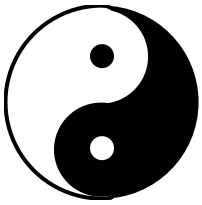
Developing Efficient Graphics Software:
The Yin and Yang of Graphics



A SIGGRAPH 2002 Course

Course Organizer

Keith Cok

SGI

Course Speakers

Keith Cok

Thomas True

SGI

Contents

List of Figures

List of Tables

Preface

About the Speakers

Keith Cok
SGI
18201 Von Karman Ave., Suite 100, Irvine CA 92612
cok@sgi.com

Keith Cok is an Engineering Program Manager at SGI. He currently manages a government related program at SGI and also consults with software developers in porting, optimizing, and differentiating their graphics applications. His primary interests include high-performance graphics, animation, scientific visualization, and simulation of natural phenomena. Prior to joining SGI, he spent time as an independent software developer writing a particle animation system used in several television and film shots. He also worked at TRW designing spacecraft and astronaut training simulators for NASA. Keith has given a number of technical talks at various local and international conferences. Keith received a BS in Mathematics from Calvin College, Michigan and an MS in Computer Science from Purdue University.

Thomas True
SGI
1600 Amphitheatre Pkwy., Mountain View, CA 94043
true@sgi.com

Thomas True is a Member of the Technical Staff at SGI where he currently works developing novel ways to utilize system hardware and software to create high-performance graphics software applications. In this role, he assists software developers in tuning their graphics applications. His primary areas of interest include low-level graphics system software, graphics APIs, user interaction, digital media, rendering, and animation. He received an MS in Computer Science from Brown University in 1992 when he completed his thesis on volume warping under the direction of Dr. John Hughes and Dr. Andries van Dam. He presented this research at IEEE Visualization '92. He received a BS in Computer Science from the Rochester Institute of Technology in 1988. Prior to joining SGI, Thomas developed graphics system software at Digital Equipment Corporation.

Acknowledgments

This course is based on our experience with real applications outside of SGI or in conjunction with partnerships through SGI Applications Consulting. We thank all of the graphics software developers and researchers who are pushing the envelope in graphics technology; without them, there would be no content for this course.

We also thank our management, Brian Thatch and Ann Johnson, for giving us the opportunity to develop this course and the course reviewers who gave us much needed feedback.

We gratefully acknowledge the past contributors to this course: Alan Commike, Roger Corron, and Bob Kuehne.

Course Resources on the Web

The course notes and slides are available on the SGI Developer's Toolbox (free registration required):

`http://toolbox.sgi.com/toolbox/documents/OpenGL/DEGS`

Section 1

Course Introduction

Computer systems, and in particular graphics systems, change quickly. Vendors introduce new products several times a year and new features and better performance are available with accelerating frequency. It is not surprising then that a common misconception in the computing industry today is that to make slow software work more quickly, you simply obtain a bigger and faster computer. However, this approach is expensive and often unworkable. Anticipated performance is often disappointing and published benchmarks are often misleading. A more feasible and cost-effective approach to improving software performance is to measure the current software performance, and then optimize the software to meet the anticipated graphics and system performance.

This SIGGRAPH 2002 course was developed for those software graphics developers who are interested in developing interactive graphics applications that perform well. The course is not targeted at a specific class of graphics applications, such as visual simulation or CAD, but instead focuses on the general elements required for highly interactive 2D and 3D applications.

Any graphics application has bottlenecks or areas within an application that limit overall performance. Simply buying faster graphics may not increase performance, as the bottleneck may lie elsewhere in the system. Therefore, it is important to understand where a bottleneck is and understand its underlying cause. So, this course starts with an overview in section 2 of the different components of a graphics computer system and how those components interact with an application.

Simply eliminating a bottleneck is not enough, because other bottlenecks will appear. In addition, bottlenecks change from system to system, so performance on one machine does not imply proportional performance on another machine. Fortunately, the goal in tuning an application is not to merely eliminate all the bottlenecks - an impossible task. A better goal is to achieve a *balance* across the different hardware components and subsystems. A useful metaphor for this balance (and fun diversion from the topic of computer hardware) is the Chinese concept of yin and yang. Quoting from the Skeptics Dictionary (`http://skepdic.com/yinyang.html`):

> According to traditional Chinese philosophy, yin and yang are the two primal cosmic principles of the universe. Yin (Mandarin for moon) is the passive, female principle. Yang (Mandarin for sun) is the active, masculine principle. According to legend, the Chinese emperor Fu Hsi claimed that the best state for everything in the universe is a state of harmony represented by a balance of yin and yang.

Although the ideas behind yin and yang do not exactly map to the main goal of application tuning, the basic concept of balance is key. If the repurposing of this ancient Chinese philosophy can be forgiven, the goal in tuning an application is to obtain harmony, a state of blissful balancing of application load across

the hardware provided in a computer. Throughout the remainder of this course, the yin/yang symbol appears in the margin to denote a section of interest that discusses harmonious application balance. A consequence of trying to obtain balanced hardware usage is the need to understand how that hardware operates so that an application can best take advantage of it.

Finding a bottleneck is useless unless you can fix it. So, this course introduces in sections 3 and 4 alternate ways to create and structure applications more efficiently. Of course, the additional performance must be verified and sections 5 and 6 explain techniques to quantify and optimize application performance.

In these sections, the course also uses another icon: the winged foot of Mercury. This icon indicates an explicit performance hint or suggestion. The goal of this course is not, however, to give explicit hints, but to encourage overall understanding of an application and its interaction with the computer on which it runs. Therefore, scanning the course for these icons and following the hint without understanding the surrounding concepts and content will not be of much value. Furthermore, much larger performance increases can be obtained by implementing the concept, as opposed to implementing a specific suggestion. Be sure that you understand why a particular suggestion is given, where it will work, and most importantly, the context of the section surrounding the suggestion.

Section 2

System Design and Architecture

The computer hardware on which an application runs can vary dramatically from system to system and vendor to vendor. Therefore, understanding some of the architectural issues of hardware systems can improve your understanding of application hardware utilization. Tuning an application based on this understanding can, in turn, lead to overall application and graphics performance improvements through more effective use of hardware resources. This section describes application and hardware interaction topics that you should consider when you access the performance of a graphics application.

2.1 Data Access Rates

Before diving in and examining the details of computer architecture, it is important to first understand two key measures of performance that are relevant to computer design. These performance metrics are *bandwidth* and *latency*. Bandwidth is the amount of data per time unit that can be transmitted to a device. Latency is the amount of time it takes to fully transfer a single unit of data to a device. The difference between the two is quite clear, but the interaction between the two is not.

Different hardware systems often have very different bandwidth abilities in different portions of a system. For example, the 33-MHz, 32-bit PCI bus has a theoretical bandwidth of 133 Mbytes/second(MB/s), calculated simply by multiplying 33M cycles/second * 32 bits/cycle (or 4 bytes/cycle) to yield 133 MB/s. The 66-MHz, 32-bit AGP graphics bus has a theoretical bandwidth of either 264 MB/s (or 528 MB/s depending on whether data transfer happens on both edges of the clock cycle). Other systems have vastly greater bandwidths.

Latency can be measured between many points in a system, so it is helpful to know where latency is important to an application. Profiling an application can yield insight into where critical latencies are encountered. Profiling is discussed in later sections, but the key result of profiling shows which routines take up the most time. These time-intensive routines can be either computationally complex or doing much simpler tasks that are latency critical. Latencies vary dramatically within a system. For example, network latencies can be many milliseconds (or even seconds), whereas latencies for data in L2 cache operate in tens of nanoseconds (Figure 2.1).

Now that a few typical latencies and bandwidths have been discussed, how do the two interact? When transferring data from one piece of hardware to another, both measures are important. Latency is most often a factor when many operations are being performed, each with a latency that is large relative to length of the overall operation. Latency is critical when accessing memory; for example, as the access times for portions of main memory are slower than those of cache memories.

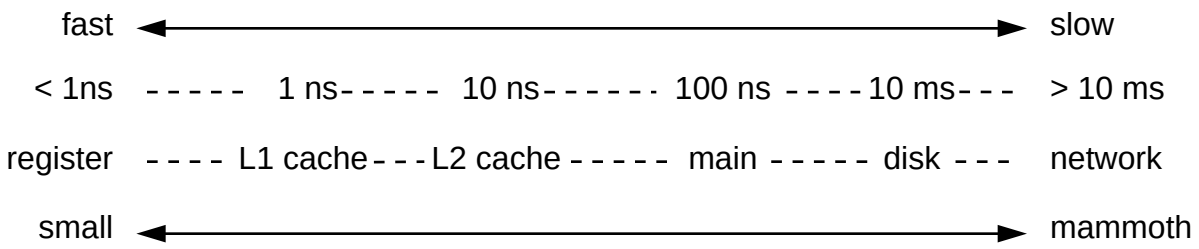


Figure 2.1: Approximate Data Latencies and Capacities of Typical System Components.

A hypothetical graphics device is used to illustrate the effects that latencies can have on a running program. Assume that this system consists of a data source (memory) and a data sink (graphics) where the bandwidth between source and sink is 1 MB/s and the latency is 100 ms. The hypothetical application programming interface (API) in this example is a call that blocks (is synchronous) while downloading a texture. The transfer time for a 100-MB download of a texture (assuming no other delays in retrieving the data) then takes 100 seconds. Because the latency involved in transferring this texture is 100 ms or 0.1 second, then the overall time to transfer this texture is 100.1 seconds. However, if 100 1-MB textures are downloaded, the transfer time per texture is 1 second, for a total of 100 seconds. Adding in the latency of 0.1 second per texture, we add a cumulative additional 10 seconds, which increases the total transfer time by 10% or 110 seconds total. A developer who is aware of this issue could design methodologies such as creating a large texture with many small subtextures within it to avoid many small data transfers that could negatively impact performance. Though contrived, this example illustrates that latency can be an issue that affects application performance, and that developers must be aware of hardware latencies so that the effects can be minimized.

2.2 Component I/O

Computer systems are constructed from a wide variety of components, each with its own characteristics. Aside from the obvious differences in core functionality among network interfaces, hard disk drives, graphics accelerators, and serial port controllers are the less obvious differences in the way these systems respond to input.

Some systems are said to *block* when input or output is requested. Blocking is the process of preventing the controlling program from proceeding in its current thread of execution until the device being communicated with finishes its operation. Blocking operation of a system is also known as *synchronous operation*. Other devices operate asynchronously, or in a non-blocking mode, allowing data to be queried and program execution to continue regardless of the guery result.

Other differences among devices are the rates at which they can communicate data back to the host, the latency involved in these data transfers, and how various buffers and caches mitigate the effects of these differences among devices. Subsequent sections discuss these issues in more detail.

The architecture of a specific computer system is important to consider when designing software for that system. Specifically, its important to consider which subsystems an application interacts with and how that interaction occurs. There are several distinct systems on a computer, each of which uses some interconnect fabric or “glue” (shown as a single block in Figure 2.2) to communicate with one another. Understanding this fabric and where the devices are located on this fabric is extremely important in both determining where application bottlenecks occur and avoiding bottlenecks when designing new software systems.

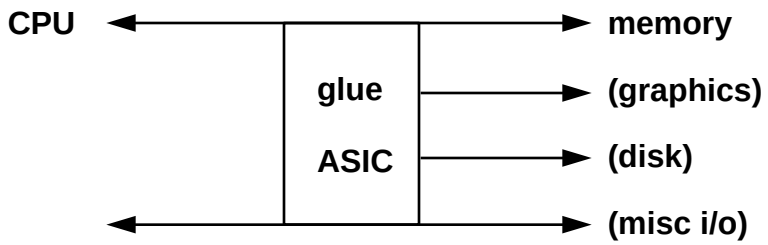


Figure 2.2: An Abstract Computer System Fabric.

Interconnect fabrics vary dramatically from system to system. On low-end systems, the fabric is often a bus on which all devices share access through some hardware arbitration mechanism. The fabric can be a point-to-point peer connection, which allows individual devices to communicate with preallocated guaranteed bandwidth. In other fabrics, some systems might live on a bus, while others in that system live on a peer interface. The differences in application performance among these types of systems can be dramatic depending on how an application uses various components.

Because the focus of this course is on writing graphics applications, it is especially important to understand the specifics of how graphics hardware interfaces with CPU, memory, and disk. A diverse mix of computer systems exists on which an application might be run. This diversity ranges from systems that have a shared bus (PCI) with local texture and framebuffer, to systems that have a dedicated bus to the graphics (AGP) with some local texture cache, main memory texture cache, and local framebuffer, to systems on a dedicated bus with all texture and framebuffer allocated from main memory (Silicon Graphics® O2® visual workations). Each of these architectures has certain advantages and disadvantages, but you cannot expect an application to fully realize the performance of these platforms without considering the differences among them.

A concrete example of these differences is shared-bus systems. Graphics systems that use a shared-bus architecture share bandwidth with other devices on that bus. This sharing impacts applications that are attempting to transfer large amounts of data to or from the graphics pipe while other devices are using the bus. Large texture downloads, framebuffer readbacks, or other high-bandwidth uses of the graphics hardware are likely to encounter bottlenecks as other parts of the system utilize the bus. Regardless of the type of system that is used, the key to high-performance applications is to fully utilize the entire system, balancing the workload among all of the components that are needed so that more application work can be performed more quickly.

2.3 Memory

Previous sections have described the effects of latencies and bandwidths on hypothetical activities. This section of the course discusses memory hierarchies and how applications interact with data within memory. This section also describes how memory hierarchies work in general, but many details are beyond the scope of this course, such as instruction vs. data caches, details behind cache mappings (direct, n-way associative), translation look-aside buffers, and many others.

2.3.1 CPU and Memory Interaction

Before jumping into the discussion on memory, it is important to understand how the system CPU interacts with memory. Figure 2.3 depicts a simplistic CPU to illustrate the lengthy path that application data must

travel before it is useful. In this figure, main memory lives on the far side of all of the caches, and data must be successively cached down to the registers before it can be operated on by the CPU. This means that keeping often-used data localized in memory is a good idea, as it can improve cache efficiencies dramatically. In fact, the premise behind caches is that the data that is near the data that is currently being operated upon is much more likely to be needed next. This design criterion means that data locality affects performance, because access to cache memory is significantly faster than to main memory.

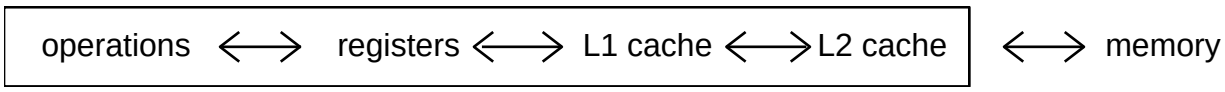


Figure 2.3: Abstract CPU.

Application data transfers to the graphics hardware that avoid pushing data through the CPU can significantly improve performance. Graphics structures such as OpenGL® display lists and Microsoft® Direct3D® vertex buffers often can be pre-compiled into a state so that a single function call simply transfers the display list directly from main memory to the graphics hardware via a technique such as direct memory access (DMA). This technique allows large amounts of graphics data to be rendered without any complex calculations occurring on that data at run time.

2.3.2 Virtual Memory

Most current operating systems work under a memory scheme known as *virtual memory*. Virtual memory is a method of managing memory that allows applications access to data storage space that is sized significantly larger than the amount of physical RAM in a system. Addressing schemes vary, but 32-bit applications can typically address more than GB of memory when only a small fraction of that is physically available. Virtual memory systems perform this task by managing a list of active memory segments known as *pages*. For details behind this operation, and for a general discussion of computer systems, see *Principles of Computer Architecture* [43] or a good introductory computer architecture book for elaboration.

Pages of memory are blocks of *address space* of a fixed size. Memory address space is simply a hardware mapping of all available memory locations to a numbering scheme. A simplistic mapping for a 16-byte memory system might have valid memory addresses of 0x00 to 0x10. The size of pages of memory varies from system to system but is typically constant on a specific running system. However, many hardware systems allow the page size to be changed; some operating systems allow this to be changed dynamically as a tunable parameter. Knowing the page size and page boundary for the specific system on which an application is running can be very useful. Specific page sizes and functions to retrieve page size and page boundary vary by operating system. Pages are important structures to understand because they are used as the coarsest level of data caching that occurs in virtual memory systems.

As applications use memory and address space for code and data storage, more and more pages of that address space are allocated and used. Eventually, more pages are in use than are available in physical system RAM. At that point, the virtual memory manager decides to move some infrequently used pages for that application from main memory to disk. This process is known as *paging*. Each time a page of memory is requested, the memory manager determines whether it already exists in main memory. If it does, no action is required; if it does not, the memory manager determines whether space is available in RAM for that page. If space is available in RAM for the needed page, no action is required. If space is not available, a page of resident data must be written to disk before reading the desired page from

disk. In all cases, the desired page is then copied from disk to the available page location in RAM. When an application pages, disk I/O occurs, which impacts both the application and the overall system performance. Because maintaining the integrity of a running application is essential, the paging process operates in a fairly resource-intensive way to ensure that data is properly preserved. Because of these constraints, keeping data in as few pages as possible is important to ensure high-performance applications. Applications that typically use very large datasets, which cause the system to page, may benefit from implementing its own data paging strategy. Application-specific paging can be written to be much more efficient than the general OS paging mechanism.

