Chapter 11

Tuning Graphics Applications: Fundamentals

Tuning your software makes it use hardware capabilities more effectively. This chapter looks at tuning graphics applications. It discusses pipeline tuning as a conceptual framework for tuning graphics applications, and introduces some other fundamentals of tuning:

- "Why Is Tuning Useful?" on page 210
- "What Is Pipeline Tuning?" on page 210
- "Tuning Animation" on page 214
- "Optimizing Cache and Memory Use" on page 215
- "Taking Timing Measurements" on page 218

Writing high-performance code is usually more complex than just following a set of rules. Most often, it involves making trade-offs between special functions, quality, and performance for a particular application. For more information about the issues you need to consider, and for a tuning example, look at the following chapters in this book:

- Chapter 12, "Tuning the Pipeline"
- Chapter 13, "Tuning Graphics Applications: Examples"
- Chapter 14, "System-Specific Tuning"

After reading these chapters, experiment with the different techniques described to help you decide where to make these trade-offs.

Note: If optimum performance is crucial, consider using the IRIS Performer rendering toolkit. See "Maximizing Performance With IRIS Performer" on page 8.

Why Is Tuning Useful?

Because the hardware of Silicon Graphics systems is so fast, you may think that tuning your application won't make a significant difference in perceived performance. This is often not the case: both the capabilities of your system and the speed at which your application runs are limiting factors. Even the fastest machine can render only as fast as the application can drive it. Simple changes in application code can therefore make a dramatic difference in rendering time. In addition, the flexibility of Silicon Graphics systems provides the opportunity to make the most appropriate tradeoffs between image quality and performance for your application.

What Is Pipeline Tuning?

Traditional software tuning focuses on finding and tuning hot spots, the 10% of the code in which a program spends 90% of its time. Pipeline tuning uses a different approach: it looks for bottlenecks, overloaded stages that are holding up other processes.

At any time, one stage of the pipeline is the bottleneck. Reducing the time spent in the bottleneck is the only way to improve performance. Speeding up operations in other parts of the pipeline has no effect. Conversely, doing work that further narrows the bottleneck, or that creates a new bottleneck somewhere else, is the only thing that further degrades performance. The workload can be increased at other parts of the pipeline without degrading performance, as long as that part does not become a new bottleneck. In this way, an application can sometimes be altered to draw a higher-quality image with no performance degradation.

Different programs stress different parts of the pipeline, so it's important to understand which elements in the graphics pipeline are the bottlenecks for your program.

Three-Stage Model of the Graphics Pipeline

The graphics pipeline in all Silicon Graphics workstations consists of three conceptual stages (see Figure 11-1):

1. **The CPU subsystem.** The application program running on the CPU, feeding commands to the graphics subsystem.
2. **The geometry subsystem.** The per-polygon operations, such as coordinate transformations, lighting, texture coordinate generation, and clipping.

3. **The raster subsystem.** The per-pixel operations, such as the simple operation of writing color values into the framebuffer, or more complex operations like depth buffering, alpha blending, and texture mapping.

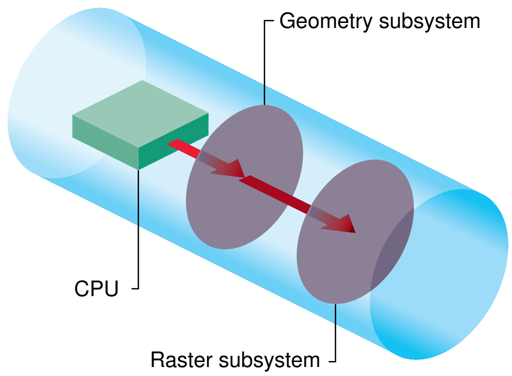

Figure 11-1 Three-Stage Model of the Graphics Pipeline

Note that on some platforms, part of the geometry subsystem is implemented within the CPU subsystem, but the conceptual model is still useful. Note also that this three-stage model is simpler than the actual hardware implementation in the various models in the Silicon Graphics product line, but it is detailed enough for all but the most subtle tuning tasks.

