must be used to determine the color of the resulting pixel. Because mathematical points and lines have no area, and therefore cannot be sampled, it is necessary to define a geometry to be used for their sampling. Points are best thought of as circles of diameter one, centered around the exact mathematical point. Lines are rectangles of width one, centered around the exact mathematical line.

Figure 15-1 shows an antialiased line.

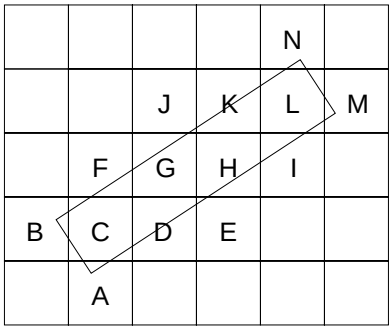

Figure 15-1 Antialiased Line

15.1.1 Subpixel Positioning

Vertices, after they have been transformed and projected to screen coordinates, are rounded to the nearest pixel center, rather than being treated with full precision. A prerequisite for accurate scan conversion of points, lines, and polygons is ensuring that their vertices are projected to the screen with *subpixel* precision. When you enable `subpixel()` mode, you defeat the default behavior of rounding projected vertices to the nearest pixel center. Exact vertex position is made available to the sampling hardware.

Use `subpixel()` to control the placement of point, line, and polygon vertices in screen coordinates:

```
void subpixel(Boolean b)
```

When `subpixel()` is FALSE, vertices are *snapped* (aligned) to the center of the nearest pixel after they have been transformed to screen coordinates. When `subpixel()` is TRUE, vertices are positioned with fractional precision.

The default setting for `subpixel()` depends on the system type. On IRIS Indigo Entry, XS, XS24, and Elan systems, polygons are always drawn with MicroPixel™ subpixel accuracy, regardless of the setting of `subpixel()`. On VGX, VGXT, SkyWriter, and RealityEngine systems, the default for `subpixel()` is FALSE, but performance is enhanced when subpixel positioning is enabled. It is thus a good idea to call `subpixel(TRUE)` when writing GL applications for these systems.

15.1.2 How Subpixel Positioning Affects Lines

In addition to its effect on vertex position, `subpixel()` also modifies the scan conversion of lines. Specifically, non-subpixel-positioned lines are drawn *closed*, meaning that connected line segments both draw the pixel at their shared vertex. subpixel-positioned lines are drawn *half open*, meaning that connected line segments share no pixels. Thus subpixel-positioned lines produce better results when `logicop` or `blendfunction()` are used, but will produce different, possibly undesirable results in 2-D applications where the endpoints of lines have been carefully placed.

For example, using the standard 2-D projection shown below, subpixel-positioned lines match non–subpixel-positioned lines pixel for pixel, except that they omit either the right-most or top-most pixel.

```
ortho2(left–0.5, right+0.5, bottom–0.5,top+0.5);
viewport (left,right,bottom,top);
```

Thus the non-subpixel-positioned line drawn from (0,0) to (0,2) fills pixels (0,0), (0,1), and (0,2), while the subpixel-positioned line drawn between the same coordinates fills only pixels (0,0) and (0,1).

On IRIS Indigo systems, subpixel-positioned lines are drawn closed rather that half-open. `subpixel()` is not supported by all models for all primitives. Refer to the `subpixel()` man page for details.

15.2 Blending

The pixel color value in the frame buffer is replaced with the incoming pixel color when pixels are drawn. When operating in RGB mode, however, it is possible to replace the color components of the frame buffer (destination) pixel with values that are a function of both the incoming (source) pixel color components and of the current frame buffer color components. This operation is called *blending*. Not all systems support blending. See the *blendfunction(3G)* man page for details.

The antialiasing techniques described in Section 15.3 require blending when operating in RGB mode. Blending has other uses, including drawing transparent objects, and compositing images. Blending is specified with the `blendfunction()` command:

```
void blendfunction (long sfactr,long dfactr);
```

The `blendfunction()` arguments *sfactr* and *dfactr* specify how destination pixels are computed, based on the incoming (source) pixel values and the current framebuffer values. The token specified for *sfactr* selects the blending factor used to scale the contribution from the source RGBA values. The token specified for *dfactr* selects the blending factor used to scale the contribution from the destination RGBA values.

Note: RealityEngine systems provide the ability to specify a constant color for blending. Use `blendcolor()` to set the values of the color components for the blending functions `BF_CA`, `BF_MCA`, `BF_CC`, and `BF_MCC`.

Blending factors *sfactr* and *dfactr* are chosen from the list of tokens in Table 15-1.

Token	Calculated Value	Notes
BF_ZERO	0.0	
BF_ONE	1.0	
BF_SA	Source Alpha / 255	
BF_MSA	1.0 – (source Alpha / 255)	
BF_DA	Source Alpha / 255	Requires alpha bitplanes
BF_MDA	1.0 – (source Alpha / 255)	Requires alpha bitplanes
BF_SC	Source RGBA / 255	*dfactr* only
BF_MSC	1.0 – (source RGBA / 255)	*dfactr* only
BF_DC	Destination RGBA / 255	*sfactr* only
BF_MDC	1.0 – (destination RGBA / 255)	*sfactr* only
BF_MIN_SA_MDA	Min (BF_SA, BF_MDA)	Requires alpha planes, changes expression
BF_CC	Constant RGBA / 255	RealityEngine only
BF_MCC	1-(constant RGBA / 255)	RealityEngine only
BF_CA	Constant alpha / 255	RealityEngine only
BF_MCA	1-(constant alpha / 255)	RealityEngine only
BF_MIN	Min(1, destination RGBA / source RGBA)	RealityEngine only
BF_MAX	Max(1, destination RGBA / source RGBA)	RealityEngine only

Table 15-1 Blending Factors

All the blending factors are defined to have values between 0.0 and 1.0 inclusive. In most cases, this range is obtained by dividing a color component value, in the range 0 through 255, by 255. The blending arithmetic is done such that a blending factor with a value of 1.0 has no effect on the color component that it weights. Also, a blending factor of 0.0 forces its corresponding color component to 0. Because the weighting factors are all positive, and because each weighted sum is clamped to 255, colors blend toward white, rather than wrapping back to low-intensity values.

Each frame buffer color component is replaced by a weighted sum of the current value and the incoming pixel value. This sum is clamped to a maximum of 255.

By default, *sfactr* is set to `BF_ONE` and *dfactr* is set to `BF_ZERO`, resulting in simple replacement of the framebuffer color components:

```
Rdestination = Rsource
Gdestination = Gsource
Bdestination = Bsource
Adestination = Asource
```

When `blendfunction()` is called with *sfactr* set to a value other than `BF_ONE`, or *dfactr* set to a value other than `BF_ZERO`, a more complex expression defines the frame buffer replacement algorithm.

```
Rdest = min (255, ((Rsource * sfactr) + (Rdest * dfactr)))
Gdest = min (255, ((Gsource * sfactr) + (Gdest * dfactr)))
Bdest = min (255, ((Bsource * sfactr) + (Bdest * dfactr)))
Adest = min (255, ((Asource * sfactr) + (Adest * dfactr)))
```

Blending factors `BF_DA`, `BF_MDA`, and `BF_MIN_SA_MDA` require alpha bitplanes for correct operation. Blending functions specified without using these three symbolic constants work correctly, regardless of the availability of alpha bitplanes.

Use the following command, testing for a nonzero return value, to determine if your machine has alpha bitplanes:

```
getgdesc(GD_BITS_NORM_SNG_ALPHA)
```

Blending factors BF_SC, BF_MSC, BF_DC, and MF_MDC weight each color component by the corresponding weight component. For example, you can scale each framebuffer color component by the incoming color component with the blending function:

```
blendfunction (BF_DC,BF_ZERO)
```

```
Rdestination = min (255, (Rsource * (Rdestination / 255)))
Gdestination = min (255, (Gsource * (Gdestination / 255)))
Bdestination = min (255, (Bsource * (Bdestination / 255)))
Adestination = min (255, (Asource * (Adestination / 255)))
```

The special blending factor BF_MIN_SA_MDA is intended to support polygon antialiasing, as described in Section 15.3.3. It must be used only for *sfactr*, and only while *dfactr* is BF_ONE. In this case, the blending equations are:

```
blendfunction (BF_MIN_SA_MDA,BF_ONE)
```

```
Rdestination = min (255, ((Rsource * sfactr) + Rdestination)
Gdestination = min (255, ((Gsource * sfactr) + Gdestination)
Bdestination = min (255, ((Bsource * sfactr) + Bdestination)
sfactr = min ((Asource/255), (1.0 - (Adestination/255)))
Adestination = sfactr + Adestination
```

This special blending function accumulates pixel contributions until the pixel is fully specified, then allows no further changes. Frame buffer alpha bitplanes, which must be present, store the accumulated contribution percentage, or “coverage”.

Although many blending functions are supported, the following function stands out as the single most useful one.

```
blendfunction (BF_SA,BF_MSA)
```

It weights incoming color components by the incoming alpha value, and frame buffer components by one minus the incoming alpha value. In other words, it blends between the incoming color and the frame buffer color, as a function of the incoming alpha. This function renders transparent objects by drawing them correctly when they are drawn in sorted order from farthest to nearest, specifying opacity as incoming alpha.