Figure 2.4 shows a hypothetical application with an address space that ranges from page 0 to page *n*, and a system with many physical pages of RAM. In this example application, pages 0 through 9 are active; the application has stored data in them, and pages 0 and 1 are physically resident in RAM. For this example, the memory manager has decreed that the application can use only two pages, so any application data that resides on pages other than the two pages in RAM are paged to and from disk.

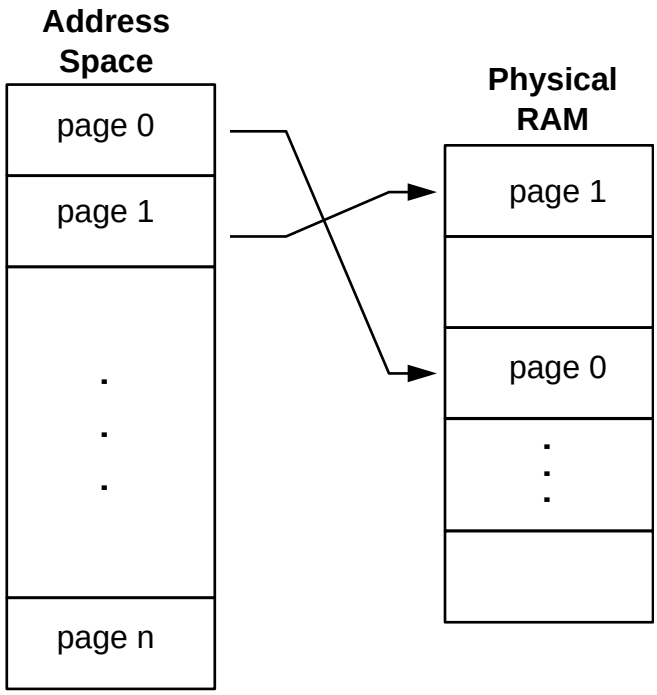


Figure 2.4: Virtual Memory Mapping Active Pages into RAM.

If the application in this example needs to retrieve vertex data from each of the 10 pages in use by the application, then each page must be cached into RAM. This process likely will require eight paging operations, which can be quite expensive given that disk access is slower than RAM access. However, if the application could rearrange data such that it all resided on one page, the virtual memory manager would not be required to page, and access times for this data would improve dramatically. This property of one piece of data residing "close" to another piece of data in memory is known as *data locality*. If data locality can be improved by storing frequently-used data in adjacent memory, performance may improve as well. Understanding data access patterns is the key to understanding and improving data locality.

When data resides on pages in main memory, it then must be transferred to the CPU (see Figure 2.3) for operations to be performed on it. The process by which data is copied from main memory into cache memory is similar to the process by which data is paged into main memory. As memory locations are

required by the operating program, they must be copied into the registers. Figure 2.5 shows the data arrangement of cache lines in pages and both caches.

To get data to the registers, active data must first be cached into L2 and then L1 caches. Data is transferred from pages in main memory to L2 cache in chunks known as *cache lines*. A cache line is a linear block of address space of a system-dependent size. Level-2 (L2) caches are typically sized between 32 and 128 bytes in length. As data is required by the CPU, data from L2 cache must be copied into a faster level-1 (L1) cache of a system-dependent size, typically 32 bytes. Finally, the actual data required from within the L1 cache is copied into the registers where the CPU operates on it. This is the physical path through which data must flow to enable the CPU to operate on it.

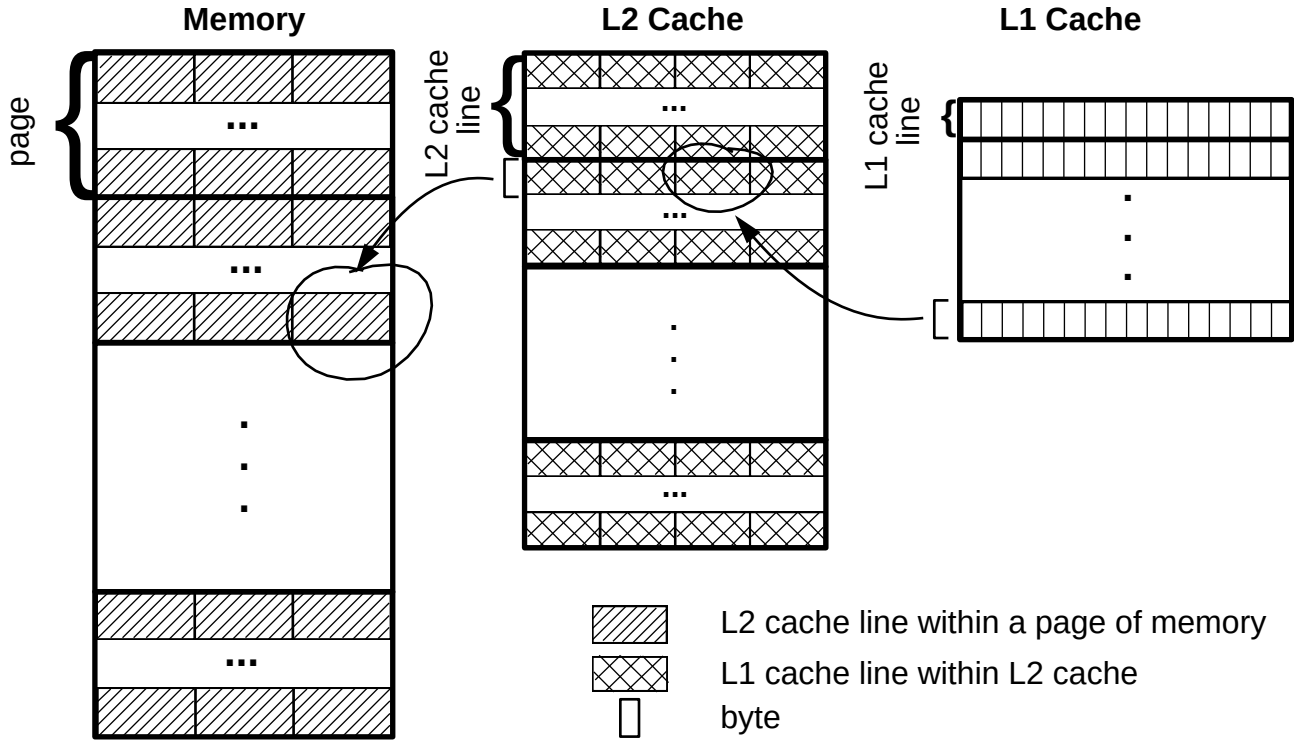


Figure 2.5: Cache Line Structure. Pages of Memory Composed of Multiple L2 Cache Lines; L2 Cache Composed of Multiple L1 Cache Lines; and L1 Cache Composed of Individual Bytes.

The process by which requested data is copied into the registers is important because the consequences of its action are one of the primary factors that limit application performance. As data is needed by the CPU, controlling circuitry checks to see if that data is in the registers. If the data is not immediately available, the controller checks the L1 cache for the data. If it is again unavailable, the controller checks the L2 cache. Finally, if the data is still not available, a cache line that contains the required data is copied from a page in main memory (assuming that the page is already resident in RAM, and not paged to disk) and is propagated through L2 and L1 cache, ultimately depositing the requested data in a register. This process is depicted in Figure 2.6, which shows the data request procedure as a flowchart.

Though this discussion of memory and how it works is straightforward, the relevance to application performance may not be immediately clear. *Data locality* is the ultimate point of any discussion of how memory works. Keeping data closer together keeps data in faster and faster memories in the memory hierarchy. Conversely, data that is widely dispersed in memory is accessed through slower layers in the memory hierarchy. The effects of data locality are best demonstrated through two examples.

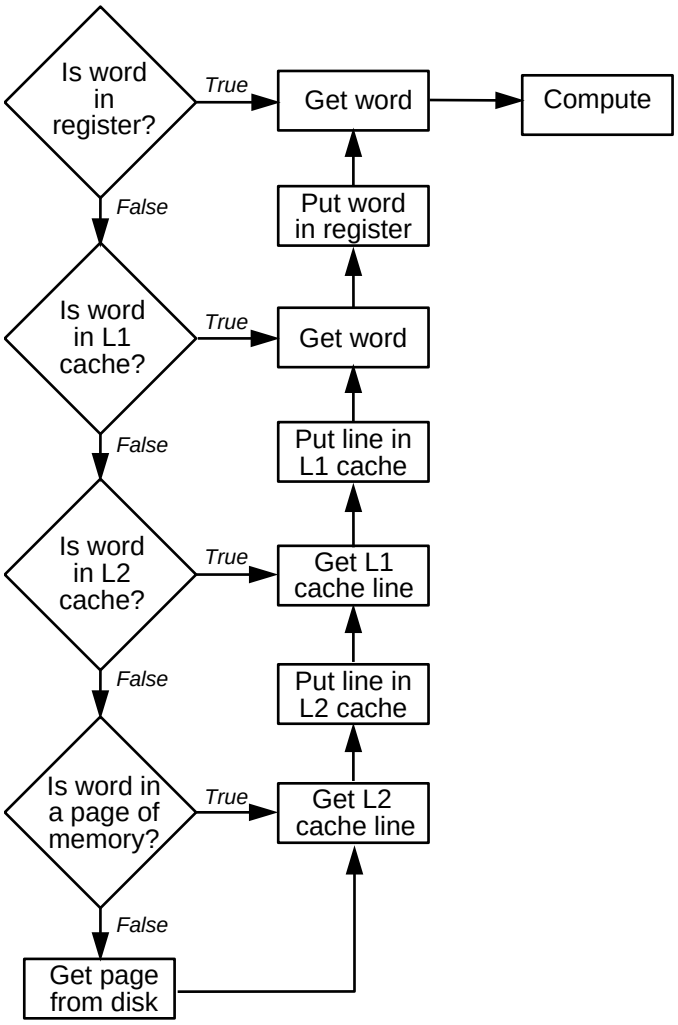


Figure 2.6: Register Data Request Flowchart.

In these examples, the CPU is performing an operation that requires 2 bytes of data, each in a register. The computer on which this operation is running has the following access times: L1 cache, 1 ns; L2 cache, 10 ns; main memory, 100 ns. These access times are the largest contributors to overall data access time. In the first example, the 2 bytes of data are resident on two different pages of memory, so for each byte of data to be accessed, a cache line must be copied from main memory into the cache. Thus, to access main memory, it takes 100 ns + the L2 cache access time (10 ns) + the L1 cache time (1 ns), or 111 ns for each data byte to be copied from main memory to a register. Therefore, for the first example, the total time to prepare memory for the operation to occur is 222 ns. Note that in this example, the 2 bytes are the data of interest, but complete L2 cache lines that contain the bytes of interest are copied from main memory, and L1 cache lines that contain the bytes are copied from L2 cache to L1 cache. Finally, a word that contains each byte of interest is copied to each register location.

In the second example, both data bytes live on the same page in memory and on the same L2 cache line (though far enough apart that they don't fit on the same L1 cache line). This operation requires a much smaller time to set up than the operation in the first example. Again, it takes 100 ns to access the main memory page to copy data to L2 cache, 10 ns to access the L2 cache twice to copy each byte to a different L1 cache line, and two 1-ns accesses of the L1 cache to load the registers. In this example, the total time

to prepare the operation is 122 ns, which is nearly half of the previous example's overall time. As these examples show, localizing data can clearly benefit application performance. Remember this cache effect when you design graphics data structures to hold objects to be rendered, state information, visibility lists, and so forth. Simple changes in the data structure organization might gain a few frames per second in the application frame rate.

Another example of how data locality can be advantageous to a graphics application is through a graphics construct known as a *vertex array*. Vertex arrays allow the CPU to efficiently utilize graphics data for purposes such as transformations and lighting. This efficiency is primarily because vertex arrays are arranged contiguously in memory; therefore, subsequent accesses to vertex data are likely to be found in a cache. For example, if a hypothetical L2 cache uses 128-byte lines, then four 32-bit floats can live on a single cache line which allows fast access to each of them. However, because most applications do more than render flat-shaded triangles, these vertices need normals too. If a large contiguous array is allocated in memory for the vertices, another for the normals, another for the color, and so on, it is possible that — due to the implementations of the L2 caches — these arrays may overlap in cache and still incur trips to main memory for access. Interleaved vertex arrays are a solution to this problem. In this case, vertex, normal, and color data are arrayed one after another in memory; therefore, in a 128-byte cache line implementation, all three are likely to live in nonoverlapping L2 cache at once, thus improving performance.

A number of techniques exist for mitigating the effects of cache on data access performance; however, these techniques are more adequately addressed in later sections of this course, which discuss language and code optimizations.

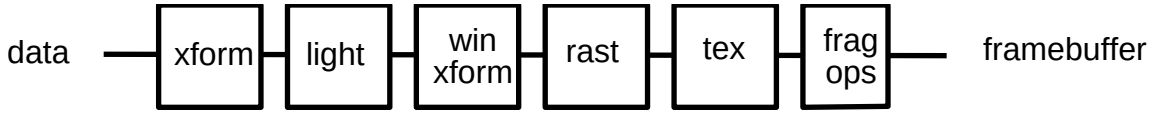
Understanding the path through which data must flow to the CPU is key because of the latencies involved in accessing data from various memory caches. Keeping data packed closely in memory ensures that subsequent data accesses will occur from memory that is already resident in cache; and therefore, the algorithms operating on that data will be much faster.

2.4 Graphics

The graphics subsystem is responsible for rendering and displaying application data. The rendering process, also know as the *graphics pipeline*, is typically implemented as a combination of CPU-based software and dedicated graphics hardware. The hardware functionality within this subsystem and the physical connection between it and the other parts of a system play a large role in the overall performance of a graphics application. This section reviews the graphics rendering pipeline and describes how special-purpose dedicated hardware can be used to implement it as well as the impact these different hardware implementations have on overall application performance.

2.4.1 Graphics Pipeline

The process of rendering interactive graphics can best be described as a series of distinct operations that are performed on a set of input data. This data, often referred to as a *primitive*, typically takes the form of triangles, triangle strips, image data, points, and lines. Each primitive enters the process as a set of vertex data in a world coordinate system and leaves as a set of pixels in the framebuffer. The set of stages, which performs this transformation, is known collectively as the graphics pipeline. This pipeline, shown in Figure 2.7, is implemented within the system by a combination of dedicated graphics hardware and host-based software.



xform:	Transform geometry to eye space
light:	Lighting operations
win xform:	Clipping, perspective transform, and viewport operations
rast:	Rasterize
tex:	Texture, fog, and anti−aliasing operations
frag ops:	Stencil, depth, alpha, and other operations

Figure 2.7: Graphics Pipeline.

When dedicated graphics hardware is available within a system, this hardware – commonly referred to as a graphics processing unit or GPU – implements various stages of the graphics pipeline while stages not explicitly implemented are still performed in software that is executing on the host CPU. More sophisticated GPUs implement more stages in hardware while less sophisticated systems leave more stages in software. Understanding the performance of a graphics application requires an understanding of how the GPU and the CPU divide the graphics processing.

The frontend of the graphics pipeline consists of the geometry processing stages. These geometry processing stages operate on vertex data. These stages map graphics primitives from world space into eye space, and then perform lighting and shading calculations prior to backface culling, transforming, and clipping the resulting primitives to the viewport. The final stages of primitive assembly and triangle setup convert the resulting primitives into fragments for rasterization. This geometry processing portion of the graphics pipeline is illustrated in Figure 2.8. On GPUs with hardware-accelerated transform and lighting (T & L) functionality, these stages of the graphics pipeline occur in dedicated circuitry that is commonly referred to as a transform engine. Operations not implemented by the hardware transform engine are performed by software that is running on the host CPU. And, as one might expect, the result of executing these operations on the host CPU is significantly reduced performance.

The backend of the graphics pipeline consists of the rasterization stages. These stages operate on screen-space fragments and determine the final colors of pixels within the framebuffer by performing texturing, filtering, blending and fog operations. Visibility testing, whether a pixel is visible, also occurs within these stages of the pipeline. This rasterization section of the graphics rendering pipeline is pictured in Figure 2.9. In hardware, these functions are implemented by the rendering engine. Most graphics systems today implement this functionality within the GPU.

The graphics pipeline describe above is a traditional rendering pipeline with fixed-function transform, lighting, and pixel shading functionality. As such, the graphics rendering hardware implements solely the fixed functions of graphics APIs for which it was designed. In recent years, however, GPU technology has advanced beyond this traditional fixed-function model to provide programmable capabilities [38, 25, 14]. Because the GPU is like a specially designed CPU for graphics processing, it is only natural that the GPU should evolve to become user programmable. In the GPU, this programmability takes the form of small assembly-language-like programs that replace the hardwired T & L and pixel shading functionality of traditional graphics processors.

