Optimization for Real-Time Graphics Applications

Sharon Rose Clay
Silicon Graphics Computer Systems

1 Introduction

Real-time entertainment applications are very sensitive to image quality, performance, and system cost. Graphics workstations provide full product lines with a full range of price points and performance options. At the high end, they provide many traditional Image Generator features such as real-time texture mapping and full scene antialiasing. They can also support many channels, or players, per workstation to offset the cost of getting the high-end features. At the low end, they have entry prices and performance that are often competitive with PCs. Graphics workstations can provide a very powerful, flexible solution with a rich development environment. Additionally, because of binary compatibility across product lines and standards in graphics APIs, graphics workstations offer the possibility of portability of both applications and databases to different and future architectures. However, this power and flexibility increases the complexity for achieving the full quoted performance from such a machine. This paper presents a strategy for performance for developing and tuning real-time graphics applications on graphics workstations.

The following topics are covered:

- typical application requirements for graphics workstations
- multi-processing issues for graphics subsytems
- graphics workstation pipelines and performance trade-offs
- strategies for diagnosing pipeline bottlenecks

- database structure for traversal
- designing and tuning a real-time application
- run-time diagnostics and load-management strategies
- tools for debugging graphics performance

Developing a designed-for-performance application requires understanding the potential performance problems, identifying which factors are limiting performance, and then making the trade-offs to achieve maximum frame rate with the highest quality scene content.

2 Background

Tuning both application and database is an essential part of the development process on a graphics workstation. Traditionally, both high-end image generators and low-end PC graphics platforms have been very restrictive in the type of application that can be supported and the scene content that can be rendered at a given rate. Graphics workstations offer:

- a wide range of graphics subsystems tightly coupled with today's fastest CPUs,
- high-bandwidth connections between the main CPU and the graphics subsystem (1.2GBytes/sec for the Silicon Graphics RealityEngine system bus),
- scalable (binary compatible) product lines and graphics standards for portability to different architectures, and hopefully future architectures,
- native, optimized, standard rendering libraries, such as OpenGL,
- the flexibility of being able to make performance trade-offs to optimize performance, maximize scene content, and minimize cost — careful tuning can yield tremendous results resulting in a flexible, low cost solution with high scene content,
- sophisticated development environments.

Why Tune?

The down-side of these features is that the tuning process is not only essential, it can be complex. Tuning an application to be performance-portable to different architectures is additionally complex. Unfortunately, tuning is one of those tasks that is often put off until the point of crisis.

Top 10 rationalizations for tuning avoidance:

10. The machine specifications should take into account a "real" application, so tuning should not be necessary.
9. We can worry about performance after implementation.
8. If we design correctly, we won't have to tune.
7. We will tune after we fix all of the bugs
   (also known as: The next release will be the performance version)
6. CPUs are going to be faster by the time we release so we don't have to tune our code.
5. We will always be limited by "that other thing" so tuning won't help.

4. The compiler should produce good code so we don't have to
3. We have this guru who will do all of the performance tuning for us.
2. The demo looks pretty fast.
1. Tuning will destroy our beautiful code.

An understanding of tuning issues and methods should hopefully make the above rationalizations unnecessary.

Tune early. Tune often.

Getting Started: Assessing Requirements and Predicting Performance

The first step, both in designing a real-time application and tuning an existing application, is to assess the requirements for image quality and predict the time required to produce that image quality. If there is a large gap between these calculations, trade-offs may have to be made in the application to balance scene content with performance to produce the most compelling result.

One of the most important parameters in the effectiveness of a simulated environment is *frame rate* — the rate at which new images are presented. The faster new frames can be displayed, the smoother and more compelling the animation will be. Constraints on the frame-rate can determine how much time there is to produce a scene.[i]

Entertainment applications typically require a frame rate of at least 20 frames per second (fps.), and more commonly 30fps. High-end simulation applications, such as flight trainers, will accept nothing less than 60fps. If, for example, we allow two milliseconds (msecs.) of initial overhead to start frame processing, one msec. for screen clear or background, and two msecs. for a window of safety, a 60pfs. application has, optimistically, about 11 msecs. to process a frame and a 30fps. application has 28 msecs.

Another important requirement for Visual Simulation and Entertainment applications is minimizing *rendering latency* — the time from when a user event occurs, such as change of view, to the time when the last pixel for the corresponding frame is displayed on the screen. Minimizing latency is also very important for the basic interactivity of non-real time applications.

A Typical Frame

The basic graphics elements that contribute to the time to render a frame are:

i. There are also additional requirements associated with frame-rate, such as low variability of frame rates and the handling of overload conditions when frame rates are missed. These issues are discussed later in Section 9.

screen clear (color and z-buffer clear or reset),

amount of data transferred to the graphics subsystem,

selected attributes for geometry, such as lighting, texturing, atmospheric effects and number of different attribute sets,

viewing transformations of geometry,

the number of pixels produced for the frame (resolution multiplied by *depth complexity*),

video refresh to output final image from framebuffer memory.

An estimation of expected performance should take into account all of these frame components, plus possible overhead due to interactions between components. An estimation of the time in milliseconds required to render a frame will then translate into an expected frame rate.

Screen clear time is like a fixed tax on the time to render a scene and for rapid frame rates, may be a measurable percentage of the frame interval. Because of this, most architectures have some sort of screen clear optimization. For example, the Silicon Graphics RealityEngine™ has a special screen clear that is well under one millisecond for a full high-resolution framebuffer (1280x1024). Video-refresh also add to the total frame time and is discussed in Section 4.

The size and contents of full databases vary tremendously among different applications. However, for context, we can guess at reasonable hypothetical scene content, given the high frame rates required for real-time graphical applications and current capabilities of graphics workstations.

The number of polygons possible in a 60fps. or 30fps. scene is affected by the many factors discussed in this paper, but needless to say, it can be quite different than the peak polygon transform rate of a machine. Current graphics workstations can manage somewhere between 1500 and 5000 triangles at 60pfs. and 7000-10,000 triangles at 30fps. Typical attributes specified for triangles include some combination of normals, colors, texture coordinates, and associated textures. For entertainment applications, the amount of dynamic objects and geometry changing on a per-frame basis is probably relatively high. For handling general dynamic coordinate systems of moving objects, matrix transforms are most convenient. Such objects usually also have relatively high detail (50-100 polygons). These numbers imply that we can easily imagine having half to a full megabyte of just geometric graphics data per frame.

Depth-complexity is the number of times, on average, that a given pixel is written. A depth-complexity of one means that every pixel on the screen is touched one time. This is a resolution-independent way of measuring the fill requirements of an application. Visual simulation applications tend to have depth-complexity between two and three for high-altitude applications, and depth-complexity between three and five for ground-based applications. Depth-complexity can be reduced through aggressive database optimizations, discussed in Section 7. Resolutions for visual simulation applications also vary widely. For entertainment, VGA(640x480) resolution is common. A 60fps. application at VGA resolution with depth-complexity five will require a fill rate of 100 million pixels per second (MPixels/sec.). In a single frame, there may can easily be one-two million pixels that must be processed.

The published specs of the machine can be used to make a rough estimate of predicted frame rate for the expected scene content. However, this prediction will probably be very optimistic. Performance prediction is covered in detail in Section 5. An understanding of the graphics architecture enables more realistic calculations.

Graphics Architectures

The type of system resources available and their organization has a tremendous effect on the application architecture. Architecture issues for graphics subsystems are discussed in detail in Section 3 andSection 4.

On traditional image-generators, the main application is actually running on a remote host with a low-bandwidth network connection between the application running on the main CPU and the graphics subsystem. The full graphics database resides in the graphics subsystem. Tuning applications on these machines is a matter of tuning the database to match set performance specifications. At the other extreme, we have PCs. Until recently, almost all of the graphics processing for PCs has traditionally been done by the host CPU and there has been little or no dedicated graphics hardware. Recently, there have been many new developments in this area with dedicated graphics cards developed by independent vendors for general PC buses. Some of these cards have memories for textures and even resident databases.

Graphics workstations fall between these two extremes. They traditionally have separate processors that make up a dedicated graphics subsystem.They may also have multiple host CPUs. Some workstations, such as Silicon Graphics, have a tightly coupling between the CPU and graphics subsystems with system software, compilers, and libraries. However, there are also independent vendors, such as EvansSutherland, Division, and Kubota, producing both high and low end graphics boards for general workstations.

The growing acceptance and popularity of standards for 3D graphics APIs, such as OpenGL™, is making it possible to develop applications that are portable between vastly different architectures. Performance, however, is typically not portable between architectures so an application may still require significant tuning (rewriting) to run reasonable on the different platforms. In some cases, the standard API library may have to be bypassed altogether if it is not the fastest method of rendering on the target machine. For the Silicon Graphics product line, this has been solved with a software application layer that is specifically targeted at real-time 3D graphics applications and gives peak performance across the product line [Rohlf94]. Writing/tuning rendering software is discussed in Section 5.

