Chapter 12

Tuning the Pipeline

This chapter looks in some detail at tuning the graphics pipeline. It presents a variety of techniques for optimizing the different parts of the pipeline, providing code fragments and examples as appropriate. You learn about:

CPU Tuning: Basics

The first stage of the rendering pipeline is traversal of the data and sending the current rendering data to the rest of the pipeline. In theory, the entire rendering database (scene graph) must be traversed in some fashion for each frame because both scene content and viewer position can be dynamic.

To get the best possible CPU performance, follow these two overall guidelines:

- Compile your application for optimum speed.

  Compile all object files with at least **-O2**. Note that the compiler option for debugging, **-g**, turns off all optimization. If you must run the debugger on optimized code, you can use **-g3** with **-O2** with limited success. If you're not compiling with **-xansi** (the default) or **-ansi**, you may need to include **-float** for faster floating-point operations.

  On certain platforms, other compile-time options, such as **-mips3** or **-mips4,** are available.

- Use a simple data structure and a fast traversal method.

  The CPU tuning strategy focuses on developing fast database traversal for drawing with a simple, easily accessed data structure. The fastest rendering is achieved with an inner loop that traverses a completely flattened (non-hierarchical) database. Most applications cannot achieve this level of simplicity for a variety of reasons. For example, some databases occupy too much memory when completely flattened. Note also that you run a greater risk of cache misses if you flatten the data.

When an application is CPU limited, the entire graphics pipeline may be sitting idle for periods of time. The following sections describe techniques for structuring application code so that the CPU doesn't become the bottleneck.

Immediate Mode Drawing Versus Display Lists

In deciding whether to use immediate mode drawing or display lists, consider the following advantages and disadvantages of display lists. In the end, the characteristics of your application determine which approach to take.

Display lists have the following advantages:

- You don't have to optimize traversal of the data yourself; in fact, display list traversal is typically well-tuned and runs more efficiently than user programs.
- Display lists manage their own data storage. This is particularly useful for algorithmically generated objects.
- Display lists are significantly better for remote graphics over a network. The display list can be cached on the remote CPU so that the data for the display list does not have to be re-sent every frame. Furthermore, the remote CPU handles much of the responsibility for traversal.
- Display lists are preferable for direct rendering if they contain enough primitives (a total of about ten) because display lists are stored efficiently. If the lists are short, the setup performance cost is not offset by the more efficient storage or saving in CPU time.
- For information on display lists on Indigo2 IMPACT systems, see "Using Display Lists Effectively" on page 280.

Display lists do have drawbacks that may affect some applications:

- The most troublesome drawback of display lists is data expansion. To achieve fast, simple traversal on all systems, all data is copied directly into the display list. Therefore, the display list contains an entire copy of all your data plus additional overhead for each command.
- If vertices are shared in structures more complex than the OpenGL primitives (line strip, triangle strip, triangle fan, quad strip), they are stored more than once.
- If the database becomes sufficiently large, paging eventually hinders performance. Therefore, when contemplating the use of OpenGL display lists for really large databases, consider the amount of main memory.
- Compiling display lists may take some time.

For moderate amounts of static data, there is seldom a good reason not to use display lists. Note the reason that the following section on immediate mode drawing is much longer than the section on display lists: Display lists are optimized by the system, and therefore require less work. That alone may be a good reason to prefer display lists under the appropriate circumstances.

CPU Tuning: Immediate Mode Drawing

Immediate mode drawing means that OpenGL commands are executed when they're called, rather than from a display list. This style of drawing provides flexibility and control over both storage management and drawing traversal. The trade-off for the extra control is that you have to write your own optimized subroutines for data traversal. Tuning therefore has two parts:

- "Optimizing the Data Organization"
- "Optimizing Database Rendering Code"

While you may not use each technique in this section, minimize the CPU work done at the per-vertex level and use a simple data structure for rendering traversal.

There is no recipe for writing a peak-performance immediate mode renderer for a specific application. To predict the CPU limitation of your traversal, design potential data structures and traversal loops and write small benchmarks that mimic the memory demands you expect. Experiment with optimizations and benchmark the effects. Experimenting on small examples can save time in the actual implementation.

Optimizing the Data Organization

It is common for scenes to have hierarchical definitions. Scene management techniques may rely on specific hierarchical information. However, a hierarchical organization of the data raises several performance concerns:

- The time spent traversing pointers to different sections of a hierarchy can create a CPU bottleneck.

  This is partly because of the number of extra instructions executed, but it is also a result of the inefficient use of cache and memory. Overhead data not needed for rendering is brought through the cache and can push out needed data, causing subsequent cache misses.

- Traversing hierarchical structures can cause excessive memory paging.

  Hierarchical structures can be distributed throughout memory. It is difficult to be sure of the exact amount of data you are accessing and where it is located; traversing hierarchical structures can therefore access a costly number of pages.