In the geometry-processing section of the graphics pipeline these assembly-language-like programs take as input the current vertex coordinates and attributes in source registers along with the constants

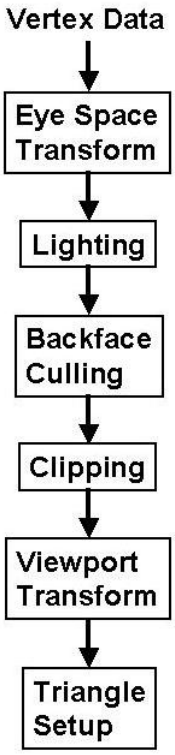


Figure 2.8: Geometry Processing Pipeline.

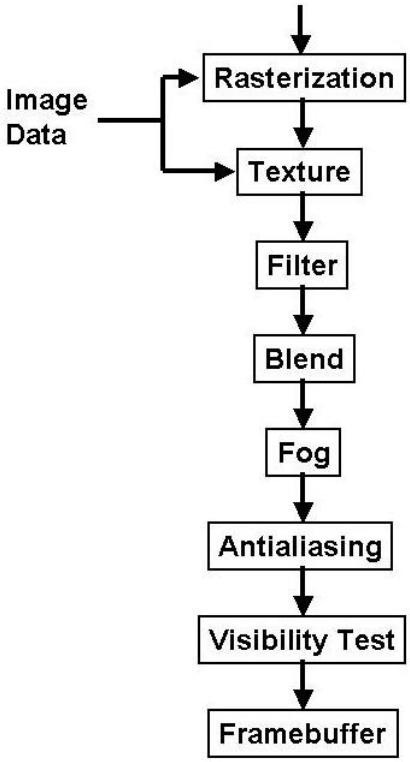


Figure 2.9: Rasterization Pipeline.

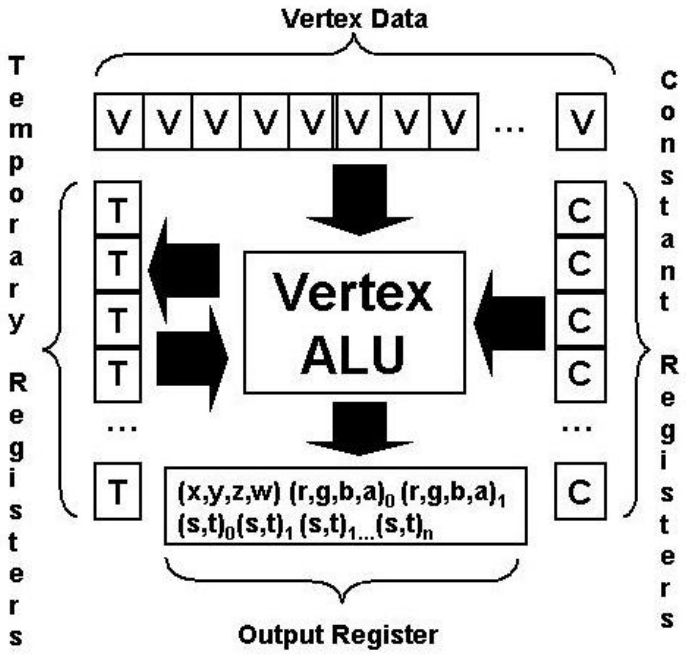


Figure 2.10: Programmable Vertex Shader Data Flow.

that contain the current transform and lighting parameters and produce as output the resulting vertex in output registers. This data flow is pictured in Figure 2.10. The input vertex data takes the form of streams of independent vertex data rather than primitives to exploit the parallel nature of the processing task. By implementing a SIMD programming model, the same instructions are executed for each vertex. Backface culling, clipping, and the viewport transform, which require primitive-relative information, are performed by subsequent implementation-specific processing techniques as they would be in the case of a fixed function pipeline. This programmable transform and lighting functionality within a GPU is typically referred to as programmable vertex shaders. The addition of this programmability feature to the geometric process section of the graphics pipeline is pictured in Figure 2.11.

For the rasterization section of the user-programmable graphics pipeline, assembly-language-like programs take as input multiple texel, color, and constant values and perform mathematical operations on these values to calculate the final color value for a pixel. These programs can be divided into two basic stages. The first stage samples the input textures by using the original input texture coordinates or new texture coordinates that result from program instructions. This stage permits dependent texture reads and is called the texture shader. The instructions in the second stage blend the sampled texture values with the input color values to calculate the final pixel color. This second stage is referred to as the color shader. The data flow is pictured in Figure 2.12. These assembly-language-like programmable units that operate on individual pixels within a GPU are commonly referred to as programmable pixel shaders. To see how programmable pixel shaders fit within the traditional rasterization section of the graphics pipline, refer to Figure 2.13.

Now, you might be asking yourself why we went through all this. Well, all of this has an impact on performance. Complex lighting and shading models that previously were not implemented by the fixed-function processing within the GPU and as a result required software processing on the host CPU can now be performed faster by custom vertex shading programs that execute on the GPU. And, pixel shading operations that required multiple passes to implement all the texturing and blending operations, can now

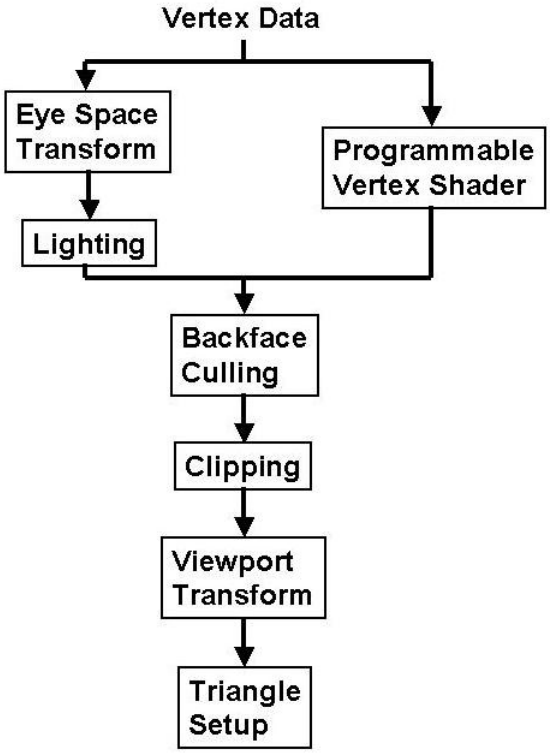


Figure 2.11: Geometry Process Pipeline with Programmable Vertex Shaders.

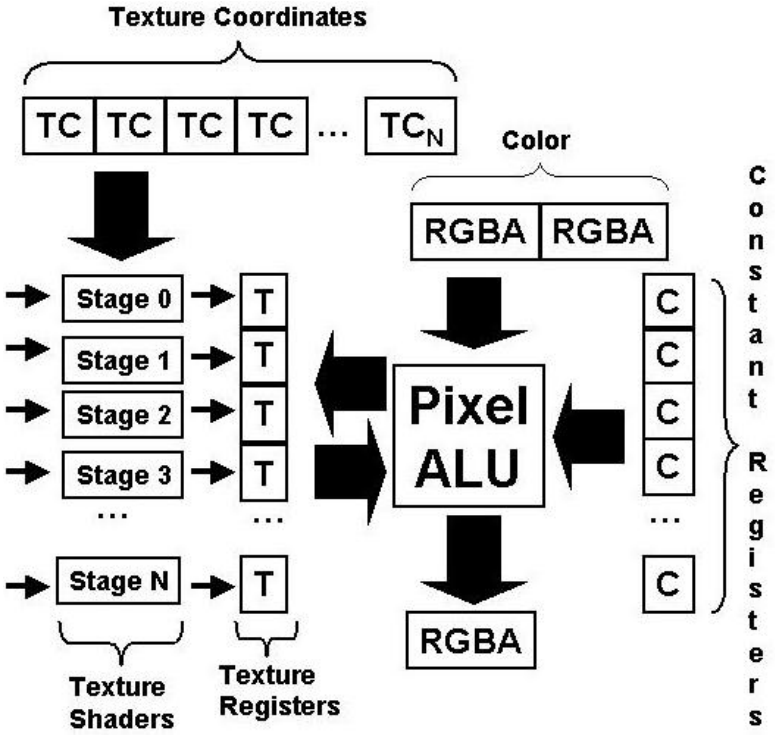


Figure 2.12: Programmable Pixel Shader Data Flow.

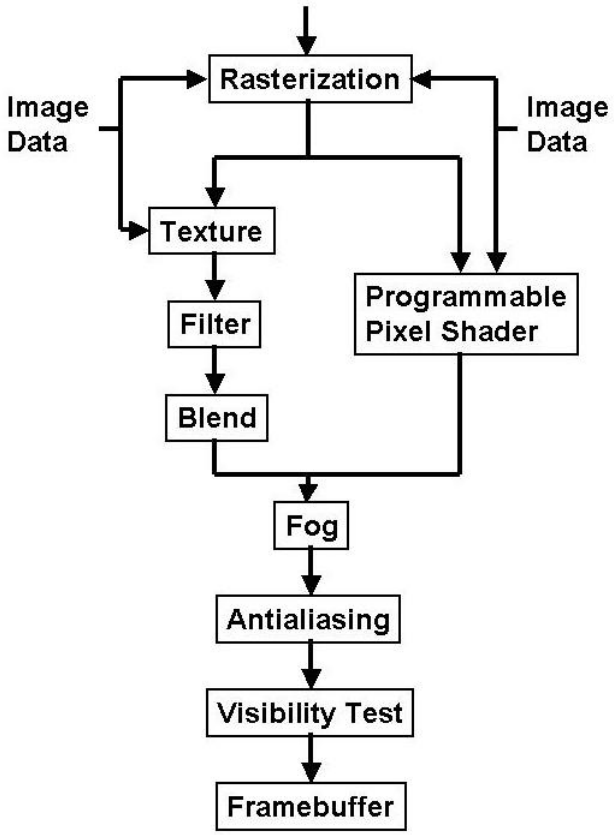


Figure 2.13: Rasterization Pipeline with Programmable Pixel Shaders.

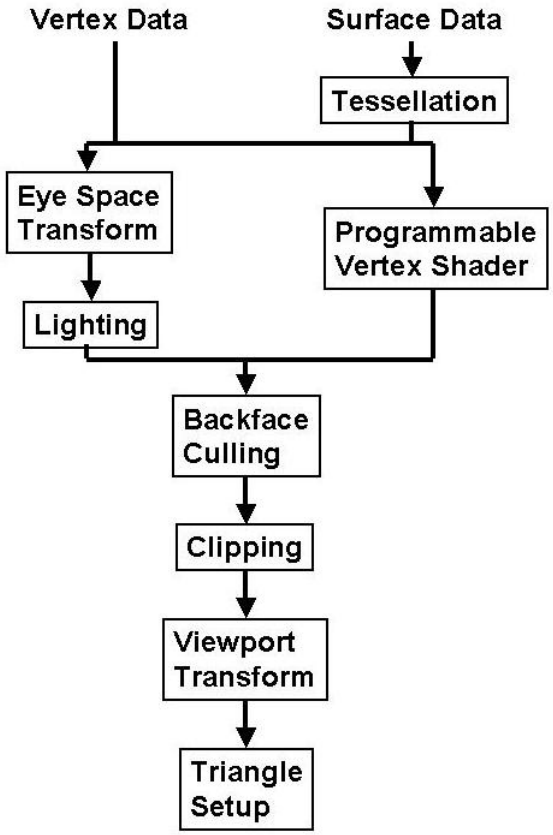


Figure 2.14: Geometry Process Pipeline with Programmable Vertex Shaders and Tesselation.

be programmed in fewer passes. Both of these user-programmable functions within the graphics rendering pipeline leverage the dedicated graphics hardware to improve the overall performance of a graphics application.

The next step in the implementation of the graphics rendering pipeline in hardware is the support for the automatic tesselation of higher-order surfaces within the GPU. This is an active area of research that has led to some initial implementations [57, 13, 14]. The goal here is to allow an application to pass higher-order surfaces to the graphics hardware. The GPU then tesselates the geometry and calculates the required vertex data prior to transform and lighting. The resulting performance advantage to an application is that this approach requires less data to be passed over the interconnect between the CPU/system memory and the graphics subsystem. The limited bandwidth of this interconnect can prove to be a bottleneck as described in Section 2.4.3. So, transferring less data in this case reduces the bandwidth requirements. The addition of this functionality to the geometry processing portion of graphics pipeline is pictured in Figure 2.14.

2.4.2 Graphics Memory

Another critical part of the graphics subsystem is the local graphics memory. This memory resides directly within the graphics subsystem. This memory is different from any system memory that may be allocated by the OS specifically for graphics usage. One example of this type of memory is AGP memory on a system that uses an AGP interconnect to the graphics subsystem. There are two aspects of graphics memory to consider in the scope of application performance: the amount of dedicated graphics memory

```
Pixel          = Color Read
                 + Z Read
                 + Texture Lookup
                 + Color Write
                 + Z Write

Bytes/Pixel = 4 + 4 + 4 + 4 + 4
Bytes/Pixel = 20 Bytes

Frame          = Horizontal Resolution
                 x Vertical Resolution
                 x Depth Complexity
                 x Bytes/Pixel

Bytes/Frame = 1280 x 1024 x 2.5 x 20
            = 65.5 Mbytes

Bytes/Sec      = 65.5 x 60
               = 3.7 GB/sec
```

Figure 2.15: Frame Buffer Bandwidth Requirements for an Application Running at 1280x1024 Resolution at 60 Frames/Second with an Average Depth Complexity of 2.5.

available and the bandwidth between the GPU and the graphics memory.

In typical graphics subsystems today, the amount of onboard graphics memory ranges from 8 MB to 256 MB. This memory is typically divided between the framebuffer, Z buffer, local texture store, and sometimes display list storage and other auxiliary offscreen buffers. Caching textures and display lists locally in the graphics subsystem eliminates the latency required to fetch them from main system memory. However, when an insufficient amount of graphics memory is available, these objects must be paged over the graphics interconnect as required by the GPU. The limited bandwidth of the graphics interconnect as described in Section 2.4.3 negatively impacts the performance of this operation and the overall performance of the application. As such, it is important that applications efficiently use the onboard graphics memory. Numerous techniques to optimize the use of graphics memory will be presented in later sections of the course.

The second issue relating to graphics memory in the context of a graphics application is the memory bandwidth between the GPU and the graphics memory. Pixels are rendered by reading from and writing to the color and z-buffers and performing lookups into texture data. For a typical graphics application, this rendering process completes two to three times for every pixel in a frame as objects occlude or hide other objects. The repeated filling of the same pixel is known as *depth complexity*. More information on depth complexity and how it can impact the performance of an application is discussed in Section 3.4.1. However, to understand the frame buffer bandwidth that is required for a typical graphics application, examine Figure 2.15, which demonstrates an application that is running at a resolution of 1280x1024 at a speed of 60 Hz with an average depth complexity of 2.5. The application requires 3.7 GB/s of bandwidth between the GPU and frame buffer memory. Bumping the refresh rate up to a more typical 72 frames/s adds another gigabyte for a total of 4.7 GB/s.

As one can conclude from this example, frame buffer bandwidth is one of the key limiters to the fill performance of a graphics application. Applications whose performance is limited by the framebuffer

bandwidth are described as fill-limited. (See Section 4.3.3) When analyzing application performance it is important to understand the available bandwidth between the GPU and the frame buffer memory as well as the subsequent requirements of the application. And then, it is important to efficiently use this available bandwidth.

2.4.3 Graphics Interconnect Limitations

Another important aspect of the graphics subsystem is the physical connection or the fabric that connects the graphics subsystem with main memory and the CPUs. Of particular relevance is the peak and sustainable bandwidth among the principal components. The physical connection can take the form of a bus or switched hub, depending on the overall architecture of the system. This connection, no matter what form it takes, has a limited bandwidth that can hinder application performance if it is not used effectively.

Typically, low-end graphics adapters sit directly on the 132-MB/s PCI bus where they must compete for bus bandwidth with other PCI devices. In this scenario, graphics data that is transferred between system memory and dedicated memory in the graphics subsystem must pass through the CPU, thereby increasing the requirements on the CPU as well as the risk of an application becoming CPU-bound.

Meanwhile, high-end graphics cards might use an AGP or other proprietary bus connection that offers exclusive bandwidth between system memory and graphics. Implemented via DMA, graphics data can be transferred from system memory to video memory in the graphics subsystem without increasing the load on the CPU. This reduces the risk that an application will become CPU-bound. Currently, AGP offers an exclusive 512-MB/s or 1024-MB/s transfer path between system memory and graphics.

Another approach is the unified memory architecture (UMA). In UMA machines, a dedicated bus handles the flow of data between the CPU and graphics. A comparison of the various architectures can be seen in Figure 2.16.

2.5 Complex Pipeline Architectures

The previous sections described the architecture of single graphics-adapter (also know as a single-head or pipe) systems. These systems are the most common types found on desktops. However, larger architectures of either multiple pipes within a system or numbers of smaller systems clustered together provide a different set of application goals and a different set of design and usage patterns.

