Fortunately, since the program is drawing a sphere, you can eliminate depth buffering and still render a correct image by discarding quads that face away from the viewer (the “front” faces, given the orientation of quads in this model):

```
glDisable(GL_DEPTH_TEST);
glEnable(GL_CULL_FACE);
glCullFace(GL_FRONT);
```

This pushes performance up to nearly 260 frames per second. Further improvements are possible. The program’s performance is still far from the upper limit determined by the rate at which the screen can be cleared.

```
        GLfloat x, y, z;
        x = sin(longitude * dToR) * cos(latitude * dToR);
        y = sin(latitude * dToR);
        z = cos(longitude * dToR) * cos(latitude * dToR);
        glNormal3f(x, y, z);
        glVertex3f(x, y, z);
        x = sin(longitude * dToR) * cos((latitude+10) * dToR);
        y = sin((latitude+10) * dToR);
        z = cos(longitude * dToR) * cos((latitude+10) * dToR);
        glNormal3f(x, y, z);
        glVertex3f(x, y, z);
        }
    glEnd()
    }
```

Of course, this yields a less smooth-looking sphere. When tuning, you often need to make such trade-offs between image quality and drawing performance, or provide controls in your application that allow end users to make the trade-offs.

In this particular case, the improvement up to 200 frames per second becomes apparent only because the program is single-buffered. If the program used double-buffering, performance wouldn't increase beyond the frame rate of the monitor (typically 60 or 72 frames per second), so there would be no performance penalty for using a higher-quality sphere.

If performance is truly critical and sphere intersections aren't likely, consider rendering more vertices at the edge of the silhouette and fewer at the center.

Testing Again for Fill Limitation

If you now shrink the window, performance increases again. This indicates that the program is again fill-limited. To increase performance further, you need to fill fewer pixels, or make pixel-fill less expensive by changing the pixel-drawing mode.

This particular application uses just one special per-fragment drawing mode: depth buffering. Depth buffering can be eliminated in a variety of special cases, including convex objects, backdrops, ground planes, and height fields.

Working on a Geometry-Limited Program

Previous tests determined that the program is geometry-limited. The next step is therefore to pinpoint the most severe problems and to change the program to alleviate the bottleneck.

Since the purpose of the program is to draw a lighted sphere, you can't eliminate lighting altogether. The program is already using a fairly simple lighting model (a single infinite light and a nonlocal viewer), so there's not much performance to be gained by changing the lighting model.

Smooth Shading Versus Flat Shading

Smooth shading requires more computation than flat shading, so consider changing

```
glShadeModel(GL_SMOOTH);
```

to

```
glShadeModel(GL_FLAT);
```

This increases performance to about 2.75 frames per second. Since this isn't much better than 2.5 frames per second, this discussion continues to use smooth shading.

Reducing the Number of Polygons

Since a change in lighting and shading does not improve performance significantly, the best option is to reduce the number of polygons the program is drawing.

One approach is to tesselate the sphere more efficiently. The simple sphere model used in the program has very large numbers of very small quadrilaterals near the poles, and comparatively large quadrilaterals near the equator. Several superior models exist, but to keep things simple, this discussion continues to use the latitude/longitude tesselation.

A little experimentation shows that reducing the number of quadrilaterals in the sphere causes a dramatic performance increase. When the program places vertices every 10 degrees, instead of every degree, performance skyrockets to nearly 200 frames per second:

```
    for (latitude = -90; latitude < 90; latitude += 10) {
        glBegin(GL_QUAD_STRIP);
        for (longitude = 0; longitude <= 360; longitude += 10) {
```

```
void
RunTest(void){...
      glNewList(1, GL_COMPILE);
      for (latitude = -90; latitude < 90; ++latitude) {
            glBegin(GL_QUAD_STRIP);
            for (longitude = 0; longitude <= 360; ++longitude) {
                  GLfloat x, y, z;
                  x = sin(longitude * dToR) * cos(latitude * dToR);
                  y = sin(latitude * dToR);
                  z = cos(longitude * dToR) * cos(latitude * dToR);
                  glNormal3f(x, y, z);
                  glVertex3f(x, y, z);
                  x = sin(longitude * dToR) * cos((latitude+1) * dToR);
                  y = sin((latitude+1) * dToR);
                  z = cos(longitude * dToR) * cos((latitude+1) * dToR);
                  glNormal3f(x, y, z);
                  glVertex3f(x, y, z);
                  }
            glEnd();
            }
      glEndList();

      printf("%.2f frames per second\n", Benchmark(Test));
      }
```

This version of the program achieves a little less than 2.5 frames per second, a noticeable improvement.