This sample program, *blendcircs.c*, illustrates image composition. Three colored circles are blended such that the first one is weighted by 0.5, the second by 0.35, and the third by 0.15. The blending function `blendfunction (BF_SA,BF_ONE)` is used, causing the colors to be added to each other, rather than blended. The order in which the circles are drawn makes no difference. Because the three weights add up to exactly 1.0, no clamping is done.

```
#include <stdio.h>
#include <gl/gl.h>
#define WINSIZE 400
#define RGB_BLACK 0x000000
#define RGB_RED 0x0000ff
#define RGB_GREEN 0x00ff00
#define RGB_BLUE 0xff0000
main()
{
    if (getgdesc(GD_BITS_NORM_SNG_RED) == 0) {
        fprintf(stderr, "Single buffered RGB not available\n");
        return 1;
    }
    if (getgdesc(GD_BLEND) == 0) {
        fprintf(stderr, "Blending not available\n");
        return 1;
    }
    prefsize(WINSIZE, WINSIZE);
    winopen("blendcircs");
    mmode(MVIEWING);
    RGBmode();
    gconfig();
    mmode(MVIEWING);
    ortho2(-1.0, 1.0, -1.0, 1.0);
    glcompat(GLC_OLDPOLYGON, 0);
    blendfunction(BF_SA, BF_ONE);
    cpack(RGB_BLACK);
    clear();
    cpack(0x80000000 | RGB_RED); /* red with alpha=128/255 */
    circf(0.25, 0.0, 0.7);
    sleep(2);
    cpack(0x4f000000 | RGB_GREEN); /* green with alpha=79/255 */
    circf(-0.25, 0.25, 0.7);
    sleep(2);
    cpack(0x30000000 | RGB_BLUE); /* blue with alpha=48/255 */
    circf(-0.25, -0.25, 0.7);
    sleep(10);
    gexit();
    return 0;
```

```
}
```

15.3 One-Pass Antialiasing—the Smooth Primitives

Aliasing artifacts are especially objectionable in image animations, because jaggies often introduce motion unrelated to the actual direction of motion of the primitives. The techniques described in this section improve the sampling quality of primitives without requiring that the primitives be drawn more than once. These techniques therefore perform well enough to animate complex scenes.

Note: Not all systems support smoothing and not every system supports every type of smooth primitive, so refer to the man pages for details about smoothing on different systems.

Modes are provided to support the drawing of antialiased points and lines in both color map and RGB modes. Because their interactions are more critical to the antialiasing quality, antialiased polygons are supported only in the more general RGB mode. If you are drawing an image composed entirely of points and/or lines, the routines in this section are always the right choice for antialiasing. If you include polygons in the image, you should consider both the techniques described in this section and the multipass accumulation technique described in Section 15.4.

15.3.1 High-Performance Antialiased Points—pntsmooth

By default, IRIS-4D Series systems sample points by selecting and drawing the pixel nearest the exact projection of the mathematical point. You can enable subpixel sampling and use `pntsmooth()` to draw antialiased points.

```
subpixel(TRUE);
pntsmooth(SMP_ON);
```

When you enable `subpixel()` mode, you defeat the default behavior of rounding projected vertices to the nearest pixel center. Exact point position is made available to the sampling hardware. By enabling `pntsmooth()` mode, you replace the default sampling of points with coverage sampling of a unit-diameter* circle centered around the exact mathematical point position. All that remains is instructing the system on how to use the computed pixel

coverage to blend between the background color and the point color at each pixel. This instruction differs based on whether the drawing is done in color map mode or in RGB mode.

When you enable `pntsmooth()` while in color map mode, the antialiasing hardware uses computed pixel coverage to replace the 4 least significant bits of the point's color. Therefore, for color map antialiased points to blend correctly, you must initialize a 16-entry colormap block (whose lowest entry location is a multiple of 16) to a ramp between the background color (lowest index) and the point color (highest index). Before drawing points, clear the background to the same color used as background in the colormap ramp.

When you draw a point with a color index in the range of the specified ramp, pixels in the area of the exact mathematical point are written with color indices that select ramp values based on the fraction of the pixel that is obscured by the point's unit-diameter circle. Because the sampling hardware modifies only the 4 least significant bits of the point's color, you can initialize and use multiple color ramps, each with a different point color, in the same image. Note that all ramps must blend to the same background color, which must be the color of the background used for the image.

This sample program, *pnt.cm.c*, illustrates the difference in image quality when you use `pntsmooth()` and `subpixel()` together to antialias color map points. The antialiased points and lines drawn by these example programs look better if you set gamma correction to 2.4, instead of the default value of 1.7.

```
/*
* Drag a string of color map antialiased points with the cursor.
* Disable antialiasing while the left mouse button is depressed.
* Disable subpixel positioning while the middle mouse button is depressed.
*/

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

#define WINSIZE 400
#define RAMPBASE 64  /* avoid the first 64 colors */
#define RAMPSIZE 16
```

---

* `pntsize()` and `pntsizef()` also affect antialiased points, but may not support the range of sizes supported by aliased points. See the *pntsize*(3G) man page for details.

```
#define RAMPSTEP (255 / (RAMPSIZE-1))
#define MAXPOINT 25

Device devs[2] = {MOUSEX,MOUSEY};
```

```
main()
{
    short val, vals[2];
    long i, xorg, yorg;
    float vert[2], x, y, interp;
    if (getgdesc(GD_PNTSMOOTH_CMODE) == 0) {
    fprintf(stderr, "Color map mode point antialiasing not available\n");
    return 1;
    }
    if (getgdesc(GD_BITS_NORM_DBL_CMODE) < 8) {
    fprintf(stderr, "Need 8 bitplanes in doublebuffer color map mode\n");
    return 1;
    }
    prefsize(WINSIZE, WINSIZE);
    winopen("pntsmooth.index");
    mmode(MVIEWING);
    ortho2(-0.5, WINSIZE-0.5, -0.5, WINSIZE-0.5);
    doublebuffer();
    gconfig();
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);
    getorigin(&xorg, &yorg);
    for (i = 0; i < RAMPSIZE; i++)
        mapcolor(i + RAMPBASE, i * RAMPSTEP, i * RAMPSTEP, i * RAMPSTEP);
    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
        color(RAMPBASE);
        clear();
        getdev(2,devs,vals);
        x = vals[0] - xorg;
        y = vals[1] - yorg;
        pntsmooth(getbutton(LEFTMOUSE) ? SMP_OFF : SMP_ON);
        subpixel(getbutton(MIDDLEMOUSE) ? FALSE : TRUE);
        color(RAMPBASE+RAMPSIZE-1);
        bgnpoint();
            for (i=0; i<=MAXPOINT; i++) {
                 interp = (float)i / (float)MAXPOINT;
                 vert[0] = 100.0 * interp + x * (1.0 - interp);
                 vert[1] = 100.0 * interp + y * (1.0 - interp);
                 v2f(vert);
            }
        endpoint();
        swapbuffers();
    }
    gexit();
    return 0;
```

```
}
```

Notice how smoothly the antialiased points move as you move the cursor. Now defeat the antialiasing by pressing the left mouse button, and notice that the points move less smoothly, and that they do not line up nearly as well as the antialiased points. The image quality degrades in exactly the same way when you defeat subpixel positioning by pressing the middle mouse button.

The antialiased points look good when they are not drawn touching each other. However, when you move the cursor near the lower-left corner of the window, causing the points to bunch together, the image quality again degrades. This is because pixels that are obscured by more than one point take as their value the color computed for the last point drawn. There is no general solution to the problem of overlapping primitives while drawing in color map mode.

The problem of overlapping primitives is handled well when antialiasing in RGB mode. When you enable `pntsmooth()` in RGB mode, the antialiasing hardware uses computed pixel coverage to scale the alpha value of the point's color. If the alpha value of the incoming point is 1.0, scaling it by the computed pixel coverage results in a pixel alpha value that is directly proportional to pixel coverage. For RGB antialiased points to draw correctly, set `blendfunction()` to merge new pixel color components into the frame buffer using the incoming alpha value.