A common thread is that multiprocessing of some form has been a key component of the high-performance graphics platforms and is working its way down to the low-end platforms.

3 Multi-Processing for High-Performance Graphics

A significant part of designing and tuning an application is determining the best way to utilize the system processing resources and determining if additional system resources will benefit performance. Any tuning strategy requires an understanding of how the different components in a system interact to affect performance. There are many elements to system performance beyond the guiding frames-per-second:

Throughput — maximizing frame rates is equivalent to maximizing throughput: producing the most output possible per unit of time.

Bandwidth — The major datapaths through the system must have sufficient bandwidth or entire parts of a system may be under-utilized. The connections of greatest concern would be 1) that between the host computer and the graphics subsystem, 2) the paths of access to database memory and 3) disk access for the application and the graphics subsystem. It is particularly important that bandwidth specs not assume tiny datasets that will not scale to a real application.

Processor utilization — Will the system get good utilization of the available hardware or will some processors be sitting idle while others are overloading (will you get what you paid for). Good processor utilization is essential for a system to realize its potential throughput. Achieving this in a dynamic environment requires load balancing mechanisms.

Scalability — If performance is a problem, will the system support the addition of extra processors to improve throughput, and will improved performance scale with the addition of new processors. Additionally, as new processors are added, will load-balancing enable a real application to see the improved performance, or will it only show up in benchmarks.

Latency — What is the maximum interval of time from when a user initiated an input and the moment the final pixel of the corresponding new frame is presented. Low latency is critical to interactive real-time entertainment applications.

Synchronization overhead — how much overhead is incurred when tasks communicate information. This is particularly an issue for the very dynamic database of an interactive, real-time entertainment application: both the main application and the graphics subsystem need efficient access to the current state of the database.

These measures of performance can be applied to both the system as a whole, and to individual subsystems.

Methods of Multiprocessing

Because graphics applications have many very different tasks that must be executed every frame, they are well suited to division among multiple tasks, and multiple processors if available. Multiprocessing can also be used to achieve better utilization and throughput of a single processor.

The partitioning and ordering of the separate tasks has direct consequences on the performance of the system. A task may be executed in a pipelined, or in a concurrent fashion. *Pipelining* uses an assembly-line model where a task is decomposed into stages of operations that can be performed sequentially. Each stage is a separate processor working on a separate part of a frame and passing it to the next stage in the line. *Concurrent* processing has multiple tasks simultaneously working on different parts of the same input, producing a single result.

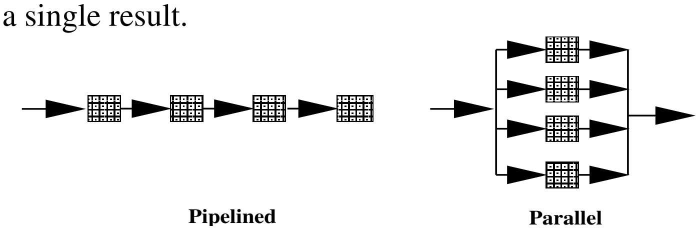


FIGURE 1. Pipelined vs. Parallel Processors

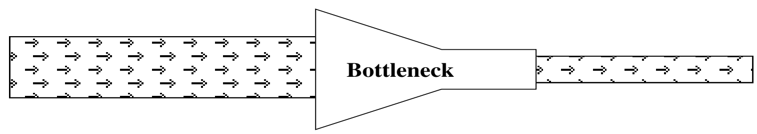
Both the host and graphics subsystems may employ both pipelining and parallelism as a way of using multiple processors to achieve higher performance. The general theoretical multiprocessing issues apply to both graphics applications and graphics subsystems. Additionally, there are complexities that arise with the use of special purpose processors, and from the great demands of graphics applications.

Many graphics tasks are easily decomposed into pipelined architectures. Typically, there is a main graphics pipeline, with parallelism within stages of the pipeline. Individual pipeline stages may themselves be sub-pipelines, or have parallel concurrent processors. Additionally, there may be multiple parallel graphics pipelines working concurrently.

Pipelining vs. Parallelism

Pipelined systems need minimal synchronization because each task is working on its own data for a different frame, or part of a frame, in an ordered fashion so synchronization is implicit. Longer pipelines have increased throughput — producing new results in quick succession because the task has been broken up into many trivial stages that all execute quickly. However, each stage in a pipeline will add latency to the system because the total amount of time through the pipeline is the number of stages multiplied by the step time, which is the speed of the slowest stage. While every step produces a new output, the total amount of time to produce a single output may get longer. The addition of a new pipeline stage will presumably decrease the step time, but probably not enough to avoid overall increased latency.

Pipelined systems will always run at the speed of the slowest stage, and no faster. The limiting stage in a pipelined system is appropriately called a *bottleneck*.



Pipeline tuning amounts to determining which stage in the pipeline is the bottleneck and reducing the work-load of that stage. This can be quite difficult in a graphics application because through the course of rendering a frame, the bottleneck changes dynamically. Furthermore, one cannot simply take a snapshot of the system to see where the overriding bottleneck is. Finally, improving the performance of the bottleneck stage can actually reduce total throughput if the another bottleneck results elsewhere. Bottleneck tuning methods are discussed in Section 5.

Tune the slowest stage of the pipeline.

Concurrent architectures do not suffer from the throughput vs. latency trade-off because each of the tasks will directly produce part of the output. However, synchronization and load-balancing are major issues. If processors are assigned to separate tasks that can be run in parallel, then there is the chance that some tasks will take very little time to complete and those processors will be idle. If a single task is distributed over several processors, then there is the overhead of starting them off and recombining the output results. However, the latter has a better chance of producing an easily-scalable system because repetitive tasks, such as transforming vertices of polygons, can be distributed among multiple concurrent processors. Concurrent parallel architectures are also easier to tune because it is quite apparent who is finishing last.

SIMD vs. MIMD

The processor organization in the system also needs to be considered. There are two types of processor execution organization: SIMD or MIMD. SIMD (single instruction multiple data) processors operate in lock-step where all processors in the block are executing the same code. These processors are ideal for the concurrent distributed-task model and require less overhead at the start and end of the task because of the inherent constraints they place on the task distribution. SIMD processors are common in graphics subsystems. However, MIMD (multiple instruction multiple data) do better on complex tasks that have many decision points because they can each branch independently. As with pipelined architectures, the slowest processor will limit the rate of final output.

In actual implementation, graphics architectures are a creative mix of pipelining and concurrency. There may be parallel pipelines with the major pipeline stages implemented as blocks of parallel processors.

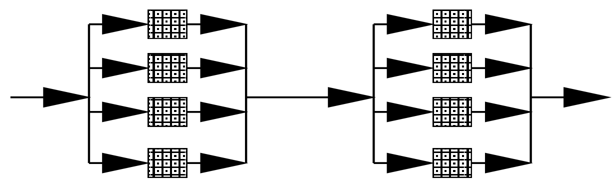


FIGURE 2. Parallel Pipeline

Individual processors may then employ significant sub-pipelining within the individual chips. Systems may be made scalable by allowing the ability to add parallel blocks.

4 Performance Issues in Graphics Pipelines

This section briefly reviews the basic rendering processes in the context of presenting the basic computational requirements. Additionally, ways in which the rendering task can be partitioned for implementation in hardware and corresponding performance trade-offs are also discussed. Tuning an application to a graphics pipeline is discussed in detail in Section 5.

The task of rendering three-dimensional graphics primitives is very demanding in terms of memory accesses, integer, and floating-point calculations. There are impressive software rendering packages that handle three dimensional texture-mapped geometry and can generate on the order of 1MPixels/sec on current CPUs. However, the task of rendering graphics primitives is very naturally suited to distribution among separate, specialized pipelined processors. Many of the computations that must be performed are also very repetitive, and so can take advantage of parallelism in a pipeline. This use of special-purpose processors to implement the rendering process is based on some basic assumptions about the requirements of a typical target application. The result can be orders of magnitude increases in rendering performance.

The Rendering Pipeline

The rendering process naturally lends itself to a simple pipeline abstraction. The *rendering pipeline* can generally be thought of as having three main stages:

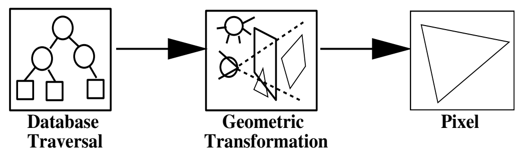


FIGURE 3. The Rendering Pipeline

Each of these stages may be implemented as a separate subsystem. These different stages are all working on different sequential pieces of rendering primitives for the current frame. A more detailed picture of the rendering pipeline is shown in Figure 4. An understanding of the computations that occur at each stage in the rendering process is important for understanding a given implementation and the performance trade-offs made in that implementation. The following is an overview of the basic rendering pipeline, the computational requirements of each stage, and the performance issues that arise in each stage[i][Foley90,Akeley93,Harrell93,Akeley89].