The amount of work required from the different pipeline stages varies among applications. For example, consider a program that draws a small number of large polygons. Because there are only a few polygons, the pipeline stage that does geometry operations is lightly loaded. Because those few polygons cover many pixels on the screen, the pipeline stage that does rasterization is heavily loaded.

To speed up this program, you must speed up the rasterization stage, either by drawing fewer pixels, or by drawing pixels in a way that takes less time by turning off modes like texturing, blending, or depth-buffering. In addition, because spare capacity is available in the per-polygon stage, you can increase the work load at that stage without degrading performance. For example, you can use a more complex lighting model, or define geometries such that they remain the same size but look more detailed because they are composed of a larger number of polygons.

Isolating Bottlenecks in the Pipeline: Overview

The basic strategy for isolating bottlenecks is to measure the time it takes to execute a program (or part of a program) and then change the code in ways that don't alter its performance (except by adding or subtracting work at a single point in the graphics pipeline.) If changing the amount of work at a given stage of the pipeline does not alter performance appreciably, that stage is not the bottleneck. If there is a noticeable difference in performance, you've found a bottleneck.

- **CPU bottlenecks.** The most common bottleneck occurs when the application program does not feed the graphics subsystem fast enough. Such programs are called *CPU limited*.

  To test for a CPU bottleneck, remove as much graphics work as possible, while preserving the behavior of the application in terms of the number of instructions executed and the way memory is accessed. Often, changing just a few OpenGL calls is a sufficient test. For example, replacing vertex and normal calls like **glVertex3fv()** and **glNormal3fv()** with color subroutine calls like **glColor3fv()** preserves the CPU behavior while eliminating all drawing and lighting work in the graphics pipeline. If making these changes does not significantly improve performance, then your application has a CPU bottleneck. For more information, see "CPU Tuning: Basics" on page 223.

- **Geometry bottlenecks.** Programs that create bottlenecks in the geometry (per-polygon) stage are called *transform limited*. To test for bottlenecks in geometry operations, change the program so that the application code runs at the same speed and the same number of pixels are filled, but the geometry work is reduced. For example, if you are using lighting, call **glDisable()** with a GL_LIGHTING argument to turn off lighting temporarily. If performance improves, your application has a per-polygon bottleneck. For more information, see "Tuning the Geometry Subsystem" on page 238.

- **Rasterization bottlenecks.** Programs that cause bottlenecks at the rasterization (per-pixel) stage in the pipeline are *fill-rate limited*. To test for bottlenecks in rasterization operations, shrink objects or make the window smaller to reduce the number of active pixels. This technique won't work if your program alters its behavior based on the sizes of objects or the size of the window. You can also reduce the work done per pixel by turning off per-pixel operations such as depth-buffering, texturing, or alpha blending. If any of these experiments speeds up the program, it has a per-pixel bottleneck. For more information, see "Tuning the Raster Subsystem" on page 243.

In general, it is easiest to first determine if your application is host (CPU) limited using *osview* and checking whether the CPU usage is near 100%. Then check whether the application is fill (per-pixel) limited by shrinking the window. You then only have to prove that it's geometry limited. The *osview* program also includes statistics that indicate whether the performance bottleneck is in the graphics subsystem or in the host.

Many programs draw a variety of things, each of which stresses different parts of the system. Decompose such a program into pieces and time each piece. You can then focus on tuning the slowest pieces. For an example of such a process, see Chapter 13, "Tuning Graphics Applications: Examples."

Factors Influencing Performance

Pipeline tuning is discussed in detail in Chapter 12, "Tuning the Pipeline." Table 11-1 provides an overview of factors that may limit rendering performance and the part of the pipeline they belong to.

