Appendix C

Benchmarking Libraries: libpdb and libisfast

When optimizing an OpenGL application, there are two problems you need to address:

- When you're writing an OpenGL application, it's difficult to know whether a particular feature (like depth buffering or texture mapping) is fast enough to be useful.
- If you want your application to run fast on a variety of machines, while taking advantage of as many hardware features as possible, you need to write code that makes configuration decisions at runtime.

For the OpenGL predecessor IRIS GL, you could call **getgdesc()** to determine whether a feature had hardware support. For example, you could determine whether a Z buffer existed. If it did, you might assume that Z buffering was fast, and therefore your application would use it.

In OpenGL, things are more complicated. All the core features are provided, even when there is no hardware support for them and they must be implemented completely in software. There is no OpenGL routine that reports whether a feature is implemented partly or completely in hardware.

Furthermore, features interact in unpredictable ways. For example, a machine might have hardware support for depth buffering, but only for some comparison functions. Or depth buffering might be fast only as long as stencilling is not enabled. Or depth buffering might be fast when drawing to a window, but slow when drawing to a pixmap. And so on. A routine that identifies hardware support for particular features is actually a lot more complicated and less useful than you might think.

To decide whether a given OpenGL feature is fast, you have to measure it. Since the performance of a section of graphics code is dependent on many pieces of information from the runtime environment, no other method is as well-defined and reliable.

Keep in mind that while the results of the libisfast routines are interesting, they apply to limited special cases. Always consider using a more general tool like Open Inventor or IRIS Performer.

Performance measurement can be tricky:

- You need to handle the cases when you're displaying over a network, as well as locally.
- Think about flushing the graphics pipeline properly, and accounting for the resulting overhead.
- Measuring all the features needed by your application may take a while. Save performance measurements and reuse them whenever possible; users won't want to wait for measurements each time the application starts.
- Consider measuring things other than graphics: Disk and network throughput, processing time for a particular set of data, performance on uniprocessor and multiprocessor systems.

Libraries for Benchmarking

This appendix describes two libraries that can help with all of the tasks just mentioned:

- libpdb (**P**erformance **D**ata**B**ase). Routines for measuring execution rates and maintaining a simple database.
- libisfast. A set of routines demonstrating libpdb that answer common questions about the performance of OpenGL features (using reasonable but subjective criteria).

These libraries can't substitute for comprehensive benchmarking and performance analysis, and don't replace more sophisticated tools (like IRIS Performer and IRIS Inventor) that optimize application performance in a variety of ways. However, they can handle simple tasks easily.

Using libpdb

libpdb provides five routines:

- **pdbOpen()** opens the performance database.
- **pdbReadRate()** reads the execution rate for a given benchmark (identified by a machine name, application name, benchmark name, and version string) from the database.

- **pdbMeasureRate()** measures the execution rate for a given operation.
- **pdbWriteRate()** writes the execution rate for a given benchmark into the database.
- **pdbClose()** closes the performance database and writes it back to disk if necessary.

All libpdb routines return a value of type pdbStatusT, which is a bitmask of error conditions. If the value is zero (PDB_NO_ERROR), the call completed successfully. If the value is nonzero, it is a combination of one or more of the conditions listed in Table C-1:

Table C-1 Errors Returned by libpdb Routines

Error	Meaning
PDB_OUT_OF_MEMORY	Attempt to allocate memory failed.
PDB_SYNTAX_ERROR	Database contains one or more records that could not be parsed.
PDB_NOT_FOUND	Database does not contain the record requested by the application.
PDB_CANT_WRITE	Database file could not be updated.
PDB_NOT_OPEN	**pdbOpen()** was not invoked before calling one of the other libpdb routines.
PDB_ALREADY_OPEN	**pdbOpen()** was called while the database is still open (e.g., before **pdbClose()** is invoked).