When the **glClear()**, **glNormal3f()**, and **glVertex3f()** calls are again replaced with **glColor3f()**, the program runs at roughly 4 frames per second. This implies that the program is no longer CPU limited, so you need to look further to find the bottleneck.

Testing for Fill Limitation

To check for a fill limitation, reduce the number of pixels that are filled. The easiest way to do that is to shrink the window. If you try that, you see that the frame rate doesn’t change for a smaller window, so the program must now be geometry-limited. As a result, it’s necessary to find ways to make the processing for each polygon less expensive, or to render fewer polygons.

```
      1  0.01s(  0.5)  2.1s( 97.7)      _ioctl
                       (/usr/lib/libc.so.1:ioctl.s)
      1  0.01s(  0.5)  2.1s( 98.2)       __glInitAccum64
                       (/usr/lib/libGLcore.so:../soft/so_accumop.c)
      1  0.01s(  0.5)  2.2s( 98.6)       _bzero
                       (/usr/lib/libc.so.1:bzero.s)
      1  0.01s(  0.5)  2.2s( 99.1)       GetClock (perf:perf.c)
      1  0.01s(  0.5)  2.2s( 99.5)       strncpy
                       (/usr/lib/libc.so.1:strncpy.c)
      1  0.01s(  0.5)  2.2s(100.0)      _select
                       (/usr/lib/libc.so.1:select.s)

    219   2.2s(100.0)  2.2s(100.0)        TOTAL
```

Almost 60% of the program’s time for a single frame is spent computing trigonometric functions (__sin and __cos).

There are several ways to improve this situation. First consider reducing the resolution of the quad strips that model the sphere. The current representation has over 60,000 quads, which is probably more than is needed for a high-quality image. After that, consider other changes. For example:

- Consider using efficient recurrence relations or table lookup to compute the regular grid of sine and cosine values needed to construct the sphere.
- The current code computes nearly every vertex on the sphere twice (once for each of the two quad strips in which a vertex appears), so you could achieve a 50% reduction in trigonometric operations just by saving and re-using the vertex values for a given line of latitude.

Because exactly the same sphere is rendered in every frame, the time required to compute the sphere vertices and normals is redundant for all but the very first frame. To eliminate the redundancy, generate the sphere just once, and place the resulting vertices and surface normals in a display list. You still pay the cost of generating the sphere once, and eventually may need to use the other techniques mentioned above to reduce that cost, but at least the sphere is rendered more efficiently:

```
void
Test(void) {
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      glCallList(1);
      }
....
```