Table 11-1 Factors Influencing Performance

Performance Parameter	Pipeline Stage
Amount of data per polygon	All stages
Time of application overhead	CPU subsystem
Transform rate & mode setting for polygon	Geometry subsystem
Total number of polygons in a frame	Geometry & Raster subsystem
Number of pixels filled	Raster subsystem
Fill rate for the given mode settings	Raster subsystem
Time of screen and/or depth buffer clear	Raster subsystem

Tuning Animation

Tuning animation requires attention to some factors not relevant in other types of applications. This section first looks at factors contributing to animation speed, then provides some advice for optimizing an animation's performance.

Factors Contributing to Animation Speed

The smoothness of an animation depends on its frame rate. The more frames rendered per second, the smoother the motion appears. The basic elements that contribute to the time to render a frame are shown in Table 11-1 above.

Smooth animation also requires double buffering. In double buffering, one framebuffer holds the current frame, which is scanned out to the monitor by video hardware, while the rendering hardware is drawing into a second buffer that is not visible. When the new framebuffer is ready to be displayed, the system swaps the buffers. The system must wait until the next vertical retrace period between raster scans to swap the buffers, so that each raster scan displays an entire stable frame, rather than parts of two or more frames.

Frame rates must be integral multiples of the screen refresh time, which is 16.7 msec (milliseconds) for a 60-Hz monitor. If the draw time for a frame is slightly longer than the time for *n* raster scans, the system waits until the *n+1st* vertical retrace before swapping buffers and allowing drawing to continue, so the total frame time is (n+1)*16.7 msec.

To summarize: A change in the time spent rendering a frame has no visible effect unless it changes the total time to a different integer multiple of the screen refresh time.

If you want an observable performance increase, you must reduce the rendering time enough to take a smaller number of 16.7 msec raster scans. Alternatively, if performance is acceptable, you can add work without reducing performance, as long as the rendering time does not exceed the current multiple of the raster scan time.

To help monitor timing improvements, turn off double buffering. If you don't, it's difficult to know if you're near a 16.7 msec boundary.

Optimizing Frame Rate Performance

The most important aid for optimizing frame rate performance is taking timing measurements in single-buffer mode only. For more detailed information, see "Taking Timing Measurements" on page 218.

In addition, follow these guidelines to optimize frame rate performance:

- Reduce drawing time to a lower multiple of the screen refresh time. This is the only way to produce an observable performance increase.
- Perform non-graphics computation after **glXSwapBuffers()**.

  A program is free to do non-graphics computation during the wait cycle between vertical retraces. Therefore, issue a **glXSwapBuffers()** call immediately after sending the last graphics call for the current frame, perform computation needed for the next frame, then execute OpenGL calls for the next frame, call **glXSwapBuffers()**, and so on.
- Do non-drawing work after a screen clear.

  Clearing a full screen takes time. If you make additional drawing calls immediately after a screen clear, you may fill up the graphics pipeline and force the program to stall. Instead, do some non-drawing work after the clear.

Optimizing Cache and Memory Use

This section first provides some background information about the structure of the cache and about memory lookup. It then gives some tips for optimizing cache and memory use.

How Memory Is Organized

Most systems do not have an unlimited amount of fast memory. To approach this ideal, system memory is structured as a hierarchy that contains a small amount of faster, more expensive memory at the top and a large amount of slower memory at the base.

The hierarchy is organized from registers in the CPU at the top down to the disks at the bottom. As memory locations are referenced, they are automatically copied into higher levels of the hierarchy, so data that is referenced most often migrates to the fastest memory locations.

Here are the areas you should be most concerned about:

- The cache feeds data to the CPU, and cache misses can slow down your program.

  Each processor has instruction caches and data caches. The purpose of the caches is to feed data and instructions to the CPU at maximum speed. When data is not found in the cache, a cache miss occurs and a performance penalty is incurred as data is brought into the cache.