- Complex operations may need access to both the geometric data and other scene information, complicating the data structure.
- Caching behavior is often difficult to predict for dynamic hierarchical data structures.

For these reasons, hierarchy should be used with care. In general, store the geometry data used for rendering in static, contiguous buffers, rather than in the hierarchical data structures.

- Flatten your rendering data (minimize the number of levels in the hierarchy) as much as cache and memory considerations and your application constraints permit.

  This is an example of a decision that should be guided by the choice of system on which your application will run. For system-specific tuning information, see Chapter 14, “System-Specific Tuning.”

- Balance the data hierarchy. This makes application culling (the process of eliminating objects that don’t fall within the viewing frustum) more efficient and effective.

Optimizing Database Rendering Code

This section includes some suggestions for writing peak-performance code for inner rendering loops.

Ideally, an application spends most of its time traversing the database and sending data to the graphics pipeline. Instructions in the display loop are executed many times every frame, creating hot spots. Any extra overhead in a hot spot is greatly magnified by the number of times it is executed.

When using simple, high-performance graphics primitives, the application is even more likely to be CPU limited. The data traversal must be optimized so that it does not become a bottleneck.

During rendering, the sections of code that actually issue graphics commands should be the hot spots in application code. These subroutines should use peak-performance coding methods. Small improvements to a line that is executed for every vertex in a database accumulate to have a noticeable effect when the entire frame is rendered.

The rest of this section looks at examples and techniques for optimizing immediate-mode rendering:

- "Examples for Optimizing Data Structures for Drawing"
- "Examples for Optimizing Program Structure"
- "Using Specialized Drawing Subroutines and Macros"
- "Preprocessing Drawing Data: Introduction"
- "Preprocessing Meshes Into Fixed-Length Strips"
- "Preprocessing Vertex Loops"

Examples for Optimizing Data Structures for Drawing

Follow these suggestions for optimizing how your application accesses data:

- **One-Dimensional Arrays**. Use one-dimensional arrays traversed with a pointer that always holds the address for the current drawing command. Avoid array-element addressing or multidimensional array accesses.

  ```
  bad:  glVertex3fv(&data[i][j][k]);
  good: glVertex3fv(dataptr);
  ```

- **Adjacent structures**. Keep all static drawing data for a given object together in a single contiguous array traversed with a single pointer. Keep this data separate from other program data, such as pointers to drawing data, or interpreter flags.
- **Flat structures.** Use flat data structures and do not use multiple pointer indirection when rendering:

```
bad:    glVertex3fv(object->data->vert);
ok:     glVertex3fv(dataptr->vert);
best:   glVertex3fv(dataptr);
```

The following code fragment is an example of efficient code to draw a single smooth-shaded, lit polygon from quad-word aligned data. Notice that a single data pointer is used. It is updated once at the end of the polygon, after the **glEnd()** call.

```
glBegin(GL_QUADS);
glNormal3fv(ptr);
glVertex3fv(ptr+4);
glNormal3fv(ptr+8);
glVertex3fv(ptr+12);
glNormal3fv(ptr+16);
glVertex3fv(ptr+20);
glNormal3fv(ptr+24);
glVertex3fv(ptr+28);
glEnd();
ptr += 32;
```

Examples for Optimizing Program Structure

- **Loop unrolling (1).** Avoid short, fixed-length loops, especially around vertices. Instead, unroll these loops:

```
bad:
    for(i=0; i < 4; i++){
        glColor4ubv(poly_colors[i]);
        glVertex3fv(poly_vert_ptr[i]);
    }

good:
    glColor4ubv(poly_colors[0]);
    glVertex3fv(poly_vert_ptr[0]);
    glColor4ubv(poly_colors[1]);
    glVertex3fv(poly_vert_ptr[1]);
    glColor4ubv(poly_colors[2]);
    glVertex3fv(poly_vert_ptr[2]);
    glColor4ubv(poly_colors[3]);
    glVertex3fv(poly_vert_ptr[3]);
```

- **Loop unrolling (2).** Minimize the work done in a loop to maintain and update variables and pointers. Unrolling can often assist in this:

  bad:
  ```
  glNormal3fv(*(ptr++)); glVertex3fv(*(ptr++));
  ```
  or
  ```
  glNormal3fv(ptr); ptr += 4;
  glVertex3fv(ptr); ptr += 4;
  ```
  good:
  ```
  glNormal3fv(*(ptr)); glVertex3fv(*(ptr+1));
  glNormal3fv(*(ptr+2)); glVertex3fv(*(ptr+3));
  ```
  or
  ```
  glNormal3fv(ptr); glVertex3fv(ptr+4); glNormal3fv(ptr+8);
  glVertex3fv(ptr+12);
  ```
  Note: On some processors, such as the R8000,™ loop unrolling may hurt performance more than it helps, so use it with caution. In fact, unrolling too far hurts on any processor because the loop may use an excessive portion of the cache. If it uses a large enough portion of the cache, it may interfere with itself; that is, the whole loop won't fit (not likely) or it may conflict with the instructions of one of the subroutines it calls.