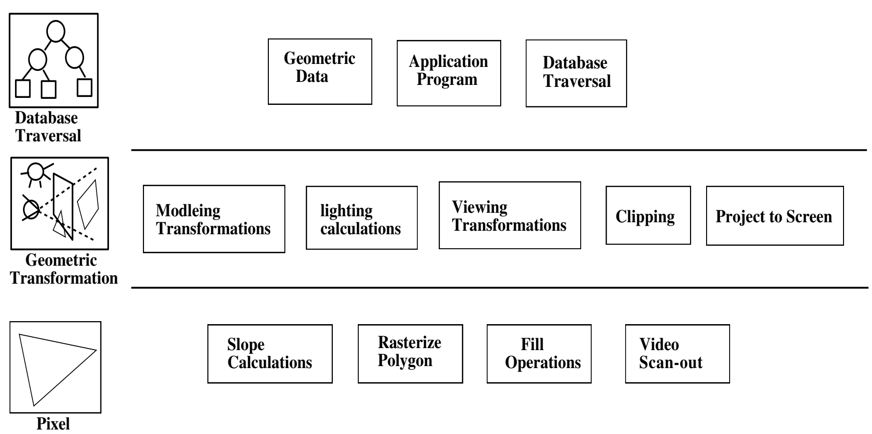


FIGURE 4. The Detailed Stages of the Rendering Pipeline

i. See [Foley90], Chapter 18, for a detailed discussion of this topic.

The CPU Subsystem (Host)

At the top of the graphics pipeline is the main real-time application running on the host. If the host is the limiting stage of the pipeline, the rest of the graphics pipeline will be idle.

The graphics pipeline might really be software running on the host CPU. In which case, the most time consuming operation is likely to be the processing of the millions of pixels that must be rendered. For the rest of this discussion, we assume that there is some dedicated graphics hardware for the graphics subsystem.

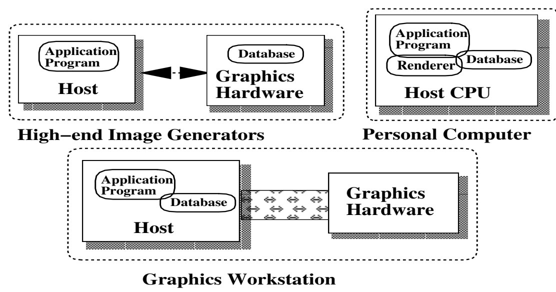


FIGURE 5. Host-Graphics Organizations

Database Traversal

The application may itself be multiprocessed and running on one or more CPUs. The host and the graphics pipeline may be tightly connected, sharing a high speed system bus, and possibly even access to host memory. Such buses currently run at several hundred MBytes/sec, up to 1.2GBytes/sec. However, in many high-end visual simulation systems, the host is actually a remote computer that drives the graphics subsystem over a network (SCRAMnet, 100 Mbits/sec, or even ethernet at 10Mbits/sec).The first stage of the rendering pipeline is traversal of the database and sending the current rendering data on to the rest of the graphics pipeline. In theory, the entire rendering database, or *scene graph*, must be traversed in some fashion for every frame because both scene content and viewer position are dynamic. Because of this, there are three major parts of the database traversal stage: processing to determine current viewing parameters (usually part of the main application), determining which parts of the scene graph are contained within the viewing frustum (culling), and the actual drawing traversal that issues rendering commands for the visible parts of the database. These components form a traversal pipeline of three stages: Application, Cull, and Draw:

FIGURE 6. Application Traversal Process Pipeline

Possibilities for the application processes are discussed further in Figure 6. This section will be focussing on the drawing traversal stage of the application.

Some graphics architectures impose special requirements on the drawing traversal task, such as requiring that the geometry be presented in sorted order from front to back, or requiring that data be presented in large, specially formatted chunks as display lists.

There are three main types of database drawing traversal:

immediate mode,
display list mode,
retained data.

Immediate Mode Drawing Traversal

In the first two, the rendering database lives in main memory. For immediate mode rendering, the database is actually shared with the main application on the host, as shown in Figure 7. The application is responsible for traversing the database and sending geometry directly to the graphics pipeline. This mode is the most memory efficient and flexible for dynamic geometry. However, the application is directly responsible for the low-level communication with the graphics subsystem.

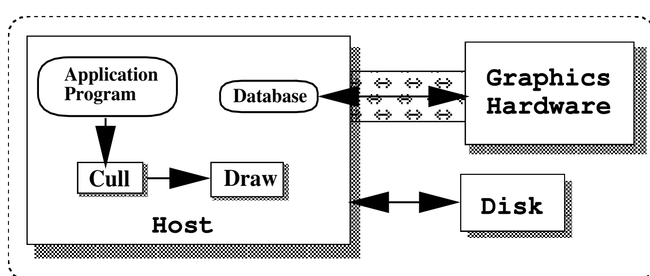


FIGURE 7. Architecture with Shared Database

Display List Traversal

In display-list mode, pieces of the database are compiled into static chunks that can then be sent to the graphics pipe. In this case, the display list is a separate copy of the database that can be stored in main memory in an optimized form for feeding the rest of the pipeline. The database traversal task is to hand the correct chunks to the graphics pipeline. These display lists can usually be edited, or re-created easily for some additional performance cost. For both of these types of drawing traversals, it is essential that the application be using the fastest possible API for communication with the graphics subsystem. An inefficient host-graphics interface for such operations as issuing polygons and vertices could leave the rest of the graphics pipeline starved for data.

Use only the fastest interface and routines when communicating with the graphics pipeline.

There is potentially a significant amount of data that must be transferred to the graphics pipeline every frame. If we consider just a 5K triangle frame, that corresponds to

(5000 tris) * (3 vertices/tri * (8 floats/vertex) * (4bytes/float)) = 480KBytes

→ 28.8 MBytes/sec for a 60fps. update rate.

for just the raw geometric data. The size of geometric data can be reduced through the use of primitives that share vertices, such as triangle strips, or through the use of high-level primitives, such as surfaces, that are expanded in the graphics pipeline (this is discussed further in Section 7). In addition to geometric data, there may also be image data, such as texture maps. It is unlikely that the data for even a single frame will fit a CPU cache so it is important to know the rates that this data can be pulled out of main memory. It is also desirable to not have the CPU be tied up transferring this data, but to have some mechanism whereby the graphics subsystem can pull data directly out of main memory, thereby freeing up the CPU to do other computation. For highly interactive and dynamic applications, it is important to have good performance on transfers of small amounts of data to the graphics subsystems since many small objects may be changing on a per-frame basis.

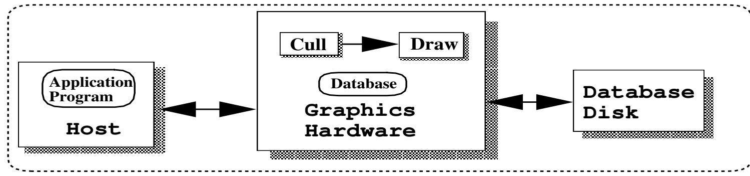


FIGURE 8. Architecture with Retained Data

Retained Database Traversal

If the database, and display list, is stored in the graphics pipeline itself, as shown in Figure 8, separate from main memory, it is a retained display list. Retained display lists are traversed only by the graphics pipeline and are required if there is very low bandwidth between the host and the graphics subsystem. The application only sends small database edits and updates (such as the new viewing parameters and a few matrices) to the graphics pipe on a per-frame basis. Pieces of the database may be paged directly off local disks (also at about 10MBytes/sec.). Retained mode offers less much flexibility and power over editing the database, but also can remove the possible bandwidth bottleneck at the head of the graphics pipeline.

The use of retained databases can enable additional processing of the total database by the graphics subsystem. For example, partitioning the database may be done order to implement sophisticated optimization and rendering techniques. One common example is the separation of static from moving objects for the implementation of algorithms requiring sorting. The cost may be additional loss of power and control over the database due to limitations on database construction, such as the number of moving objects allowed in a frame.

Graphics Subsystems

The Geometry Subsystem

The second two stages of the rendering pipeline, Figure 3, are commonly called *The Geometry Subsystem* and *The Raster Subsystem*, respectively. The geometry subsystem operates on the geometric primitives (surfaces, polygons, lines, points). The actual operations are usually per-vertex operations. The basic set of operations and estimated computational complexity includes [Foley90]: modeling transformation of the vertices and normals from eye-space to world space, per-vertex lighting calculations, viewing projection, clipping, and mapping to screen coordinates. Of these, the lighting calculations are the most costly. A minimal lighting model typically includes emissive, ambient diffuse, and specular illumination for infinite lights and viewer. The basic equation that must be evaluated for each color component (R, G, and B) is [OpenGL93]:

RGBemissive_mat +
RGBambient_model*RGBambient_mat +
RGBambient_mat*RGBambient_light +
RGBdiffuse_mat*RGBdiffuse_light * (light_vector . normal)

Specular illumination adds an additional term (exponent can be approximated with table lookup):

$$RGBspecular\_light*RGBspecular\_mat*(half\_angle\ .\ normal)^{shininess}$$

Much of this computation must be re-computed for additional lights. Distance attenuation, local viewing models and local lights add significant computation.

