Appendix F

Programming Tips

This appendix lists some tips and guidelines that you might find useful. Keep in mind that these tips are based on the intentions of the designers of OpenGL, not on any experience with actual applications and implementations. This appendix has the following major sections:

- "OpenGL Correctness Tips"
- "OpenGL Performance Tips"
- "GLX Tips"

OpenGL Correctness Tips

- Perform error checking often. Call **glGetError()** at least once each time the scene is rendered to make certain error conditions are noticed while you are developing your application. Once development is completed, removing the error checking can help improve performance.
- Do not count on the error behavior of an OpenGL implementation—it might change in a future release of OpenGL. For example, OpenGL 1.1 ignores matrix operations invoked between **glBegin()** and **glEnd()** commands, but a future version might not. Put another way, OpenGL error semantics may change between upward-compatible revisions.
- Check the extension string to verify that any extensions you want to use are supported. This is particularly relevant for routines that must be accessed through a function pointer retrieved by functions such as **glXGetProcAddress()** or **wglGetProcAddress()**.
- If you need to collapse all geometry to a single plane, use the projection matrix. If the modelview matrix is used, OpenGL features that operate in eye coordinates (such as lighting and application-defined clipping planes) might fail.
- Do not make extensive changes to a single matrix. For example, do not animate a rotation by continually calling **glRotate*()** with an incremental angle. Rather, use **glLoadIdentity()** to initialize the given matrix for each frame, and then call **glRotate*()** with the desired complete angle for that frame.
- Count on multiple passes through a rendering database to generate the same pixel fragments only if this behavior is guaranteed by the invariance rules established for a compliant OpenGL implementation. (See Appendix G for details on the invariance rules.) Otherwise, a different set of fragments might be generated.
- Do not expect errors to be reported while a display list is being defined. The commands within a display list generate errors only when the list is executed.
- Place the frustum's near plane as far from the viewpoint as possible to optimize the operation of the depth buffer.
- Call **glFlush()** to force all previous OpenGL commands to be executed. Do not count on **glGet*()** or **glIs*()** to flush the rendering stream. Query commands flush as much of the stream as is required to return valid data but don't guarantee completion of all pending rendering commands.

- Turn dithering off when rendering predithered images (for example, when **glCopyPixels()** is called).
- Make use of the full range of the accumulation buffer. For example, if accumulating four images, scale each by one-quarter as it is accumulated.
- If exact two-dimensional rasterization is desired, you must carefully specify both the orthographic projection and the vertices of primitives that are to be rasterized. The orthographic projection should be specified with integer coordinates, as shown in the following example:

  ```
  gluOrtho2D(0, width, 0, height);
  ```

  where *width* and *height* are the dimensions of the viewport. Given this projection matrix, polygon vertices and pixel image positions should be placed at integer coordinates to rasterize predictably. For example, **glRecti(**0, 0, 1, 1**)** reliably fills the lower left pixel of the viewport, and **glRasterPos2i(**0, 0**)** reliably positions an unzoomed image at the lower left of the viewport. Point vertices, line vertices, and bitmap positions should be placed at half-integer locations, however. For example, a line drawn from (*x1*, 0.5) to (*x2*, 0.5) will be reliably rendered along the bottom row of pixels into the viewport, and a point drawn at (0.5, 0.5) will reliably fill the same pixel as **glRecti(**0, 0, 1, 1**)**.

  An optimum compromise that allows all primitives to be specified at integer positions, while still ensuring predictable rasterization, is to translate *x* and *y* by 0.375, as shown in the following code fragment. Such a translation keeps polygon and pixel image edges safely away from the centers of pixels, while moving line vertices close enough to the pixel centers.

  ```
  glViewport(0, 0, width, height);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0, width, 0, height);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glTranslatef(0.375, 0.375, 0.0);
  /* render all primitives at integer positions */
  ```

- Avoid using negative *w* vertex coordinates and negative *q* texture coordinates. OpenGL might not clip such coordinates correctly and might make interpolation errors when shading primitives defined by such coordinates.

- Do not assume the precision of operations based on the data types of parameters to OpenGL commands. For example, if you are using **glRotated()**, you should not assume that the geometric processing pipeline maintains double-precision floating-point accuracy during its operation. It is possible that the parameters to **glRotated()** are converted to a different data type before processing.