- **Loops accessing buffers.** Minimize the number of different buffers accessed in a loop:

  bad:
  ```
  glNormal3fv(normaldata);
  glTexCoord2fv(texdata);
  glVertex3fv(vertdata);
  ```
  good:
  ```
  glNormal3fv(dataptr);
  glTexCoord2fv(dataptr+4);
  glVertex3fv(dataptr+8);
  ```

- **Loop end conditions.** Make end conditions on loops as trivial as possible; for example, compare the loop variable to a constant, preferably zero. Decrementing loops are often more efficient than their incrementing counterparts:

  bad:
  ```
  for (i = 0; i < (end-beginning)/size; i++)
      {...}
  ```
  better:
  ```
  for (i = beginning; i < end; i += size)
      {...}
  ```

```
good:
    for (i = total; i > 0; i--)
        {...}
```

- **Conditional statements.**
  - Use **switch** statements instead of multiple **if-else-if** control structures.
  - Avoid **if** tests around vertices; use duplicate code instead.
- **Division.** Avoid division. Shift or multiply by a reciprocal instead:

  `f = x * 0.5 instead of f = x / 2.0`

  Integer division is even slower than floating-point division.

  `i = j >> 1 instead of i = j/2`
- **Subroutine prototyping.** Prototype subroutines in ANSI C style to avoid runtime typecasting of parameters:

```
void drawit(float f, int count)
{
  .......
}
```

- **Typecasting.** Avoid typecasting of values, which happens at run-time:

  `val = (float) *f;`

  Instead, use typecasting of pointers, which occurs at compile time and is efficient:

```
int *ptr;
*(float *) ptr = float_val;
float_val = *(float *) ptr;
```

- **Multiple polygons.** Send multiple polygons between **glBegin()/glEnd()** whenever possible:

```
glBegin(GL_TRIANGLES)
....
..../* many triangles */
....
glEnd

glBegin(GL_QUADS)
....
..../* many quads */
....
glEnd
```

Using Specialized Drawing Subroutines and Macros

This section looks at several ways to improve performance by making appropriate choices about display modes, geometry, and so on.

- **Geometry display choices.** Make decisions about which geometry to display and which modes to use at the highest possible level in the program organization.

  The drawing subroutines should be highly specialized leaves in the program's call tree. Decisions made too far down the tree can be redundant. For example, consider a program that switches back and forth between flat-shaded and smooth-shaded drawing. Once this choice has been made for a frame, the decision is fixed and the flag is set. For example, the following code is inefficient:

```
/* Inefficient way to toggle modes */
draw_object(float *data, int npolys, int smooth)  {
int i;
glBegin(GL_QUADS);
for (i = npolys; i > 0; i--) {
    if (smooth) glColor3fv(data);
    glVertex3fv(data + 4);
    if (smooth) glColor3fv(data + 8);
    glVertex3fv(data + 12);
    if (smooth) glColor3fv(data + 16);
    glVertex3fv(data + 20);
    if (smooth) glColor3fv(data + 24);
    glVertex3fv(data + 28);
}
glEnd();
```

  Even though the program chooses the drawing mode before entering the **draw_object()** routine, the flag is checked for every vertex in the scene. A simple **if** test may seem innocuous; however, when done on a per-vertex basis, it can accumulate a noticeable amount of overhead.

  Compare the number of instructions in the disassembled code for a call to **glColor3fv()**, first without, and then with, the **if** test.

  Assembly code for a call without **if** test (six instructions):

```
lw a0,32(sp)
lw t9,glColor3fv
addiu a0,a0,32
jalr ra,t9
nop
lw gp,24(sp)
```

Assembly code for a call with an **if** test (eight instructions):

```
lw t7,40(sp)
beql t7,zero,0x78
nop
lw t9,glColor3fv
lw a0,32(sp)
jalr ra,t9
addiu a0,a0,32
lw gp,24(sp)
```

Notice the two extra instructions required to implement the **if** test. The extra **if** test per vertex increases the number of instructions executed for this otherwise optimal code by 33%. These effects may not be visible if the code is used only to render objects that are always graphics limited. However, if the process is CPU-limited, then moving decision operations such as this **if** test higher up in the program structure improves performance.

Preprocessing Drawing Data: Introduction

Putting some extra effort into generating a simpler database makes a significant difference when traversing that data for display. A common tendency is to leave the data in a format that is good for loading or generating the object, but not optimal for actually displaying it. For peak performance, do as much of the work as possible before rendering.

Preprocessing turns a difficult database into a database that is easy to render quickly. This is typically done at initialization or when changing from a modeling to a fast-rendering mode. This section discusses "Preprocessing Meshes Into Fixed-Length Strips" and "Preprocessing Vertex Loops" to illustrate this point.

Preprocessing Meshes Into Fixed-Length Strips

Preprocessing can be used to turn general meshes into fixed-length strips.