A trivial accept/reject clipping step can be inserted before lighting calculations to save expensive lighting calculations on geometry outside the viewing frustum. However, if an application can do a coarse cull of the database during traversal, a trivial reject test may be more overhead than benefit. Examples of other potential operations that may be computed at this stage include primitive-based antialiasing and occlusion detection.

This block of floating-point operations is an ideal case for both sub-pipelining and block parallelism. For parallelism, knowledge about following issues can help application tuning:

MIMD vs. SIMD,

how streams of primitives are distributed to the processors,

is the output of these processors remain separate in parallel streams for the next major stage, or re-combined into a single output stream for re-distribution.

The first issue, MIMD vs. SIMD, affects how a pipeline handles changes in the primitive stream. Such changes might include alterations in primitive type, state changes, such as the enabling or disabling of lighting, and the occurrence of a triangle that needs to be clipped to the viewing frustum. SIMD processors have less overhead in their setup for changes. However, since all of the processors must be executing the same code, changes in the stream can significantly degrade processor utilization, particularly for highly parallel SIMD systems. MIMD processors are flexible in their acceptance of varied input, but can be more somewhat more complex to setup for given operations, which will include the processing state changes. This overhead can also degrade processor utilization. However adding more processors can be added to balance the cost of this overhead.

The distribution of primitives to processors can happen in several ways. An obvious scheme is to dole out some fixed number of primitives to processors. This scheme also makes it possible to easily re-combine the data for another distribution scheme for the next major stage in the pipeline. MIMD processors could also receive entire pieces of general display lists, as might be done for parallel traversal of a retained database.

The application can affect the load-balancing of this stage by optimizing the database structure for the distribution mechanism, and controlling changes in the primitive stream.

Order rendering to benefit the most expensive pipeline stage.

After these floating point operations to light and transform geometry, there are fixed-point operations to calculate slopes for the polygon edges and additional slopes for z, colors, and possibly texture coordinates. These calculations are simple in comparison to the floating point operations. The more significant characteristic here is the explosion of vertex data into pixel data and how that data is distributed to downstream raster processors.

The Raster Subsystem

There is an explosion of both data and processing that is required to rasterize a polygon as individual pixels. Typically, these operations include depth comparison, gouraud shading, color blending, logical operations, texture mapping and possibly antialiasing. These operations require accessing of various memories, both reading for inputs to comparisons and the blending and texturing operations, and writing of the updated depth and color information and status bits for logical operations. In fact, the memory accesses can be more of a performance burden than the simple operations being computed. Of course, this is not true if complex per-pixel shading algorithms, such as Phong Shading, are in use. For antialiasing methods using super-sampling, some of these operations (such as z-buffering) may have to be done for each sub-sample. For interpolation of pixel values for antialiasing, each pixel may also have to visit the memory of its neighbors. Texture interpolation for smoothing the effects of minification and magnification can also cause many memory accesses for each pixel. An architecture might choose to keep some memory local to the pixel processor, in which case, fill operations that only access local processor memory would probably be faster.

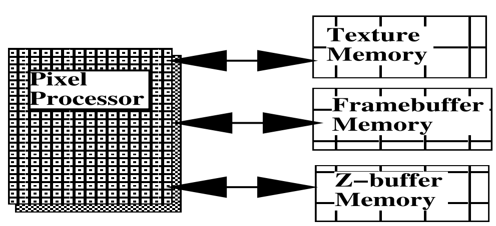


FIGURE 9. Pixel Operations do many Memory Accesses

The vast number of pixel operations that must be done would swamp single-processor architectures, but are ideal for wide parallelism. The Silicon Graphics RealityEngine™ has 80 Image Processors on just one of up two four raster subsystem boards. The following is a simplified example of how parallelism can be achieved in the Raster Subsystem. The screen is subdivided into areas for some number of rasterizing engines that take polygons and produce pixels. Each rasterizor has a number of pixel processors that are each responsible for a sub-area of its parent rasterizing engine and writes directly into framebuffer memory. Thus, the Raster Subsystem may have concurrent sub-pipelines.

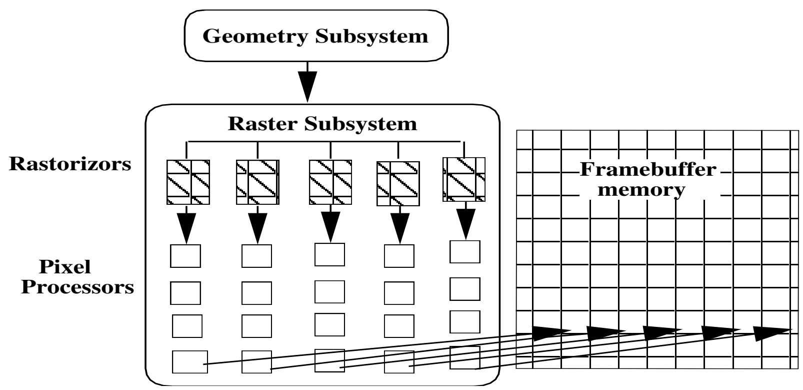


FIGURE 10. Parallelism in the Raster Subsystem

In some architectures, such as the Silicon Graphics VGX™, the rasterizors get interleaved vertical spans on the screen and the pixel processors get interleaved pixels within those spans. The Silicon Graphics Reality-Engine™ uses a similar scheme, but with more complex interleaving for better load balancing and many more pixel processors[Akeley93].

Certain operations that can cause an abort of the writing to a pixel, such as a failed z-buffer test, can be used short-circuit further more expensive pixel operations. If the application can draw front to back, or draw large front polygons first, a speedup might be realized.

Depending on the distribution strategy, MIMD processors in this stage might show more benefit from such short-circuit operations. The distribution strategy typically employs an interleaved partitioning of the frame-buffer. This optimizes memory accesses and promotes good processor utilization. The possible down side is that most processors will need to see most primitives. The complexity in figuring out which primitive go to which processors may cause processors to receive input for which they do no work. Because of this overhead, small polygons can have less efficient fill characteristics.

Bus Bandwidth

The bottleneck of a pipeline may not be one of the actual stages, but in fact one of the buses connecting two stages, or logic associated with it. There may be logic for parsing the data as it is comes off the bus, or distributing the data among multiple downstream receivers. Any connection that must handle a data explosion, such as the connection between the Geometry and Raster subsystems, is a potential bottleneck. The only way to reduce these bottlenecks is to reduce the amount of raw data that must flow through it, or to send data that requires less processing. The most important connection is the one that connects the graphics pipeline to the host because if that connection is a bottleneck, the entire graphics pipeline will be under-utilized.

The use of FIFO buffers between pipeline stages provides necessary padding that protects a pipeline from the affects of small bottlenecks and smooths the flow of data through the pipeline. Large FIFOs at the front of the pipeline and between each of the major stages can effectively prevent a pipeline from backing up through upstream stages and sitting idle while new data is still waiting at the top to be presented. This is useful important for fill-intensive applications which tend to bottleneck the very last stages in the pipeline. However, once a FIFO fills, the upstream stage will back up.

Fill the pipeline from back to front.

Video Refresh

The final stage in the frame interval is the time spent waiting for the video scan-out to complete for the new frame to be displayed. This period, called a *field*, is the time from the first pixel on the screen until the last pixel on the screen is scanned out to video. For a 60Hz video refresh rate, the time could be as much as 16.7msecs. Graphics workstations typically use a double-buffered framebuffer so that for an extra field of latency, the system can achieve frame-rates equal to the scan-out rate. A double-buffered system will toggle between two framebuffers, outputting the contents of one framebuffer while the other is receiving rendering results. The framebuffers cannot be swapped until the previous video refresh has completed. This will force the frame rate of the application to run at a integer multiple of the video refresh rate. In the worst case, if the rendering for one frame completed just after a new video refresh was started, the application could theoretically have to wait for the entire refresh period, waiting for an available framebuffer to receive rendering for the next frame.

A double-buffered application will always have a frame rate that is an integer multiple of the video refresh rate.

Dealing with Latency

The time for video refresh is also the lower bound on possible latency for the application. The typical double-buffered application will have a minimum of two fields of latency: one for drawing the next frame while the current one is being scanned, and then a second field for the frame to be scanned. This assumes that the frame rate of the application is equal to the field rate of the video. In reality, a double-buffered application will have a latency that is at least

$$2 * N * field\_time$$

where N is the number of fields per application frame.