As computing and graphics power increases and the cost of these systems drops, systems (either single-system-image or cluster-of-workstations) with multiple graphics pipelines (or pipes) are increasingly common. Systems of this size are typically used to drive large displays, such as wall or room displays, with each pipe driving a portion of that display. One common example of a system of this type is the CAVE [7]. This usage model is relatively simple to implement, because each graphics pipe contains a simple section of the overall view-frustum and the output can simply be sent to a display device. This section describes more complicated usage models in which the resultant pipe outputs are not simply sent to a display device, but recombined in another pipe for scaling the graphics loads. A more general way to use multiple graphics pipelines to render imagery is to subdivide the entire final graphics image and send it to individual pipes; then, recombine these images in the final display, regardless of how that final display device is configured.

Why would an application use a single-system-image (SSI) or cluster-of-workstations (COW) system? An application uses an SSI or a COW system to display data sets that are much more complex than those viewable on a single-pipe system. Through the use of multiple pipelines, aggregate system performance is improved to a point where the problem can be interactively visualized. While it is true that individual

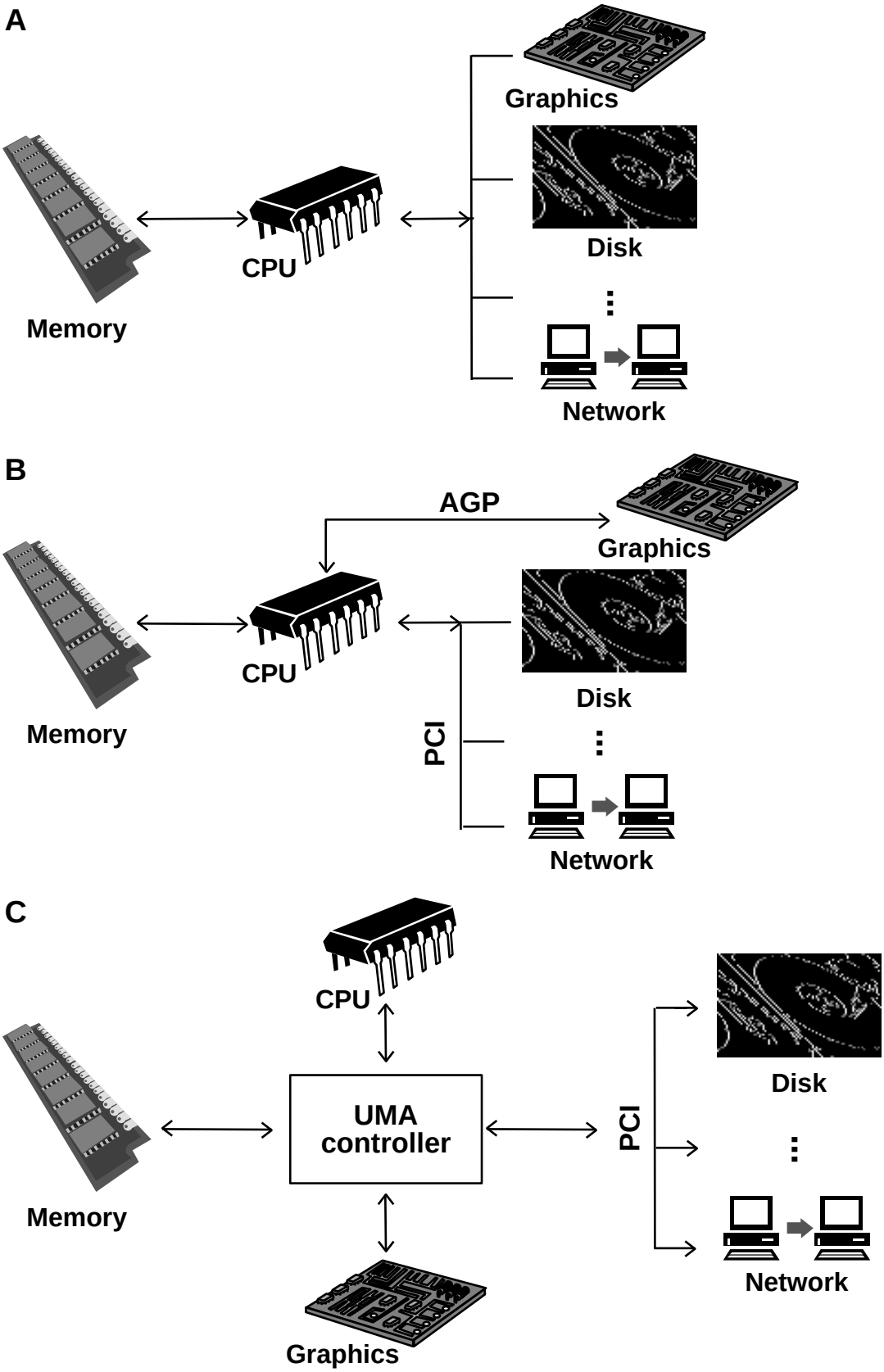


Figure 2.16: Schematic of system interconnection architectures: (A) PCI, (B) AGP, and (C) UMA.

graphics adapters that are available on the desktop today are more powerful than more costly systems that were available a few years ago, system users always want to display more. Regardless of how fast individual systems perform, a combination of multiple graphics adapters can always attack larger problems.

Both SSI and COW systems address a similar problem and raise the system interface issues (for example, PCI vs. AGP) to a different level. How do these systems differ, for what applications are systems like these appropriate, and how do applications utilize them effectively? The next few subsections address these and other issues in large-system interactive graphics application utilization.

2.5.1 Single System with Multiple Pipes

The defining features of SSI architectures are multiple graphics pipes with a single set of system resources including memory and CPU, which all communicate over a high-bandwidth, low-latency interconnect. In SSI systems, all resources are available to applications through traditional programming techniques. Displays and windows are simply opened by specifying a target graphics adapter, for example, a specific display and screen for X Window System™ applications. Figure 2.17 shows how a system might be configured. In this diagram, the system consists of four dedicated rendering pipelines (indicated by monitors) that render. The results then transfer internally across the system bus or hub and recombined on a fifth pipe. Processes are threaded (or even forked) across multiple CPUs, and functions are executed directly through standard programming language bindings. In an SSI system, data is shared either implicitly, as occurs with threaded programs, or explicitly, as occurs with forked programs that use shared-memory arenas. In both cases, explicit or implicit memory sharing, the data resides within a single logical memory subsystem, which allows easy and direct access to data across multiple processes and threads.

The key difference between systems of this type and clusters is the bandwidth and latency of interfaces among graphics pipes. In systems of this sort, sharing data from main memory to and from individual graphics pipelines is both high bandwidth and low latency.

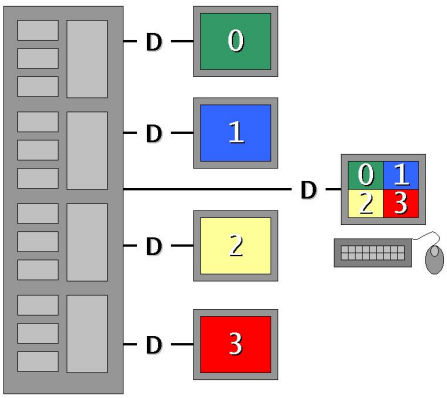


Figure 2.17: Example SSI system configuration.

2.5.2 Cluster-of-Workstation Pipes

Typically, the defining feature of COW systems is cost. COWs are often systems with much less integrated hardware and more off-the-shelf components. Though systems of this sort can potentially have

high-performance graphics, often they have lower cost and lower quality graphics. In COW systems, applications must be explicitly aware of the differences among individual systems(nodes) in the cluster, as well as the individual system capabilities and the link performance and topology of the system connections. Figure 2.18 shows an example of a configuration. Programming interfaces are explicitly parallel or happen through an abstraction layer such as a message-passing interface. Examples of these interfaces include OpenMP™ [4] and MPI [3], although many others exist. Another technique is object distribution via an object layer such as CORBA [1]. Link connection and topology are key factors in constructing and using a cluster to both determine the amount of data that can be distributed to all nodes (within the application per-frame time constraints) and the latency involved (through number-of-hops in the topology) to transfer that data.

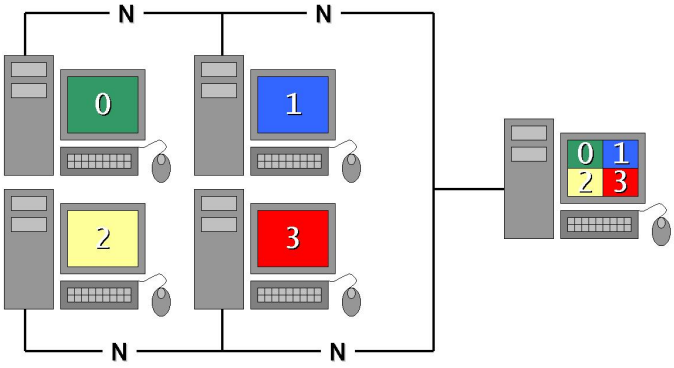


Figure 2.18: Example COW system configuration.

2.5.3 SSI/COW Usage Models

A variety of interesting usage patterns exist for both COW and SSI systems. In both systems, three factors are key to maximizing utilization and ultimately achieving good results:

- Choosing an appropriate problem decomposition
- Understanding the system architecture
- Understanding the differences among individual systems, or nodes, within a system

First you must understand and choose an appropriate problem decomposition: image-space, time-based, geometry-based, and depth-based. The application architecture might demand one particular decomposition, particularly if the application is to be modified to encompass an SSI/COW structure. In addition, each of these techniques require that you understand the possible output display configuration.

In image-space decomposition, the configuration may consist of either a single large image that is subdivided into a set of smaller subimages or a set of abutting images, perhaps not even in the same plane, such as in a CAVE. The scene geometry is divided, rendered, and then recombined(tiled) together for the final display.

A second decomposition is geometry-based. In this configuration, each pipe views the entire view-volume, and each pipe receives some portion of the geometry to render. The images from each pipe are then recombined with their depth buffers on the final pipeline. Care must be taken, however, to ensure an

adequate balance between the different pipes. Different graphical objects have different geometric requirements, and furthermore, the pipes may have different capabilities. In reality, the application must examine the geometric workload within a graphical object and allocate the workload accordingly. Additionally, the time necessary to recombine the images with their depth buffers can greatly exceed the time to recombine images in other decomposition methods. An advantage of this decomposition method, however, is that the geometry does not necessarily have to be sorted according to screen space or depth.

Another decomposition, often used in simulators, is time-based. In this decomposition, each subsequent frame is rendered on an additional pipe; then, the results are displayed sequentially on the output device (or pipe). In time-based decompositions, pipes are arranged in a ring-buffer; each pipe, once finished, begins working on the next available frame. For non-interactive graphics applications, this technique is often used to render frames of animations. In this decomposition, ensure that each pipe can accomplish its workload in the alloted time frame.

Yet another decomposition is depth-space tiling. In this decomposition, each pipe handles the same screen-space area, but each renders a specific depth-section of the database (which itself is sorted by depth.) For example, on a 3 pipe system, each pipe creates a view frustum of one-third of the total depth. This requires that the screen-space depth data of each piece of geometry be computed. This differs from the image-space decomposition described previously in which the database is divided into eye-space sections. Each pipe renders its own piece of the geometry, and the rendered sets of geometry are combined into the final image. The decomposition must balance its workload as well. In this method, the individual depth-section might have to be recalculated for each frame, depending on the time and data requirements.

If enough resources are available, combining these techniques can yield very interesting and scalable results. For example, an application might use a time-based decomposition, but for each frame within that time-buffer, you can subdivide those frames spatially. Decomposition combinations such as these are extraordinarily powerful but require a significant investment in software architecture to utilize multiple-pipe systems effectively. Appendix B contains some pseudocode that details each of these decomposition methods.

Whatever technique you chose, each image is first rendered by using a different graphics pipe and then recomposited together. Techniques for recompositing include, in the case of wall or CAVE configurations, allowing the images to simply be projected on surfaces that physically abut each other, thereby creating the illusion of a seamless image. The second, more challenging technique, involves rendering images on a number of pipes, and then capturing those pixels (and potentially depth information) and returning them to the final graphics pipe where they are recomposited and sent to the display device. Examples of all these techniques are shown in Figure 2.19.

The COW or SSI system architecture largely determines the decomposition method. In some architectures bandwidth may not be available to pass back image sections to a final pipe for recomposition. Or potentially, if bandwidth is available, the latencies involved may be too long for a copy to occur per frame. For example, in a COW latencies may be several milliseconds, but in an SSI system, latencies may be several microseconds. In clusters, at this point in time, a good strategy is to avoid these latencies whenever possible by transmitting synchronous data across the network fabric as infrequently as possible. This also implies that for COWs depth-space compositing can be difficult due to the latencies – especially in interactive applications. Specifically, a good technique in COWs is to project the resultant displays to recomposite the image. Similarly, in an SSI system where latencies are lower, it is much more feasible to transmit portions of the resultant image to a single pipe for recompositing.

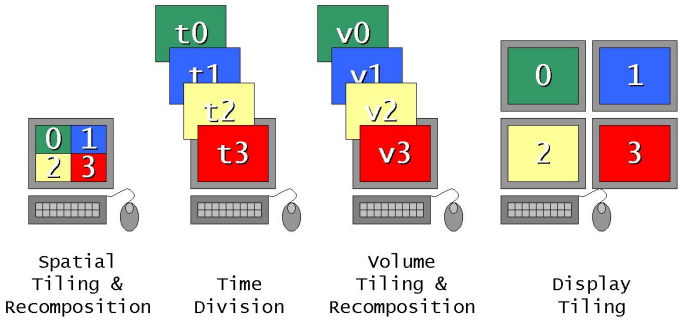


Figure 2.19: Example tiling techniques for scaling graphics performance.

The differences among individual systems(nodes) within a system further impact the decomposition method and final software architecture. When you use depth-space composition or large image reconstruction completing this final step on a system with additional resources makes obvious sense as the pixel demands on this system are larger. In a more general sense, balancing the load among systems in either a COW or an SSI system is essential to maximize the performance of the overall system. Both geometric and fill requirements should be balanced for each individual node and pipe within a system so that each pipe is kept busy, but only for as long as the time constraints on interactivity allow.

Section 3

Graphics Performance

To evaluate the performance of a software application, it is necessary to first understand the raw performance of the underlying computer system hardware. This measurement of raw system performance provides a baseline by which to compare the performance of a software application. Therefore, to analyze the graphics performance of an application, it is important to understand the raw performance of the computer graphics hardware within the system. This section examines typical graphics performance measures, the impact of the system architecture on performance, and caveats that can impact the overall graphics performance of a benchmark as well as an application.

3.1 Performance Measurements

Before jumping in and analyzing the performance of a graphics system, it is important to first understand how such performance is measured, and specified. Traditionally, graphics hardware throughput has been characterized in terms of *fill rate* and *polygon rate* while the performance of a graphics application has been measured in terms of *frame rate*. Recently however, other metrics of graphics performance have started to appear in the marketing literature of graphics hardware subsystems. These new measures include such metrics as: memory bandwidth, texel fill rate, and operations per second. The goal of this section is to define each of these performance metrics and demonstrate methods in which each of them can be objectively measured. It is important to independently verify the raw performance of a system prior to analyzing the performance of a graphics application.

3.1.1 Fill Rate

Fill rate is a measure of the speed at which primitives are converted to *fragments* and drawn into the framebuffer. Fragments not only represent the raw color data that appears in an image, but also the pixels in the framebuffer with color, alpha, depth and other data. As a result, the fill rate measures the performance of the back end of the graphics pipeline. Fill rates have historically been reported as the number of pixels drawn per second. However, this number is virtually meaningless without additional information about the types of pixels (and more correctly, types of fragments) that are involved in the measurement. When you review product literature and documentation that describes performance results, read carefully to find additional information about the tests. Specifically, look for the bit-depths of the fragments used (32-bit RGBA, 8-bit RGB, and so on), whether the fragments were textured, what type of texture interpolation was used, and other such details.

Recently, vendors of graphics cards have started reporting the fill rate in terms of texels per second rather than pixels per second. This tactic is misleading and serves to inflate the result as typical hardware has multiple parallel texture units and therefore performs multiple texture lookups per pixel. The real fill rate is then the texel fill rate divided by the number of texture units.

Another caveat to fill rate, is image fill versus texture fill. Texture fill data is typically local to the graphics memory within the graphics subsystem while image fill must typically be downloaded from system memory. Consider an image processing or compositing application that manipulates large film plates in memory prior to rendering. Rendering in this case takes the form of an OpenGL `glDrawPixels` operation after each change to an image. Because this data must be downloaded over the graphics interconnect prior to being drawn each frame, this performance will be burdened by the host download bandwidth. With texture fill rate as the metric, the texture data is kept local to the graphics subsystem and as a result does not measure the time it takes the texture or image to download from system memory.