This sample program, *pnt.rgb.c*, illustrates RGB mode point antialiasing.

```
/*
 * Drag a string of RGB antialiased points with the cursor.
 * Change from a merge-blend to an accumulate-blend when the left
 * mouse button is depressed.Use the "smoother" antialiasing sampling
 * algorithm when the middlemouse button is depressed.
 */

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

#define WINSIZE 400
#define MAXPOINT 25

Device devs[2] = {MOUSEX,MOUSEY};

main()
{
```

```
    short val, vals[2];
    long i, xorg, yorg;
    float vert[2], x, y, interp;

    if (getgdesc(GD_PNTSMOOTH_RGB) == 0) {
        fprintf(stderr, "RGB mode point antialiasing not available\n");
        return 1;
    }
    prefsize(WINSIZE, WINSIZE);
    winopen("pntsmooth.rgb");
    mmode(MVIEWING);
    ortho2(-0.5, WINSIZE-0.5, -0.5, WINSIZE-0.5);
    doublebuffer();
    RGBmode();
    gconfig();
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);
    getorigin(&xorg, &yorg);
    subpixel(TRUE);
    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
        cpack(0);
        clear();
        getdev(2,devs,vals);
        x = vals[0] - xorg;
        y = vals[1] - yorg;
        cpack(0xffffffff);
        blendfunction(BF_SA,getbutton(LEFTMOUSE) ? BF_ONE : BF_MSA);
        pntsmooth(getbutton(MIDDLEMOUSE)?(SMP_ON | SMP_SMOOTHER) : SMP_ON);
        bgnpoint();
            for (i=0; i<=MAXPOINT; i++) {
                interp = (float)i / (float)MAXPOINT;
                vert[0] = 100.0 * interp + x * (1.0 - interp);
                vert[1] = 100.0 * interp + y * (1.0 - interp);
                v2f(vert);
        }
        endpoint();
    swapbuffers();
    }
    gexit();
    return 0;
}
```

Unlike the color map antialiased points, the RGB antialiased points look good when they are bunched together. This is because RGB blending allows multiple points to contribute to a single pixel in a meaningful way.

In this demonstration a blend function that interpolates between the incoming and frame buffer color components, based on the incoming alpha, is used by default.

```
blendfunction(BF_SA,BF_MSA)
```

Press the left mouse button to switch to a blend function that simply accumulates color, again scaled by incoming alpha:

```
blendfunction(BF_SA,BF_ONE)
```

The difference between `blendfunction(BF_SA,BF_MSA)` and `blendfunction(BF_SA,BF_ONE)` is more apparent when you draw lines (see Section 15.3.2). In this demonstration, note that bunched points are a little brighter when you select the accumulating blending function.

You can switch from the standard antialiasing sampling algorithm to a "smoother" algorithm by pressing the middle mouse button. Not all systems support the higher-quality point sampling algorithm. Refer to the `pntsmooth()` manual page for details. This algorithm modifies more pixels per antialiased point than does the standard antialiasing algorithm. As a result, it produces slightly higher-quality antialiased points, at the price of slightly reduced performance. Set the "smoother" algorithm by calling

```
pntsmooth(SMP_ON | SMP_SMOOTHER);
```

Because RGB mode antialiased points are blended into the frame buffer, they can be drawn in any color and over any background. Unless you want to draw transparent, antialiased points, however, be sure to specify alpha as 1.0 when drawing antialiased RGB points.

15.3.2 High-Performance Antialiased Lines—linesmooth

By default, IRIS-4D Series systems sample lines by selecting and drawing the pixels nearest the projection of the mathematical line.You can enable subpixel sampling and use `linesmooth()` to draw antialiased lines.

```
subpixel(TRUE);
linesmooth(SML_ON);
```

By enabling `linesmooth()` mode, you replace the default sampling of lines with coverage sampling of a unit-width* rectangle centered around the exact mathematical line. All that remains is instructing the system on how to use the computed pixel coverage to blend between the background color and the line color at each pixel. This instruction differs based on whether the drawing is done in color map mode or in RGB mode.

When you enable `linesmooth()` while in color map mode, the antialiasing hardware uses computed pixel coverage to replace the 4 least significant bits of the line's color. Therefore, for color map antialiased lines to appear correct, you must initialize a 16-entry colormap block (whose lowest entry location is a multiple of 16) to a ramp between the background color (lowest index) and the line color (highest index). Before drawing lines, clear the background to the same color used as background in the color map ramp.

When you draw a line with a color index in the range of the specified ramp, pixels in the area of the exact mathematical line are written with color indices that select ramp values based on the fraction of the pixel that is obscured by the line's unit-width rectangle. Because the sampling hardware modifies only the 4 least significant bits of the line's color, you can initialize and use multiple color ramps, each with a different line color, in the same image. Note that all ramps must blend to the same background color, which must be the color of the background used for the image.

Note: To improve antialiasing performance on the IRIS Indigo, set `zsource(ZRC_COLOR)`.

---

* `linewidth()` and `linewidthf()` also affect antialiased lines, but may not support the range of sizes supported by aliased lines. See the *linewidth*(3G) man page for details.

This sample program, *line.cm.c*, illustrates the difference in image quality when you use `linesmooth()` and `subpixel()` together to antialias color map lines. The program draws a single straight line, made up of several individual line segments.

```
/*
 *Drag a string of color map antialiased line segments with the cursor.
 *Disable antialiasing while the left mouse button is depressed.
 *Disable subpixel positioning while the middle mouse button is depressed.
 */

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

#define WINSIZE 400
#define RAMPBASE 64 /* avoid the first 64 colors */
#define RAMPSIZE 16
#define RAMPSTEP (255 / (RAMPSIZE-1))
#define MAXVERTEX 10

Device devs[2] = {MOUSEX,MOUSEY};

main()
{
    short val, vals[2];
    long i, xorg, yorg;
    float vert[2], x, y, interp;

    if (getgdesc(GD_LINESMOOTH_CMODE) == 0) {
    fprintf(stderr, "Color map mode line antialiasing not available\n");
    return 1;
    }
    if (getgdesc(GD_BITS_NORM_DBL_CMODE) < 8) {
    fprintf(stderr, "Need 8 bitplanes in doublebuffer color map mode\n");
    return 1;
    }
    prefsize(WINSIZE, WINSIZE);
    winopen("linesmooth.index");
    mmode(MVIEWING);
    ortho2(-0.5, WINSIZE-0.5, -0.5, WINSIZE-0.5);
    doublebuffer();
    gconfig();
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);
```

```
    getorigin(&xorg, &yorg);
    for (i = 0; i < RAMPSIZE; i++)
        mapcolor(i + RAMPBASE, i * RAMPSTEP, i * RAMPSTEP, i * RAMPSTEP);

    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
        color(RAMPBASE);
        clear();
        getdev(2,devs,vals);
        x = vals[0] - xorg;
        y = vals[1] - yorg;
        linesmooth(getbutton(LEFTMOUSE) ? SML_OFF : SML_ON);
        subpixel(getbutton(MIDDLEMOUSE) ? FALSE : TRUE);
        color(RAMPBASE+RAMPSIZE-1);
        bgnline();
            for (i=0; i<=MAXVERTEX; i++) {
                interp = (float)i / (float)MAXVERTEX;
                vert[0] = 100.0 * interp + x * (1.0 - interp);
                 vert[1] = 100.0 * interp + y * (1.0 - interp);
                v2f(vert);
            }
        endline();
    swapbuffers();
    }
    gexit();
    return 0;
}
```

Notice how smooth the edges of the antialiased lines are, and how smoothly they move as you move the cursor. Now defeat the antialiasing by pressing the left mouse button, and notice that the lines become jagged. When you defeat subpixel positioning by pressing the middle mouse button, the individual line segments that make up the long line remain antialiased, but they no longer combine to form a single straight line. This is because the endpoints of the segments have been coerced to the nearest pixel centers, which are rarely on the exact mathematical line. Thus, you can antialias lines, unlike points, while `subpixel()` mode is `FALSE`. However, the image quality is still greatly enhanced when you enable subpixel positioning of vertices.

Like color map antialiased points, color map antialiased lines interact poorly when they intersect on the screen. The problem of overlapping primitives is handled well when antialiasing in RGB mode. When you enable `linesmooth()` in RGB mode, the antialiasing hardware uses computed pixel coverage to scale the alpha value of the line's color. If the alpha value of the incoming line is 1.0, scaling it by the computed pixel coverage results in a pixel alpha value that is directly proportional to pixel coverage.

For RGB antialiased lines to draw correctly, set `blendfunction()` to merge new pixel color components into the frame buffer using the incoming alpha value.

This sample program, *line.rgb.c*, illustrates RGB mode line antialiasing:

```
/*
 * Rotate a pinwheel of antialiased lines drawn in RGB mode.
 * Change to the "smoother" sampling function when the left mouse button
 * is depressed.
 * Change to the "end-corrected" sampling function when the middle mouse
 * button is depressed.
 * Change to a "color index like" blend function when the i-key is pressed.
 * Change from merge-blend to accumulate-blend when the a-key is depressed.
 * Disable subpixel positioning when the s-key is depressed.
 */

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

#define WINSIZE 400
#define MAXLINE 48
#define ROTANGLE (360.0 / MAXLINE)

float vert0[2] = {0.0,0.0};
float vert1[2] = {0.8,0.0};

main()
{
    int i;
    int smoothmode;
    short val;

    if (getgdesc(GD_LINESMOOTH_RGB) == 0) {
        fprintf(stderr, "RGB mode line antialiasing not available\n");
        return 1;
    }
    prefsize(WINSIZE, WINSIZE);
    winopen("linesmooth.rgb");
    mmode(MVIEWING);
    ortho2(-1.0,1.0,-1.0,1.0);
    doublebuffer();
    RGBmode();
    gconfig();
    qdevice(ESCKEY);
```

```
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);

    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
        cpack(0);
        clear();
        cpack(0xffffffff);
        smoothmode = SML_ON;
        if (getbutton(LEFTMOUSE))
            smoothmode |= SML_SMOOTHER;
        if (getbutton(MIDDLEMOUSE))
            smoothmode |= SML_END_CORRECT;
        linesmooth(smoothmode);
        if (getbutton(IKEY))
            blendfunction(BF_SA,BF_ZERO);
        else if (getbutton(AKEY))
            blendfunction(BF_SA,BF_ONE);
        else
            blendfunction(BF_SA,BF_MSA);
        subpixel(getbutton(SKEY) ? FALSE : TRUE);
        pushmatrix();
        rot(getvaluator(MOUSEX) / 25.0,'z');
        for (i=0; i<MAXLINE; i++) {
            bgnline();
                v2f(vert0);
                v2f(vert1);
            endline();
            rot(ROTANGLE,'z');
        }
        popmatrix();
        swapbuffers();
    }
    gexit();
    return 0;
}
```

