Miscellaneous OpenGL Extensions

This chapter explains how to use several extensions that are not easily grouped with texturing, imaging, or GLX extensions, providing example code as needed. You learn about:

The Polygon Offset Extension

The polygon offset extension, EXT_polygon_offset, is useful for rendering hidden-line images, rendering solids with highlighted edges, and applying decals to surfaces.

The extension is useful, for example, when you want to render outlines on surfaces. The lines are not rasterized the same way the polygons are. This results in slightly different depth values which, when used by the depth buffering mechanism, cause some of the line pixels to show, and some not to show. This is because triangles and lines produce different depth values and the system cannot tell ahead of time what will be in front. The extension offsets polygon pixels in the depth buffer so that lines and points no longer interact in that way, resulting in clean polygon outlines.

Using the extension, you can displace the depth values of fragments generated by rendering polygons by a fixed bias plus an amount. The amount is proportional to the maximum absolute value of the depth slope of the polygon, measured and applied in window coordinates. This displacement

- allows lines (or points) and polygons in the same plane to be rendered without interaction: the lines are rendered either completely in front of or behind the polygons (depending on the sign of the offset factor)
- allows multiple coplanar polygons to be rendered without interaction, if different offset factors are used for each polygon

Polygon Offset Example Program

This section illustrates how to use the polygon offset extension by providing code fragments from an example program *hiddenline.c* that displays three images:

- The left image shows an ordinary wireframe image.
- The center image shows a hidden-line image without polygon offset.
- The right image shows a hidden-line image using polygon offset to reduce artifacts.

Figure 8-1 shows the output that the program generates. You are encouraged to run the program yourself to see the difference more clearly than the printed version can show.