Using the Profiler

The program still renders less than 0.8 frames per second. Because eliminating all graphics output had almost no effect on performance, the program is clearly CPU limited. Use the profiler to determine which function accounts for most of the execution time.

```
% cc -o perf -O -p perf.c -lGLU -lGL -lX11
% perf
% prof perf
-------------------------------------------------------------
Profile listing generated Wed Jul 19 17:17:03 1995
    with:       prof perf
-------------------------------------------------------------

samples   time    CPU    FPU   Clock   N-cpu  S-interval Countsize
    219   2.2s  R4000  R4010 100.0MHz   0     10.0ms     0(bytes)

Each sample covers 4 bytes for every 10.0ms (0.46% of 2.1900sec)
----------------------------------------------------------------------
-p[rocedures] using pc-sampling.
Sorted in descending order by the number of samples in each procedure.
Unexecuted procedures are excluded.
-----------------------------------------------------------------------

samples   time(%)      cum time(%)      procedure (file)

    112   1.1s( 51.1)  1.1s( 51.1)      __sin
                                       (/usr/lib/libm.so:trig.s)
     29  0.29s( 13.2)  1.4s( 64.4)      Test (perf:perf.c)
     18  0.18s(  8.2)  1.6s( 72.6)      __cos (/usr/lib/libm.so:trig.s)
     16  0.16s(  7.3)  1.8s( 79.9)      Finish
                       (/usr/lib/libGLcore.so:../EXPRESS/gr2_context.c)
     15  0.15s(  6.8)  1.9s( 86.8)      __glexpim_Color3f
                       (/usr/lib/libGLcore.so:../EXPRESS/gr2_vapi.c)
     14  0.14s(  6.4)    2s( 93.2)      _BSD_getime
                       (/usr/lib/libc.so.1:BSD_getime.s)
      3  0.03s(  1.4)  2.1s( 94.5)      __glim_Finish
                       (/usr/lib/libGLcore.so:../soft/so_finish.c)
      3  0.03s(  1.4)  2.1s( 95.9)      _gettimeofday
                       (/usr/lib/libc.so.1:gettimeday.c)
      2  0.02s(  0.9)  2.1s( 96.8)      InitBenchmark (perf:perf.c)
      1  0.01s(  0.5)  2.1s( 97.3)      __glMakeIdentity
                       (/usr/lib/libGLcore.so:../soft/so_math.c)
```