The following sample code shows a commonly used, but inefficient, way to write a triangle strip render loop:

```
float* dataptr;
...
while (!done) switch(*dataptr) {
    case BEGINSTRIP:
        glBegin(GL_TRIANGLE_STRIP);
        dataptr++;
        break;
    case ENDSTRIP:
        glEnd();
        dataptr++;
        break;
    case EXIT:
        done = 1;
        break;
    default: /* have a vertex !!! */
        glNormal3fv(dataptr);
        glVertex3fv(dataptr + 4);
        dataptr += 8;
}
```

This traversal method incurs a significant amount of per-vertex overhead. The loop is evaluated for every vertex and every vertex must also be checked to make sure that it is not a flag. This wastes time and also brings all of the object data through the cache. This practice reduces the performance advantage of using triangle strips. Any variation of this code that has per-vertex overhead is likely to be CPU limited for most types of simple graphics operations.

Preprocessing Vertex Loops

Preprocessing is also possible for vertex loops:

```
glBegin(GL_TRIANGLE_STRIP);
for (i=num_verts; i > 0; i--) {
    glNormal3fv(dataptr);
    glVertex3fv(dataptr+4);
    dataptr += 8;
    }
glEnd();
```

For peak immediate mode performance, precompile strips into specialized primitives of fixed length. Only a few fixed lengths are needed. For example, use strips that consist of 12, 8, and 2 primitives.

Note: The optimal length may vary depending on the hardware the program runs on. For more information, see Chapter 14, “System-Specific Tuning.”

These specialized strips are then sorted by size, resulting in the efficient loop shown in this sample code:

```
/* dump out N 8-triangle strips */
for (i=N; i > 0; i--) {
    glBegin(GL_TRIANGLE_STRIP);
    glNormal3fv(dataptr);
    glVertex3fv(dataptr+4);
    glNormal3fv(dataptr+8);
    glVertex3fv(dataptr+12);
    glNormal3fv(dataptr+16);
    glVertex3fv(dataptr+20);
    glNormal3fv(dataptr+24);
    glVertex3fv(datatpr+28);
    ...
    glEnd();
    dataptr += 64;
}
```

A mesh of length 12 is about the maximum for unrolling. Unrolling helps to reduce the overall cost-per-loop overhead, but after a point, it produces no further gain.

Note: Over-unrolling eventually hurts performance by increasing code size and reducing effectiveness of the instruction cache. The degree of unrolling depends on the processor; run some benchmarks to understand the optimal program structure on your system.

CPU Tuning: Display Lists

In display-list mode, pieces of the database are compiled into static chunks that can then be sent to the graphics pipeline. In this case, the display list is a separate copy of the database that can be stored in main memory in a form optimized for feeding the rest of the pipeline. The database traversal task is to hand the correct chunks to the graphics pipeline. Display lists can be recreated easily with some additional performance cost.

Tuning for display lists focuses mainly on reducing storage requirements. Performance improves if the data fit in the cache because this avoids cache misses when the data is retraversed.

Follow these rules to optimize display lists:

- If possible, compile and execute a display list in two steps instead of using GL_COMPILE_AND_EXECUTE.
- Call **glDeleteLists()** to delete display lists that are no longer needed.

  This frees storage space used by the deleted display lists and expedites the creation of new display lists.
- Avoid duplication of display lists.

  For example, if you have a scene with 100 spheres of different sizes and materials, generate one display list that is a unit sphere centered about the origin. Then for each sphere in the scene, follow these steps:

  1. Set the material for the current sphere.
  2. Issue the necessary scaling and translation commands for sizing and positioning the sphere.
  3. Invoke **glCallList()** to draw the unit sphere display list.

  In this way, a reference to the unit sphere display list is stored instead of all of the sphere vertices for each instance of the sphere.
- Make the display list as flat as possible, but be sure not to exceed the cache size.

  Avoid using an excessive hierarchy with many invocations of **glCallList()**. Each **glCallList()** invocation causes a table lookup to find the designated display list. A flat display list requires less memory and yields simpler and faster traversal. It also improves cache coherency.

  On the other hand, excessive flattening increases the size. For example, if you're drawing a car with four wheels, having a hierarchy with four pointers from the body to one wheel is preferable to a flat structure with one body and four wheels.

Display lists are best used for static objects. Do not put dynamic data or operations in display lists. Instead, use a mixture of display lists for static objects and immediate mode for dynamic operations.

Note: See Chapter 14, "System-Specific Tuning," for potential display list optimizations on the system you are using.

CPU Tuning: Advanced Techniques

After you've applied the techniques discussed in the previous sections, consider using these advanced techniques to tune CPU-limited applications:

- "Mixing Computation With Graphics"
- "Examining Assembly Code"
- "Using Additional Processors for Complex Scene Management"
- "Modeling to the Graphics Pipeline"

Mixing Computation With Graphics

When you are fine-tuning an application, interleaving computation and graphics can make it better balanced and therefore more efficient. Key places for interleaving are after **glXSwapBuffers()**, **glClear()**, and drawing operations that are known to be fill limited (such as drawing a backdrop or a ground plane or any other large polygon).