Every program must call **pdbOpen()** before using the database, and **pdbClose()** when the database is no longer needed. **pdbOpen()** opens the database file (stored in $HOME/.pdb2 on UNIX systems) and reads all the performance measurements into main memory. **pdbClose()** releases all memory used by the library, and writes the database back to its file if any changes have been made by invoking **pdbWriteRate()**.

```
pdbStatusT pdbOpen(void);
pdbStatusT pdbClose(void);
```

pdbOpen() returns

- PDB_NO_ERROR on success
- PDB_OUT_OF_MEMORY if there was insufficient main memory to store the entire database
- PDB_SYNTAX_ERROR if the contents of the database could not be parsed or seemed implausible (for example a nonpositive performance measurement)

- PDB_ALREADY_OPEN if the database has been opened by a previous call to **pdbOpen()** and not closed by a call to **pdbClose()**

pdbClose() returns

- PDB_NO_ERROR on success
- PDB_CANT_WRITE if the database file is unwritable for any reason
- PDB_NOT_OPEN if the database is not open

Normally applications should look for the performance data they need before going to the trouble of taking measurements. **pdbReadRate()**, which is used for this, has the following prototype:

```
pdbStatusT pdbReadRate (const char* machineName,const char* appName,
          const char* benchmarkName,const char* versionString, double* rate )
```

machineName A zero-terminated string giving the name of the machine for which the measurement is sought. If NULL, the default machine name is used. (In X11 environments, the display name is an appropriate choice, and the default machine name is the content of the DISPLAY environment variable.)

appName Name of the application. This is used as an additional database key to reduce accidental collisions between benchmark names.

benchmarkName Name of the benchmark.

versionString The fourth argument is a string identifying the desired version of the benchmark. For OpenGL performance measurements, the string returned by **glGetString(GL_VERSION)** is a good value for this argument. Other applications might use the version number of the benchmark, rather than the version number of the system under test.

rate A pointer to a double-precision floating-point variable that receives the performance measurement (the "rate") from the database. The rate indicates the number of benchmark operations per second that were measured on a previous run. If **pdbReadRate()** returns zero, then it completed successfully and the rate is returned in the last argument. If the requested benchmark is not present in the database, it returns PDB_NOT_FOUND. Finally, if **pdbReadRate()** is called when the database has not been opened by **pdbOpen()**, it returns PDB_NOT_OPEN.

Example for pdbRead

```
main() {
        double rate;
        pdbOpen();
        if (pdbReadRate(NULL, "myApp", "triangles",
           glGetString(GL_VERSION), &rate)
               == PDB_NO_ERROR)
        printf("%g triangle calls per second\n", rate);
        pdbClose();
        }
```

When the application is run for the first time, or when the performance database file has been removed (perhaps to allow a fresh start after a hardware upgrade), **pdbReadRate()** is not able to find the desired benchmark. If this happens, the application should use **pdbMeasureRate()**, which has the following prototype, to make a measurement:

```
pdbStatusT pdbMeasureRate (pdbCallbackT initialize, pdbCallbackT operation,
                          pdbCallbackT finalize, int calibrate, double* rate)
```

initialize A pointer to the initialization function. The initialization function is run before each set of operations. For OpenGL performance measurement, it's appropriate to use **glFinish()** for initialization, to make sure that the graphics pipe is quiet. However, for other performance measurements, the initialization function can create test data, preload caches, and so on. May be NULL, in which case no initialization is performed.

operation A pointer to the operation function. This function performs the operations that are to be measured. Usually you'll want to make sure that any global state needed by the operation is set up before calling the operation function, so that you don't include the cost of the setup operations in the measurement.

finalize A pointer to a finalization function. This is run once, after all the calls to the operation function are complete. In the example above, **glFinish()** ensures that the graphics pipeline is idle. It may be NULL, in which case no finalization is performed. The finalization function must be calibrated so that the overhead of calling it may be subtracted from the time used by the operation function. If the fourth argument is nonzero, then **pdbMeasureRate()** calibrates the finalization function. If the fourth argument is zero, then **pdbMeasureRate()** uses the results of the previous calibration. Recalibrating each measurement is the safest approach, but it roughly doubles the amount of time needed for a measurement. For OpenGL, it should be acceptable to calibrate once and recalibrate only when using a different X11 display.

rate A pointer to a double-precision floating-point variable that receives the execution rate. This rate is the number of times the operation function was called per second. **pdbMeasureRate()** attempts to compute a number of repetitions that results in a run time of about one second. (Calibration requires an additional second.) It's reasonably careful about timekeeping on systems with low-resolution clocks.

pdbMeasureRate() always returns PDB_NO_ERROR.

