Appendix B

Benchmarks

This appendix contains a sample program you can use to measure the performance of an OpenGL operation. For an example of how the program can be used with a small graphics applications, see Chapter 13, "Tuning Graphics Applications: Examples."

```
/*****************************************************************
 * perf - framework for measuring performance of an OpenGL operation
 *
 * Compile with: cc -o perf -O perf.c -lGL -lX11
 *
*****************************************************************/


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


char* ApplicationName;
double Overhead = 0.0;
int VisualAttributes[] = { GLX_RGBA, 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;
        }

/*****************************************************************
 * 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.
************************************************************************/
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's 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;

        /* 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
Test(void) {
        /* Replace this code with the operation you want to measure: */
        glColor3f(1.0, 1.0, 0.0);
        glRecti(0, 0, 32, 32);
        }

/*************************************************************************
 * RunTest - initialize the rendering context and run the test
*************************************************************************/
void
RunTest(void) {
        if (Overhead == 0.0)
                InitBenchmark();
```

```
        /* Replace this sample with initialization for your test: */

        glClearColor(0.5, 0.5, 0.5, 1.0);
        glClear(GL_COLOR_BUFFER_BIT);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0.0, WindowWidth, 0.0, WindowHeight, -1.0, 1.0);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        printf("%.2f operations 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) {
                        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);
        }

/*****************************************************************
 * 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);
}
```