A **glXSwapBuffers()** call creates a special situation. After calling **glXSwapBuffers()**, an application may be forced to wait for the next vertical retrace (in the worst case, up to 16.7 msecs) before it can issue more graphics calls. For a program drawing 10 frames per second, 15% of the time (worst case) can be spent waiting for the buffer swap to occur.

In contrast, non-graphic computation is not forced to wait for a vertical retrace. Therefore, if there is a section of computation that must be done every frame that includes no graphics calls, it can be done after the **glXSwapBuffers()** instead of causing a CPU limitation during drawing.

Clearing the screen is a time-consuming operation. Doing non-graphics computation immediately after the clear is more efficient than sending additional graphics requests down the pipeline and being forced to wait when the pipeline's input queue overflows.

Experimentation is required to

- determine where the application is reliably graphics limited
- ensure that inserting the computation does not create a new bottleneck

For example, if a new computation references a large section of data that is not in the data cache, the data for drawing may be swapped out for the computation, then swapped back in for drawing, resulting in worse performance than the original organization.

Examining Assembly Code

When tuning inner rendering loops, examining assembly code can be helpful. Use *dis* to disassemble optimized code for a given procedure, and correlate assembly code lines with line numbers from the source code file. This is especially helpful for examining optimized code. The **-S** option to *cc* produces a *.s* file of assembly output, complete with your original comments.

You need not be an expert in MIPS assembly code to interpret the results. Just looking at the number of extra instructions required for an apparently innocuous operation is informative. Knowing some basics about MIPS assembly code can be helpful for finding performance bugs in inner loops. See *MIPS RISC Architecture*, by Gerry Kane, listed in "Background Reading" on page xxv for additional information.

Using Additional Processors for Complex Scene Management

If your application is running on systems with multiple processors, consider supplying an option for doing scene management on additional processors to relieve the rendering processor from the burden of expensive computation.

Using additional processors may also reduce the amount of data rendered for a given frame. Simplifying or reducing rendering for a given scene can help reduce bottlenecks in all parts of the pipeline, as well as the CPU. One example is removing unseen or backfacing objects. Another common technique is to use an additional processor to determine when objects are going to appear very far away and use a simpler model with fewer polygons and less expensive modes for distant objects. This is known as level-of-detail rendering.

Modeling to the Graphics Pipeline

The modeling of the database directly affects the rendering performance of the resulting application and therefore needs to match the performance characteristics of the graphics pipeline and make trade-offs with the database traversals. Graphics pipelines that support connected primitives, such as triangle meshes, benefit from having long meshes in the database. However, the length of the meshes affects the resulting database hierarchy, and long strips through the database do not cull well with simple bounding geometry.

Model objects with an understanding of inherent bottlenecks in the graphics pipeline:

- Pipelines that are severely fill-limited benefit from having objects modeled with cut polygons and more vertices and fewer overlapping parts, which decreases depth complexity.
- Pipelines that are easily geometry- or host-limited benefit from modeling with fewer polygons.

There are several other modeling tricks that can reduce database complexity:

- Use textured polygons to simulate complex geometry. This is especially useful if the graphics subsystem supports the use of textures where the alpha component of the texture marks the transparency of the object. Textures can be used as cut-outs for objects like fences and trees.
- Use textures for simulating particles, such as smoke.
- Use textured polygons as single-polygon billboards. Billboards are polygons that are fixed at a point and rotated about an axis, or about a point, so that the polygon always faces the viewer. Billboards are useful for symmetric objects such as light posts and trees, and also for volume objects such as smoke. Billboards can also be used for distant objects to save geometry. However, the managing of billboard transformations can be expensive and affect both the cull and the draw processes.

Tuning the Geometry Subsystem

This section presents techniques that you can use to tune the geometry subsystem, discussing the following topics:

- "Using Fast Drawing Modes"
- "Using Peak Performance Primitives for Drawing"
- "Optimizing Lighting Performance"
- "Using Expensive Modes Efficiently"
- "Advanced Transform-Limited Tuning Techniques"

Using Fast Drawing Modes

Use flat shading whenever possible. This reduces the number of lighting computations from one per vertex to one per primitive, and also reduces the amount of data that must be passed from the CPU through the graphics pipeline for each primitive. This is particularly important for high-performance line drawing.

Using Peak Performance Primitives for Drawing

This section describes how to draw geometry with optimal primitives. Consider these guidelines to optimize drawing:

- Use connected primitives (line strips, triangle strips, triangle fans, and quad strips). Put at least 8 primitives in a sequence, 12 to 16 if possible.