Notice that the RGB mode antialiased lines look good where they intersect at the center of the pinwheel. This is because RGB blending allows multiple lines to contribute to a single pixel in a meaningful way. In this demonstration a blend function that interpolates between the incoming and frame buffer color components, based on the incoming alpha, is used by default:

```
blendfunction(BF_SA,BF_MSA)
```

You can switch to a blend function that simply accumulates color, again scaled by incoming alpha, by pressing the **<A>** key:

```
blendfunction(BF_SA,BF_ONE)
```

This blending function makes the slight noise at the center of the pinwheel disappear, because these pixels all accumulate and clamp at full brightness. This technique works well with white lines on a black background, but does not do well in other situations.

You can simulate the appearance of color map mode lines by pressing the **<I>** key, which forces a blending function that overwrites pixels:

```
blendfunction(BF_SA,BF_ZERO);
```

When you defeat subpixel positioning of line endpoints by pressing the **<S>** key, the pinwheel ceases to behave like a rigid object, and instead appears to wiggle and twist as it is rotated.

You can switch from the standard antialiasing sampling algorithm to a "smoother" algorithm by pressing the left mouse button. Not all systems support the higher-quality line sampling algorithm. Refer to the man page for details. This algorithm modifies more pixels per unit line length than does the standard antialiasing algorithm. As a result, it produces slightly higher-quality antialiased lines, at the price of slightly reduced performance. Set the "smoother" algorithm by calling `linesmooth (SML_ON | SML_SMOOTHER).`

Notice that when it is selected, lines at all angles appear to have the same width, and the "cloverleaf" pattern at the center of the pinwheel disappears. When you rotate the pinwheel with the left mouse button pressed, the only image artifact that remains is the sudden changing of line length observed at the ends of the lines. Press the middle mouse button to select a sampling algorithm that correctly samples line length as well as line cross-section. Invoke this "end-corrected" algorithm by calling `linesmooth(SML_ON | SML_END_CORRECT).`

When you select both "smoother" and "end-correct", the rotating pinwheel appears absolutely rigid, with even width lines and no jagged edges.

Because RGB antialiased lines are blended into the frame buffer, they can be drawn in any color over any background. Unless you want to draw transparent, anti-aliased lines, however, be sure to specify alpha as 1.0 when drawing antialiased RGB lines.

Note: Because RGB antialiased lines are blended, they interact well at intersections. However, when two RGB antialiased lines are drawn between the same vertices, the line quality is reduced noticeably. When the polygons in a standard geometric model are drawn as lines, either explicitly or using polymode, lines at the edges of adjacent polygons are drawn twice, and therefore do not antialias well in RGB mode. For best results, modify the database traversal so that edges of adjacent polygons are drawn only once.

15.3.3 High-Performance Antialiased Polygons—polysmooth

By default, IRIS-4D Series systems sample polygons by selecting and drawing the pixels whose exact center points are within the boundary described by the projection of the mathematical polygon edges.

You can enable subpixel sampling and use `polysmooth()` to draw antialiased polygons.

```
subpixel(TRUE);
polysmooth(PYSM_ON);
```

When you enable subpixel mode, you defeat the default behavior of rounding projected vertices to the nearest pixel center. Exact polygon vertex positions are made available to the sampling hardware. By enabling `polysmooth()` mode, you replace the default sampling of polygons with coverage sampling—the fraction of each pixel covered by the polygon is computed. All that remains is instructing the system how to use the computed pixel coverage to blend between the background color and the polygon color at each pixel. Because this blending operation is more critical for polygon antialiasing than it is for point or line antialiasing, polygon antialiasing is supported only in RGB mode, not in color map mode.

This sample program, *poly.rgb.c*, draws a single antialiased triangle:

```
/*
 * Rotate a single antialiased triangle drawn in RGB mode.
 * Disable antialiasing when the left mouse button is depressed.
 * Disable subpixel positioning when the middle mouse button is depressed.
 */

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

```
#define WINSIZE 400

float vert0[2] = {0.0,0.0};
float vert1[2] = {0.8,0.0};
float vert2[2] = {0.4,0.4};

main()
{
    short val;

    if (getgdesc(GD_POLYSMOOTH) == 0) {
        fprintf(stderr, "polygon antialiasing not available\n");
        return 1;
    }
    prefsize(WINSIZE, WINSIZE);
    winopen("polysmooth.rgb");
    mmode(MVIEWING);
    ortho2(-1.0,1.0,-1.0,1.0);
    doublebuffer();
    RGBmode();
    gconfig();
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);
    blendfunction(BF_SA,BF_MSA);
    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
        cpack(0);
        clear();
        cpack(0xffffffff);
        polysmooth(getbutton(LEFTMOUSE) ? PYSM_OFF : PYSM_ON);
        subpixel(getbutton(MIDDLEMOUSE) ? FALSE : TRUE);
        pushmatrix();
        rot(getvaluator(MOUSEX) / 25.0,'z');
        rot(getvaluator(MOUSEY) / 10.0,'x');
        bgnpolygon();
            v2f(vert0);
            v2f(vert1);
            v2f(vert2);
        endpolygon();
        popmatrix();
        swapbuffers();
    }
    gexit();
    return 0;
}
```

Move the cursor left and right to rotate the triangle, and note the smoothness of its edges. When you move the cursor toward the top of the screen, the triangle rotates away from you until it becomes perpendicular to your viewing direction. Note that when it is perpendicular, it disappears completely. This is because the projection of a triangle on edge covers no area on the screen, and therefore all pixel coverages are zero.

When you press the left mouse button, the triangle is drawn aliased. When you press the middle mouse button, the triangle vertices are no longer subpixel-positioned. Notice that the edges remain smooth, but that the triangle motion is no longer smooth, and the triangle no longer appears rigid.

This simple example of a single antialiased triangle drawn on a black background works correctly with the standard blending function:

```
blendfunction(BF_SA,BF_MSA)
```

However, when multiple antialiased triangles are drawn with adjacent edges, the standard blending function no longer produces good results.

This sample program, *poly2.rgb.c*, draws a bowtie-shaped object, constructed of four triangles in a planar mesh:

```
/*
 * Rotate a patch of antialiased triangles drawn in RGB mode.
 * Disable special polygon-blend when the left mouse button is depressed.
 * Disable subpixel positioning when the middle mouse button is depressed.
 */

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

#define WINSIZE 400

float vert0[2] = {0.0,0.0};
float vert1[2] = {0.0,0.4};
float vert2[2] = {0.4,0.1};
float vert3[2] = {0.4,0.3};
float vert4[2] = {0.8,0.0};
float vert5[2] = {0.8,0.4};
```

```
main()
{
 short val;