3.1.2 Triangle Rate

Triangle rate is a measure of the speed at which triangles can be processed by the graphics pipeline. As such, the triangle rate measures the performance of the transform and lighting stages at the front end of the graphics pipeline. Triangle rates are reported as the number of triangles that can be drawn per second. Triangle rates, like fill rates, are almost meaningless without additional supporting information. Check hardware information carefully for specifics about these triangles: were they lit or unlit, were they textured or untextured, what was the pixel size, and other such details. Polygon rates are often bottlenecks in application domains such as CAD and manufacturing simulation.

3.1.3 Memory Bandwidth

Memory bandwidth, the number of bytes of data which can be moved between the frame buffer memory and the GPU, is typically measured in gigabytes per second(GB/s). This statistic started to appear on graphics product spec sheets with the introduction of programmable pixel shaders. Because pixel shaders operate on multiple textures and other per-pixel values that are stored in graphics memory – often reading and writing these values multiple times – the higher the memory bandwidth, the higher the performance of programmable pixel shaders and the general fill performance of an application.

3.1.4 Operations Per Second

With the advent of programmable GPUs, the operations-per-second specification indicates the number of vertex and pixel shader instructions that can be executed in one second. This metric has a direct correlation to the number of vertices which can be processed per second and ultimately the geometry performance of an application.

3.1.5 Frame Rate

Another measure of performance that is more typical of graphics applications specifications than raw hardware throughput is that of the frame rate. The frame rate is the number of frames rendered and displayed per second. This measurement is also often specified in Hz, where 30 frames/second is 30 Hz. For applications like simulation and animation it is important to maintain a consistent and visually acceptable frame rate to maintain temporal continuity. As such, the performance of applications like this is

typically expressed in terms of the frame rate achieved. Ultimately, good frame rate performance depends on good fill rates and triangle rates. However, as explained in Section 3.4.2 improving the fill or polygon performance of an application does not always improve the overall frame rate.

3.2 Application Impact

It is important to consider how the fill rate and triangle rate can impact the overall performance of an application. Fill rates directly correspond to the rasterization phase of the pipeline that is referred to in Figure 2.9; whereas, triangle rates directly correspond to the transformation and lighting phase of the pipeline in Figure 2.8. Achieving a good balance between these two phases is essential to good application performance. For example, if a graphics vendor claims a high triangle rate, but a low fill rate, the card is of little use in the flight simulation space because of the type of graphics data that is typically used in flight simulations. Flight simulations draw relatively few polygons, but most are large, textured, and fogged, and often overlap (think of trees that are in front of buildings that are in front of layers of ground terrain). Thus, they have high depth-complexity. In another example, if graphics hardware claims a high fill rate, but low triangle rate, it will likely be a poor CAD performer. CAD applications typically draw many small polygons without using much fill capacity (no texture, no fog, no per-pixel effects). In either application scenario, CAD or flight-simulation, some performance of the graphics hardware is often underutilized; and if it were more fully utilized, more complex or detailed scenes could be rendered. Balance is key.

Examining the details behind the reported fill rate and polygon rate numbers can yield information about whether an application will be able to perform up to these published standards. However, even armed with all this information, hardware vendors do not provide data on many variables that affect application performance. Ultimately, to measure the real performance of system graphics, you must test(benchmark) the hardware yourself.

3.3 Benchmarking

After carefully examining a graphics hardware vendor's reported performance numbers, it can be illuminating to try to duplicate those numbers. Small test programs are useful to characterize performance in a scenario that is similar to the vendor's tests, or you can use a focused test application such as SPECglperf® [2] to characterize very specific portions of the graphics hardware.

Benchmarking a system to obtain real application data numbers, however, can be very difficult. A system must be "quiet" without extraneous processes running, which could potentially modify the measured applications behavior. So, as a general rule, kill all unnecessary services/daemons before benchmarking. A second issue to be aware of when benchmarking is that of *frame-rate quantization*. Frame-rate quantization is the characteristic of a graphics system to swap buffers (draw the backbuffer to the visible screen area) only at the next vertical retrace interval. As a result, an application can block while it waits for the next vertical retrace. Frame-rate quantization is described in more detail in Section 3.4.2; but in the meantime, this implies that you should use single-buffer mode for benchmarking because single-buffer rendering does not wait for the next vertical refresh before it swaps buffers. Though single-buffered rendering introduces a visual artifact known as *tearing*, the application draws as fast as possible, which ensures accurate frame-rate measurement.

Several design parameters to keep in mind when you write test applications. First, keep data structures as small as possible and as tightly packed in memory as possible. Closely packed data is more likely to be kept in cache and therefore, more likely to accurately characterize the true performance of the graphics

hardware and avoid performance issues with the memory subsystem. Documentation can be sketchy about the default graphics hardware state (for example: is lighting enabled, is depth buffering enabled?). Be as explicit as possible about setting the graphics state to ensure that the test can be reliably duplicated on other platforms. Fully specify as much of the state as possible, paying particular attention to the state that is commonly used in your application. Finally, test a lot of data for a long time. Highly accurate timers are not available on all platforms; so, to lessen the effects of less-precise timers on the results, test data for a length of time that is much greater than a single frame. Similarly, use large enough data to ensure that the desired effect is accurately measured, not the setup/shutdown costs that are associated with each frame of drawing.

An alternative to writing your own benchmark application is to use the SPECglperf application, which is available through the SPECopc^{SM} Web site [2]. SPECglperf is designed to allow you to test most of the OpenGL pipeline within a simple script-based test framework. SPECglperf scripts test a variety of parameters in many combinations, thus, providing an automated way of gathering performance data across a set of rendering conditions. SPECglperf also has been highly tuned and optimized according to the guidelines that are outlined above to accurately measure the raw graphics performance of a system.

Upon testing graphics hardware with either a test framework, SPECglperf, or some other tool, performance still may not be as high as expected from the graphics hardware. Many hardware accelerators are only "fast" when they use very specific data formats or state settings, which are implemented in hardware on your graphics subsystem. Another name for these "fast" formats is *native formats*, which indicates that they are used internally by the graphics hardware. To find the native formats, try changing vertex data formats to vertex arrays, compiled arrays, tristrips, or quadstrips. Change pixel format data among RGBA, RGB, AGBR, BGR. Change light types from directional to local, and change lighting modes, and texture modes. Vary all of the important parameters in an application space to determine which combination yields both the highest performance and the desired quality for that application.

3.4 Performance Caveats and Pitfalls

When you measure both the raw graphics performance of the system and the overall performance of a graphics application, you must be aware of caveats that can adversely affect the performance. Fully understanding these pitfalls will help mitigate their effects.

3.4.1 Depth Complexity

Fill rate consists of more than the number of fragments drawn to the framebuffer and transferred to the screen. While the pipeline draws geometry to the framebuffer, fragments can be filled multiple times. For example, if a polygon at some far distance in the framebuffer is first drawn and then another is drawn in front of it, the second polygon is be drawn completely, overwriting some of the more distant polygons. Pixels in which the two polygons intersect are written to, or filled, twice. The phenomenon of writing the same framebuffer fragments multiple times yields a measurement know as the *depth complexity* of a scene. Depth complexity is an average measurement of how many times a single-image pixel has been filled prior to display. The overall performance of applications with a high depth complexity is often limited by the fill rate of the graphics subsystem. As a result, fill rate is often a bottleneck for application domains such as flight simulation, which have high depth complexity.

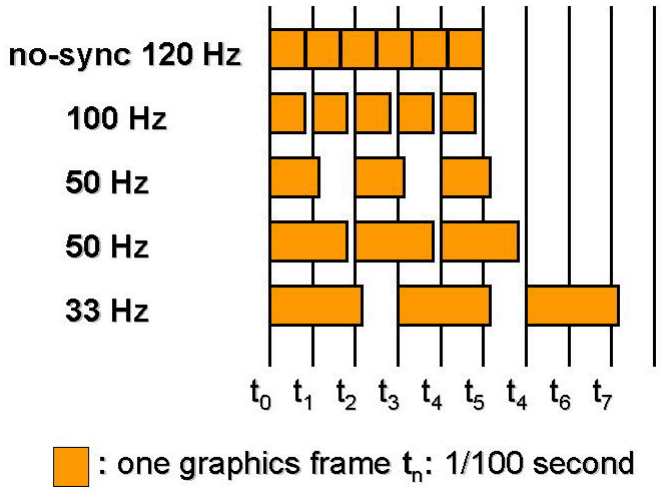


Figure 3.1: Effects of Frame-Rate Quantization.

3.4.2 Frame-Rate Quantization

A very simplified render loop for a double-buffered application entails drawing to the back buffer, issuing a buffer swap command to the graphics hardware to bring the back buffer to the front, and then drawing to the back again. Between issuing the buffer swap and the next graphics command, the graphics system must wait for the current frame to finish scanning to the output device. This render loop, combined with the display refresh rate, determines the effective frame rate of a double-buffered application. Specifically, this frame rate is an integer multiple of the output device refresh rate. Double-buffering also introduces at least one frame of latency into the application, because the scene drawn at time τ does not appear to the user until the next buffer swap. On a 72-Hz output device, this implies a potential maximum latency of 13.89 milliseconds extra per frame. This synchronization of the buffer swap operation to the vertical retract is also known as *framelock*.

Figure 3.1 demonstrates the impact that framelock can have on the performance of an application. The top line of this example shows disabled framelock such that buffer swaps do not need to wait for the next vertical retrace. In 5 one-hundredths of a second, 6 frames can be drawn for an effective frame rate of 120 Hz. Conversely, if framelock is enabled as shown on the second line, the effects of frame-rate quantization yield a performance of only 100 Hz, because the application stalls each frame until the next vertical retrace. To see how this stall can adversely affect performance, consider line 3 where the frame draw time exceeds the hundredth of a second refresh rate of the display only slightly but the frame rate is effectively halved to 50 Hz. Notice in line 4 how the frame draw time can be actually increased, but the application frame rate does not improve. In an application that falls into this category, more graphics processing can be performed to yield higher-quality results without impacting the performance of an application. And finally, notice again how the frame rate of an application is effectively halved as soon as the frame time exceeds the vertical retrace time.

Not all graphics hardware supports framelock functionality. So, how does one determine if frame rate quantization is a big problem within an application? The best way is to disable double buffering within an application and measure the performance to determine if it improves. A notable improvement in application performance indicates that frame rate quantization is playing a role in the performance. Another way is to disable framelock. Some hardware vendors provide a switch to do this. Check hardware documen-

tation for such a switch. Once it has been determined that frame rate quantization is indeed a problem, measure the time that is required to draw a frame and compare this to the time between vertical retrace operations to more fully understand the impact.

Although frame rate quantization can be viewed as a problem, it can also be viewed as an opportunity to enhance the graphics of an application without changing the frame rate. To accomplish this, increase the graphics complexity of the application such that the frame render time expands to consume the time when the application would ordinarily stall as it waits for the next vertical retrace. In this case, an application can be enhanced simply by using the available bandwidth within the system more effectively.

3.4.3 Specified vs. Measured Performance

After running benchmarks on graphics hardware, it is always interesting to compare the results with those specified by the hardware vendor. When you do this comparison you will most likely identify some differences. It is helpful to understand these differences in order to decipher your benchmark results and determine that you are getting the best possible performance from the graphics hardware. Possible differences include geometry rate and fill rate.

Geometry Rate

One reason that the geometry rate that you obtained does not match the geometry rate that the vendor specified is that the triangle sizes in your test did not match those that the hardware vendor used. To improve the fill rate, the vendor may have used very small 1 or 2 pixel triangles to calculate the published results, while the triangle in your benchmark may be a more reasonable size and closer to the triangle sizes in your application.

The geometry rate also may differ if the vendor used different lighting parameters or had lighting turned off altogether. Enabling fewer lighting parameters or disabling lighting reduces the number of required geometric calculations and ultimately improves the graphics hardware performance. (In a real application one would never totally disable lighting.)

Fill Rate

To improve the fill rate performance specification, the vendor may have not actually rendered the primitives into the frame buffer. Or, your results may differ from those of the vendor because different pixel operations were performed during rasterization. Often, a vendor will disable visability tests like the depth test inorder to eliminate the overhead of the Z buffer read.

3.4.4 Hardware Fast Paths

Section 2.4.1 described the graphics rendering pipeline and how it is typically implemented as a combination of dedicated hardware: a GPU and software running on the host CPU. Fast rendering operations directly supported in the underlying hardware are known as *hardware fast paths*. Meanwhile, primitives, states, and rendering modes for which no direct hardware support exists are rendered in a less optimal software path running on the host CPU. When this occurs within an application, the application falls off the fast path, which results in a significant decrease in rendering performance. Historically, rendering modes that cause an application to fall off the fast path include anti-aliased polygons, anti-aliased wide lines, and local light sources. As is described in Section 4.3.4, unexpected CPU activity is a sign that an application

has fallen off the hardware fast path. This unexpected CPU activity is software doing what the hardware cannot do. When benchmarking with SPECglperf or another test, another sign that an application has fallen off the fast path is a large reduction in the performance after changing a single state variable.

Although Section 2.4.1 described GPU-based graphics subsystems with hardware support geometric-processing operations, often, only a limited number of paths are fully implemented in hardware. For example, some machines may only accelerate geometric operations that involve one infinite light; others may not accelerate lights at all. Some GPUs may transform geometric data faster for triangle strips of even lengths rather than odd (due to parallelism in the geometry engines). Understanding which operations in this portion of the graphics pipeline are performed in hardware, and to what degree, is critical for building fast graphics applications.

The same is true of rasterization operations. Although any (or all) rasterization operations can be incorporated into hardware, sometimes only a limited subset actually are. Reasons for this limitation are many, including cost, complexity, chip (die) space, target market applicability, and CPU speed. Some hardware may accelerate textures only of certain formats (ABGR and not RGBA); whereas, others may not accelerate texture at all, but instead target markets such as CAD where texture is (as of yet) unimportant. It is important to know what is and is not implemented in hardware to construct a well-performing graphics application.

As a result, when you develop an application, it is important to know the hardware fast paths and design the application to stay on those paths whenever possible inorder to achieve the best performance. One technique to determine hardware fast paths is to read vendor-supplied documentation. If your vendor does not supply fast-path documentation or it is unclear, ask the vendor to supply this information. Another way to determine hardware fast paths is to use SPECglperf or a similar test suite or test program as described in Section 3.3. However, when you use a test program be careful not to introduce other bottlenecks that may invalidate the results. When targeting more than one platform, use a least-common denominator approach if possible, to stay on the intersection between the different hardware fast paths. If graphics state and modes are forcing an application off a fast path, change the code within the parameters of the application to more fully exercise the rendering features of the graphics hardware. Another method for mitigating the effects of fast and slow paths between different graphics hardware for an application is to test questionable rendering modes at startup and perform subsequent graphics operations based on the outcome of these tests.

3.4.5 Concluding Remarks

When graphics performance on specific graphics hardware has been characterized, the task then turns to realizing such performance in an application. Unfortunately, it is almost impossible to attain manufacturer-specified levels of performance in a real application. The interactions among the various components in a computer system may allow an application to perform very close to rated performance on one platform, but not on the next. But, by understanding the graphics performance of different hardware platforms, steps can be taken in the design and implementation of graphics applications to mitigate these differences. These steps are the topics of the remainder of the course.

Section 4

System Performance Analysis

The application tuning process can best be described as four nonexclusive phases as shown in Figure 4.1. The first phase, performance quantification, compares how an application performs against the ideal system performance. The second phase examines how system configuration impacts performance. The third phase performs an analysis of the graphics subsystem implementation and usage to determine when an application is CPU or graphics-bound. The fourth and final stage focuses on bottleneck elimination.

Before examining each stage of the process in detail, you should understand that the process that is described here is iterative and never really complete. When a bottleneck or application performance problem has been identified and addressed, the tuning process should restart in search of the next performance bottleneck. Code changes as well as hardware and software configuration changes can cause performance bottlenecks to shift among the different stages of the rendering process and between the CPU and graphics subsystem. As a result, performance tuning is an ongoing process.

4.1 Quantify: Characterize and Compare

To balance the demands of an application program and the computer graphics hardware, examine the application graphics requirements. Your goal is to collect basic information that will enable you to determine what the application is doing without regard for the underlying computer graphics hardware. This exercise should both help you determine the load on the system and inspire thoughts on applications changes that could improve performance.

4.1.1 Characterize Application

Application Space

Application type plays a significant role in determining the graphics demands on a system. Is the application a 3D modeling application that uses a large amount of graphics primitives with complex lighting and texture mapping, an imaging application that performs mostly 2D pixel-based operations, or a scientific visualization application that renders large amounts of geometry and texture? A good place to start is to know the application space.

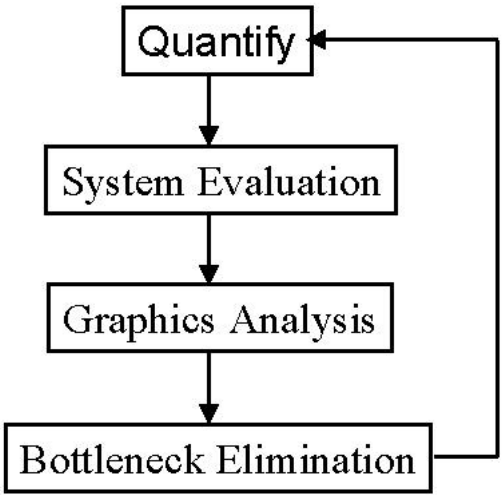


Figure 4.1: A Four Step Process.