  Connected primitives are desirable because they reduce the amount of data sent to the graphics subsystem and the amount of per-polygon work done in the pipeline. Typically, about 12 vertices per **glBegin()/glEnd()** are required to achieve peak rates (but this can vary depending on the hardware you're running on). For lines and points, it is especially beneficial to put as many vertices as possible in a **glBegin()/glEnd()** sequence. For information on the most efficient vertex numbers for the system you are using, check Chapter 14, "System-Specific Tuning."

- Use "well-behaved" polygons—convex and planar, with only three or four vertices.

  Concave and self-intersecting polygons must be tessellated by GLU before they can be drawn, and are therefore prohibitively expensive. Nonplanar polygons and polygons with large numbers of vertices are more likely to exhibit shading artifacts.

  If your database has polygons that are not well-behaved, perform an initial one-time pass over the database to transform the troublemakers into well-behaved polygons and use the new database for rendering. Using connected primitives results in additional gains.

- Minimize the data sent per vertex.

  Polygon rates can be affected directly by the number of normals or colors sent per polygon. Setting a color or normal per vertex, regardless of the **glShadeModel()** used, may be slower than setting only a color per polygon, because of the time spent sending the extra data and resetting the current color. The number of normals and colors per polygon also directly affects the size of a display list containing the object.

- Group like primitives.

Optimizing Lighting Performance

OpenGL offers a large selection of lighting features: Some are virtually "free" in terms of computational time, others offer sophisticated effects with some performance penalty. The penalties some features carry may vary depending on the hardware you're running on. Be prepared to experiment with the lighting configuration.

As a general rule, use the simplest possible lighting model, a single infinite light with an infinite viewer. For some local effects, try replacing local lights with infinite lights and a local viewer.

You normally won't notice a performance degradation when using one infinite light, unless you use lit textures or color index lighting.

Use the following settings for peak performance lighting:

- single infinite light
- RGB mode
- nonlocal viewing—GL_LIGHT_MODEL_LOCAL_VIEWER set to GL_FALSE in **glLightModel()** (the default)
- single-sided lighting—GL_LIGHT_MODEL_TWO_SIDE set to GL_FALSE in **glLightModel()** (the default)
- GL_COLOR_MATERIAL disabled
- GL_NORMALIZE disabled—because this is usually necessary when the model-view matrix includes a scaling transformation, consider preprocessing the scene to eliminate scaling.

Lighting Operations With Noticeable Performance Costs

Follow these guidelines to achieve peak lighting performance:

- Avoid using multiple lights.

  There may be a sharp drop in lighting performance when switching from one light to two lights, but the drop for additional lights is likely to be more gradual.
- Avoid using local lights.

  Local lights are noticeably more expensive than infinite lights.

- Don’t change material parameters frequently.

  Changing material parameters can be expensive. If you need to change the material parameters many times per frame, consider rearranging the scene traversal to minimize material changes. Also consider using **glColorMaterial()** to change specific parameters automatically, rather than using **glMaterial()** to change parameters explicitly.

  The following code fragment illustrates how to change ambient and diffuse material parameters at every polygon or at every vertex:

```
glColorMaterial(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE);
glEnable(GL_COLOR_MATERIAL);
...
/* Set ambient and diffuse material parameters: */
glColor4f(red, green, blue, alpha);
/* Draw triangles: */
glBegin(GL_TRIANGLES);
...
glEnd();
```

Lighting Operations With Significant Performance Costs

The features described in this section are classified as having significant costs because any one of them, when added to high-performance lighting, causes a substantial drop in performance. However, while adding more features has still higher cost, the additional penalty is small compared to the initial drop from high-performance lighting. Follow these guidelines to limit use of lighting features that significantly reduce performance:

- Use expensive features with care.

  Setting GL_LIGHT_MODEL_LOCAL_VIEWER to GL_TRUE with **glLightModel()**, while using infinite lights only, reduces performance by a small amount. However, each additional local light noticeably degrades the transform rate.

  Two-sided lighting illuminates both sides of a polygon. This is much faster than the alternative of drawing polygons twice. However, using two-sided lighting is significantly slower than one-sided lighting for a single rendering of an object.

- Use unit-length normals.
- Avoid scaling operations if possible.
- Avoid changing the GL_SHININESS material parameter if possible. Setting a new GL_SHININESS value requires significant computation each time.

Practices to Avoid

Avoid using lighting calls inside a **glBegin()/glEnd()** sequence.

If possible, avoid calls to **glMaterial()** during a **glBegin()/glEnd()** drawing sequence, as this has a serious performance impact. While making such calls to change colors by changing material properties is possible, the performance penalty makes it unadvisable. Use **glColorMaterial()** instead.

Using Expensive Modes Efficiently

OpenGL offers many features that create sophisticated effects with excellent performance. However, these features have some performance cost, compared to drawing the same scene without them. Use these features only where their effects, performance, and quality are justified.

- Turn off features when they are not required.

  Once a feature has been turned on, it can slow the transform rate even when it has no visible effect.