    if (getgdesc(GD_POLYSMOOTH) == 0) {
        fprintf(stderr, "polygon antialiasing not available\n");
        return 1;
    }
    prefsize(WINSIZE, WINSIZE);
    winopen("polysmooth2.rgb");
    mmode(MVIEWING);
    ortho(-1.0,1.0,-1.0,1.0,-1.0,1.0);
    doublebuffer();
    RGBmode();
    gconfig();
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);
    polysmooth(PYSM_ON);
    shademodel(FLAT);
    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
        cpack(0);
        clear();
        cpack(0xffffffff);
        if (getbutton(LEFTMOUSE))
            blendfunction(BF_SA,BF_MSA);
        else
            blendfunction(BF_MIN_SA_MDA,BF_ONE);
        subpixel(getbutton(MIDDLEMOUSE) ? FALSE : TRUE);
        pushmatrix();
        rot(getvaluator(MOUSEX) / 25.0,'z');
        rot(getvaluator(MOUSEY) / 10.0,'x');
        bgntmesh();
            v2f(vert0);
            v2f(vert1);
            v2f(vert2);
            v2f(vert3);
            v2f(vert4);
            v2f(vert5);
        endtmesh();
        popmatrix();
        swapbuffers();
     }
    gexit();
    return 0;
}
```

Notice that the internal edges of the four triangles that make up the bowtie are visible. Press the left mouse button, enabling the special polygon blending function, and note that these internal edges disappear. They are visible when you use the standard blending function because the standard blending function operates with uncorrelated coverages, such as those generated by antialiased points and lines.

Two adjacent polygons generate pixel coverages that are highly correlated—they always sum to 100% for pixels covered by the shared edge—and are therefore inappropriate for the standard blending function. Consider, for example, a pixel that is covered 60% by the first polygon that intersects it, and 40% by a second polygon adjacent to the first. Assuming white polygons and a black background, the first polygon raises the pixel intensity to 0.6, which is the correct value. However, the second polygon raises the pixel intensity to only 0.76, rather than to 1.0 as is desired. This is because the standard blending function assumes that the 60% and 40% coverages are uncorrelated, so 60% of the additional 40% is assumed to have been covered by the original 60%. Thus in uncorrelated coverage arithmetic, 60% plus 40% equals 76%, not 100%.

The special blending function `blendfunction(BF_MIN_SA_MDA,BF_ONE)` works with correlated coverages, the kind generated by antialiased polygon images. As the example code illustrated, the correlated blend does a good job with polygonal data. It is, however, much more difficult to use correlated blending than uncorrelated blending.

The requirements for using the correlated blending function are:

- You must have alpha bitplanes.
- You must draw polygons in order from the nearest to the farthest (not farthest to nearest as in the other antialiasing examples).
- You must not draw backfacing polygons (use `backface()`).
- The background color bitplanes, including the alpha bitplanes, must be cleared to zero before drawing starts.
- If the background is any color other than black, it must be filled as a polygon (i.e. not with a `clear()` command) after all polygons are drawn.
- You must draw all primitives (points, lines, and polygons) using the correlated blending function.

The correlated blending function works by accumulating pixel coverage in the frame buffer alpha bitplanes. The coverage granted each pixel write is limited by the total remaining at that pixel. When no coverage is left, additional writes to that pixel are ignored.

Because polygons must be drawn in depth-sorted order, you cannot use the *z*-buffer to eliminate hidden surfaces. Thus, polygon antialiasing, unlike point and line antialiasing, requires significant changes to the way the object data are traversed. It is therefore more difficult to use than are point and line antialiasing. If performance is not an absolute requirement, the accumulation buffer technique described in Section 15.4 is a better choice for polygon antialiasing.

This sample program, *poly3.rgb.c*, demonstrates correct polygon antialiasing of two cubes against a non-black background:

```
/*
 * Rotate two antialiased cubes in RGB mode.
 * Disable antialiasing by depressing the left mouse button.
 */

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

#define WINSIZE 400
#define SIZE  (0.2)
#define OFFSET(0.5)
#define CUBE0 OFFSET
#define CUBE1 (-OFFSET)

float vert0[4] = {-SIZE,-SIZE, SIZE};
float vert1[4] = { SIZE,-SIZE, SIZE};
float vert2[4] = {-SIZE, SIZE, SIZE};
float vert3[4] = { SIZE, SIZE, SIZE};
float vert4[4] = {-SIZE, SIZE,-SIZE};
float vert5[4] = { SIZE, SIZE,-SIZE};
float vert6[4] = {-SIZE,-SIZE,-SIZE};
float vert7[4] = { SIZE,-SIZE,-SIZE};

float cvert0[2] = {-1.0,-1.0};
float cvert1[2] = { 1.0,-1.0};
float cvert2[2] = { 1.0, 1.0};
float cvert3[2] = {-1.0, 1.0};
```

```
main()
{
    short val;
    float xang;

    if (getgdesc(GD_POLYSMOOTH) == 0) {
        fprintf(stderr, "polygon antialiasing not available\n");
        return 1;
 }
    prefsize(WINSIZE, WINSIZE);
    winopen("polysmooth3.rgb");
    mmode(MVIEWING);
    ortho(-1.0,1.0,-1.0,1.0,-1.0,1.0);
    doublebuffer();
    RGBmode();
    gconfig();
    device(ESCKEY);
    qdevice(LEFTMOUSE);
    blendfunction(BF_MIN_SA_MDA,BF_ONE);
    subpixel(TRUE);
    backface(TRUE);
    shademodel(FLAT);

    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
        cpack(0);
        clear();
        polysmooth(getbutton(LEFTMOUSE) ? PYSM_OFF : PYSM_ON);
        pushmatrix();
        xang = getvaluator(MOUSEY) / 5.0;
        rot(xang,'x');
        rot(getvaluator(MOUSEX) / 5.0,'z');
        if (xang < 90.0) {
            drawcube(CUBE0);
            drawcube(CUBE1);
        } else {
            drawcube(CUBE1);
            drawcube(CUBE0);
        }
        popmatrix();
        drawbackground();
        swapbuffers();
    }
    gexit();
    return 0;
}
```

```
drawcube(offset)
float offset;
{
   pushmatrix();
   translate(0.0,0.0,offset);
   bgntmesh();
      v3f(vert0);
      v3f(vert1);
      cpack(0xff0000ff);
      v3f(vert2);
      v3f(vert3);
      cpack(0xff00ff00);
      v3f(vert4);
      v3f(vert5);
      cpack(0xffff0000);
      v3f(vert6);
      v3f(vert7);
      cpack(0xff00ffff);
      v3f(vert0);
      v3f(vert1);
   endtmesh();
   bgntmesh();
      v3f(vert0);
      v3f(vert2);
      cpack(0xffff00ff);
      v3f(vert6);
      v3f(vert4);
   endtmesh();
   bgntmesh();
      v3f(vert1);
      v3f(vert7);
      cpack(0xffffff00);
      v3f(vert3);
      v3f(vert5);
   endtmesh();
   popmatrix();
   }

   drawbackground() {
      cpack(0xffffffff);
      bgnpolygon();
         v2f(cvert0);
         v2f(cvert1);
         v2f(cvert2);
         v2f(cvert3);
      endpolygon();
   }
```

Note that the nearer cube is drawn first, that cube faces are not sorted because back face elimination handles the sorting of convex solids, and that the background is drawn last as a single polygon.

When you press the left mouse button, antialiasing is disabled, but the correlated blend function remains enabled. Otherwise, the drawing order of the primitives would have to be changed.

15.4 Multipass Antialiasing with the Accumulation Buffer

This section describes techniques for computing pixel area coverage for various primitives, and using this coverage information to blend pixels into the framebuffer. This technique, called *accumulation*, is an iterative process that converges on a very accurate image. It easily handles all combinations of points, lines, and polygons, but it cannot typically be used to generate an interactive image. Accumulation also has applications in other advanced rendering techniques.

Accumulation is somewhat like blending, in that multiple images are composited to produce the final image. It differs from blending, however, in that its operation is completely separated from the rendering of a single frame. The accumulation buffer is an extended range bitplane bank in the normal frame buffer. You do not draw images into it; rather, images drawn in the front or back buffer of the normal frame buffer are added to the contents of the accumulation buffer after they are completely rendered.

15.4.1 Configuring the Accumulation Buffer

Before you can use the accumulation buffer, you must allocate bitplanes for it. `acsize()` specifies the number of bitplanes to be allocated for each color component in the accumulation buffer:

```
void acsize(long planes)
```

The number of bits that can be allocated to the accumulation buffer varies, depending on the system type. Sizes of 0 and 16 are used for most IRIS-4D systems, IRIS Indigo uses 32, and RealityEngine uses either a 12-bit unsigned or 24-bit signed accumulation buffer.

Color components in the accumulation buffer are signed values, so the range for each component depends on the size of the accumulation buffer. For example, a 16-bit accumulation buffer actually allocates 64 bitplanes, 16 each for red, green, blue, and alpha. You must call `gconfig()` after `acsize()` to activate the new specification.

15.4.2 Using the Accumulation Buffer

After bitplanes have been allocated for the accumulation buffer, you can use the `acbuf()` command to add the contents of the front or back bitplanes of the normal frame buffer to the accumulation buffer, and to return the accumulation buffer contents to either the front or back bitplanes. Call `acbuf()` only while the normal frame buffer is in RGB mode.

`acbuf()` operates on the accumulation buffer, which must already have been allocated using `acsize()` and `gconfig()`. When *op* is AC_CLEAR, each component of the accumulation buffer is set to value. When *op* is AC_ACCUMULATE, pixels are taken from the current `readsource()` (front, back, or *z*-buffer). Pixel components red, green, blue and alpha are each scaled by *value*, which must be in the range -256.0 *value* 256.0, and added to the current contents of the accumulations buffer.

Finally, when *op* is `AC_RETURN`, pixels are taken from the accumulation buffer. Each pixel component is scaled by *value*, which must be in the range 0.0 through 1.0, clamped to the integer range 0 through 255, and returned to the currently active drawing buffer (as specified by the `frontbuffer()`, `backbuffer()`, and `zdraw` commands). All special pixel operations—including z-buffer, blending function, logical operation, stenciling, and texture mapping—are ignored during this transfer. (These commands implement several other accumulation buffer operations. See the man pages for details on these operations.)

Accumulation buffer pixels map one-to-one with pixels in the window. All accumulation buffer operations affect the pixels within the viewport, limited by the screen mask and by the edges of the window itself. Like front, back, and z-buffer pixels, accumulation buffer pixels corresponding to window pixels that are obscured by another window, or are not on the screen, are undefined.

You can use the accumulation buffer to average many renderings of the same scene into one final image. By jittering the viewing frustum slightly for each

image, you can produce a single antialiased image as the result of many averaged images. For this to work, you must:

1. Completely render the image for each pass, including using the z-buffer to eliminate hidden surfaces, if appropriate.
2. Enable subpixel positioning of all primitives used (see `subpixel`).
3. Slightly perturb the viewing frustum before rendering each image. By slightly perturbing the projection transformation before rendering each image, you can effectively move the sample position in each pixel away from the pixel center. This is particularly easy to implement when you use an orthographic projection.

This sample program, *acbuf.rgb.c*, draws an antialiased circle using a 2-D orthographic projection:

```
/*
*Draw an antialiased circle using the accumulation buffer.
*Disable antialiasing when the left mouse button is depressed.
*Disable subpixel positioning when the middle mouse button is depressed.
*/

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

#defineWINSIZE100
#defineSAMPLES3
#defineDELTA(2.0/(WINSIZE*SAMPLES))

main()
{
    longx,y;
    shortval;

    if(getgdesc(GD_BITS_ACBUF)==0){
        fprintf(stderr,"accumulation buffer not available\n");
        return1;
}
    prefsize(WINSIZE,WINSIZE);
    winopen("acbuf.rgb");
    mmode(MVIEWING);
    glcompat(GLC_OLDPOLYGON,0);              /*point sample the circle*/
    doublebuffer();
    RGBmode();
    acsize(16);
```

```
    gconfig();
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);

    while(!(qtest() && qread(&val) == ESCKEY && val==0)){
        subpixel(getbutton(MIDDLEMOUSE) ? FALSE : TRUE);
        if(getbutton(LEFTMOUSE)){
        drawcirc(0.0,0.0);
        }else{
            acbuf(AC_CLEAR,0.0);
            for(x=0;x<SAMPLES;x++){
            for(y=0;y<SAMPLES;y++){
            drawcirc((x-(SAMPLES/2))*DELTA,(y-(SAMPLES/2))*DELTA);
            acbuf(AC_ACCUMULATE,1.0);
            }
        }
        acbuf(AC_RETURN,1.0/(SAMPLES*SAMPLES));
        }
        swapbuffers();
    }
    gexit();
    return0;
}
drawcirc(xdelta,ydelta)
    floatxdelta,ydelta;
    {
    ortho2(-1.0+xdelta,1.0+xdelta,-1.0+ydelta,1.0+ydelta);
    cpack(0);
    clear();
    cpack(0xffffffff);
    circf(0.0,0.0,0.8);
}
```

The circle is drawn nine times on a regular three-by-three grid. After the ninth accumulation, the resulting image is returned to the back buffer, and the buffers are swapped, making the antialiased circle visible. Note that the edges of the circle are smooth, and that when you press the left mouse button, the edges become jagged (aliased). Also note the reduction in image quality when you defeat subpixel positioning by pressing the middle mouse button.

Some other points about this sample program:

- The orthographic projection is perturbed by multiples of DELTA, a constant that is a function of the ratio of units in orthographic coordinates to window coordinates, and of the resolution of the subsampling.

  Note that you cannot use the viewport to jitter the sample points, both because the viewport is specified with integer coordinates, and because pixels near the viewport boundary would sample incorrectly.

- Old polygon mode is defeated. Otherwise the circle would be drawn with the old fill style, rather than the new point-sampled style. Point sampling and subpixel positioning are both required to use the accumulation buffer accurately.

- Each drawing pass clears the back buffer to black, then draws the circle. In general, all drawing operations (such as clearing and using the z-buffer) must be duplicated for each pass.

You can perform accumulation buffer antialiasing with perspective projections as well as orthographic projections. The following subroutines do all the arithmetic required to implement pixel jitter using the perspective and window projection calls:

```
#include <math.h>

void subpixwindow(left,right,bottom,top,near,far,pixdx,pixdy)
float left,right,bottom,top,near,far,pixdx,pixdy;

{
   short vleft,vright,vbottom,vtop;
   float xwsize,ywsize,dx,dy;
   int xpixels,ypixels;
   getviewport(&vleft,&vright,&vbottom,&vtop);
   xpixels = vright - vleft + 1;
   ypixels = vtop - vbottom + 1;
   xwsize = right - left;
   ywsize = top - bottom;
   dx = -pixdx * xwsize / xpixels;
   dy = -pixdy * ywsize / ypixels;
   window(left+dx,right+dx,bottom+dy,top+dy,near,far);
}

void subpixperspective(fovy,aspect,near,far,pixdx,pixdy)
Angle fovy;
```

```
float aspect, near, far, pixdx, pixdy;