Testing for CPU Limitation

An application may be CPU limited, geometry limited, or fill limited. Start tuning by checking for a CPU bottleneck. Replace the **glVertex3f()**, **glNormal3f()**, and **glClear()** calls in **Test()** with **glColor3f()** calls. This minimizes the number of graphics operations while preserving the normal flow of instructions and the normal pattern of accesses to main memory.

```
void
Test(void) {
      float latitude, longitude;
      float dToR = M_PI / 180.0;

      glColor(0, 0, 0);

      for (latitude = -90; latitude < 90; ++latitude) {
            glBegin(GL_QUAD_STRIP);
            for (longitude = 0; longitude <= 360; ++longitude) {
                  GLfloat x, y, z;
                  x = sin(longitude * dToR) * cos(latitude * dToR);
                  y = sin(latitude * dToR);
                  z = cos(longitude * dToR) * cos(latitude * dToR);
                  glColor3f(x, y, z);
                  glColor3f(x, y, z);
                  x = sin(longitude * dToR) * cos((latitude+1) * dToR);
                  y = sin((latitude+1) * dToR);
                  z = cos(longitude * dToR) * cos((latitude+1) * dToR);
                  glColor3f(x, y, z);
                  glColor3f(x, y, z);
                  }
            glEnd();
            }
      }
```

```
/**********************************************************************
 * main - create window and context, then pass control to ProcessEvents
**********************************************************************/
int
main(int argc, char* argv[]) {
        Display *dpy;
        XVisualInfo *vi;
        XSetWindowAttributes swa;
        Window win;
        GLXContext cx;

        ApplicationName = argv[0];

        /* Get a connection: */
        dpy = XOpenDisplay(NULL);
        if (!dpy) Error("can't open display");

        /* Get an appropriate visual: */
        vi = glXChooseVisual(dpy, DefaultScreen(dpy),
                              VisualAttributes);
        if (!vi) Error("no suitable visual");

        /* Create a GLX context: */
        cx = glXCreateContext(dpy, vi, 0, GL_TRUE);

        /* Create a color map: */
        swa.colormap = XCreateColormap(dpy, RootWindow(dpy,
                                  vi->screen), vi->visual, AllocNone);

        /* Create a window: */
        swa.border_pixel = 0;
        swa.event_mask = ExposureMask | StructureNotifyMask |
                                                          KeyPressMask;
        win = XCreateWindow(dpy, RootWindow(dpy, vi->screen), 0, 0,
                       300, 300, 0, vi->depth, InputOutput, vi->visual,
                       CWBorderPixel|CWColormap|CWEventMask, &swa);
        XStoreName(dpy, win, "perf");
        XMapWindow(dpy, win);

        /* Connect the context to the window: */
        glXMakeCurrent(dpy, win, cx);

        /* Handle events: */
        while (1) ProcessEvents(dpy);
        }
```

```
                        case Expose:
                                redraw = 1;
                                break;
                        case ConfigureNotify:
                                glViewport(0, 0,
                                        WindowWidth =
                                              event.xconfigure.width,
                                        WindowHeight =
                                              event.xconfigure.height);
                                redraw = 1;
                                break;
                        case KeyPress:
                                (void) XLookupString(&event.xkey, buf,
                                        sizeof(buf), &keysym, NULL);
                                switch (keysym) {
                                        case XK_Escape:
                                                exit(EXIT_SUCCESS);
                                        default:
                                                break;
                                        }
                                break;
                        default:
                                break;
                        }
                } while (XPending(dpy));

        if (redraw) RunTest();
        }

/**********************************************************************
 * Error - print an error message, then exit
 *********************************************************************/
void
Error(const char* format, ...) {
        va_list args;

        fprintf(stderr, "%s:  ", ApplicationName);

        va_start(args, format);
        vfprintf(stderr, format, args);
        va_end(args);

        exit(EXIT_FAILURE);
        }
```

```
        glEnable(GL_DEPTH_TEST);

        glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
        glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
        glLightfv(GL_LIGHT0, GL_POSITION, direction);
        glEnable(GL_LIGHT0);
        glEnable(GL_LIGHTING);

        glMaterialfv(GL_FRONT, GL_AMBIENT, ambientMat);
        glMaterialfv(GL_FRONT, GL_SPECULAR, specularMat);
        glMateriali(GL_FRONT, GL_SHININESS, 128);

        glEnable(GL_COLOR_MATERIAL);
        glShadeModel(GL_SMOOTH);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(45.0, 1.0, 2.4, 4.6);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        gluLookAt(0,0,3.5,  0,0,0,  0,1,0);

        printf("%.2f frames per second\n", Benchmark(Test));
        }

/**********************************************************************
 * ProcessEvents - handle X11 events directed to our window
 *
 * Run the measurement each time we receive an expose event.
 * Exit when we receive a keypress of the Escape key.
 * Adjust the viewport and projection transformations when the window
 * changes size.
 **********************************************************************/
void
ProcessEvents(Display* dpy) {
        XEvent event;
        Bool redraw = 0;

        do {
                char buf[31];
                KeySym keysym;

                XNextEvent(dpy, &event);
                switch(event.type) {
```

```
void
Test(void) {
        float latitude, longitude;
        float dToR = M_PI / 180.0;

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        for (latitude = -90; latitude < 90; ++latitude) {
                glBegin(GL_QUAD_STRIP);
                for (longitude = 0; longitude <= 360; ++longitude) {
                      GLfloat x, y, z;
                      x = sin(longitude * dToR) * cos(latitude * dToR);
                      y = sin(latitude * dToR);
                      z = cos(longitude * dToR) * cos(latitude * dToR);
                      glNormal3f(x, y, z);
                      glVertex3f(x, y, z);
                      x = sin(longitude * dToR) * cos((latitude+1) *
                                                                dToR);