Primitive Types

Determine the primitive types, such as triangles, quads, and pixel rectangles, that the application uses and whether a predominant primitive type exists. Identify if the primitives are generally 2D or 3D and if they are rendered individually or as strips. Primitives passed to the graphics hardware as strips use inherently less bandwidth, which is important during the analysis process. The easiest way to determine this information is to examine the source code and the graphics API calls.

Primitive Counts

Determine the average number of primitives that are rendered per frame by instrumenting the code to count the number of primitives between buffer swaps or screen updates. For primitives that are sent in lists, report the number of lists and the number of primitives per list. Add instrumentation in such a way that it can be enabled and disabled easily with an environment variable or compiler flag. Consider enabling and using run-time instrumentation to load-balance as application hardware utilization changes. Instrumentation also provides a chance to examine the graphics code to determine how the primitives are being packaged and sent to the graphics hardware. Later in this section, you will learn about tools to trace per-frame primitive information.

When gathering primitive counts and other data, it is important that you use the application and exercise code paths as a real user would use them. The work process that a user encounters day in and day out is the most useful to consider. It is also important to exercise multiple code paths when you gather performance data.

After you determine the number of primitives, calculate the amount of per-primitive data that must be transferred to the rendering pipeline. This exercise can be a revelation, stimulating thought about bandwidth- saving alternatives. For example, consider the worst case as illustrated in Figure 4.2. To render a triangle with color, normal, and texture data requires 56 bytes of data per vertex and 168 bytes per triangle. Rendering the three triangles individually requires 504 bytes of data (Figure 4.2A); rendering the triangles as a strip requires only 280 bytes of data (Figure 4.2A), which saves 224 bytes. In an

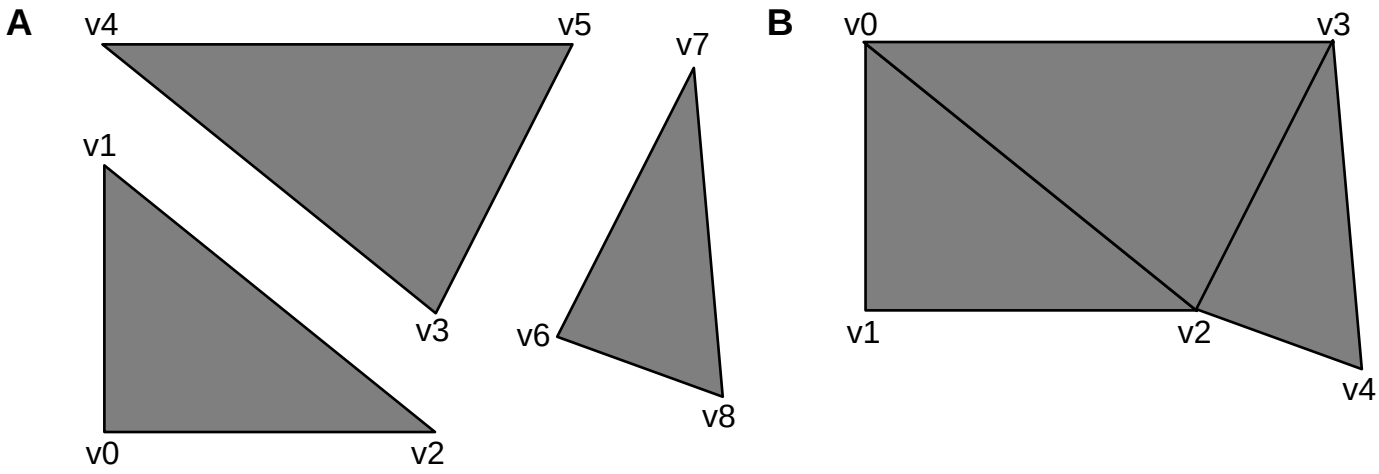


Figure 4.2: Worst Case Per-Vertex Data for Triangles. (A) Shown are three triangles, each vertex containing position (XYZW), color (RGBA), normal (XYZ), and texture (STR). Rendering a single triangle requires 56 bytes of data per vertex, resulting in a total of 168 bytes of data. The set of triangles therefore requires 504 bytes of data. (B) The same triangles from A are now combined into a triangle strip. Each vertex still requires 56 bytes of data, but because only 5 vertices are used, the total amount of data is 280 bytes, saving 224 bytes.

actual application, this savings increases dramatically. For example, rendering 5000 independent triangles requires 820 KB of data. However, combining the triangles into a single strip requires only 273 KB of data, roughly 300% less data.

Lighting Requirements

To fully quantify the graphics requirements of an application, it is critical that you consider the following lighting variables:

- Number of light sources
- Local or infinite light sources
- Lighting model
- Local or nonlocal viewpoint
- If both sides of polygons are lit

You can easily find lighting information by looking at the graphics API calls in the application source code.

All of the listed lighting variables affect the number and complexity of calculations that must be performed in the lighting equations. For example, local lights require that you calculate an attenuation factor, which makes local lights more expensive than infinite light sources. Furthermore, a local viewpoint is more costly, because the direction between the viewpoint and each vertex must be calculated. With an infinite viewer, the direction between each vertex and the viewpoint remains constant. Two-sided lighting requires that the lighting be done twice, once for the front face of a polygon and a second time for the

back face. For OpenGL, review the OpenGL Programming Guide [44] to obtain more information about how different lighting parameters can change the computation complexity and performance of the lighting model equation.

Frame Rate

Measure the frame rate to determine the number of frames that can be rendered per second. The best way to determine frames per second is to add instrumentation code to the application that counts the number of buffer swaps or screen updates per second. Be aware that swaps may occur at screen-refresh boundaries and that single-buffer mode can eliminate some potential measurement artifacts here, as described in more detail in Section 3.4.2. Some systems provide hooks and tools into the hardware, which can measure framebuffer swaps for any application.

4.1.2 Compare Results

After you collect the above data, you can compare the current performance of the application to the ideal performance on a particular system. Methods for determining this measure of ideal performance are described in Section 3.3. Use this comparison to determine if the application performance is what you expect given the capabilities of the available hardware.

Compare how the application data that was gathered earlier compares with the data that is either supplied by the manufacturer or obtained via a test program. However, keep in mind the performance caveats described in Section 3.4. Do not forget that data supplied by the manufacturer is optimal and may not be realistic. Does the application use primitives that the hardware vendor recommended and accelerated? Also, remember that the application may need time to generate the data to render for each frame. This time is not included in the optimal system graphics performance.

How does the comparison look? Are the primitive count/sec and the frame rate roughly equivalent to either the quotes from the manufacturer or information that was obtained from a benchmark test program? If so, then tuning the graphics code will not improve the user experience. In this case, the core application must be tuned to realize a performance boost. Please refer to Section 5 for more information on general application code tuning. If the rates are not equivalent, then the application graphics are performing poorly and will benefit if you tune them to reach the balancing point between the demands of the application and the capabilities of the system. Subsequent steps in this process examine how the system configuration and application software design could create an imbalance between different aspects of the system that could impact overall performance.

4.2 Examine the System Configuration

Often the first system component that is considered when you examine the graphics performance is the graphics hardware. However, it is better to first examine the other system components to determine how they are configured and how they might affect rendering performance. Eliminate other system components from the performance tuning equation before you examine the graphics hardware.

A complete examination of the system configuration involves two steps. First, examine the actual physical resources of a system to ensure that they are adequate for an application. Second, examine how the various system components are set up and configured.

4.2.1 Resources

Memory

Insufficient memory in a system can cause excessive paging. Understand the memory requirements of your application and compare them with the available memory in the system. If disk activity is high while an application is running memory page swapping may be occurring; this is a symptom of having insufficient physical memory or inefficient application memory usage. Swapping memory pages to disk negatively impacts performance. Try to keep data small and in-cache as much as possible by creating and using small, tightly packed, and efficient data structures. Large models and databases add to the overall memory footprint of an application.

Consider how system memory stores graphics data. Some systems implement a UMA where the framebuffer resides in system memory, and other systems use AGP where some textures and most graphics data are stored in system memory before a high-speed transfer to the framebuffer. These two approaches to graphics hardware can affect performance in different ways.

In a UMA system, a set amount of system memory is reserved for the framebuffer at boot time. This memory is not available to application programs and is never released. The performance advantage of this approach is that graphics data can be rendered directly into the framebuffer, which removes the cost of the additional copy from system memory to dedicated video memory that is found in more traditional hardware. One caveat of this approach is that this memory is never available to an application. As a result, if you do not boost the physical system memory accordingly when you configure the system, an application that fits on a traditional system may swap on a UMA system.

On a system built around AGP, system memory holds graphics data, but this memory is not reserved for the framebuffer and can be allocated and freed as necessary so that the application may use it. System memory provides an application with space for textures and other graphics data that otherwise would not fit in dedicated graphics memory. Copying data from system memory to video memory is implemented as a DMA over AGP. One disadvantage of AGP texturing is that memory access to nonresident textures requires a full fetch from main memory with all the attendant performance implications of main memory access.

Learn the memory access times and bus speeds of the system. Compare these with the amount of data that the application moves around when rendering. Determine if the optimal data transfer time per unit of the application time exceeds the time that which the memory and bus can provide. No matter how fast the system CPUs are, the overall performance in some application domains is limited by the bus speeds on which the CPUs reside. For example, in current Intel® memory controller-based workstations, overall performance is governed by the front-side bus between the CPU and main memory.

Disk

Consider how the disk subsystem might affect the graphics performance of an application. In addition to the type of disk, for example, IDE, SCSI, and fibre channel, consider the actual location of the disks and the application requirements. Streaming video to the screen from a slow disk is physically impossible, regardless of the speed of the graphics hardware. Store data and textures on local disks, because fetching data across a network can be a significant bottleneck. Choose disks with the lowest latencies and seek times. Once again, the disk requirements vary greatly by application, so use appropriate disk resources for the specific application.

4.2.2 Configuration

Display

Ensure that the latest driver is installed on the system before you examine the display configuration. Manufacturers constantly fix bugs and tune drivers; the latest driver typically performs best. If it is unclear whether a new driver offers the best performance, run a benchmark test to compare the performances of the new and old drivers. Use the driver that offers the best performance for the application.

Almost all combinations of operating systems and window systems provide methods for setting the configuration of the graphics display. This functionality dictates how the window system uses the graphics hardware, and consequently, how an application uses the graphics hardware. Consider how the current

active display configuration relates to the actual hardware in the graphics subsystem as described in Section 2.4. The display configuration should be set to take full advantage of the features that are implemented in hardware and necessary for the application.

When display information is queried by an application, the window system passes the display capability information back to the application. An improperly configured display impacts performance by forcing operations to be performed in software on the host CPU — operations that could have been performed by the graphics hardware — which effectively forces an application off the fast path. Therefore, it is important to confirm that display properties are set properly within the window system before you consider the display properties that are available to an application. More often than not, poor performance or some aspect of it can be attributed to a poorly configured display that does not take full advantage of hardware

features. A number of visuals can match the needs of an application. It is important to understand the performance of the selected visual as it may not be the best performing or most feature rich.

Once the display is configured properly it is the responsibility of the application to use an appropriate

configuration for the underlying graphics hardware. One way to ensure that this occurs is to have an application run simple benchmarks tests at startup that exercise frequently used functionality. Use the results of these tests to determine on an optimum display configuration. The following display parameters are important to consider.

Pixel Formats / Visuals The pixel formats/visuals that are available dictate the color depth and the availability of auxiliary buffers such as depth and stencil. Determine how the available pixel formats or

visuals compare with those that an application requires. Have a backup strategy if the application cannot get the desired pixel format. For example, if the display is configured so that no pixel formats or visuals are available with destination alpha, an application that draws alpha-blended shapes forces the graphics driver to perform alpha blending in software. A backup strategy for this scenario might be to use stippled alpha rather than blended alpha.

Color Buffer Choose a visual that matches the color precision needs of your application. For example,

a system may support visuals with 12 bits of precision available per color-component, but may not have alpha planes available in this configuration. Secondly, choose visuals that match, and only just match, the requirements for the application. Visuals with more precision per pixel induce extra fill work, and can be a potential bottleneck.

Screen Resolution The screen resolution determines the number of pixels that must be filled for a given

frame. Determine the optimal screen resolution for an application. An application may run faster at 1024×768 than at 1280×1024 because there are fewer pixels to fill. However, a lower resolution sacrifices visual quality, which may not be an acceptable trade-off.

Depth Buffer The depth buffer configuration indicates the resolution of the Z buffer. Determine how the resolution of the configured Z buffer compares to the requirements of the application. A visual or pixel format that does not support a hardware Z buffer forces depth testing to be performed in software. The actual resolution of the Z buffer is important as well. Too many bits of precision increases the fill overhead per pixel; whereas too few bits of precision creates visual artifacts known as *Z-fighting* or *flimmering*.

Auxiliary Buffers Typically, several auxiliary buffers exist for a particular visual, and it is important that you select a visual with appropriate auxiliary buffers. Additional buffers available include stencil, accumulation, and stereo. Certain combinations of auxiliary buffers may force the rendering driver off the fast path. This is especially true with auxiliary buffers that are not resident in local graphics memory.

Buffer Swap Characteristics Determine if buffer swaps are tied to the vertical retrace of the graphics display. If so, an application that can render a frame faster than the screen refresh rate (normally 60 Hz or 75 Hz) stalls to wait for a vertical retrace and buffer swap to complete. This anomaly is called *frame rate quantization* and is described in more detail in Section 3.4.2. Many hardware graphics drivers now enable users to disconnect buffer swaps from the vertical retrace which improves performance by allowing an application to render to the back buffer as quickly as possible. Keep in mind that enabling this disconnect may introduce unacceptable tearing in the display.

Network

The network can also play a role in the performance of an interactive graphics application. Use caution when you load data and textures from a remote file system during rendering; network traffic and latencies affect performance. Also, consider what else might be happening on the network to cause a system "hiccup" that would impact performance. For example, something as simple as receiving an e-mail, generating a DNS lookup, or redrawing a simple animated gif on a Web page consumes CPU cycles and system bandwidth that would have been otherwise devoted to the application. Another issue to consider is remote rendering. Is all data and rendering being performed locally, or are remote machines being used to augment the CPU processing requirements? If so, understand the capabilities of all systems in a remote-rendering scenario and the available bandwidth between them.

4.3 Graphics Analysis

An analysis of the graphics system and the graphics performance of an application requires that you understand how the performance of an application oscillates between the CPU and the graphics hardware subsystem. It is also important that you understand how the architecture of the graphics subsystem affects performance. A computer graphics application is either CPU-bound or graphics-bound at any moment during its execution. An application oscillates like a pendulum between varying degrees of these two states while rendering execution swings from CPU-based tasks to graphics-based tasks. Tuning an application attempts to improve the balance between these two extremes. As with yin and yang, the ideal state of rendering is a healthy balance of CPU usage and special-purpose dedicated graphics hardware usage. However, before you can make the appropriate lifestyle adjustments to achieve this balance, you must be able to recognize a few warning signs of imbalance.

An application passed data through a graphics library that prepares the data to pass over the interconnect fabric (see Section 2.4.3). At this point, the graphics commands enter a command buffer, often a first in, first out (FIFO) buffer. A FIFO is a mechanism designed to mitigate the effects of the differing rates of graphics data generation and graphics data processing. However, this FIFO cannot handle extreme differences between the generation and processing rates.

4.3.1 Ideal Performance

Ideal graphics application performance is defined as simultaneously using all of the CPU and all of the graphics. Said differently, ideal performance is when the CPU is running at 100% utilization executing application code while the graphics subsystem is running at 100% utilization rendering graphics data. In a perfect world, this is how an application would behave. Unfortunately, ideal performance is rarely achieved. Instead, during program execution, application performance typically swings between being CPU-bound or graphics-bound.

4.3.2 CPU-Bound

When the graphics subsystem processes data in the FIFO faster than the CPU can place new data into the FIFO, the FIFO empties, which causes the graphics hardware to stall while waiting for data to render. In this case, an application is CPU-bound because the overall performance of the application is governed by how fast the CPU can process data to be rendered. Here, the balance between the stages of the rendering pipeline done in hardware and in software is such that all available CPU cycles are consumed preparing data to be rendered while additional unused bandwidth may be available in the graphics subsystem. An application in this state can also be described as being host-limited. In this scenario, the CPU is running at 100% utilization, while the graphics subsystem is running at less than 100% utilization and may even be idle.

4.3.3 Graphics-Bound

If the graphics subsystem is processing data slower than the FIFO is being filled, the FIFO issues an interrupt which causes the CPU to wait until sufficient space is available in the FIFO so that it can continue sending data. This condition is known as a pipeline *stall*. The implications of stalling the pipeline are that the application processing stops as well and waits until the hardware again begins processing data again. An application in this state is graphics-bound such that the overall performance is determined by how fast the graphics hardware can render the data that the CPU sends. A graphics application that is not CPU-bound is graphics-bound. A graphics-bound application can be either fill-limited or geometry-limited. In this situation, the graphics subsystem is running at 100% utilization, while the CPU is running at less than 100% utilization.