{
    float fov2,left,right,bottom,top;
    fov2 = ((fovy*M_PI) / 1800) / 2.0;
    top = near / (fcos(fov2) / fsin(fov2));
    bottom = -top;
    right = top * aspect;
    left = -right;
subpixwindow(left,right,bottom,top,near,far,pixdx,pixdy);
}
```

In many applications, you can condition use of the accumulation buffer on user input. For example, when mouse position determines view angle, you can accumulate and display a progressively higher-quality antialiased image while the mouse is stationary. At any time during an antialiasing accumulation, the contents of the accumulation buffer represent a better image than the aliased image. You might choose a sample pattern that optimizes the intermediate results, then display each intermediate result, rather than waiting for the accumulation to be complete.

The antialiasing example implements a box filter—samples are evenly distributed inside a square pixel, and each sample has the same effect on the resulting image. Antialiasing filter quality improves when the samples are distributed in a circular pattern that is larger than a pixel, perhaps with a diameter of 0.75 pixels or so. You can further improve the filter quality by shaping it as a symmetric Gaussian function, either by changing the density of sample locations within the circle, or by keeping the sample density constant and assigning different weights to the samples. The weight of a sample is specified by *value* when you call `acbuf(AC_ACCUMULATE, value)`. A circularly symmetric Gaussian filter function yields smoother edges than does a unit-size box filter.

See */usr/people/4Dgifts/examples/acbuf* for examples of different filters.

Regardless of the filter function, fewer samples are required to achieve a given antialiasing quality level when the samples are distributed in a random fashion, rather than in regular rows and columns.

The accumulation buffer has many rendering applications other than antialiasing. For example, to limit depth of field, you can average images projected from slightly different viewpoints and directions. To produce motion blur, you can average images with moving objects rendered in different

locations along their trajectories. To implement a filter kernel, you can convolve images with `rectcopy()` and the accumulation buffer. Because the accumulation buffer operates on signed color components, and clamps these components to the display range of 0 through 255 when they are returned to the display buffer, you can implement filters with negative components in their kernels.

Additional details of the theory and use of the accumulation buffer, as well as example images, are available in "The Accumulation Buffer: Hardware Support for High-Quality Rendering," in *SIGGRAPH'90 Conference Proceedings*, Volume 24, Number 3, August 1990.

15.5 Antialiasing on RealityEngine Systems

This section discusses advanced features that are available only on systems with RealityEngine graphics, so you may want to skip to Chapter 16 if you do not have one of these systems.

The techniques discussed in the previous sections of this chapter suggest some different ways to reduce aliasing. RealityEngine graphics offers high performance on all the traditional antialiasing methods presented in the previous sections of this chapter. In addition, RealityEngine supports real-time antialiasing through the advanced feature of *multisampling*.

The scan conversion hardware in most graphics systems samples points, lines, and polygons with a single sample located at the center of each pixel. See Chapter 2 for more information on point-sampling.

When these single-sampled primitives are rendered, aliasing artifacts can appear. Aliasing occurs because the pixels that are only partially covered by the primitive do not get colored if the center of the pixel is not covered.

Not having enough sample points within the pixel to adequately determine the amount of pixel coverage is called *undersampling*, which results in an aliased image. A sufficient sampling method accounts for the areas of all the polygons that contribute to the shading of each pixel, rather than just a single sample point. That way, the pixel can be accurately shaded to a value that represents all polygons that are visible within that pixel.

15.5.1 Multisample Antialiasing

In single-pass multisample antialiasing, up to 16 subsamples can be evaluated at each pixel without repeatedly rendering the frame and accumulating the results. Multisampling provides for greater accuracy when rendering primitives while still maintaining a high level of geometry performance. Depth and stencil values are also evaluated and stored at each subsample, if those features are enabled.

Figure 15-2 shows example dot patterns for multisampling. Each dot represents a subsample within a single pixel. The example dot patterns shown here are *not* very efficient sampling locations. The actual sample patterns used by the hardware are efficient.

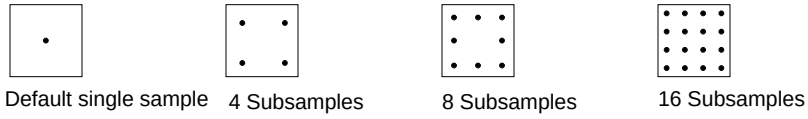

Figure 15-2 Example Multisample Patterns

15.5.2 Configuring Your System for Multisampling

In its default configuration, the standard framebuffer is configured to store a single value at each pixel. In multisampling, the multisample framebuffer is configured to store 0, 4, 8, or 16 subsamples for each pixel. The default multisample size is zero, corresponding to point-sampling.

Storage for multiple subsamples is allocated from graphics memory, as is storage for framebuffers and other features. Rather than limit the way you can use multisampling with other features by establishing fixed boundaries for memory partitions, RealityEngine allows a flexible configuration that lets you choose the combination of features that best suits your application needs.

Your RealityEngine system contains either 1, 2, or 4 Raster Manager (RM) boards. The base configuration contains one RM board. Each additional RM increases the pixel throughput, the memory for graphics features, and the amount of screen resolution available from a variety of user-selectable video formats. One additional RM lets you use either advanced graphics features such as multisampling, or lets you select a high-resolution video format. The

maximum configuration of four RMs provides both advanced graphics features and high screen resolution.

To set up multisampling, you must configure the multisample buffer with the number of samples to use. The combination of the number of RMs installed in your system, the depth of the color framebuffer (either 8 bits or 12 bits per component), the screen resolution, and the use of other features such as *z*-buffering, stereo buffering, or stenciling determines the number of samples available.

The framebuffer resolution that you select can affect the features that are available, such as the number of subsamples you can use, or whether color computations are performed at 8 or 12 bits per component. You can balance the trade-off between screen real estate and sample size to suit your needs.

Multisampling can be used only in RGB mode when the draw mode is `NORMALDRAW`. Because it is not possible to allocate a multisample buffer in color index mode, `multisample()` is always ignored in color index mode. When a multisample buffer is configured, multisampling is enabled by default.

Selecting the Number of Samples

Evaluating a greater number of samples provides more accuracy. You can select 0, 4, 8, or 16 samples per pixel. Use `mssize()` to configure the number of subsamples in the multisample buffer. `mssize()` takes three arguments:

samples	Number of subsamples to use, dependent on system configuration. Use either 0 (default single sampling), 4, 8, or 16.
zsise	Number of bits of z-buffer data to store at each subsample. Use either 0 or 32.
ssize	Number of bits of stencil data to store at each subsample. Use either 0, 1, or 8.

The GL allocates framebuffer memory when you request rendering modes. When you issue a `gconfig()`, the system attempts to honor all of your requests. If the system is unable to honor all of the requests, it may reduce the sample size or disable the use of other options such as the accumulation buffer or stereo buffering. For example, you may be granted the requested number of multisamples, but not a hardware accumulation buffer. In that case, a software accumulation buffer is substituted for the hardware accumulation buffer.

Another example is a request for 4 multisamples and stereo double buffering. If not enough resources are available to supply all of these requests, the multisample size request may be approved, but stereo buffering may be denied.

Figure 15-3 shows a conceptual diagram of how graphics memory, screen resolution, and framebuffer requests determine the configuration granted.

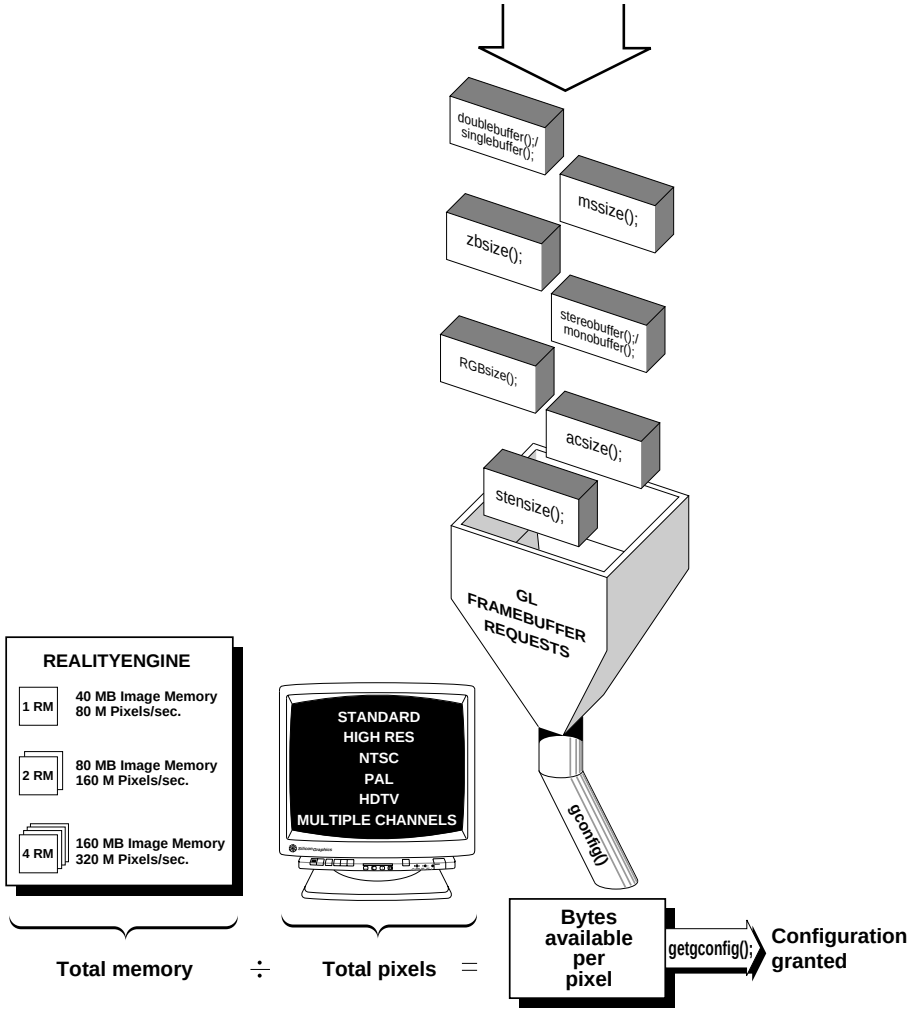

Figure 15-3 Flexible Framebuffer Configuration

The GL makes reasonable assumptions about the priority of the requests when it allocates memory.

You can query the system for the resources actually received. Use `getgconfig()` to read back the framebuffer configuration. `getgconfig()` returns the configured size of a buffer in the current drawmode.

Programs should be coded to request their most desirable configuration first, then back off and try configurations that are less stringent, but still acceptable.

The following sample code fragment illustrates this heuristic approach to configuring the RealityEngine framebuffer.

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

main()
{
   winopen("Request");

   /*this code requests 8 multisamples, but is willing to accept 4 or none*/
   RGBmode();              /* RGB mode */
   doublebuffer();         /* double buffering */
   zbsize(0);              /* deallocate main zbuffer */
   mssize(8,32,0);          /* 8 multisamples with zbuffer */
   gconfig();

   /* check for at least 4 multisamples */
   if (getgconfig(GC_MS_SAMPLES) < 4)
   {
      /* did not get enough multisamples */
      mssize(0,0,0);   /* remove multisample request */
      zbsize(32);      /* restore main zbuffer */
      gconfig();
   }
}
```

Note: As illustrated in the sample code, be sure to restore the *z*-buffer if multisampling is not used.

Freeing Standard Framebuffer Memory

If an `mssize()` request is honored, the configuration requests for the z-buffer and stenciling apply to the multisample buffer rather than the standard framebuffer. However, memory is *not* automatically deallocated from the

standard framebuffer memory for z-buffer and stencil usage. Therefore, you should free this memory by setting the standard z-buffer size, as well as the standard stencil size, to zero. The default stencil size is already zero unless you have changed it, but the default z-buffer size is 32 bits.

Note: Unless you need the standard z-buffer in addition to the multisample buffer, deallocate memory from the standard z-buffer and stencil planes when using multisampling. The default 32-bit z-buffer on the main framebuffer uses up valuable memory that may be needed for multisampling. If multisampling is not going to be used—perhaps because a multisample request was requested, but not granted—remember to restore the z-buffer size to 32 bits.

Use `zbsize()` and `stensize()` to allocate / deallocate z-buffer and stencil memory from the standard framebuffer.

`zbsize(`*n*`)`	specifies the number of z-buffer bits allocated for the standard framebuffer. When using multisampling, no z-buffer data is stored with the standard framebuffer sample.
`stensize(`*n*`)`	specifies the number of stencil bits allocated for the standard framebuffer. When using multisampling, no stencil data is stored with the standard framebuffer sample.

You can use `RGBsize()` to reduce the color resolution in order to gain more memory for framebuffer requests. Specify the number of bits to be allocated per component. RealityEngine supports either 8 or 12 bits per component.

Summary

The following list summarizes the steps involved in using multisampling:

1. Set the drawing and color modes for normal drawing and RGB color:

   `drawmode(NORMALDRAW);` (default)

   `RGBmode();`

2. Specify the other framebuffer modes:

   `singlebuffer()` (default) or `doublebuffer();`

   `monobuffer()` (default) or `stereobuffer();`

3. Specify the number of samples:

   `mssize(`*`samples`*`,`*`zsize`*`,`*`ssize`*`);`

4. Deallocate z-buffer and stencil bits from the main framebuffer:

   `stensize(0);`

   `zbsize(0)`

5. Configure the GL:

   `gconfig();`

6. Enable multisampling:

   `multisample(TRUE);` (default)

Note: `multisample()` can be called at any time during the rendering of a scene, except between bgn/end calls. Mixing multisampling and default sampling in the same scene is permitted but not recommended. The reason for this is that while the color information in the standard framebuffer is continuously updated with multisample data, the standard z-buffer is not. Therefore, z data updated at subsample locations is lost while multisampling is turned off.