                      y = sin((latitude+1) * dToR);
                        z = cos(longitude * dToR) * cos((latitude+1) *
                                                                dToR);
                      glNormal3f(x, y, z);
                      glVertex3f(x, y, z);
                      }
                glEnd();
                }
        }

/*********************************************************************
 * RunTest - initialize the rendering context and run the test
 ********************************************************************/
void
RunTest(void) {
        static GLfloat diffuse[] = {0.5, 0.5, 0.5, 1.0};
        static GLfloat specular[] = {0.5, 0.5, 0.5, 1.0};
        static GLfloat direction[] = {1.0, 1.0, 1.0, 0.0};
        static GLfloat ambientMat[] = {0.1, 0.1, 0.1, 1.0};
        static GLfloat specularMat[] = {0.5, 0.5, 0.5, 1.0};

        if (Overhead == 0.0)
                InitBenchmark();

        glClearColor(0.5, 0.5, 0.5, 1.0);

        glClearDepth(1.0);
```

```
        /* Select a run time appropriate for our timer resolution: */
        runTime = ChooseRunTime();

        /*
         * Measure successively larger batches of operations until we
         * find one that's long enough to meet our runtime target:
         */
        reps = 1;
        for (;;) {
                /* Run a batch: */
                FinishDrawing();
                start = WaitForTick();
                for (i = reps; i > 0; --i)
                        (*operation)();
                FinishDrawing();

                /* If we reached our target, bail out of the loop: */
                current = GetClock();
                if (current >= start + runTime + Overhead)
                        break;

                /*
                 * Otherwise, increase the rep count and try to reach
                 * the target on the next attempt:
                 */
                if (current > start)
                        newReps = reps *(0.5 + runTime /
                                         (current - start - Overhead));
                else
                        newReps = reps * 2;
                if (newReps == reps)
                        reps += 1;
                else
                        reps = newReps;
                }

        /* Subtract overhead and return the final operation rate: */
        return (double) reps / (current - start - Overhead);
        }
/**********************************************************************
 * Test - the operation to be measured
 *
 * Will be run several times in order to generate a reasonably accurate
 * result.
**********************************************************************/
```

```
void
InitBenchmark(void) {
        double runTime;
        long reps;
        double start;
        double finish;
        double current;

        /* Select a run time appropriate for our timer resolution: */
        runTime = ChooseRunTime();

        /* Wait for the pipe to clear: */
        FinishDrawing();

        /* Measure approximate overhead for finalization and timing
         * routines: */
        reps = 0;
        start = WaitForTick();
        finish = start + runTime;
        do {
                FinishDrawing();
                ++reps;
                } while ((current = GetClock()) < finish);

        /* Save the overhead for use by Benchmark(): */
        Overhead = (current - start) / (double) reps;
        }

/**********************************************************************
 * Benchmark--measure number of caller operations performed per second
 *
 * Assumes InitBenchmark() has been called previously, to initialize
 * the estimate for timing overhead.
*********************************************************************/
double
Benchmark(void (*operation)(void)) {
        double runTime;
        long reps;
        long newReps;
        long i;
        double start;
        double current;

        if (!operation)
                return 0.0;
```

```
/**********************************************************************
 * FinishDrawing - wait for the graphics pipe to go idle
 *
 * This is needed to make sure we're not including time from some
 * previous uncompleted operation in our measurements. (It's not
 * foolproof, since we can't eliminate context switches, but we can
 * assume our caller has taken care of that problem.)
**********************************************************************/
void
FinishDrawing(void) {
        glFinish();
        }


/**********************************************************************
 * WaitForTick - wait for beginning of next system clock tick; return
 * the time
**********************************************************************/
double
WaitForTick(void) {
        double start;
        double current;

        start = GetClock();

        /* Wait for next tick: */
        while ((current = GetClock()) == start)
                ;

        /* Start timing: */
        return current;
        }


/**********************************************************************
 * InitBenchmark - measure benchmarking overhead
 *
 * This should be done once before each risky change in the
 * benchmarking environment. A “risky” change is one that might
 * reasonably be expected to affect benchmarking overhead. (For
 * example, changing from a direct rendering context to an indirect
 * rendering context.)  If all measurements are being made on a single
 * rendering context, one call should suffice.
**********************************************************************/
```

```
char* ApplicationName;
double Overhead = 0.0;
int VisualAttributes[] = { GLX_RGBA, GLX_RED_SIZE, 1, GLX_GREEN_SIZE,
        1, GLX_BLUE_SIZE, 1, GLX_DEPTH_SIZE, 1, None };
int WindowWidth;
int WindowHeight;

/**********************************************************************
 * GetClock - get current time (expressed in seconds)
**********************************************************************/
double
GetClock(void) {
        struct timeval t;

        gettimeofday(&t);
        return (double) t.tv_sec + (double) t.tv_usec * 1E-6;
        }

/**********************************************************************
 * ChooseRunTime - select an appropriate runtime for benchmarking
**********************************************************************/
double
ChooseRunTime(void) {
        double start;
        double finish;
        double runTime;

        start = GetClock();

        /* Wait for next tick: */
        while ((finish = GetClock()) == start)
                ;

        /* Run for 100 ticks, clamped to [0.5 sec, 5.0 sec]: */
        runTime = 100.0 * (finish - start);
        if (runTime < 0.5)
                runTime = 0.5;
        else if (runTime > 5.0)
                runTime = 5.0;

        return runTime;
        }
```