Example 8-1 Polygon Offset Example Program

```
/*********************************************************************
 * Uses PolygonOffset to draw hidden-line images. PolygonOffset
 * shifts z values of polygons by an amount proportional to their slope
 * in screen z. This keeps the lines, which are drawn without
 * displacement, from interacting with their respective polygons, and
 * and thus eliminates line dropouts.
 */

#include <GL/glx.h>
#include <GL/glu.h>
#include <X11/keysym.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAXQUAD 6

typedef float Vertex[3];

typedef Vertex Quad[4];

Quad quads[MAXQUAD] = {     /*data to define six faces of a unit cube */
    0,0,0, 1,0,0, 1,1,0, 0,1,0,
    0,0,1, 1,0,1, 1,1,1, 0,1,1,
    0,0,0, 1,0,0, 1,0,1, 0,0,1,
    0,1,0, 1,1,0, 1,1,1, 0,1,1,
    0,0,0, 0,0,1, 0,1,1, 0,1,0,
    1,0,0, 1,0,1, 1,1,1, 1,1,0
};
```

```
#define WIREFRAME       0
#define HIDDEN_LINE     1

/*** function prototypes go here  ***/
static int dimension = 3;

main(int argc, char** argv) {
    Display *dpy;
    XVisualInfo *vi;
    XSetWindowAttributes swa;
    Window win;
    GLXContext cx;

    /*** X Window System setup goes here ***/

    /* check for the polygon offset extension */
    if (!query_extension("GL_EXT_polygon_offset"))
        error(argv[0], "polygon_offset extension is not available");

    /* set up viewing parameters */
    glMatrixMode(GL_PROJECTION);
    gluPerspective(20, 1, 0.1, 20);
    glMatrixMode(GL_MODELVIEW);
    glTranslatef(0, 0, -15);

    /* set other relevant state information */
    glEnable(GL_DEPTH_TEST);
    glPolygonOffsetEXT(1.5, 0.000001);

    /* process events until the user presses ESC */
    while (1) process_input(dpy, win);
}

static void
draw_scene(int mx, int my) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix();
    glTranslatef(-1.7, 0.0, 0.0);
    cubes(mx, my, WIREFRAME);
    glPopMatrix();

    glPushMatrix();
    cubes(mx, my, HIDDEN_LINE);
    glPopMatrix();
```

```
    glPushMatrix();
    glTranslatef(1.7, 0.0, 0.0);
    glEnable(GL_POLYGON_OFFSET_EXT);
    cubes(mx, my, HIDDEN_LINE);
    glDisable(GL_POLYGON_OFFSET_EXT);
    glPopMatrix();
}

static void
cubes(int mx, int my, int mode) {
    int x, y, z, i;

    /* track the mouse */
    glRotatef(mx / 2.0, 0, 1, 0);
    glRotatef(my / 2.0, 1, 0, 0);

    /* draw the lines as hidden polygons */
    glTranslatef(-0.5, -0.5, -0.5);
    glScalef(1.0/dimension, 1.0/dimension, 1.0/dimension);
    for (z = 0; z < dimension; z++) {
        for (y = 0; y < dimension; y++) {
            for (x = 0; x < dimension; x++) {
                glPushMatrix();
                glTranslatef(x, y, z);
                glScalef(0.8, 0.8, 0.8);
                for (i = 0; i < MAXQUAD; i++)
                    draw_hidden(quads[i], mode);
                glPopMatrix();
            }
        }
    }
}

static void
fill(Quad quad) {
    glBegin(GL_QUADS);
    glVertex3fv(quad[0]);
    glVertex3fv(quad[1]);
    glVertex3fv(quad[2]);
    glVertex3fv(quad[3]);
    glEnd();
}
```

```
static void
outline(Quad quad) {
    /* draw an outlined polygon */
    glBegin(GL_LINE_LOOP);
    glVertex3fv(quad[0]);
    glVertex3fv(quad[1]);
    glVertex3fv(quad[2]);
    glVertex3fv(quad[3]);
    glEnd();
}

static void
draw_hidden(Quad quad, int mode) {
    /* draw outline using white, optionally fill interior with black */
    glColor3f(1, 1, 1);
    outline(quad);

    if (mode == HIDDEN_LINE) {
        glColor3f(0, 0, 0);
        fill(quad);
    }
}

static void
process_input(Display *dpy, Window win) {
    XEvent event;
    static int prevx, prevy;
    static int deltax = 90, deltay = 40;

    do {
        char buf[31];
        KeySym keysym;

        XNextEvent(dpy, &event);
        switch(event.type) {
        case Expose:
            break;
        case ConfigureNotify:
            glViewport(0, 0, e+vent.xconfigurewidth,
                       event.xconfigureheigth);
            break;
        case ButtonPress:
            prevx = event.xbutton.x;
            prevy = event.xbutton.y;
            break;
```

```
            case MotionNotify:
                deltax += (event.xbutton.x - prevx); prevx =
                            event.xbutton.x;
                deltay += (event.xbutton.y - prevy); prevy =
                            event.xbutton.y;
                break;
            default:
                break;
            }
        } while (XPending(dpy));

        draw_scene(deltax, deltay);
        glXSwapBuffers(dpy, win);
    }
```

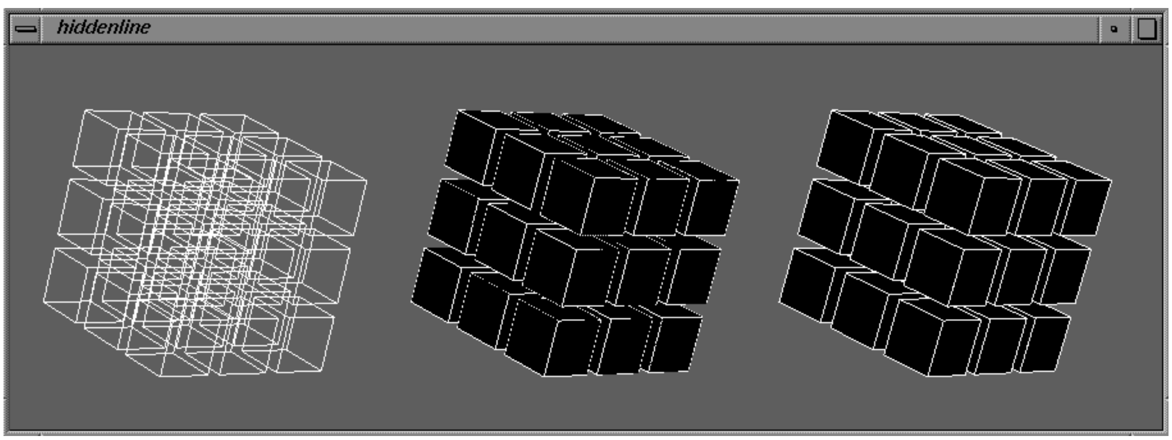

Figure 8-1 Polygon Offset Extension Example

New Functions

glPolygonOffsetEXT().