  For example, the use of fog can slow the transform rate of polygons even when the polygons are too close to show fog, and even when the fog density is set to zero. For these conditions, turn off fog explicitly with

  ```
  glDisable(GL_FOG)
  ```

- Minimize expensive mode changes and sort operations by the most expensive mode. Specifically, consider these tips:
  - Use small numbers of texture maps to avoid the cost of switching between textures. If you have many small textures, consider combining them into a single larger, tiled texture. Rather than switching to a new texture before drawing a textured polygon, choose texture coordinates that select the appropriate small texture tile within the large texture.
  - Avoid changing the projection matrix or changing **glDepthRange()** parameters.
  - When fog is enabled, avoid changing fog parameters.
  - Turn fog off for rendering with a different projection (for example, orthographic) and turn it back on when returning to the normal projection.
- Beware of excessive mode changes, even mode changes considered cheap, such as changes to shade model, depth buffering, and blending function.

Advanced Transform-Limited Tuning Techniques

This section describes advanced techniques for tuning transform-limited drawing. Follow these guidelines to draw objects with complex surface characteristics:

- Use texture to replace complex geometry.

  Textured polygons can be significantly slower than their non-textured counterparts. However, texture can be used instead of extra polygons to add detail to a geometric object. This can greatly simplify geometry, resulting in a net speed increase and an improved picture, as long as it does not cause the program to become fill-limited. Texturing performance varies across the product line, so this technique might not be equally effective on all systems. Experimentation is usually necessary.

- Use **glAlphaFunc()** in conjunction with one or more textures to give the effect of rather complex geometry on a single polygon.

  Consider drawing an image of a complex object by texturing it onto a single polygon. Set alpha values to zero in the texture outside the image of the object. (The edges of the object can be antialiased by using alpha values between zero and one.) Orient the polygon to face the viewer. To prevent pixels with zero alpha values in the textured polygon from being drawn, call:

  ```
  glAlphaFunc(GL_NOTEQUAL, 0.0)
  ```

  This effect is often used to create objects like trees that have complex edges or many holes through which the background should be visible (or both).

- Use culling on a separate processor to eliminate objects or polygons that will be out of sight or too small to see.

Tuning the Raster Subsystem

An explosion of both data and operations is required to rasterize a polygon as individual pixels. Typically, the operations include depth comparison, Gouraud shading, color blending, logical operations, texture mapping, and possibly antialiasing. This section discusses the following techniques for tuning fill-limited drawing:

- “Using Backface/Frontface Removal”
- “Using Expensive Per-Fragment Operations Efficiently”
- “Balancing Polygon Size and Pixel Operations”
- “Clearing the Color and Depth Buffers Simultaneously”

Using Backface/Frontface Removal

To reduce fill-limited drawing, use backface and frontface removal. For example, if you are drawing a sphere, half of its polygons are backfacing at any given time. Backface and frontface removal is done after transformation calculations but before per-fragment operations. This means that backface removal may make transform-limited polygons somewhat slower, but make fill-limited polygons significantly faster. You can turn on backface removal when you are drawing an object with many backfacing polygons, then turn it off again when drawing is completed.

Using Expensive Per-Fragment Operations Efficiently

Use expensive per-fragment operations with care. Per-fragment operations, in order of increasing cost (with flat-shading being the lowest and multisampling the highest) are

1. flat-shading
2. Gouraud shading
3. depth buffering
4. alpha blending
5. texturing
6. multisampling

Each operation can independently slow down the pixel fill rate of a polygon, although depth buffering can help reduce the cost of alpha blending or multisampling for hidden polygons.

Some of this information depends on the particular system the program is running on:

- Texturing is less expensive than alpha blending on new-generation hardware only.
- Alpha blending is less expensive than depth buffering on Indy systems.
- Beware of fill operations that are executed on the host for your graphics platform (for example, texturing on Extreme™ or Elan™ graphics).

Using Depth-Buffering Efficiently

Any rendering operation can become fill-limited for large polygons. Clever structuring of drawing can eliminate the need for certain fill operations. For example, if large backgrounds are drawn first, they do not need to be depth buffered. It is better to disable depth buffering for the backgrounds and then enable it for other objects where it is needed.

For example, flight simulators use this technique. The sky and ground are drawn with depth buffering disabled, then the polygons lying flat on the ground (runway and grid) are drawn without suffering a performance penalty. Finally, depth buffering is enabled for drawing the mountains and airplanes.

There are many other special cases in which depth buffering might not be required. For example, terrain, ocean waves, and 3D function plots are often represented as height fields (X-Y grids with one height value at each lattice point). It's straightforward to draw height fields in back-to-front order by determining which edge of the field is furthest away from the viewer, then drawing strips of triangles or quadrilaterals parallel to that starting edge and working forward. The entire height field can be drawn without depth testing provided it doesn't intersect any piece of previously-drawn geometry. Depth values need not be written at all, unless subsequently-drawn depth buffered geometry might intersect the height field; in that case, depth values for the height field should be written, but the depth test can be avoided by calling

```
glDepthFunc(GL_ALWAYS)
```

Other Considerations

- Use alpha blending with discretion.