OpenGL Performance Tips

Following are some performance tips for your consideration, regardless of whether of which rendering pipeline (i.e., fixed-function or programmable) you're using. Generally speaking, these tips will help improve the performance of any application:

- Try to minimize state changes during rendering by grouping primitives of like OpenGL state. When you issue a draw call (e.g., **glVertex()**, **glDrawArrays()**), OpenGL does an internal state validation check, usually called *validation*. Validation can be an expensive operation in terms of performance, so it benefits your application's performance to minimize validations.
- Use query functions when your application requires just a few state values for its own computations. If your application requires several state values from the same attribute group, use **glPushAttrib()** and **glPopAttrib()** to save and restore them.
- Some OpenGL implementations benefit from storing vertex data in vertex arrays. Use of vertex arrays reduces function call overhead. Some implementations can improve performance by engaging in batch processing or reusing processed vertices.
- Generate object names (e.g., by calling **glGen*()**), rather than assigning names of your own choosing. This practice is required for OpenGL Version 3.1 and greater, but in versions prior to OpenGL Version 3.1, user-assigned object names require a lookup in internal implementation-specific tables (and almost always on the host CPU, as compared to the GPU). This lookup may add significant overhead to binding objects if a large number of names are used, especially if identical names are used for different types of objects (e.g., if a texture object and a pixel-buffer object use the same unsigned-integer value).
- In many cases, using indexed vertex arrays rendered with **glDrawElements()** may take advantage of hardware caching of transformed vertices.

- If the situation allows it, use **gl*TexSubImage()** to replace all or part of an existing texture image, rather than the more costly operations of deleting and creating an entire new image.
- If your OpenGL implementation supports a high-performance working set of resident textures, try to make all of your textures resident—that is, make them fit into the high-performance texture memory. If necessary, reduce the size or internal format resolution of your textures until they all fit into memory. If such a reduction creates intolerably fuzzy textured objects, you may give some textures lower priority, which will, when push comes to shove, leave them out of the working set.
- If you need to render the same pixel rectangle multiple times, store the image in a texture map and render it by using a screen-aligned quadrilateral rather than by calling **glDrawPixels().**
- Use a single **glClear()** call per frame if possible. Do not use **glClear()** to clear small subregions of the buffers; use this routine only for complete or nearly complete clears.
- Call **glFinish()** only when you are measuring the execution time of OpenGL rendering. Remove this function from your program once you're done making measurements.
- OpenGL commands are automatically flushed (as if you had called **glFlush**()) when a double-buffered window is swapped.

When using the fixed-function pipeline, here are some performance considerations to keep in mind:

- Use **glColorMaterial()** when only a single material property is being varied rapidly (at each vertex, for example). Use **glMaterial()** for infrequent changes or when more than a single material property is being varied rapidly.
- Use **glLoadIdentity()** to initialize a matrix, rather than load your own copy of the identity matrix.
- Use specific matrix calls such as **glRotate*()**, **glTranslate*()**, and **glScale*()**, rather than compose your own rotation, translation, or scale matrices and call **glMultMatrix()**.
- Use texture objects to encapsulate texture data. Place all the **glTexImage*()** calls (including mipmaps) required to completely specify a texture and the associated **glTexParameter*()** calls (which set texture properties) into a texture object. Bind this texture object to select the texture.

- Use evaluators even for simple surface tessellations to minimize network bandwidth in client–server environments.
- Provide unit-length normals if it's possible to do so, and avoid the overhead of GL_NORMALIZE. Avoid using **glScale*()** when doing lighting because it almost always requires that GL_NORMALIZE be enabled.
- Set **glShadeModel()** to GL_FLAT if smooth shading isn't required.
- Use a single call to **glBegin(**GL_TRIANGLES**)** to draw multiple independent triangles, rather than calling **glBegin(**GL_TRIANGLES**)** multiple times or calling **glBegin(**GL_POLYGON**)**. Even if only a single triangle is to be drawn, use GL_TRIANGLES, rather than GL_POLYGON. Use a single call to **glBegin(**GL_QUADS**)** in the same manner, rather than calling **glBegin(**GL_POLYGON**)** repeatedly. Likewise, use a single call to **glBegin(**GL_LINES**)** to draw multiple independent line segments, rather than calling **glBegin(**GL_LINES**)** multiple times.
- Favor the use of vertex buffer objects over display lists.
- In general, use the vector forms of commands to pass precomputed data, and use the scalar forms of commands to pass values that are computed near call time.
- Be sure to disable expensive rasterization and per-fragment operations when drawing or copying images. OpenGL will even apply textures to pixel images if asked.
- Unless absolutely necessary, avoid having different front and back polygon modes.

When you're using vertex or fragment shaders in your applications, here are some tips that may help improve the application's performance:

- Pack individual scalar values (such as uniform variables or vertex attributes) into vector data types. In the case of uniform variables, when the GLSL linker resolves scalar values, they are assigned unique uniform vector values, even if only a single component of the vector is used. Helping the linker out by making that mapping explicit in your shader may improve performance.
- Using uniform buffer objects (in particular, the shader variety) will help significantly reduce the overhead of loading uniform variables when changing between shaders.

GLX Tips

- Use **glXWaitGL()**, rather than **glFinish()**, to force X rendering commands to follow GL rendering commands.
- Likewise, use **glXWaitX()**, rather than **XSync()**, to force GL rendering commands to follow X rendering commands.
- Be careful when using **glXChooseVisual()**, because Boolean selections are matched exactly. Since some implementations won't export visuals with all combinations of Boolean capabilities, you should call **glXChooseVisual()** several times with different Boolean values before you give up. For example, if no single-buffered visual with the required characteristics is available, check for a double-buffered visual with the same capabilities. It might be available, and it's easy to use.