One obvious way to reduce rendering latency is to reduce the frame time to draw the scene. Another method is to allow certain external inputs, namely viewer position, into later stages of the graphics pipeline. An interesting method to address both of these problems is presented in [Regan94]. A rendering architecture is proposed that handles viewer orientation after rendering to reduce both reduce latency and drawing. The architecture renders a full encapsulating view around the viewer's position. The viewer orientation is sampled after rendering by a separate pipeline that runs at video refresh rate to produce the output RGB stream for video. Additionally, only objects that are moving need to be redrawn as the viewer changes his orientation. Changes in viewer position could also be tolerated by setting a maximum tolerable error in object positions and sizes. Complex objects could even be updated at a slower rate than the application frame rate since their previous renderings still update correctly with viewer orientation.

These principles of sampling viewer position as late as possible, and decoupling of object rendering rate from viewer update rate can also be applied to applications.

5 Optimizing Performance of a Graphics Pipeline

This section discusses tuning an application for a graphics pipeline. The *Law of Diminishing Returns* definitely applies here: for an application designed with performance in mind, it is typically fairly easy to achieve about 60% of expected optimal performance. A bit of work can get you to 75% or 80% and then it starts to get more difficult. A major reason for this is the pure complexity of having so many parameters interacting which further affects performance behavior. The key is in identifying and isolating the current problems. The goal is a balanced pipeline where no one stage is an overwhelming bottleneck.

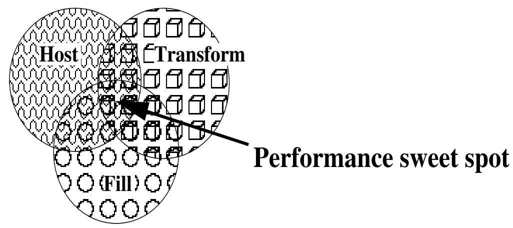


FIGURE 11. A Balanced Pipeline

The focus of this section is the development of the following basic tuning strategy:

1. Design for performance
2. Estimate expected performance
3. Measure and evaluate current performance
4. Isolate performance problems
5. Balance operations
6. Repeat

Design For Performance

The full system pipeline should be kept in mind when designing the application. Graphics features should be chosen to balance the pipeline and careful estimations of expected performance for target databases should be made during the design phase as well as the tuning phase.

Selecting Features

Combinations of rendering features should be choosen to produce a balanced pipeline. An advantage of graphics workstations is the power to make trade-offs to maximize both performance and scene quality for a given application. If, for example, a complex lighting feature is required that will bottleneck the geometry subsystem, then possibly a more interesting fill algorithm could be used to both require less polygons being lit and achieve overall higher scene quality.

Beware of features that use multi-pass algorithms because pipelines are usually balanced with one pass through each stage. There are many sophisticated multi-pass algorithms incorporating such techniques as texture-mapping, Phong-shading, accumulation antialiasing, and other special effects, that produce high-quality images. Such features should be used sparingly and their performance impact should be well understood.

Designing Tasks

The application should also be designed with multiprocessing in mind since this is very hard to add after-the-fact. Large tasks that can be run on separate processors (preferably with minimal synchronization and sharing of data) should be identified. For ease of debugging, portability, and tuning (discussed further in Section 8) the application should support both a single process mode, and a mode where all tasks are forced into separate processes.

Design with multiprocessing in mind.

The tasks also need to be able to non-invasively monitor their own performance, and need to be designed so that they will support measurements and experiments that will need to be done later for tuning. The rendering task (discussed later in this section) must send data to the graphics pipeline in a form that will maximize pipeline efficiency. Overhead in renderer operations should be carefully measured and amortized over on-going drawing operations.

Estimating Performance for a Pipeline

Making careful performance estimations greatly enhances your understanding of the system architecture. If the target machine (or similar machine) is available, then this should be done in tandem with the analysis of current application performance and the comparison to small benchmarks until the measurements and estimations agree.

As should not be surprising by this time, estimating performance of an application for a pipeline is much more than looking at peak quoted numbers for a machine and polygon totals for a database. The following are basic steps for estimating performance:

1. Define the contents of a worst-case frame, including number of polygons and their types, number of graphics modes and changes, and average polygon sizes
2. Identify the major stages of the graphics pipeline
3. For each major stage, identify the parts of the frame that are significant for that stage
4. Estimate the time that these frame components will spend in each stage of the pipeline (if possible, verify with small benchmarks)
5. Sum the maximum stage times computed in (4) for a very pessimistic estimation
6. When the drawing order can be predicted (such as for screen clear which may be expensive in the fill stage and will always come first) more optimistic estimations can be made by assuming that time spent in upstream stages for later drawing will be concurrent with the downstream work, and thus gotten for free.

First, consider the geometry subsystem: the significant operations might be triangles (with better rates for meshed triangles), lighting operations, clipping operations, mode changes, and matrix transformations. Given this information, one can compile the following type of information:

- Triangles: percentage of triangles in meshes of different lengths
- Graphics modes: percentage of the triangles that are lit
- Mode changes: number of texture changes per frame
- Viewing Matrix transformations: number per frame
- Clipping: percentage of triangles that are trivial accept, are trivial reject, and those that intersect the viewing frustum

Given the scene characteristics, one should first write small benchmarks to get geometry subsystem performance statistics on individual components. Then, write an additional benchmark that mimics the geometry characteristics of a frame to evaluate the interactions of those components.

We then similarly examine the raster subsystem. We first need to know the relevant frame information:

- Depth complexity of the frame and target resolution
- Number of triangles in different fill modes and a few typical sizes
- Number of raster mode changes
- Time for screen clear

Again, if possible, benchmarks should be written to verify the fill rates of polygons and the cost of raster mode changes. From these estimates, one can make a best guess about the amount of time that will be spent in the raster subsystem.

We can now make coarse-grained and fine-grained estimations of frame time. An extremely pessimistic approach would be to simply add the bottleneck times for the geometry subsystem and the raster subsystem. However, if there is a sufficient FIFO between the geometry and raster subsystems, much of the operations in the geometry subsystem should overlap with the raster operations. Assuming this, a more optimistic coarse-grained estimation would be to sum the amount of time spend in the raster subsystem and the amount of time beyond that required by the geometry subsystem. A fine-grain approach would be to consider the bottlenecks for different types of drawing. Identify the parts of the scene that are likely to be fill-limited and those that are likely to be transform-limited. Then sum the bottleneck times for each.

Measuring Performance and Writing Benchmarks

Being able to make good performance measurements and write good benchmarks is essential to getting that last 20% of performance. To achieve good timing measurements, do the following:

1. Take measurements on a quiet system. Graphics workstations have fancy development environments so care must be taken that background processes, such as a graphical clock ticking off seconds, or a graphical performance monitor, etc., are not disrupting timing.
2. Use a high-resolution clock and make measurements over a period of time that is at least 100x the clock resolution.
3. If there are only very low resolution timers (less than 1millisecond) then to accurately time a frame, pick a static frame (freeze matrix transformations), run in single-buffered mode, and time the repeated drawing of that frame.
4. Make sure that the benchmark frame is repeatable so that you can return to this exact frame to compare the affects of changes.
5. Make sure that pipeline FIFOs are empty before starting timing and then before checking the time at the end of drawing. When using OpenGL, one should call **`glFinish()`** before checking the clock.
6. Verify that you can rerun the test and get consistent timings.

A generally good technique for writing benchmarks is to always start with one that can achieve a known peak performance point for the machine. If you are writing a benchmark that will do drawing of triangles, start with one that can achieve the peak triangle transform rate. This way, if a benchmark seems to be giving confusing results, you can simplify it to reproduce the known result and then slowly add back in the pieces to understand their effect.

Verify a known benchmark on a quiet system.

When writing benchmarks, separate the timings for operations in an individual stage from benchmarks that time interactions in several stages. For example, to benchmark the time polygons will spend in the geometry subsystem, make sure that the polygons are not actually being limited by the raster subsystem. One simple trick for this is to draw the polygons as 1-pixel polygons. Another might be to enable some mode that will cause a very fast rejection of polygon or pixels after the geometry subsystem. However, it is important to write both benchmarks that time individual operations in each stage, and those that mimic interactions that you expect to happen in your application.

Finding the Bottlenecks

Over the course of drawing a frame, there will likely be many different bottlenecks. If you first clear the screen and draw background polygons, you will start out fill-limited. Then, as other drawing happens, the bottleneck will move up and down the pipeline (hopefully not residing at the host). Without special tools, bottlenecks can be found only by creative experimentation. The basic strategy is to isolate the most overwhelming bottleneck for a frame and then try to minimize it without creating a worse one elsewhere.

Isolate the bottleneck stage of the pipeline.