The Vertex Array Extension

The vertex array extension, EXT_vertex_array, makes it possible to specify multiple geometric primitives with very few subroutine calls. This section first discusses why and when you might want to use the extension, then looks at how the extension works, including a discussion of static and nonstatic arrays, and how to use it.

Why and When to Use the Vertex Array Extension

Instead of calling an OpenGL procedure to pass each individual vertex, normal, or color, you can specify separate arrays of vertexes, normals, and colors, and use them to define a sequence of primitives (all of the same type) when a single call is made to **glDrawArraysEXT()**.

The extension defines a stride mechanism that lets an application choose to keep all data staggered in a single array. For example, you could combine color, vertex, and edge data and access each type as needed using the stride number. This extension also supports the rendering of individual array elements, each specified as an index into the enabled arrays.

Note: Using the vertex array extension may lead to performance improvements or performance deterioration. While using the extension reduces the number of function calls your application has to make, there is added overhead for accessing the data in the array. The break-even point differs depending on the system the application is running on. See "Optimizing Use of the Vertex Array Extension" on page 284 for information on the accelerated cases on RealityEngine systems.

On Silicon Graphics systems, single-array storage provides better performance. The extension has been optimized for floats but not for other data types. You should run some performance tests to determine whether using vertex arrays is really appropriate for your application.

How the Vertex Array Extension Works

The extension maintains individual array pointers and associated data for an array of elements: vertexes, normals, colors, color indexes, texture coordinates, and edge flags. The data associated with each array specify the following:

- the data type of the values in the array

- the number of values per element in the array (for example, vertexes of 2, 3, or 4 coordinates)
- the byte stride from one array element to the next
- the number of elements (counting from the first) that are static

  Static elements may be modified by the application, but once they are modified, the application must explicitly respecify the array before using it for rendering.

When you specify an array, the pointer and associated data are saved as client-side state, and static elements may be cached. You define an element as static, that is, guaranteed not to change, using the *count* argument to **glVertexPointerEXT().** Non-static (dynamic) elements are never accessed until a call to **glArrayElementEXT()** or **glDrawArraysEXT().**

Specifying Arrays for Different Kinds of Data

This section provides information about specifying arrays for vertices. A full description for normals, colors, color indexes, texture coordinates, and edge flags is not included; see the reference pages for more information.

- **glVertexPointerEXT()** specifies the location and data format of an array of vertex coordinates.
  - *pointer* specifies a pointer to the first coordinate of the first vertex in the array.
  - *type* specifies the data type of each coordinate in the array, and must be one of GL_SHORT, GL_INT, GL_FLOAT, or GL_DOUBLE_EXT, implying OpenGL data types short, int, float, and double, respectively.
  - *size* specifies the number of coordinates per vertex, and must be 2, 3, or 4.
  - *stride* specifies the byte offset between pointers to consecutive vertexes. If *stride* is zero, the vertex data are tightly packed in the array.
  - *count* specifies the number of vertexes, counting from the first, that are static.
- **glNormalPointerEXT()** specifies the location and data format of an array of normals.
- **glColorPointerEXT()** specifies the location and data format of an array of color components.
- **glIndexPointerEXT()** specifies the location and data format of an array of color indexes.

- **glTexCoordPointerEXT()** specifies the location and data format of an array of texture coordinates.
- **glEdgeFlagPointerEXT()** specifies the location and data format of an array of boolean edge flags.

Rendering the Arrays

By default all the arrays are disabled; they are not accessed when either **glArrayElementEXT()** or **glDrawArraysEXT()** is called. To enable or disable an individual array, call **glEnable()** or **glDisable()** with *cap* set to an appropriate value, as specified in Table 8-1.

Table 8-1 Tokens for Enabling Arrays

Array Specification Command	Enable Token
glVertexPointerEXT	GL_VERTEX_ARRAY_EXT
glNormalPointerEXT	GL_NORMAL_ARRAY_EXT
glColorPointerEXT	GL_COLOR_ARRAY_EXT
glIndexPointerEXT	GL_INDEX_ARRAY_EXT
glTexCoordPointerEXT	GL_TEXTURE_COORD_ARRAY_EXT
glEdgeFlagPointerEXT	GL_EDGE_FLAG_ARRAY_EXT

When **glArrayElementEXT()** is called, a single vertex is drawn, using vertex and attribute data taken from location *i* of the enabled arrays.