- The translation-lookaside buffer (TLB) keeps track of the location of frequently used pages of memory. If a page translation is not found in the TLB, a delay is incurred while the system looks up the page and enters its translation.

The goal of machine designers and programmers is to maximize the chance of finding data as high up in the memory hierarchy as possible. To achieve this goal, algorithms for maintaining the hierarchy, embodied in the hardware and the operating system, assume that programs have locality of reference in both time and space; that is, programs keep frequently accessed locations close together. Performance increases if you respect the degree of locality required by each level in the memory hierarchy.

Even applications that appear not to be memory intensive, in terms of total number of memory locations accessed, may suffer unnecessary performance penalties for inefficient allocation of these resources. An excess of cache misses, especially misses on read operations, can force the most optimized code to be CPU limited. Memory paging causes almost any application to be severely CPU limited.

How to Minimize Memory Paging

This section provides some guidelines for minimizing memory paging. You learn about:

- "Minimizing Lookup"
- "Minimizing Cache Misses"
- "Measuring Cache-Miss and Page-Fault Overhead"

Minimizing Lookup

To minimize page lookup, follow these guidelines:

- Keep frequently used data within a minimal number of pages (each page consists of 4K bytes—16K in high-end systems—in current versions of IRIX). Minimize the number of pages referenced in your program by keeping data structures within as few pages as possible. Use *osview* to verify that no TLB misses are occurring.

- Store and access data in flat, sequential data structures, particularly for frequently referenced data. Every pointer indirection could result in the reading of a new page. This is guaranteed to cause performance problems with CPUs like the R10000™ that try to do instructions in parallel.
- In large applications (which cause memory swapping), use **mpin()** to lock important memory into RAM.

Minimizing Cache Misses

Each processor may have first-level instruction and data caches on chip and have second-level cache(s) that are bigger but somewhat slower. The sizes of these caches vary; you can use the *hinv* command to determine the sizes on your system. The first-level data cache is always a subset of the data in the second-level cache.

A cache line is a block of consecutively-addressed words that are treated as a unit by the cache. On a read-miss, a block of several cache lines is read from memory and stored in the cache. The size of this transaction varies from machine to machine (four lines is typical). To minimize cache misses, consider these points:

- Keep frequently accessed data together. This way, the most frequently accessed data remains in the first-level cache wherever possible.
- Access data sequentially. If you're accessing words sequentially, each cache miss brings in sixteen or more words of needed data; if you're accessing every sixteenth word, each cache miss brings in one needed word and fifteen unneeded words, degrading performance by up to a factor of sixteen.
- Avoid simultaneously traversing several large buffers of data, such as an array of vertex coordinates and an array of colors within a loop. There can be cache conflicts between the buffers. Instead, pack the contents into one buffer when possible. If this packing forces a big increase in the size of the data, it may not be the right optimization for that program.

Second-level data cache misses also increase bus traffic, which can be a problem in a multi-processing application. This can happen with multiple processes traversing very large data sets. See "Immediate Mode Drawing Versus Display Lists" on page 224 for additional information.

Measuring Cache-Miss and Page-Fault Overhead

To find out if cache and memory usage are a significant part of your CPU limitation, follow these guidelines:

- Use *osview* to monitor your application.
- A more rigorous way to estimate the time spent on memory access is to compare the results of PC sampling with those of basic block counting, performing each test with and without calls to **glVertex3fv()**.
  - PC sampling, from *prof*, gives a real-time estimate of the time spent in different sections of the code.
  - Basic block counting, from *prof -pixie*, gives an ideal estimate of how much time should be spent, not including memory references.

  See the *prof* reference page for more information.

  PC sampling includes time for system overhead, so it always predicts longer execution than basic block counting. However, your PC sample time should not be more than 1.5 times the time predicted by *prof -pixie*. For detailed information on profiling, see the *MIPS Compiling and Performance Tuning Guide*, which is available in hardcopy (document number 008-2479-001) or online through IRIS InSight.