One way to isolate bottlenecks is by eliminating work at specific stages of the pipeline and then check to see if there is a significant improvement in performance. To test for a geometry subsystem bottleneck, you might force off lighting calculations, or normalization of vertex normals. To test for a fill bottleneck, disable complex fill modes (z-buffering, gouraud shading, texturing), or simply shrink the window size. However, beware of secondary affects that can confuse the results. For example, if the application adjusts what it draws based on the smaller window, the results from just shrinking the window without disabling that functionality will be meaningless. Some stages are simply very hard to isolate. One such example is the clipping stage. However, if the application is culling the database to the frustum, you can test for an extreme clipping bottleneck by simply pushing out the viewing frustum to include all of the geometry.

6 Tuning the Application

Applications usually plan on pushing graphics past their limits. If the rendering traversal is part of the application, then this traversal must be optimized so that it keeps the graphics subsystem busy. On a multiprocessing system, other operations for scene management formatting of data can be moved out of the renderer and into other processes, preferably running on other CPUs. Finally, a key part of real-time rendering is *load management*, providing a graceful response to overloading the graphics subsystem, which is discussed later in Section 8.

Tuning the Renderer

Efficient Coding

There is no escape from writing efficient code in the renderer. Immediate mode drawing loops are the most important parts since code in those loops are executed thousands of times per frame. For peak performance from these loops, one should do the following:

- Minimize the loop overhead and decision logic in the polygon loops. Unroll per-vertex code and duplicate code, as opposed to using per-vertex if-tests.
- Use a flat data structure for the draw traversal - you want to minimize the number of memory pages you need to touch in a given loop.
- Disassemble code to examine loop overhead (and to check that the compiler is doing what you expect).

Display list rendering requires less optimization because it does not require the tight loops for rendering individual polygons. However, this is at the cost of more memory usage for storing the display list and less flexibility in being able to edit the geometry in the list for dynamic objects. The extra memory required by display lists can be quite significant because there can be no vertex sharing in display lists. This can restrict the number of objects you can hold in memory and will also slow the time to page in new objects if the graphics display lists must be re-created. Additionally, display lists may need to be of a certain minimum size to be handled efficiently by the system. If there are many small moving objects in the scene, the result will be many small display lists. If you have the choice, given the option between immediate mode rendering and database paging, you might choose to use at least some immediate mode, particularly for dynamic objects.

Don't let the host be the bottleneck

IRIS Performer™, a Silicon Graphics toolkit for developing real-time graphics applications, uses a fairly aggressive technique for achieving high-performance immediate-mode rendering. Data structures for geometry enforce the use of efficient drawing primitives. Geometry is grouped into sets by type and attribute bindings (use of per-vertex or per-polygon colors, normals, and texture coordinates). For each combination of primitive and attribute binding, there is a specialized routine with a tight loop to draw the geometry in that set. The result is several hundred such routines but the use of macros makes the code easy to generate and maintain. IRIS Performer also provides an optimized display list mode that is actually an immediate mode display list and shares the application copy of data instead of copying off a separate, uneditable copy. This is discussed in [Rohlf94], and [PFPG94]. Host rendering optimization techniques 24are also discussed in detail in [GLPTT92].

Multiprocessing

Multiprocessing can be used to allow the renderer to devote its time issuing graphics calls while other tasks, such as scene and load management can be placed into other processes. There are several large tasks that are obvious candidates for such course-grained multiprocessing:

- the real-time application — processes inputs from IO, calculates new viewing parameters, positional parameters for objects, and parameters for dynamic geometry,
- scene management — culling out parts of the scene graph that are not in the viewing frustum, calculating LOD information, generating a display list for the rendering traversal,
- dynamic editing of geometric data,
- IO handling — polling external devices, database paging,

intersection traversals for collision detection,

complex simulations for various vehicles.

A combination of pipelining and parallelism can be used to get the right throughput/latency trade-off for your application and the target machine. IRIS Performer™ provides a process pipeline:

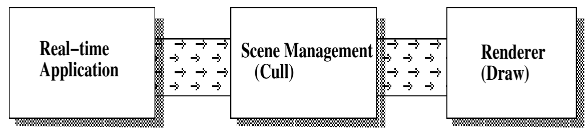


FIGURE 12. IRIS Performer Process Pipeline

This process pipeline, described in [Rohlf94], is re-configurable to allow:

a pipeline where the cull and draw are parallel processes (app->cull/draw),

a model where the cull and draw are performed by a single process that culls and renders simultaneously (app->cull_draw),

a minimal-latency model where all tasks are performed by a single process (app_cull_draw).

Multiprocessing also allows additional tasks to be done that will make the rendering task more efficient, such as:

generating a per-frame, optimized display list for the rendering task so that the drawing traversal does not need to traverse the original database,

sorting geometry by mode to minimize mode changes,

host backface removal (and removal of backfaced objects) to save additional host bandwidth,

flattening of dynamic transformations over objects of only one or two polygons.

It is important to identify which tasks must be real-time, and which can run asynchronously and extend beyond frame boundaries. Real-time tasks are those that must happen within a fixed interval of time, and severe consequences will result if the task extends beyond its frame. The main application, cull, and draw tasks are all real-time tasks. However, it might not be so traumatic if, for example, some of the collision results are a frame late. The polling of external control devices should probably be done in a separate, asynchronous process — if those results are late, extrapolation from previous results is probably better than waiting. Real-time tasks are discussed further in Section 8.

A real-time process should not poll an external device.

7 Database Tuning

Tuning databases is as important (and often just as much work) as tuning the application. The database hierarchy needs to be structured to optimize the major traversal tasks. Information cached in the database hierarchy can reduce the number of dynamic operations that must be done by traversals. Finally, the modeling of the database geometry should be done with an understanding of the performance characteristics of the graphics pipeline.

Spatial Hierarchy Balanced with Scene Complexity

The major real-time database traversals are the cull and collision traversals. Both benefit by having a database that is spatially organized, or is coherent in world space. These traversals eliminate parts of the scene graph based on bounding geometry. If a database hierarchy is organized by grouping spatially near objects, then entire sub-trees can be easily eliminated by the bounding geometry of a root node. If most nodes have bounding geometry that covers much of the database, then an excessive amount of the database will have to be traversed.

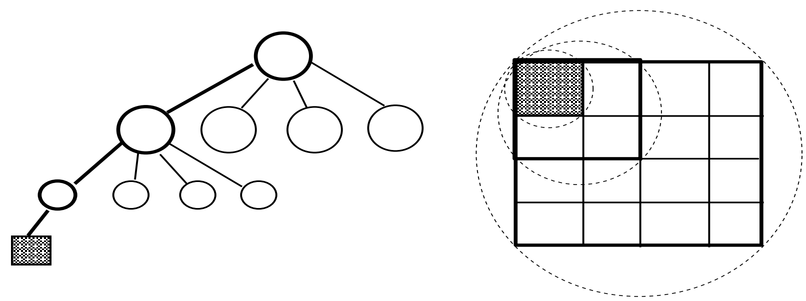


FIGURE 13. Scene-graph with Spatial Hierarchy

It is additionally helpful to have a hierarchy based on square areas so that simple bounding geometry, such as bounding spheres, can be used to optimize the traversal test.

The amount of hierarchy put in the database should balance the traversal cost of nodes with the number of children under them. A node with few children will be able to eliminate much of the database in one step. However, a deep hierarchy might be expensive to maintain as objects change and information must be propagated up the tree.

Database Instancing

Instancing in a database is where multiple parents reference a single instanced child which allows you to make performance/memory trade-offs.

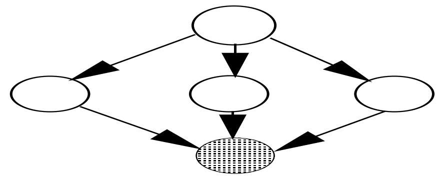


FIGURE 14. Instanced Node

Instancing saves memory but prevents a traversal from caching traversal information in the child and also prevents you from flattening inherited matrix transformations. To avoid these problems, IRIS Performer™ provides a compromise of *cloning* where nodes are copied but actual geometry is shared.

Balancing the Traversals

The amount of geometry stored under a leaf node will affect all of the traversals, but there is a performance trade-off between the spatial traversals and the drawing task. Leaf nodes with small numbers of polygons will provide a much more accurate culling of objects to the viewing frustum, thus generating fewer objects that must be drawn. This will make less work for the rendering task; however, the culling process will have to do more work per polygon to evaluate bounding geometry. If the collision traversal needs to compute intersections with actual geometry, then a similar trade-off exists: fewer polygons under a leaf node means fewer expensive polygon intersections to compute.

Modeling to the Graphics Pipeline

The modeling of the database will directly affect the rendering performance of the resulting application and so 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, will benefit from having long meshes in the database. However, the length of the meshes will affect the resulting database hierarchy and long strips through the database will not cull well with simple bounding geometry.