Even if GL_VERTEX_ARRAY_EXT is not enabled, the **glArrayElementEXT()** function executes. No drawing occurs in this case, but the attributes corresponding to enabled arrays are modified.

When **glDrawArraysEXT()** is called, *count* sequential elements from each enabled array are used to construct a sequence of geometric primitives, beginning with element *first*. The *mode* parameter specifies what kind of primitives are constructed, and how the array elements are used to construct these primitives. Accepted values for *mode* are GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_QUAD_STRIP, GL_QUADS, and

GL_POLYGON. If GL_VERTEX_ARRAY_EXT is not enabled, no geometric primitives are generated.

Vertex attributes that are modified by **glDrawArraysEXT()** have an unspecified value after **glDrawArraysEXT()** returns. For example, if GL_COLOR_ARRAY_EXT is enabled, the value of the current color is undefined after **glDrawArraysEXT()** executes. Attributes that aren't modified remain well defined.

If an error is generated during execution of **glDrawArraysEXT()**, no other operations take place and the state does not change.

Although it is not an error to respecify an array between the execution of **glBegin()** and the corresponding execution of **glEnd()**, the result of such respecification is undefined. Static array data may be read and cached by the implementation at any time. If static array data are modified by the application, the results of any subsequent **glArrayElementEXT()** or **glDrawArraysEXT()** calls are undefined.

New Functions

glArrayElementEXT(), glDrawArraysEXT(), glVertexPointerEXT(), glNormalPointerEXT(), glColorPointerEXT(), glIndexPointerEXT(), glTexCoordPointerEXT(), glEdgeFlagPointerEXT(), glGetPointervEXT().

The Multisample Extension

The multisample extension, SGIS_multisample, provides a mechanism to antialias all OpenGL primitives: points, lines, polygons, bitmaps, and images.

This section explains how to use multisampling and also discusses what happens when you use it, discussing the following topics:

- “Introduction to Multisampling” on page 155
- “Using the Multisample Extension” on page 155 and “Using Advanced Multisampling Options” on page 156
- “How Multisampling Affects Different Primitives” on page 160

Introduction to Multisampling

Multisampling works by sampling all primitives multiple times at different locations within each pixel, in effect collecting subpixel information. The result is an image that has fewer aliasing artifacts.

Because each sample includes depth and stencil information, the depth and stencil functions perform equivalently to the single-sample mode. A single pixel can have 4, 8, or 16 subsamples.

When you use multisampling and read back color, you get the resolved color value (that is, the average of the samples). When you read back stencil or depth, you typically get back a single sample value rather than the average; the sample value is typically the one closest to the center of the pixel.

When to Use Multisampling

Multisample antialiasing is most valuable for rendering polygons because it correctly handles adjacent polygons, object silhouettes, and even intersecting polygons. Each time a pixel is updated, the color sample values for each pixel are resolved to a single, displayable color.

For points and lines, the "smooth" antialiasing mechanism provided by standard OpenGL results in a higher-quality image and should be used instead (see "Antialiasing" in Chapter 7, "Blending, Antialiasing, and Fog," of the *OpenGL Programming Guide).*

The multisampling extension lets you alternate multisample and smooth antialiasing during the rendering of a single scene, so it's possible to mix multisampled polygons with smooth lines and points. See "Multisampled Points" on page 160 and "Multisampled Lines" on page 160 for more information.

Using the Multisample Extension

To use multisampling in your application, select a multisampling-capable visual by calling **glXChooseVisual()** with the following items in *attr_list*:

GLX_SAMPLES_SGIS
: Must be followed by the minimum number of samples required in multisample buffers. **glXChooseVisual()** gives preference to visuals with the smallest number of samples that meet or exceed the specified

number. Color samples in the multisample buffer may have fewer bits than colors in the main color buffers. However, multisampled colors maintain at least as much color resolution in aggregate as the main color buffers.