Example for pdbMeasureRate()

```
void SetupOpenGLState(void) {
        /* set all OpenGL state to desired values */
        }

void DrawTriangles(void) {
        glBegin(GL_TRIANGLE_STRIP);
                /* specify some vertices... */
        glEnd();
        }
main() {
                double rate;
                pdbOpen();
                if (pdbReadRate(NULL, "myApp", "triangles",
                   glGetString(GL_VERSION), &rate)
                      != PDB_NO_ERROR) {
                SetupOpenGLState();
                pdbMeasureRate(glFinish, DrawTriangles,
                    glFinish, 1, &rate);
                }
        printf("%g triangle calls per second\n", rate);
        pdbClose();
        }
```

Once a rate has been measured, it should be stored in the database by calling **pdbWriteRate()**, which has the following prototype:

`pdbStatusT pdbWriteRate (const char*` *machineName*`, const char*` *applicationName*`, const char*` *benchmarkName*`,const char*` *versionString*`, double` *rate*`)`

The first four arguments of **pdbWriteRate()** match the first four arguments of **pdbReadRate()**. The last argument is the performance measurement to be saved in the database.

pdbWriteRate() returns

- PDB_NO_ERROR if the performance measurement was added to the in-memory copy of the database
- PDB_OUT_OF_MEMORY if there was insufficient main memory to do so
- PDB_NOT_OPEN if the database is not open

When **pdbWriteRate()** is called, the in-memory copy of the performance database is marked “dirty.” **pdbClose()** takes note of this and writes the database back to disk.

Example for pdbWriteRate()

```
main() {
        double rate;
        pdbOpen();
        if (pdbReadRate(NULL, "myApp", "triangles",
           glGetString(GL_VERSION), &rate)
              != PDB_NO_ERROR) {
                SetupOpenGL();
                pdbMeasureRate(glFinish, DrawTriangles,
                    glFinish, 1, &rate);
                pdbWriteRate(NULL, "myApp", "triangles",
                    glGetString(GL_VERSION), rate);
                }
        printf("%g triangle calls per second\n", rate);
        pdbClose();
        }
```

Using libisfast

The libisfast library is a set of demonstration routines that show how libpdb can be used to measure and maintain OpenGL performance data. libisfast is based on purely subjective performance criteria. If they’re appropriate for your application, feel free to use them. If not, copy the source code and modify it accordingly.

In all cases that follow, the term “triangles” refers to a triangle strip with 37 vertices. The triangles are drawn with perspective projection, lighting, and smooth (Gouraud) shading. Unless otherwise stated, display-list-mode drawing is used. (This makes isfast yield more useful results when the target machine is being accessed over a network.)

The application must initialize isfast before performing any performance measurements, and clean up after the measurements are finished. On X11 systems these tasks are accomplished by calling

```
int IsFastXOpenDisplay(const char* displayName);
```

and

```
void IsFastXCloseDisplay(void);
```

respectively. **IsFastOpenXDisplay()** returns zero if the named display could not be opened, and nonzero if the display was opened successfully.

DepthBufferingIsFast() returns nonzero if depth buffered triangles can be drawn at least one-half as fast as triangles without depth buffering:

```
int DepthBufferingIsFast(void);
```

ImmediateModeIsFast() returns nonzero if immediate-mode triangles can be drawn at least one-half as fast as display-listed triangles:

```
int ImmediateModeIsFast(void);
```

Note that one significant use of **ImmediateModeIsFast()** may be to decide whether a "local" or a "remote" rendering strategy is appropriate. If immediate mode is fast, as on a local workstation, it may be best to use that mode and avoid the memory cost of duplicating the application's data structures in display lists. If immediate mode is slow, as is likely for a remote workstation, it may be best to use display lists for bulky geometry and textures.

StencillingIsFast() returns nonzero if stencilled triangles can be drawn at least one-half as fast as non-stencilled triangles:

```
int StencillingIsFast(void);
```

TextureMappingIsFast() returns nonzero if texture-mapped triangles can be drawn at least one-half as fast as non-texture-mapped triangles:

```
int TextureMappingIsFast(void);
```

Although the routines in libisfast are useful for a number of applications, you should study them and modify them for your own use. That way you'll explore the particular performance characteristics of your systems: their sensitivity to triangle size, triangle strip length, culling, stencil function, texture-map type, texture-coordinate generation method, and so on.