Objects can be modeled with an understanding of inherent bottlenecks in the graphics pipelines. Pipelines that are severely fill-limited will benefit from having objects modeled with cut polygons and more vertices and fewer overlapping parts which will decrease depth complexity.

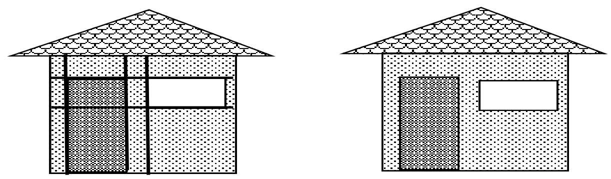


FIGURE 15. Modeling with cut polygons vs. overlapping polygons

Pipelines that are easily geometry or host limited will benefit from modeling with fewer polygons.

There are a couple of other modeling tricks that can reduce database complexity. One is to use textured polygons to simulate complex geometry. It is especially useful if the graphics subsystem supports the use of *alpha textures* where a channel of the texture marks the transparency of the object. Texture can be made as cut-outs for things like fences and trees. Textures are also useful for simulating particles, such as smoke. Textured polygons as single-polygon billboards are additionally useful. *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 just 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 impact both of the cull and draw processes.

3D Database modeling techniques like these have been in use for a long time in Visual Simulation applications.

8 Real-Time On a Workstation

Graphics workstations are attractive development platforms because they have rich and user-friendly development environments. They are not traditionally known for their real-time environments. However, extensions can be made to the basic operating system to support a real-time mode, as was done with the Silicon Graphics REACT™ extensions to IRIX™. The REACT extensions are really ways to push UNIX aside for real-time operation. It guarantees interrupt response time on a properly configured system and enables the user to have control of the scheduling of specified processors and therefore be exempt from all UNIX overhead. REACT also includes a real-time frame scheduler that can be used on the real-time processors. Finally, REACT offers time-stamps on system and user operations for real-time system performance feedback.

Running an application in performance-mode might be quite different from running it in development mode. Most obviously, a real-time application needs fast timers to be able to monitor its performance for load-management, as well as having accurate time-based animations and events. A real-time application also needs to be guaranteed worst case behavior for basic system functions such as interrupt response time. It also needs to have control over how it is scheduled with other processes on the system, and how its memory is managed. In addition, the main application needs to synchronize frame boundaries of various tasks with the graphics subsystem.

Put the system and application in real-time mode for real-time performance.

Managing System Resources for Real-Time

One type of organization is to put the rendering process on its own processor, isolated from other system activity and synchronization with other tasks. This is the organization used in IRIS Performer™[Rolf94]. To do this, the rendering process should also have its own copy of data to minimize synchronization and conflict over pages with other processors. On a general-purpose workstation, one CPU will need to be running basic system tasks and the scheduler. Additionally, a distinction should be made between tasks that must be real-time (happen reliably at fixed intervals), and those processes that may extend past frame boundaries in generating new results. Non-real-time tasks can be given lower priorities and share processors, perhaps even the system CPU.

Real-Time Graphics

Getting steady, real-time frame rates from the graphics subsystem can be a challenge on any system. One problem is handling overload conditions in the graphics subsystem. Another is the synchronization of multiple graphics systems.

Load Management

High-end image generators have frame control built into the graphics subsystem so that they can simply halting drawing at the end of a frame time. This can produce an unattractive result, but perhaps one that is less disturbing than a wild frame rate. Getting a graphics subsystem to stop based on a command sent from the host and placed in a long FIFO can be a problem. If the graphics subsystem does not have its own mechanism for managing frame rate control (as currently only high-end image-generators do) then the host will have to do it. This means leaving generous margins of safety to account for dynamic changes in graphics load and tuning the database to put an upper bound on worst-case scenes. However, some method of load management will also be required.

Load management for graphics is a nice way of saying "draw less." One convenient way to do this is by applying a scaling factor to the *levels of detail* (LODs) of the database, using lower LODs when the system is overloaded and higher LODs when it is under-utilized. A hysteresis band can be applied to avoid thrashing between high and low levels of detail. This is the mechanism used by IRIS Performer™[Rohlf94]. This technique alone is quite effective at reducing load in the geometry subsystem because object levels of detail are usually modeled with fewer vertices and polygons. The raster subsystem will see some load reduction if lower levels of detail use simpler fill algorithms, however, they will probably still require writing the same number of pixels. If the system supports it, variable screen resolution is one way to address fill limitation — though this is traditionally only available on high-end image generators. Another trick is to aggressively scale down LODs as a function of distance so that distant objects are not drawn. A fog band makes this less noticeable. However, since they will be small, they may not account for very many pixels. LOD management based on performance prediction of objects in various stage of the graphics pipeline[Funk93] can aid in choosing appropriate levels of detail. Since the computation for load management might be somewhat expensive (calculating distances, averages over previous frames, etc.) it is best done in some process other than the rendering process.

Multiple Machines

Entertainment applications typically have multiple viewpoints that must be rendered and may require multiple graphics systems. If it is desired that these channels display synchronously, then the graphics output must be synchronized, as well as the host applications driving them. There is typically some mechanism to synchronize multiple video signals. However, double-buffered machines must swap buffers during the same video refresh period. This can be done reasonably well from the front end via a high-speed network such as SCRAMnet, as was done in the CAVE environment[CN93], or with special external signals, as is done on the RealityEngine™.

9 Tuning Tools

The proper tuning tools are necessary to simply evaluate an application's performance, let alone tune it. This section describes both diagnostics that you might build into your application, and some tools that the author has found to be useful in the past.

Graphics Tuning Tools

A couple of standard tools for debugging and tuning graphics applications found to be useful on Silicon Graphics machines are **`GLdebug`** and **`GLprof`**, described in detail in [GLPTT92]. **`GLdebug`** is a tracing tool that allows you to trace the graphics calls of an application. This is quite useful because most performance bugs, such as sending down redundant normals or drawing things twice, have no obvious visual cue. The tool can also generate C code that can be used (with some massaging) to write a benchmark for the scene. **`GLprof`** is a graphics execution proffer that collects statistics for a scene and can also simulate the graphics pipeline and display pipeline bottlenecks (host, transform, geometry, scan-conversion, and fill) over the course of a frame. The **`GLprof`** statistics include counts for triangles in different modes, mode changes, matrix transformations, and also the number of polygons of different sizes in different fill modes.

System Tuning Tools

Some of the tools in the standard UNIX environment are also very useful. **`prof`**, a general profiler which does run-time sampling of program execution, allows you to find *hot spots* of execution.

Silicon Graphics provides some additional tools to help with system and real-time tuning. **`pixie`** is an extension to **`prof`** and does basic block counting and supports simulation of different target CPUs. **`par`** is a useful system tool that allows you to trace system and scheduling activity. Silicon Graphics machines also have a general system monitoring tool, **`osview`**, that allows you to externally monitor detailed system activity, including CPU load, CPU time spent in user code, interrupts, and the OS, virtual memory operations, graphics system operations, system calls, network activity, and more.

For more detailed performance monitoring of individual applications, Silicon Graphics provides a product called WorkShop that is part of the CASEVision™ tools which is a full environment for sophisticated multiprocess debugging or tuning[CASE94]. For monitoring of real-time performance of multiprocessed applications, there is the WindView™ for IRIX product based on the WindView™ product from WindRiver. WindView works with IRIX REACT to monitor use of synchronization primitives, context switching, waiting on system resources, and tracks user-defined events with time-stamps. The results are displayed in a clear graphical form. Additionally, there is the Performance Co-Pilot™ product from Silicon Graphics that can be used for full-system real-time performance analysis and tuning.

Real-Time Diagnostics

The most valuable tools may be the ones you write yourself as it is terribly difficult for outside tools to non-invasively evaluate a real-time application. Real-time diagnostics built into the application are useful for debugging, tuning, and even load-management. There are four main types of statistics: system statistics, process timing statistics, statistics on traversal operations, and statistics on frame geometry.

Applications should be self-profiling in real-time.

System statistics include host processor load, graphics utilization, time spent in system code, virtual memory operations, etc. The operating system should allow you to enable monitoring and periodic querying of these types of statistics.

Process time-stamps are taken by the processes themselves at the start and end of important operations. It is tremendously useful to keep time-stamps over several frames and then display the results as timing bars relative to frame boundaries. This allows one to monitor the timing behavior of different processes in real-time as the system runs. By examining the timing history, one can keep track of the average time each task takes for a frame, and can also detect if any task ever extends past a frame boundary. The standard deviation of task times will show the stability of the system. Process timing statistics from IRIS Performer™ are shown in Figure 16. Geometry statistics can keep track of the number of polygons in a frame, the ratio of polygons to leaf nodes in the database, frequency of mode changes, and average triangle mesh lengths. IRIS Performer™ displays a histogram of tmesh lengths, shown in the statistics above in Figure 16.