GLX_SAMPLE_BUFFERS_SGIS
: This attribute is optional. Currently there are no visuals with more than one multisample buffer, so the returned value is either zero or one. When GLX_SAMPLES_SGIS is non-zero, this attribute defaults to 1. When specified, the attribute must be followed by the minimum acceptable number of multisample buffers. Visuals with the smallest number of multisample buffers that meet or exceed this minimum number are preferred.

Multisampling is enabled by default.

- To query whether multisampling is enabled, call

  `glIsEnabled(MULTISAMPLE_SGIS)`
- To turn off multisampling, call

  `glDisable(MULTISAMPLE_SGIS)`

Using Advanced Multisampling Options

Advanced multisampling options provide additional rendering capabilities. This section discusses:

- using a multisample mask to choose how many samples are writeable
- using alpha values to feather-blend texture edges
- using the accumulation buffer with multisampling

Figure 8-2 shows how the subsamples in one pixel are turned on and off.

1. First, the primitive is sampled at the locations defined by a sample pattern. If a sample is inside the polygon, it is turned on, otherwise, it is turned off. This produces a coverage mask.
2. The coverage mask is then ANDed with a user-defined sample mask, defined by a call to **glSampleMaskSGIS() (**see “Using a Multisample Mask to Fade Levels of Detail” on page 158).

3. You may also choose to convert the alpha value of a fragment to a mask and AND it with the coverage mask from step 2.

   Enable GL_SAMPLE_ALPHA_TO_MASK_SGIS to convert alpha to the mask. The fragment alpha value is used to generate a temporary mask, which is then ANDed with the fragment mask.

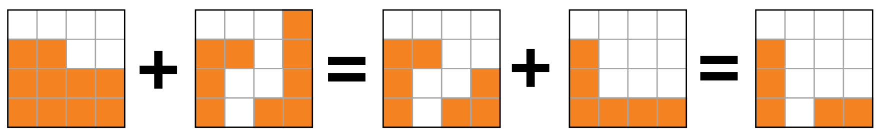

Figure 8-2 Sample Processing During Multisampling

The two processes—using a multisample mask created by **glSampleMaskSGIS()** and using the alpha value of the fragment as a mask—can both be used for different effects.

When GL_SAMPLE_ALPHA_TO_MASK_SGIS is enabled, it's usually appropriate to enable GL_SAMPLE_ALPHA_TO_ONE_SGIS to convert the alpha values to 1 before blending. Without this option, the effect would be colors that are twice as transparent.

Note: When you use multisampling, blending reduces performance. Therefore, when possible, disable blending and instead use GL_SAMPLE_MASK_SGIS or GL_ALPHA_TO_MASK.

Color Blending and Screen-Door Transparency

Multisampling can be used to solve the problem of blurred edges on textures with irregular edges, such as tree textures, that require extreme magnification. When the texture is magnified, the edges of the tree look artificial, as if the tree were a paper cutout. To make them look more natural by converting the alpha to a multisample mask, you can obtain several renderings of the same primitive, each with the samples offset by a specific amount. See "Accumulating Multisampled Images" on page 159 for more information.

The same process can be used to achieve screen-door transparency: If you draw only every other sample, the background shines through for all other samples, resulting in a transparent image. This is useful because it doesn't require the polygons to be sorted from back to front. It is also faster because it doesn't require blending.

Using a Multisample Mask to Fade Levels of Detail

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. You can use multisample masks to perform the blending from one level of detail of a model to the next by rendering the additional data in the detailed model using a steadily increasing percentage of subsamples as the viewpoint nears the object.

To achieve this blending between a simpler and a more detailed representation of an object, or to achieve screen-door transparency discussed in the previous section, either call **glSampleMaskSGIS()** or use the Alpha values of the object and call **glSampleAlphaToMaskSGIS().** Below is the prototype for **glSampleMaskSGIS()**:

```
void glSampleMaskSGIS (GLclampf value, boolean invert);
```

- *value* specifies coverage of the modification mask. Clamped to the range [0, 1]; 0 implies no coverage, and 1 implies full coverage.
- *invert* should be GL_FALSE to use the modification mask implied by value or GL_TRUE to use the bitwise inverse of that mask.

To define a multisample mask using **glSampleMaskSGIS()**, follow these steps:

1. Enable GL_SAMPLE_MASK_SGIS.
2. Call **glSampleMaskSGIS()** with, for example, *value* set to .25 and *invert* set to GL_FALSE.
3. Render the object once for the more complex level of detail.
4. Call **glSampleMaskSGIS()** again with, for example, *value* set to .25 and *invert* set to GL_TRUE.
5. Render the object for the simpler level of detail.