Fill-Limited

A fill-limited application is limited by the speed at which pixels can be updated in the framebuffer. This is common in applications that draw large polygons. In the context of the graphics pipeline as described in Section 2.4.1, an application that is fill-limited is limited by the speed at which rasterization and subsequent pipeline stages can be executed. The rasterization capabilities of the graphics accelerator card determine

the fill limit, which is specified in megapixels/s. When an application reaches the fill limit, consider increasing the application geometry load to improve the balance between fill and geometry operations.

Geometry-Limited

An application that is geometry-limited is limited by the speed at which vertices can be lit, transformed, and clipped. Programs that contain large amounts of geometry or geometric primitives that are highly tessellated can easily become geometry-limited or transform-limited. An application that is fill-limited is limited by the speed at which the per-vertex and primitive assembly operations can be performed. The geometry limit is determined by both the CPU and the graphics hardware. The limit depends on the hardware capabilities of the graphics subsystem and where the geometric calculations are performed. In this case, consider reducing geometry calculations to improve the balance between fill and geometry operations.

4.3.4 Simple Techniques for Determining CPU-Bound or Graphics-Bound

You can use numerous techniques to determine if the performance of an application is bound by the CPU or by the graphics subsystem. Use the following techniques before you try more complicated tools.

- Shrink the graphics window. If the frame rate improves, then the application is fill-limited because the overall performance is limited by the time that is required to update the graphics window. Shrinking the graphics window shrinks the viewport which in turn shrinks the size of primitives and reduces the fill requirements. Before using this technique, ensure that the behavior of the application does not change. Some applications change their behavior and render less polygons when the graphics window is made smaller. This behavior invalidates the test; not only are the fill requirements reduced, but the geometry requirements are reduced as well.

- Reduce geometry processing requirements. Use fewer or no lights, materials properties, pixel transfers, and clip planes to render. This technique reduces the geometry processing demands on the system. If the frame rate improves and the graphics subsystem is responsible for geometry processing, then an application is graphics-bound. But, if the host performs lighting and geometric processing, then an increase in the frame rate indicates that the application is CPU-limited.

- Remove all graphics API calls. This technique establishes a theoretical upper limit on the performance of an application. The quickest way to employ this technique is to build and link with a stub library. If, after removing all the graphics calls, the performance of the application does not improve, the bottleneck is clearly not the graphics system. The bottleneck is the application code in either the generation or traversal phases. Retain this stub library in your bag of tricks for further use.

- Use a system monitoring tool to trace unexpected and excessive amounts of CPU activity. This is a sure sign that an application has become CPU-bound while software rendering. Often, a simple state change can cause this problem. It is common subsystems where not all rendering modes are implemented in hardware.

Figure 4.3 shows how to combine these techniques into a comprehensive graphics performance analysis procedure. Follow this procedure as a first step in the analysis of the graphics subsystem performance.

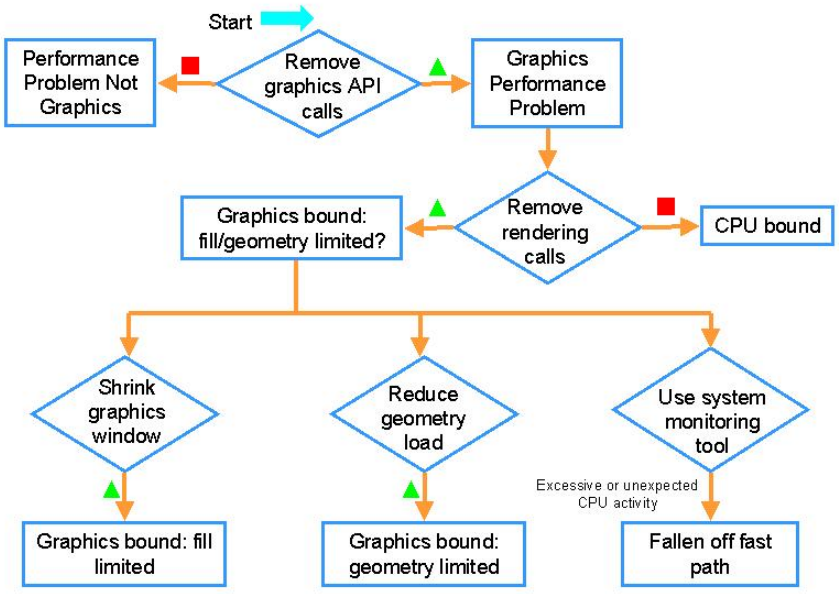


Figure 4.3: Graphics Performance Analysis Procedure.

4.3.5 Remedies

Once you determine whether an application is CPU bound or graphics bound, you can change the application to move work from one subsystem to the other. This section describes techniques to improve the performance of a CPU or graphics-limited application.

CPU-Bound

When an application is CPU bound, the goal should be to lessen the workload of the CPU. In general, there are two ways to accomplish this goal:

- Move rendering operations that are done in the host software to the GPU.
- Optimize application code so that it requires fewer CPU cycles to generate data for rendering.

Graphics Bound

When an applicaton is graphics bound, the goal should be to reduce the workload of the GPU. This goal can be accomplished via several methods, all of which fall into two general categories:

- Modify the data that is to be rendered so that it uses the hardware graphics pipeline more efficiently.
- Move operations from the GPU to the CPU.

The following section outlines numerous techniques to correct an application which is CPU bound or graphics bound.

4.4 Bottleneck Elimination

Now that you have a thorough knowledge of the system architecture, an understanding of the graphics pipeline implementation, and the ability to determine if an application is CPU-bound or graphics-bound, you can analyze the application code. This process includes identifying and removing bottlenecks.

Understanding the potential and actual bottlenecks is crucial to effectively tuning an application. Bottlenecks that limit the performance of the system will always exist within an application. The goal of tuning an application is to reach a balance among all the potential bottlenecks so that the various stages of the system and the application run as equitably as possible.

Strangely enough, a bottleneck is not always a negative situation. Sometimes, you can take advantage of a bottleneck and use the time it takes the bottleneck to clear to perform other tasks. Sometimes you can use this time to add functionality to an application. For example, in a fill-limited application, an application might add more geometry processing in the form of more sophisticated lighting and shading and/or a finer tessellation without affecting the overall performance.

Bottlenecks are not limited to the graphics subsystem and can occur in all parts of the system and arise from a number of causes. For instance, when using high-performance user-programmable GPUs which are capable of executing the complete rendering pipeline in specialized hardware, the largest problem facing the graphics applications developer is feeding the rendering pipeline. As a result, the data generation and data traversal parts of an application and the usage of memory bandwidth ultimately control the speed at which the rendering pipeline and subsequently the application can perform.

4.4.1 Graphics

Bottlenecks are most common in the graphics subsystem. Where, when, and how severe a bottleneck is depends largely on the combination of hardware and software that implements the graphics pipeline and how the application utilizes the graphics pipeline. Bottleneck elimination in this case should focus on changing the application to better utilize the graphics subsystem. In the past, tuning options were limited by the fixed-function nature of the graphics pipeline. Now, with the advent of programmable vertex and pixel shaders, the number of ways in which an application can take advantage of the performance of the hardware graphics pipeline has increased.

Non-native Graphics Formats

Pixel and texture data that is not in a format that is native to the graphics hardware must be reprocessed by the graphics driver inorder to arrange the bits into a native format before it can be rendered. An example of this process would be the conversion of ABGR data to RGBA. This increased rendering overhead could create a bottleneck within the system. A list of native data formats can typically be found in the graphics hardware documentation. To eliminate any need for bit swizzling, it is best to match image and texture formats with the framebuffer pixel format.

State Changes

The graphics subsystem is a state machine that is set up for rendering a particular primitive according to the settings of that machine. Changing state adds rendering overhead because the rendering pipeline must be revalidated after each state change before rendering can occur. Excessive state changing can cause a bottleneck in the graphics subsystem when more time is spent validating the state than rendering.

To avoid unnecessary state changes, organize data so that primitives with similar, if not identical characteristics, are rendered sequentially (without differing data in between). Although the graphics driver should be smart enough to ignore redundant state changes, it is best to avoid redundant state calls and cache important state information within the application. The following are examples of categories into which you could sort data:

- Transform
- Lighting model (one vs. two-sided, local vs. infinite)
- Texture
- Material
- Primitive (triangle strips, quads, lines)
- Color

However, use caution, and do not blindly pick a sorting methodology. Measure the relative expense of each state change and hierarchically order the sort accordingly. Also, refer to vendor documentation for hints as to the relative costs of various state changes.

Pipeline Queries

The graphics subsystem is optimized to receive graphics data and attribute information from an application and to render the resulting primitives according to the current state settings. Avoid querying the pipeline for state information because this breaks the inherent pipelining and causes the graphics subsystem to stall. Cache important state information within the application.

Inefficiently Packaged Graphics Primitives

Render similar primitives together and combine them into strips if possible to reduce the rendering overhead of setup time in the graphics subsystem. The graphics driver and hardware pipe can often pipeline the rendering of primitive strips.

Most graphics subsystems implement post-T&L vertex caches for the temporary storage of a limited number of previously transformed and lit vertices within graphics memory. Packaging vertex data into vertex arrays in OpenGL or vertex buffers in Direct3D promotes the most efficient use of this cache. When a vertex is in cache, it does not have to be resent to the GPU, retransformed, or relit, which saves both the transfer latency time and precious cycles within the GPU. Check with the hardware vendor to determine the availability and size of vertex cache as well as any requirements for their use. Some vendors require the use of glDrawElements, glDrawRangeElements, or a special OpenGL extension.

Texture Paging

Textures that do not fit in texture cache on a graphics subsystem must be transferred to the memory within the graphics subsystem prior to rendering. Traditional PCI bus-based graphics subsystems have limited local graphics memory in which to hold texture data. Such systems, therefore, are required to cache textures from system memory over the 132-MB/s shared PCI bus. In this scenario, loading and using textures that do not fit in the local texture cache can be a bottleneck. The AGP architecture solves this

problem by providing a high-speed dedicated bus to transfer the texture data from system memory to graphics memory. However, some latency remains when a texture is not resident in the texture memory on the graphics card and must be transferred from system memory. UMA systems provide support for large textures by implementing the framebuffer directly in system memory. In the case of UMA, texture download or copy of the texture data is not required for rendering.

One solution to reduce texture paging in OpenGL is to use the texture LOD extension to reduce the resolution, and subsequently the texture size, until a texture fits into texture memory. When you cannot avoid texture paging, amortize the cost of loading textures by calling `glAreTexturesResident()` to determine which textures are resident in texture memory and then render all objects that require the resident textures.

In OpenGL, use texture objects to optimize rendering and texture management. Texture objects are persistent. They can be stored in onboard texture memory, which may prevent a texture from being downloaded to the rasterizer every frame. If texture objects are not an option, consider encapsulating texture commands into a display list.

Direct3D implements a texture management system that automatically downloads textures to graphics memory as needed and uses a least-recently-used algorithm to determine which texture should be removed from texture memory to make space for a new texture. Request automatic texture management by specifying `D3DPOOL_MANAGED` when you create a texture.

Except in the case of a UMA system where graphics memory is system memory, a limit to the number of textures that can be resident in graphics memory will always exist. To help stay within the limit of the amount of texture memory that is available to keep textures resident, combine multiple small textures into one large texture as a mosaic, and change the texture coordinates accordingly to map into the larger super-texture. If possible, avoid switching between textures by rendering together primitives that use the same texture.

When changing a texture, use `glTexSubImage*()` routines in OpenGL to redefine part of an existing texture object. This step eliminates the overhead of creating a new texture object.

Lighting Model Characteristics

If you use lighting features that are unnecessary within the context of the application, such as two-sided lighting and local viewer, you will add unnecessary complexity to the lighting model and the geometry processing that is required to render a scene. Bottlenecks of this type can occur in either the graphics subsystem or the CPU depending on where the lighting model is implemented.

When a local viewer is specified, the calculation of the specular term requires the calculation of the angle between the viewpoint and each object in the scene. With an infinite viewer, this angle is not required. This produces slightly less realistic results but at a reduced computational cost. In OpenGL, non-local viewing is specified by setting `GL_LIGHT_MODEL_LOCAL_VIEWER` to `GL_FALSE` in `glLightModel`.

Specifying two-sided lighting requires that lighting model calculations be performed for both faces of each polygon. Disable two-sided lighting in OpenGL by setting `GL_LIGHT_MODEL_TWO_SIDE` to `GL_FALSE` in `glLightModel`. To use one-sided lighting effectively, all normals must be consistent with respect to the geometry. In other words, all normals must point either “out” or “in.”

Ensure that all of the lights and the state characteristics of those lights are required and add to the overall visual quality of the scene. For example, use directional or infinite light sources rather than local lights to remove the per-vertex calculation of the attenuation factor. In OpenGL, infinite or directional lights are specified with the fourth coordinate of `GL_POSITION` set to 0.0. Remove lights that do not add to the visual clarity of the scene. Each additional light requires that you evaluate the lighting model equation at

each primitive for flat shading and each vertex for Gouraud shading.

When you use Direct3D follow these general rules to optimize the performance of lighting operations:

- Use direction lights rather than point lights to remove the per-vertex calculation of the angle between each vertex and the light source.
- Use the range parameter to limit lights to only the part of the scene that they should eliminate. During execution, lighting code within the graphics driver typically exits early when a light is out of range.
- Do not use specular highlights when they are not necessary. To disable specular highlights, set the `D3DRS_SPECULARENABLE` render state to 0.

If using a lighting model that is not supported in the fixed-function hardware graphics pipeline. Consider using a programmable vertex or pixel shader to implement the custom lighting model.

Normalization

Normalization within the graphics rendering pipeline can create a bottleneck on the CPU or in the graphics hardware depending on where such calculations are performed for a given implementation. Avoid normal recalculation by the graphics rendering pipeline by ensuring that all normals are normalized within the application prior to specification to the graphics subsystem. When all normals are guaranteed to be normalized by the application, disable automatic renormalization in OpenGL by disabling `GL_NORMALIZE`.

Rasterization and Per-Fragment Operations

Using rasterization operations that your application does not require increases rendering overhead and creates a rasterization bottleneck. Rasterization operations such as texture, fog, antialiasing, and other per-fragment/per-pixel operations (blending, depth, stencil, scissor, logic operations, and dithering) could be unnecessary for an application and should be disabled when appropriate. Bottlenecks of this type can occur in either the graphics subsystem or on the host CPU, depending on the rendering pipeline split between hardware and software. Although today, these operations are most always performed by rasterization hardware of the GPU.

Examine application code to ensure that all rendering states that are enabled are required to achieve the desired images. Turn off unused features and attributes when they have no visible effect. For example, depth testing can be turned off when it is not required. In a visual simulation application, draw background objects such as the sky and ground with the depth buffer disabled; then enable the depth test for foreground objects such as mountains, trees, and buildings. In another example, if low-quality texturing is acceptible, use only bilinear filtering instead of trilinear.

If an application is currently performing multiple rendering passes in the fixed-function rasterization pipeline to implement an effect, consider using a programmable pixel shader to achieve the custom effect. Use multitexturing to apply multiple textures in a single pass.

Geometry

Processing large amounts of geometry can cause a bottleneck within the transform and lighting stages of the rendering pipeline, even on hardware-accelerated graphics subsystems. In every system, there is a point where the system becomes geometry bound, where the system cannot transform and light the amount

of geometry that is specified at satisfactory frame rates. When this state occurs, the solution is to lessen the per-frame geometric requirements. And, the best way to start is to follow the rule: if you cannot see it, do not draw it. The process by which invisible objects are removed from a scene prior to rendering is known as *culling.* Culling objects from a frame eliminates the cost of transformation and lighting of vertices and the rasterization of pixels which do not contribute to the visual fidelity of the scene. Refer to Appendix A-2 for a description of various culling techniques.

Consider using billboards to replace complex geometry as described in Appendix A-2.3. Textures also can be used to implement approximate per-pixel lighting models for hardware that does not support per-pixel lighting. These textures are commonly referred to as lightmaps. More generally, think of textures as simply one-, two- or three-dimensional lookup tables. Texture coordinates can be used to extract any specific data point within texture space and apply that point's properties to a vertex. This broadens the usefulness of textures but requires some thought to determine how to apply it within an application. Refer to [35] for further description and ideas.

After all invisible geometry has been culled from a scene, use multiple levels of detail (LODs) to render distant objects with less geometric complexity. This technique reduces the geometry processing requirements of a scene at the expense of the visual clarity of distant objects. Objects that project to a smaller screen area can be rendered with less detail and have minimal impact on the scene. More information on LOD techniques can be found in Appendix A-2.4.

Depth Complexity

Consider how many times the same pixels are filled. Avoid drawing small or occluded polygons by culling unseen or insignificant geometry. Culling is available in four types: backface, occlusion, view frustrum and contribution. For more information on these culling techniques, please refer to Appendix A-2.