15.5.3 How Multisampling Affects Rendering

The multisample buffer is a part of the color framebuffer. `multisample()` should be called only while drawmode is `NORMALDRAW`. Multisampling affects the results of rendering when multisampling is enabled (`TRUE`) and when the draw mode is `NORMALDRAW`.

When multisample is true, rendered primitives directly affect the samples in the multisample buffer. Immediately after the multisample locations at a pixel are modified, the front and/or back framebuffer colors are written with the "average" value of the multisample color values. No change is made to the standard z-buffer or stencil buffer that may be associated with the color buffers, and their contents do not affect rendering operations.

The framebuffer modes of alpha test, blending, dithering, and writemask affect the modification of the individual subsamples, and have no effect on the transfer of the average color value to the front or back color buffers. Conversely, buffer enables `frontbuffer()`, `backbuffer()`, `leftbuffer()`,

and `rightbuffer()` have no effect on the modification of the individual multisample locations; they affect only the transfer of the average color.

There are no special clear commands for the multisample buffer. Rather, the standard `clear()`, `zclear()`, `sclear()`, and `czclear()` commands affect the multisample buffer much as they affect the standard color, stencil, and *z* buffers. Clear modifies the enabled color buffers, and always modifies the color portion of each multisample location. `zclear()` operates on all multisample *z* locations. `sclear()` operates on all multisample stencil locations. `czclear()` behaves like `clear()` / `zclear()`, except that the z value is specified.

When `multisample()` is false, `polysmooth()`, `linesmooth()`, and `pntsmooth()` can be used with no performance degradation. However, unlike smooth primitives, multisampling does antialias geometry at intersection and interpenetration points.

Note: Antialiased primitives using the smooth rendering modes `pntsmooth()`, `linesmooth()`, and `polysmooth()` are superceded by multisampling— that is, these modes are undefined when multisampling is on. However, `blendfunction()` is still operational, which can adversely affect rendering performance in multisample mode. Therefore, for the best performance when multisampling, you should turn `blendfunction()` off when not performing blending. Don't use `blendfunction()` for antialiasing along with multisampling. This is especially applicable when porting previously developed code that used smooth primitives and blending for antialiasing to RealityEngine.

Multisampled Points

Until circles are implemented, points are sampled into the multisample buffer as squares centered on the exact point location.

Multisampled Lines

Lines are sampled into the multisample buffer as rectangles centered on the exact zero-area segment. The rectangle width is equal to the current linewidth. Its length is exactly equal to the length of the segment. The rectangles of colinear, abutting line segments abut exactly, so no subsamples are missed or drawn twice near the shared vertex.

Multisampled Polygons

Polygons are sampled into the multisample buffer much as they are into the standard single-sample buffer. A single color value is computed for the entire pixel, regardless of the number of subsamples at that pixel. Each multisample location is then written with this color if and only if it is geometrically within the exact polygon boundary.

If the z-buffer is enabled, the correct depth value at each multisample location is computed and used to determine whether that sample should be written or not. If stencil is enabled, the test is performed at each multisample location.

Polygon pattern bits apply equally to all multisample locations at a pixel. All sample locations are considered for modification if the pattern bit is 1. None are considered if the pattern bit is zero.

Pixels are sampled into the multisample buffer by treating each pixel as an `xzoom` × `yzoom` square, which is then sampled just like a polygon.

15.5.4 Using Advanced Multisample Options

You can generate special effects with advanced multisample options, which generally apply to a very specific application. Example applications include:

- Using the accumulation buffer with multisampled images.
- Using a multisample mask to select writeable sample locations.
- Using alpha values to feather-blend texture edges.

Accumulating Multisampled Images

The accumulation buffer averages several samples obtained by storing multiple renderings of the same primitive, each with the primitive offset by a specific amount, a technique known as *jittering*.

Just as the default single-sample location is the center of each pixel, there are default locations for the multiple sample points, located in a cluster surrounding the center of each pixel. The default locations are chosen to produce optimum rendering quality for single-pass rendering.

Superlative image quality can be achieved when several multisampled images are composited using the accumulation buffer. Each rendering pass should use a slightly different sample pattern. These patterns have been selected to produce optimum rendering quality for the corresponding number of passes.

Accumulating multisample results can also extend the capabilities of your system. For example, if you have only enough resources to allow 4 subsamples, but you are willing to render the image twice, you can achieve the same effect as multisampling with 8 subsamples.

Use `mspattern()` to select the sample pattern. Select the sample pattern for `mspattern()` according to the number of rendering passes to accumulate. Table 15-2 lists the tokens for selecting accumulation multisample patterns.

Pattern Token	Purpose
`MSP_DEFAULT`	Default multisample pattern
`MSP_2PASS_0`	First pass of a 2-pass accumulation
`MSP_2PASS_1`	Second pass of a 2-pass accumulation
`MSP_4PASS_0`	First pass of a 4-pass accumulation
`MSP_4PASS_1`	Second pass of a 4-pass accumulation
`MSP_4PASS_2`	Third pass of a 4-pass accumulation
`MSP_4PASS_3`	Fourth pass of a 4-pass accumulation

Table 15-2 Tokens for Selecting Accumulation Multisample Patterns

The pattern should be changed only between complete rendering passes. It should not be changed between the time `clear()`/`czclear()` is called and the time that the rendered image is complete.

The following example configures the framebuffer with both a multisample buffer and an accumulation buffer for a 2-pass rendering:

```
RGBmode();
doublebuffer();
acsize(12);
mssize(4,32,0);
zbsize(0);
gconfig();
lsetdepth(getgdesc(GD_ZMAX),getgdesc(GD_ZMIN));
zfunction(ZF_GEQUAL);
```

```
zbuffer(TRUE);
multisample(TRUE);
mspattern(MSP_2PASS_0);
czclear(0,0);
/* draw the scene */
acbuf(AC_CLEAR_ACCUMULATE,1.0);
mspattern(MSP_2PASS_1);
czclear(0,0);
/* draw the scene again */
acbuf(AC_ACCUMULATE,1.0);
acbuf(AC_RETURN,0.5);
swapbuffers();
```

To maintain greater precision in the accumulation buffer, substitute the following values in the `acbuf()` commands:

```
acbuf(AC_CLEAR_ACCUMULATE,2.0);
acbuf(AC_ACCUMULATE,2.0);
acbuf(AC_RETURN,0.25);
```

or, for even greater precision, use the following values:

```
acbuf(AC_CLEAR_ACCUMULATE,4.0);
acbuf(AC_ACCUMULATE,4.0);
acbuf(AC_RETURN,0.0625);
```

This is because only 8 bits out of 12 are accumulated when using a value of 1.0, 9 bits are accumulated when using a value of 2.0, and so on—up to a maximum of 12 bits with acsize(*n*).

Using a Multisample Mask

You can use a mask to specify a subset of multisample locations to be written at a pixel. This feature is useful for implementing fade-level-of-detail in visual simulation applications. Multisample masks can be used to perform the blending from one model to the next by rendering the additional data in the detail model using a steadily increasing percentage of subsamples as the viewpoint nears the object.

The mask specifies the ratio of writable and non-writable locations at each pixel. However, it does not single out specific locations for writing. Mask values range from 0 to 1, where 0 indicates that no locations are to be written and 1 indicates that all sample locations are to be written.

You can also set a Boolean to create the inverse mask. For example, `msmask(0.75, FALSE)` will generate a mask that allows 75% of the samples to be written, and `msmask(0.75, TRUE)` will generate a mask that allows the other 25% of the samples to be written.

Using Alpha and Color Blending Options

Multisampling can be used to solve the problem of blurred edges on textures with irregular edges, such as trees, that require extreme magnification. When the texture is magnified, the edges of the tree look artificial, as if the tree is a paper cutout. You can feather the edges to make them look more natural by including alpha in the multisample mask.

By using `msalpha()` and `afunction()` together, you can represent objects such as trees, bridges, and fences with pictures of those objects superimposed on top of a simple rectangle polygon. The see-through effect is achieved by enabling alpha blending and `afunction()` to ignore the area of the polygon not covered by the texture. See Chapter 18 to learn more about using textures.

You can use `msalpha()` to specify whether you want alpha values to be included in the multisample mask.

When `msalpha` is set to `MSA_MASK` or `MSA_MASK_ONE` while multisampling is enabled, alpha values generated by rasterization are converted to multisample masks immediately prior to the per-pixel alpha test. At each pixel, the resulting multisample mask is logically ANDed with the mask specified by `msmask`. Only multisample locations enabled by the resulting mask are considered for modification.

If `msalpha` is set to `MSA_MASK_ONE`, the alpha value presented to alpha test and to each multisample location is the maximum value supported by the framebuffer configuration, effectively 1.0.

When `msalpha` is set to `MSA_ALPHA`, no change is made to the alpha values generated by rasterization or to the mask specified by `msmask()`.

None of the multisample options— `msmask()`, `msalpha()` or `mspattern()`–has any effect when `multisample` is FALSE, but they are maintained for potential future use.