   This time, the complementary set of samples is used because of the use of the inverted mask.
6. Display the image.
7. Repeat the process for larger sample mask values of the mask as needed (as the viewpoint nears the object).

Accumulating Multisampled Images

You can enhance the quality of the image even more by making several passes, adding the result in the accumulation buffer. The accumulation buffer averages several renderings of the same primitive. For multipass rendering, different sample locations need to be used in each pass to achieve high quality.

When an application uses multisampling in conjunction with accumulation, it has to call **glSamplePatternSGIS()** with one of the following patterns as an argument:

- GL_1PASS_SGIS is designed to produce a well-antialiased result in a single rendering pass (this is the default).
- GL_2PASS_0_SGIS and GL_2PASS_1_SGIS together specify twice the number of sample points per pixel. You should first completely render a scene using pattern GL_2PASS_0_SGIS, then completely render it again using GL_2PASS_1_SGIS. When the two images are averaged using the accumulation buffer, the result is as if a single pass had been rendered with 2×GL_SAMPLES_SGIS sample points.
- GL_4PASS_0_SGIS, GL_4PASS_1_SGIS, GL_4PASS_2_SGIS, and GL_4PASS_3_SGIS together define a pattern of 4×GL_SAMPLES_SGIS sample points. They can be used to accumulate an image from four complete rendering passes.

Accumulating multisample results can also extend the capabilities of your system. For example, if you have only enough resources to allow four subsamples, but you are willing to render the image twice, you can achieve the same effect as multisampling with eight subsamples. Note that you do need an accumulation buffer, which also takes space.

To query the sample pattern, call **glGetIntegerv()** with *pname* set to GL_SAMPLE_PATTERN_SGIS. The pattern should be changed only between complete rendering passes.

For more information, see “The Accumulation Buffer,” in Chapter 10, “The Framebuffer,” of the *OpenGL Programming Guide.*

How Multisampling Affects Different Primitives

This section briefly discusses multisampled points, lines, polygons, pixels, and bitmaps.

Multisampled Points

If you're using multisampling, the value of the smoothing hint (GL_POINT_SMOOTH_HINT or GL_LINE_SMOOTH_HINT) is ignored. Since the quality of multisampled points may not be as good as that of anti-aliased points, remember that you can turn multisampling on and off as needed to achieve multisampled polygons and anti-aliased points.

Note: On RealityEngine systems, you achieve higher-quality multisampled points by setting point smooth hint GL_NICEST (though this mode is slower and should be used with care).

```
glHint(GL_POINT_SMOOTH_HINT, GL_NICEST)
```

The result is round points. Points may disappear or flicker if you use them without this hint. See the **Caution:** in the next section for caveats on using multisampling with smooth points.

Multisampled Lines

Lines are sampled into the multisample buffer as rectangles centered on the exact zero-area segment. Rectangle width is equal to the current linewidth. Rectangle 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.

Just like points, lines on RealityEngine systems look better when drawn "smooth" than they do with multisampling.

Caution: If you want to draw smooth lines and points by enabling GL_LINE_SMOOTH_HINT or GL_POINT_SMOOTH_HINT, you need to disable multisampling and then draw the lines and points. The trick is that you need to do this after you have finished doing all of the multisampled drawing. If you try to re-enable multisampling and draw more polygons, those polygons are not necessarily anti-aliased correctly.

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 sample is then written with this color if and only if it is geometrically within the exact polygon boundary.

If the depth-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 for each sample.

Polygon bit patterns apply equally to all sample locations at a pixel. All sample locations are considered for modification if the pattern bit is 1. None is considered if the pattern bit is 0.

Multisample Rasterization of Pixels and Bitmaps

In IRIS GL, pixels are not multisampled. In OpenGL, if multisampling is on, pixels are considered small rectangles and are subject to multisampling.

When pixels are sampled into the multisample buffer, each pixel is treated as an xzoom-by-yzoom square, which is then sampled just like a polygon.

For information about fast clears on RealityEngine, see the reference page for **glTagSampleBufferSGIX().**

New Functions

glSampleMaskSGIS(), glSamplePatternSGIS().