  The CASEVision™/WorkShop tools, in particular the performance analyzer, can also help with those measurements. *The WorkShop Overview* introduces the tools.

Taking Timing Measurements

Timing, or benchmarking, parts of your program is an important part of tuning. It helps you determine which changes to your code have a noticeable effect on the speed of your application.

To achieve performance that is demonstrably close to the best the hardware can achieve, you can first follow the more general tuning tips provided in this manual, but you then need to apply a rigorous and systematic analysis. This section looks at some important issues regarding benchmarking:

- "Benchmarking Basics"
- "Achieving Accurate Timing Measurements"
- "Achieving Accurate Benchmarking Results"

Benchmarking Basics

A detailed analysis involves examining what your program is asking the system to do and then calculating how long it should take, based on the known performance characteristics of the hardware. Compare this calculation of expected performance with the performance actually observed and continue to apply the tuning techniques until the two match more closely. At this point, you have a detailed accounting of how your program spends its time, and you are in a strong position both to tune further and to make appropriate decisions considering the speed-versus-quality trade-off.

The following parameters determine the performance of most applications:

- total number of polygons in a frame
- transform rate for the given polygon type and mode settings
- number of pixels filled
- fill rate for the given mode settings
- time of color and depth buffer clear
- time of buffer swap
- time of application overhead
- number of attribute changes and time per change

Achieving Accurate Timing Measurements

Consider these guidelines to get accurate timing measurements:

- Take measurements on a quiet system.

  Verify that no unusual activity is taking place on your system while you take timing measurements. Other graphics programs, background processes, and network activity can distort timing results. For example, do not have *osview*, *gr_osview*, or *Xclock* running while you are benchmarking.

- Choose timing trials that are not limited by the clock resolution.

  Use a high-resolution clock and make measurements over a period of time that's at least one hundred times the clock resolution. A good rule of thumb is to benchmark something that takes at least two seconds so that the uncertainty contributed by the clock reading is less than one percent of the total error. To measure something that's faster, write a loop to execute the test code repeatedly.

Note: Loops like this for timing measurements are highly recommended. Be sure to structure your program in a way that facilitates this approach.

gettimeofday() provides a convenient interface to IRIX clocks with enough resolution to measure graphics performance over several frames. Call **syssgi()** with SGI_QUERY_CYCLECNTR for high-resolution timers. If you can repeat the drawing to make a loop that takes ten seconds or so, a stopwatch works fine and you don't need to alter your program to run the test.

- Benchmark static frames.

  Verify that the code you are timing behaves identically for each frame of a given timing trial. If the scene changes, the current bottleneck in the graphics pipeline may change, making your timing measurements meaningless. For example, if you are benchmarking the drawing of a rotating airplane, choose a single frame and draw it repeatedly, instead of letting the airplane rotate and taking the benchmark. Once a single frame has been analyzed and tuned, look at frames that stress the graphics pipeline in different ways, then analyze and tune them individually.

- Compare multiple trials.

  Run your program multiple times and try to understand variance in the trials. Variance may be due to other programs running, system activity, prior memory placement, or other factors.

- Call **glFinish()** before reading the clock at the start and at the end of the time trial.

  Graphics calls can be tricky to benchmark because they do all their work in the graphics pipeline. When a program running on the main CPU issues a graphics command, the command is put into a hardware queue in the graphics subsystem, to be processed as soon as the graphics pipeline is ready. The CPU can immediately do other work, including issuing more graphics commands until the queue fills up.

  When benchmarking a piece of graphics code, you must include in your measurements the time it takes to process all the work left in the queue after the last graphics call. Call **glFinish()** at the end of your timing trial, just before sampling the clock. Also call **glFinish()** before sampling the clock and starting the trial, to ensure no graphics calls remain in the graphics queue ahead of the process you are timing.

- Consider distortions in double-buffered programs.