  Alpha blending is an expensive operation. A common use of alpha blending is for transparency, where the alpha value denotes the opacity of the object. For fully opaque objects, disable alpha blending.

- Avoid unnecessary per-fragment operations.

  Turn off per-fragment operations for objects that do not require them, and structure the drawing process to minimize their use without causing excessive toggling of modes.

- Organize drawing to minimize fill operations.

  For example, if a scene has large background polygons, draw them first without depth buffering, then render the more complex depth-buffered objects.

Balancing Polygon Size and Pixel Operations

The pipeline is generally optimized for polygons that are 10 pixels on a side. However, you may need to work with polygons larger or smaller than that depending on the other operations going on in the pipeline:

- If the polygons are too large for the fill-rate to keep up with the rest of the pipeline, the application is fill-rate limited. Smaller polygons balance the pipeline and increase the polygon rate.
- If the polygons are too small for the rest of the pipeline to keep up with filling, then the application is transform limited. Larger and fewer polygons, or fewer vertices, balance the pipeline and increase the fill rate.

Use the simplest possible fill algorithms for drawing very large polygons such as backgrounds.

Clearing the Color and Depth Buffers Simultaneously

The most basic per-frame operations are clearing the color and depth buffers. On some systems, there are optimizations for common, special, cases of these operations.

Whenever you need to clear both the color and depth buffers, don't clear each buffer independently. Instead use

```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
```

Tuning the Imaging Pipeline

This section briefly lists some ways in which you can improve pixel processing. Example 13-1 on page 249 provides a code fragment that shows how to set the OpenGL state so that subsequent calls to **glDrawPixels()** or **glCopyPixels()** will be fast.

To improve performance in the imaging pipeline, follow these guidelines:

- Byte-sized components, particularly unsigned byte components, are fast. Use pixel formats where each of the components (red, green, blue, alpha, luminance, or intensity) is 8 bits long.
- Use fewer components, for example, use GL_LUMINANCE_ALPHA or GL_LUMINANCE.

- Use color matrix and color mask to store four luminance values in the RGBA framebuffer. Use color matrix and color mask to work with one component at a time. If one component is being processed, convolution is much more efficient.

  The following code fragment uses the green component as the data source and writes the result of the operation into some (possibly all) of the other components:

```
/* Matrix is in column major order */
GLfloat smearGreenMat[16] = {
    0, 0, 0, 0,
    1, 1, 1, 1,
    0, 0, 0, 0,
    0, 0, 0, 0,
};
/* The variables update R/G/B/A indicate whether the
* corresponding component would be updated.
*/
GLboolean updateR, updateG, updateB, updateA;

...

/* Check for availability of the color matrix extension */

/* Set proper color matrix and mask */
glMatrixMode(GL_COLOR);
glLoadMatrixf(smearGreenMat);
glColorMask(updateR, updateG, updateB, updateA);

/* Perform the imaging operation */
glEnable(GL_SEPARABLE_2D_EXT);
glCopyTexSubImage2DEXT(...);
/* Restore an identity color matrix.  Not needed when the same
* smear operation is to used over and over
*/
glLoadIdentity();

/* Restore previous matrix mode (assuming it is modelview) */
glMatrixMode(GL_MODELVIEW);
...
```

- Load the identity matrix into the color matrix to turn it off.

  When using the color matrix to broadcast one component into all others, avoid manipulating the color matrix with transformation calls such as **glRotate()**. Instead, load the matrix explicitly using **glLoadMatrix().**

- Know where the bottleneck is.

  Similar to polygon drawing, there can be a pixel-drawing bottleneck due to overload in host bandwidth, processing, or rasterizing. When all modes are off, the path is most likely limited by host bandwidth, and a wise choice of host pixel format and type pays off tremendously. This is also why byte components are sometimes faster. For example, use packed pixel format GL_RGB5_A1_EXT to load texture with an GL_RGB5_A1_EXT internal format.

  When either many processing modes or a several expensive modes such as convolution are on, the processing stage is the bottleneck. Such cases benefit from one-component processing, which is much faster than multicomponent processing.

  Zooming up pixels may create a raster bottleneck.

- For simple loading, turn all pixel modes off. Turn depth test and texture off, when possible.
- A big pixel rectangle has a higher throughput (that is, pixels per second) than a small rectangle. Because the imaging pipeline is tuned to trade off a relatively large setup time with a high throughput pixel transfer efficiency, a large rectangle amortizes the setup cost over many pixels, resulting in higher throughput.
- Having no mode changes between pixel operations results in higher throughput. New high-end hardware detects pixel mode changes between pixel operations: when there is no mode change between pixel operations, the setup operation is drastically reduced. This is done to optimize for image tiling where an image is painted on the screen by drawing many small tiles.