Tuning Example

This section steps you through a complete example of tuning a small program using the techniques discussed in Chapter 12, “Tuning the Pipeline.” Consider a program that draws a lighted sphere, shown in Figure 13-1.

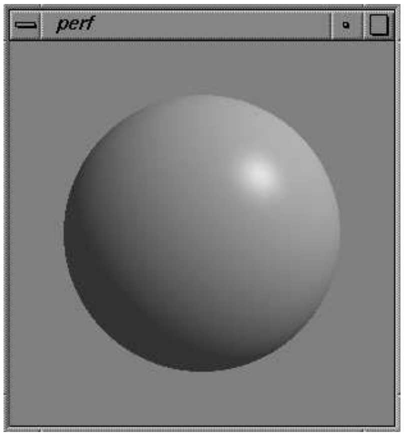

Figure 13-1 Lighted Sphere Created by perf.c

You can use the benchmarking framework in Appendix B, “Benchmarks,” for window and timing services. All you need to do is set up the OpenGL rendering context in **RunTest()**, and perform the drawing operations in **Test()**. The first version renders the sphere by drawing strips of quadrilaterals parallel to the sphere’s lines of latitude. On a 100MHz Indigo^2 Extreme system, this program renders about 0.77 frames per second.

Example 13-2 Performance Tuning Example Program

```
/*********************************************************************
   cc -o perf -O perf.c -lGLU -lGL -lX11
**********************************************************************/

#include <GL/glx.h>
#include <GL/glu.h>
#include <X11/keysym.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include <sys/time.h>
#include <math.h>
```

```
        glPixelTransferi(GL_ALPHA_BIAS, 0);

        /*
         * Disable extensions that could slow down
         * glDrawPixels.(Actually, you should check for the
         * presence of the proper extension before making
         * these calls.I omitted that code for simplicity.)
         */

#ifdef GL_EXT_convolution
        glDisable(GL_CONVOLUTION_1D_EXT);
        glDisable(GL_CONVOLUTION_2D_EXT);
        glDisable(GL_SEPARABLE_2D_EXT);
#endif

#ifdef GL_EXT_histogram
        glDisable(GL_HISTOGRAM_EXT);
        glDisable(GL_MINMAX_EXT);
#endif

#ifdef GL_EXT_texture3D
        glDisable(GL_TEXTURE_3D_EXT);
#endif

        /*
         * The following is needed only when using a
         * multisample-capable visual.
         */

#ifdef GL_SGIS_multisample
        glDisable(GL_MULTISAMPLE_SGIS);
#endif
```

Chapter 13

Tuning Graphics Applications: Examples

This chapter first presents a code fragment that helps you draw pixels fast. The second section steps through an example of tuning a small graphics program, showing changes to the program and discussing the speed improvements that result.

Drawing Pixels Fast

The code fragment in Example 13-1 illustrates how to set an OpenGL state so that subsequent calls to **glDrawPixels()** or **glCopyPixels()** will be fast.

Example 13-1 Drawing Pixels Fast

```
/*
 * Disable stuff that's likely to slow down
 * glDrawPixels.(Omit as much of this as possible,
 * when you know in advance that the OpenGL state is
 * already set correctly.)
 */
glDisable(GL_ALPHA_TEST);
glDisable(GL_BLEND);
glDisable(GL_DEPTH_TEST);
glDisable(GL_DITHER);
glDisable(GL_FOG);
glDisable(GL_LIGHTING);
glDisable(GL_LOGIC_OP);
glDisable(GL_STENCIL_TEST);
glDisable(GL_TEXTURE_1D);
glDisable(GL_TEXTURE_2D);
glPixelTransferi(GL_MAP_COLOR, GL_FALSE);
glPixelTransferi(GL_RED_SCALE, 1);
glPixelTransferi(GL_RED_BIAS, 0);
glPixelTransferi(GL_GREEN_SCALE, 1);
glPixelTransferi(GL_GREEN_BIAS, 0);
glPixelTransferi(GL_BLUE_SCALE, 1);
glPixelTransferi(GL_BLUE_BIAS, 0);
glPixelTransferi(GL_ALPHA_SCALE, 1);
```