  Because buffers can only be swapped during a vertical retrace, there is a period, between the time a **glXSwapBuffers()** call is issued and the next vertical retrace, when a program may not execute any graphics calls. A program that attempts to issue graphics calls during this period is put to sleep until the next vertical retrace. This distorts the accuracy of the timing measurement.

Note: To get accurate numbers, you must perform timing trials in single-buffer mode, with no calls to **glXSwapBuffers()**.

When making timing measurements, use **glFinish()** to ensure that all pixels have been drawn before measuring the elapsed time.

Achieving Accurate Benchmarking Results

To benchmark performance for a particular code fragment, follow these steps:

1. Determine how many polygons are being drawn and estimate how many pixels they cover on the screen. Have your program count the polygons when you read in the database.

   To determine the number of pixels filled, start by making a visual estimate. Be sure to include surfaces that are hidden behind other surfaces, and notice whether or not backface elimination is enabled. For greater accuracy, use feedback mode and calculate the actual number of pixels filled.

2. Determine the transform and fill rates on the target system for the mode settings you are using.

   Refer to the product literature for the target system to determine some transform and fill rates. Determine others by writing and running small benchmarks.

3. Divide the number of polygons drawn by the transform rate to get the time spent on per-polygon operations.

4. Divide the number of pixels filled by the fill rate to get the time spent on per-pixel operations.

5. Measure the time spent executing instructions on the CPU.

   To determine time spent executing instructions in the CPU, perform the graphics-stubbing experiment described in “Isolating Bottlenecks in the Pipeline: Overview” on page 212.

6. On high-end systems where the processes are pipelined and happen simultaneously, the largest of the three times calculated in steps 3, 4, and 5 determines the overall performance. On low-end systems, you may have to add the time needed for the different processes to arrive at a good estimate.

This process takes some effort to complete. In practice, it’s best to make a quick start by making some assumptions, then refine your understanding as you tune and experiment.

Ultimately, you need to experiment with different rendering techniques and do repeated benchmarks, especially when the unexpected happens.

Verify some of the suggestions presented in the following chapter. Try some techniques on a small program that you understand and use benchmarks to observe the effects. Figure 11-2 shows how you may actually go through the process of benchmarking and reducing bottlenecks several times. This is also demonstrated by the example presented in Chapter 13, "Tuning Graphics Applications: Examples."

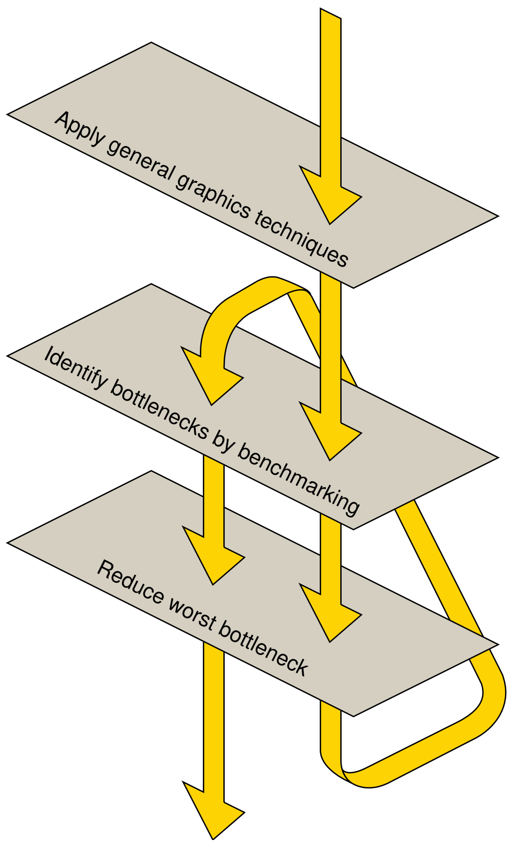

Figure 11-2 Flowchart of the Tuning Process