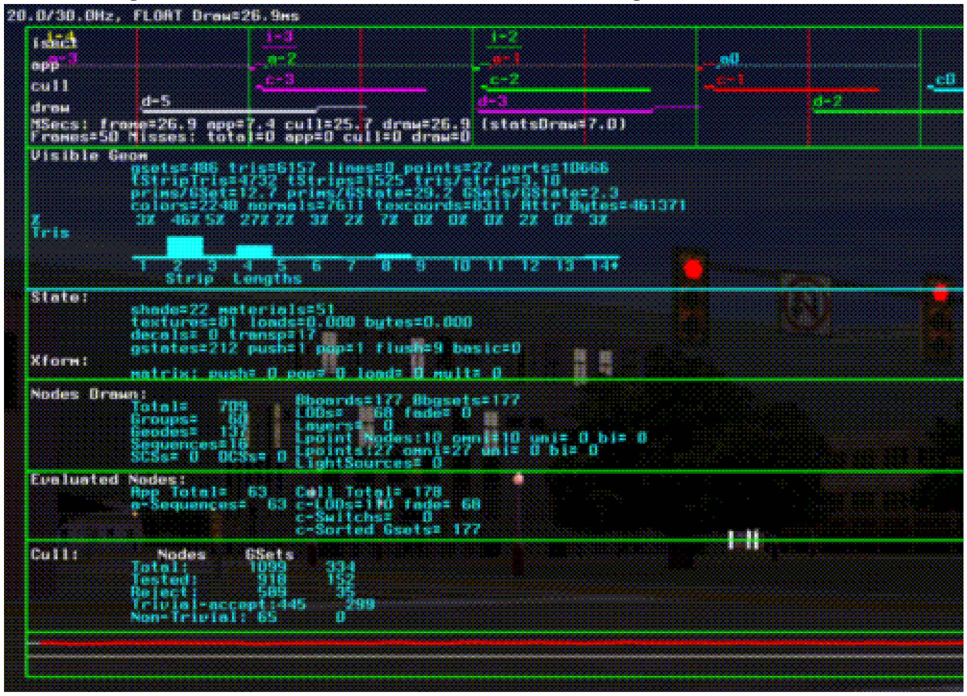


FIGURE 16. Process and Database Statistics

Traversal and geometry statistics do not need to be real-time, and may actually slow traversal operations. Therefore, they should only be enabled selectively while tuning the traversals and database. Traversal statistics can keep track of the number of different types of nodes traversed, the number of different types of operations performed, and perhaps statistics on their results. The culling traversal should keep track of the number of nodes traversed vs. the number that are trivially rejected as being completely outside the viewing frustum. A high number of trivial rejections means that the database is not spatially well organized because the travsersal should not have to examine many of those nodes.

Additionally, IRIS Performer™ supports the display of depth complexity, where the scene is painted according to how many times pixels are touched.The painted framebuffer is then read back to the host for analysis of depth complexity. This display is comfortably interactive on a VGX™ or RealityEngine™ due to special hardware support for logical operations and stenciling. Thus, you can actually drive through your database and examine depth complexity in real-time.

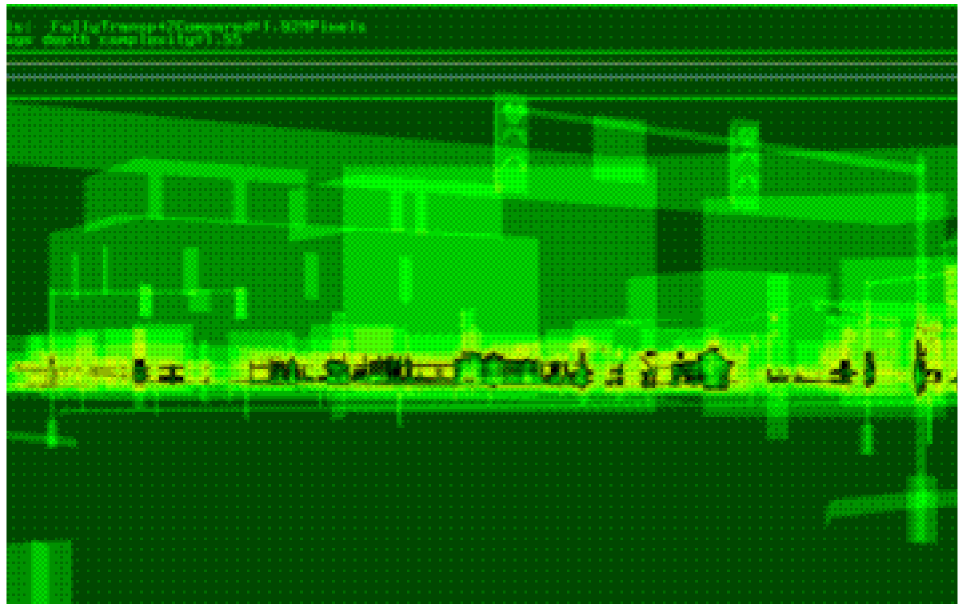


FIGURE 17. Pixel Depth Complexity Profile

10 Conclusion

Graphics workstations offer a wide array of options for balancing cost, performance, and image quality. Rich development environments plus real-time image-generator features like texture mapping and full scene anti-aliasing make graphics workstations attractive platforms for both the development and deployment of entertainment applications. Understanding the target graphics architecture and designing for performance will enable you to get the scene quality and performance you paid for.

11 Acknowledgments

The helpful comments and careful review by Rosemary Chang and Kevin Hunter, both of Silicon Graphics, greatly contributed to the quality of this paper. Their efforts and time are very much appreciated.

12 References

[Akeley89] Kurt Akeley, "The Silicon Graphics 4D/240GTX Superworkstation", *CG&A*, Vol. 9, No. 4, July 89, pp. 71-83.

[Akeley93] Kurt Akeley,"RealityEngine Graphics", *Proceedings of SIGGRAPH '93*, July 93, pp. 109-116.

[CASE94] *CASEVision*TM*/WorkShop User's Guide*, Document #007-1523-040, Silicon Graphics., 1994.

[CN93] C. Cruz-Neira, D. J. Sandin, and T.a A. DeFanti. "Surround-screen projection -based virtual reality: The design and implementation of the cave," *Proceedings of SIGGRAPH '93*, July 93, pp. 135-142.

[Deering93] Michael F. Deering"Leo: A System for Cost Effective 3D Shaded Graphics", *Proceedings of SIGGRAPH '93*, July 93, pp. 101-108.

[Foley90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics Principles and Practice, 2nd. Ed.*, Addison-Wesley, 1990.

[Fuchs89] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, L. Israel, , "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories", *Proceedings of SIGGRAPH '89, July 89*, pp. 79-88.

[Funk93] Tom Funkhouser and Carlo Sequin, "Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments", *Proceedings of SIGGRAPH '93*, July 93, pp. 247-254.

[GLPTT92] *Graphics Library Programming Tools and Techniques*, Document #007-1489-010, Silicon Graphics., 1992.

[Haeberli90] Paul Haeberli, Kurt Akeley, "The Accumulation Buffer: Hardware Support for High Quality Rendering", *Proceesings of SIGGRAPH '90, Aug. '90,* pp. 309-318.

[Harrell93] Chandlee B. Harrell and Farhad Fouladi, "Graphics Rendering Architecture for a High Performance Desktop Workstation", *Proceedings of SIGGRAPH '93*, July 93, pp. 93-99

[Molnar92] Steven Molnar, John Eyles, and John Poulton, "PixelFlow: High-Speed Rendering Using Image Composition", *Proceedings of SIGGRAPH '92, July 92*, pp. 231-240.

[OpenGL93] Jackie Neider, Tom Davis, Mason Woo, OpenGLTM Programming Guide,Addison Wesley., 1993.

[PFPG94] *IRIS Performer*TM *Programming Guide,* Document #007-1680-020, IRIS PerformerTM, Silicon Graphics., 1994.

[Potmesil89] Michael Potmesil and Eric M. Hoffert, "The Pixel Machine: A Parallel Image Com puter", Proceedings of SIGGRAPH '89, July 89, pp. 69-70.

[Rohlf94] John Rohlf and James Helman, "IRIS Performer: A High-Performance Multiprocess ing Toolkit for Real-Time 3D Graphics", Proceedings of SIGGRAPH '94, July 94, pp. 381-394.

[Regan94] Mathew Regan, Ronald Pose, "Priority Rendering with a Virtual Reality Address Recalculation Pipeline.", Proceedings of SIGGRAPH '94, July 94, pp. 155-162.

[Voorhie89] Doug Voorhies, "Reduced-complexity Graphics", *CG&A*, Vol. 9, No. 4, July 89, pp. 63-70.