4.4.2 Code and Language

Poor coding practices can be a source of application bottlenecks on the host CPU. General coding issues are addressed in Sections 5 and 6, but a few graphics-specific issues warrant discussion here.

Function Overhead

A common cause of bottlenecks is function call overhead on the transfer of data between the host and the graphics subsystem. While some systems may have a host interface that uses look-up tables for graphics API subroutines and DMA to transfer data between the CPU and graphics, most systems do not have a host interface and require a function call for each graphics API call. Function call overhead is not negligible, because the system must save the current state, push the arguments on the stack, jump to the new program location, and return and restore the original state. Using many small calls such as glVertex, instead of batching calls with aggregate functions such as glVertexArray, can cause the CPU to do excessive work and create a bottleneck on the host, which leaves the graphics subsystem underutilized.

Avoiding these types of bottlenecks is quite simple. Use primitive strips to reduce the raw amount of data sent to the pipe. Use aggregate calls such as vertex arrays and display lists in OpenGL and vertex buffers in Direct3D to reduce function-call overhead. Use vector arguments instead of individual vector elements in function calls to reduce the data copies on the stack. Another way to reduce call overhead is to eliminate function calls which set state to the same value as is already current. Do not send state information that has not changed to the graphics subsystem.

```
draw()
{
  double x1 = -0.5;
  double x2 =  0.5;
  double y1 = -0.5;
  double y2 =  0.5;

  glClear (GL_COLOR_BUFFER_BIT |
                GL_DEPTH_BUFFER_BIT);

  glBegin(GL_QUADS);
  glVertex2f(x1, y1);
  glVertex2f(x1, y2);
  glVertex2f(x2, y2);
  glVertex2f(x2, y1);
  glEnd();

  glXSwapBuffers(dpy, win);
}
```

```
33:       glVertex2f(x1, y1);
    fld         qword ptr [ebp-18h]
    fst         dword ptr [ebp-24h]
    mov         esi,esp
    push        ecx
    fstp        dword ptr [esp]
    fld         qword ptr [ebp-8]
    fst         dword ptr [ebp-28h]
    push        ecx
    fstp        dword ptr [esp]
    call        dword ptr
[__imp__glVertex2f@8 (0042b478)]
34:       glVertex2f(x1, y2);
    fld         qword ptr [ebp-20h]
    fst         dword ptr [ebp-2Ch]
    mov         esi,esp
    push        ecx
    fstp        dword ptr [esp]
    fld         qword ptr [ebp-8]
    fst         dword ptr [ebp-30h]
    push        ecx
    fstp        dword ptr [esp]
    dword ptr [__imp__glVertex2f@8
(0042b478)]
```

Figure 4.4: Call Overhead When Vertex Data Passed as Doubles.

Vertex Formats

When you use vertex arrays, consider using interleaved and precompiled vertex arrays. Interleaved arrays enable you to specify multiple arrays with a single function call. Using interleaved arrays also specifies that the data is tightly packed and can be accessed in one piece. When the data is tightly packed, the graphics subsystem can make assumptions about the layout, thereby reducing required pointer calculations during traversal. When precompiled arrays are used, data can be transferred from host memory to graphics via DMA.

Consider using a display list for static geometry that is drawn many times. However, for optimal performance within a display list, do not replace a single instantiation of an object with multiple copies. Also,

be careful not to make display lists excessively small. In this case, the overhead to traverse the display list may outweigh the time savings of immediate mode rendering. One final caveat with display lists is to understand how nested display lists may create memory fragmentation and caching problems that impact performance.

Non-native Data Format

Another source of potential graphics API overhead is passing vertex data in a non-native format. For example, if an API call is expecting vertex data as floats and the data is passed in as a double, additional CPU cycles must be used to transform the data to the required type. As an example, compare Figure 4.4 which demonstrates the assembly instructions that are required to pass vertex data as doubles to OpenGL with Figure 4.5 which passes the data in the native float data format.

Contention for a Single Shared Resource

One potential source of bottlenecks, which results more from a poor initial design rather than from a particular implementation, is contention for a single shared resource. This resource could be a graphics context, the graphics hardware, or another hardware device.

Be alert for stalls that are caused by multiple threads that are waiting to access a single graphics context, or multiple graphics contexts waiting for access to a single graphics device. Application programs that

```
draw()
{
  float x1 = -0.5;
  float x2 =  0.5;
  float y1 = -0.5;
  float y2 =  0.5;

  glClear (GL_COLOR_BUFFER_BIT |
                GL_DEPTH_BUFFER_BIT);

  glBegin(GL_QUADS);
  glVertex2f(x1, y1);
  glVertex2f(x1, y2);
  glVertex2f(x2, y2);
  glVertex2f(x2, y1);
  glEnd();

  glXSwapBuffers(dpy, win);
}
```

```
33:       glVertex2f(x1, y1);
    mov         esi,esp
    mov         eax,dword ptr [ebp-0Ch]
    push        eax
    mov         ecx,dword ptr [ebp-4]
    push        ecx
    call        dword ptr
    [__imp__glVertex2f@8 (0042b478)]
34:       glVertex2f(x1, y2);
    mov         esi,esp
    mov         edx,dword ptr [ebp-10h]
    push        edx
    mov         eax,dword ptr [ebp-4]
    push        eax
    call        dword ptr
    [__imp__glVertex2f@8 (0042b478)]
```

Figure 4.5: Call Overhead When Vertex Data Passed as Floats.

use multiple threads are becoming more and more common; however, most graphics system software is implemented such that only a single thread can draw at any moment. Even with multiple contexts, one or more per thread, access to the graphics hardware is still necessarily serialized at some level.

Mutex locks are normally used to prevent multiple threads from accessing a graphics context at the same time. However, having multiple threads drawing and waiting on a single mutex lock can cause an application bottleneck.

Bottlenecks in Non-graphics Code

A common cause of poor graphics performance in an application is one or more bottlenecks in the non-graphics code. Code that traverses the application data structures and generates the data to be rendered prior to handing it off the data to the graphics subsystem is especially suspect. Profile such code as described in Section 5 to identify and remove bottlenecks of this type.

Figure 4.6 demonstrates how API call overhead can be reduced. In this example, rendering 3 triangles as independent triangles requires 36 function calls, while using triangle strips reduces the number of calls to 20. The use of vertex arrays further reduces the number of calls to 5. The use of a single interleaved vertex array reduces the number of calls to 2, and by using a display list, the number of function calls can be reduced to 1.

4.4.3 Memory

When using high-performance user-programmable GPUs that are capable of executing the complete rendering pipeline in specialized hardware, the largest problem facing the developer of graphics applications is feeding the rendering pipeline. As a result, memory bandwidth becomes a bottleneck to obtaining ideal application performance. Inefficient storage of graphics data within memory and inefficient memory management in general can cause a bottleneck in the memory system.

Memory Allocation

Memory allocation requires a system call and an expensive kernel context switch from user mode to system mode. As a result, the allocation of memory within the rendering loop causes rendering to stall until the

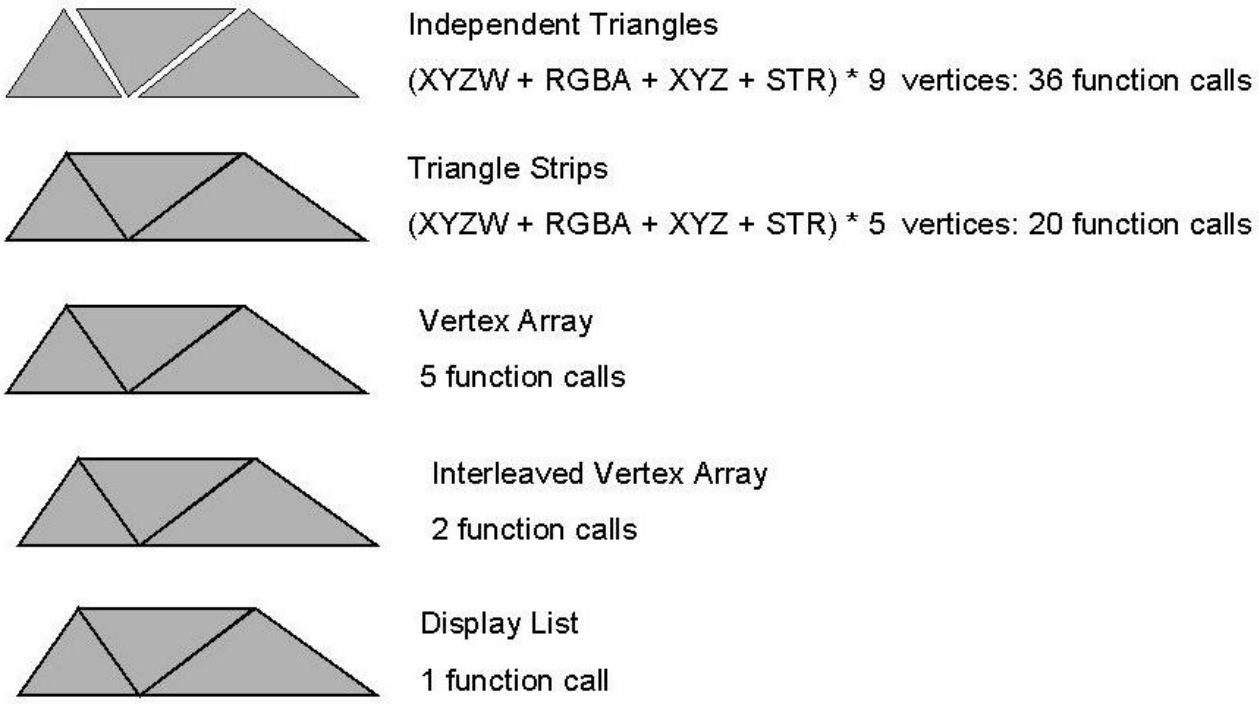


Figure 4.6: API Call Overhead.

system call returns and the user mode state is restored. To prevent a stall of this type, allocate all memory for graphics primitives before you begin the rendering loop.

Data Copies

Making local copies of data consumes CPU cycles that could otherwise be used for graphics or other data processing within the application. Avoid making local copies of per-vertex data for API calls. For example, do not copy individual X, Y, and Z coordinates into a vector to make a graphics API call when the coordinates can be sent individually.

Memory Bandwidth

Each transfer of data from memory to graphics requires overhead and system bus traffic. Amortize this overhead and maximize data bandwidth by organizing per-vertex data so that it uses vertex arrays in OpenGL or vertex buffers in Direct3D. Code that processes these data structures within the graphics driver is optimized to efficiently step through memory to obtain the per-vertex data and transfer it efficiently to the graphics hardware. Data in precompiled vertex arrays can be transferred from host memory to graphics via DMA. Display lists may also be a solution to reduce bus traffic on platforms where display list data is cached in local graphics memory. Also, check vendor documentation for extensions that permit the allocation of vertex arrays in graphics memory [6, 14].

In the case of textures, combine multiple small textures into one large texture as a mosaic; change the texture coordinates accordingly to map into the larger super texture. This action maximizes the amount of

texture data that is downloaded for the fixed overhead cost of the operation. Also when you use textures, consider using `glTexSubImage*` to redefine a subregion of an existing texture. This action optimizes

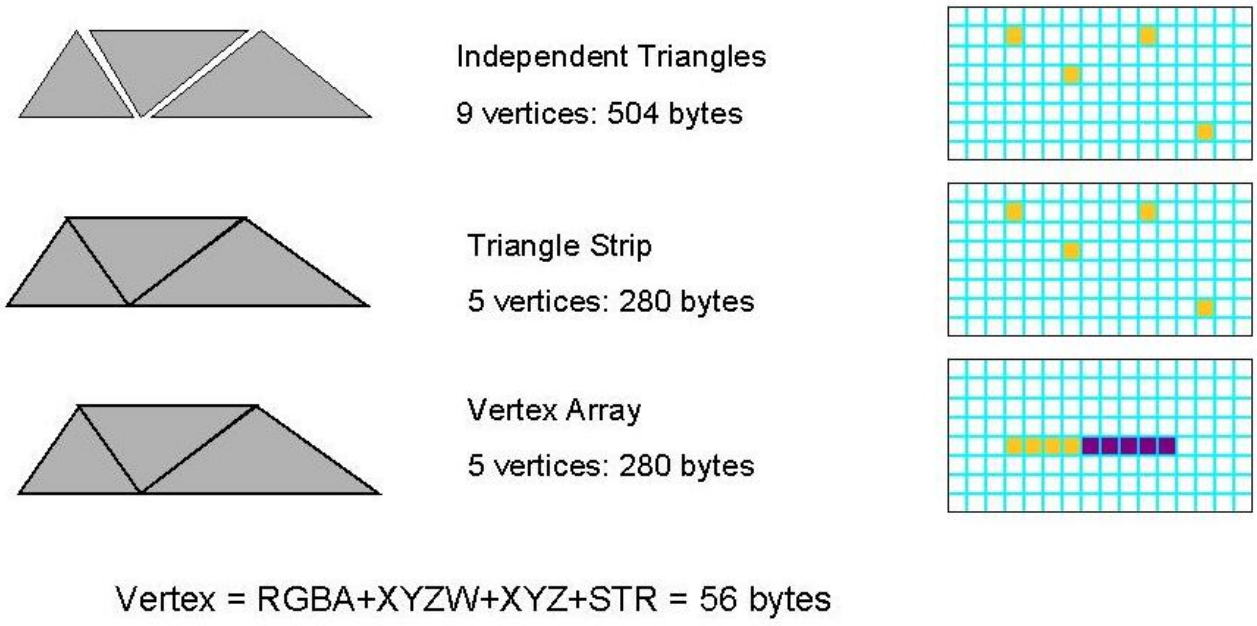


Figure 4.7: Memory Bandwidth and Fragmentation.

the use of available memory bandwidth by downloading only the portion of a texture that has changed and not the whole texture.

Another option is to pass higher-order surface geometry to the graphics rendering pipeline and let the graphics hardware perform the tessellation prior to the transform and lighting stages of geometric processing. In this case, passing surface geometry rather than individual vertex data lessens the amount of data that must be fetched from system memory and passed over the graphics interconnect.

Memory Fragmentation

Sparsely packed data causes memory fragmentation, and as a result, poor cache behavior. To avoid memory fragmentation, allocate memory for per-vertex data from a preallocated pool. This reduces expensive memory paging operations when traversing graphics data.

Figure 4.7 demonstrates how the use of triangle strips and vertex arrays can reduce memory fragmentation and maximize the use of available memory bandwidth. In this example, rendering 3 triangles as independent triangles requires 9 vertices and 504 bytes of memory; using triangle strips or a vertex array to render the same 3 triangles requires only 280 bytes of data. This modification reduces the required memory bandwidth by 45%. In the vertex array case, vertex data is contiguous in memory thereby reducing page faults and subsequent memory paging as the data is traversed.

4.4.4 CPU

Another common place to uncover system bottlenecks is the host CPU. This is especially true on systems that implement a large part of the graphics rendering pipeline in software. In this case, the most common bottleneck is geometry processing while the CPU performs all transform and lighting calculations. To remedy this situation, follow the suggestions under **Geometry** in Section 4.4.1.

4.4.5 Disk

Inefficiently storing and loading of data from disk into memory at run time can cause the file system to become a bottleneck. Ensure that texture and program data are stored locally and that the disk can handle the transfer requirements (for example, video streaming requires a disk system that can transfer the data fast enough to maintain the frame rate).

4.5 Use System Tools to Look Deeper

After you try the techniques listed above to isolate and remove bottlenecks, you might need to use system tools to probe deeper. Numerous tools exist, although different tools exist for different platforms. Unfortunately, time and space and the goal of remaining more or less platform neutral do not permit more than a brief overview here.

4.5.1 Graphics API Level

Use a graphics API tracing tool to examine the API call sequence to find excessive call overhead and unnecessary API calls. Analyze the output on a per-frame basis to establish the graphics activity per frame. Typically, the rendering loop in an application is executed per-frame, so analysis of a single frame can be applied to all frames. Analyze by examining all of the API calls between buffer swaps or screen updates. Watch for repeated calls to set graphics state and rendering modes between primitives. Tools such as OpenGL debug (see Figure 4.8), APIMON (see Figure 4.9), and ZapDB provide these capabilities.

4.5.2 Application Level

Profile the application program to determine where the most time and/or CPU cycles are spent. This helps to locate host-limiting bottlenecks in the application code. When profiling, it is important to consider not only how long a particular piece of code takes to execute, but how many times that piece of code is executed. Again, numerous tools exist depending on the target platform. Profiling is discussed in more detail with specific examples in Section 5.

4.5.3 System Level

Use a system monitoring application to examine operating system activity that is caused by the application program or perhaps an external factor. A system monitoring application will help you identify system bottlenecks. Specific things to look for include the following:

- **System/Privileged vs. User Time**

  A large percentage of time spent in system or privileged mode rather than user time can indicate excessive system call overhead.

- **Interrupt Time**

  A large percentage of time spent servicing hardware interrupts can indicate that a system is graphics-bound as the graphics hardware interrupts the CPU to prevent graphics FIFO overflow.

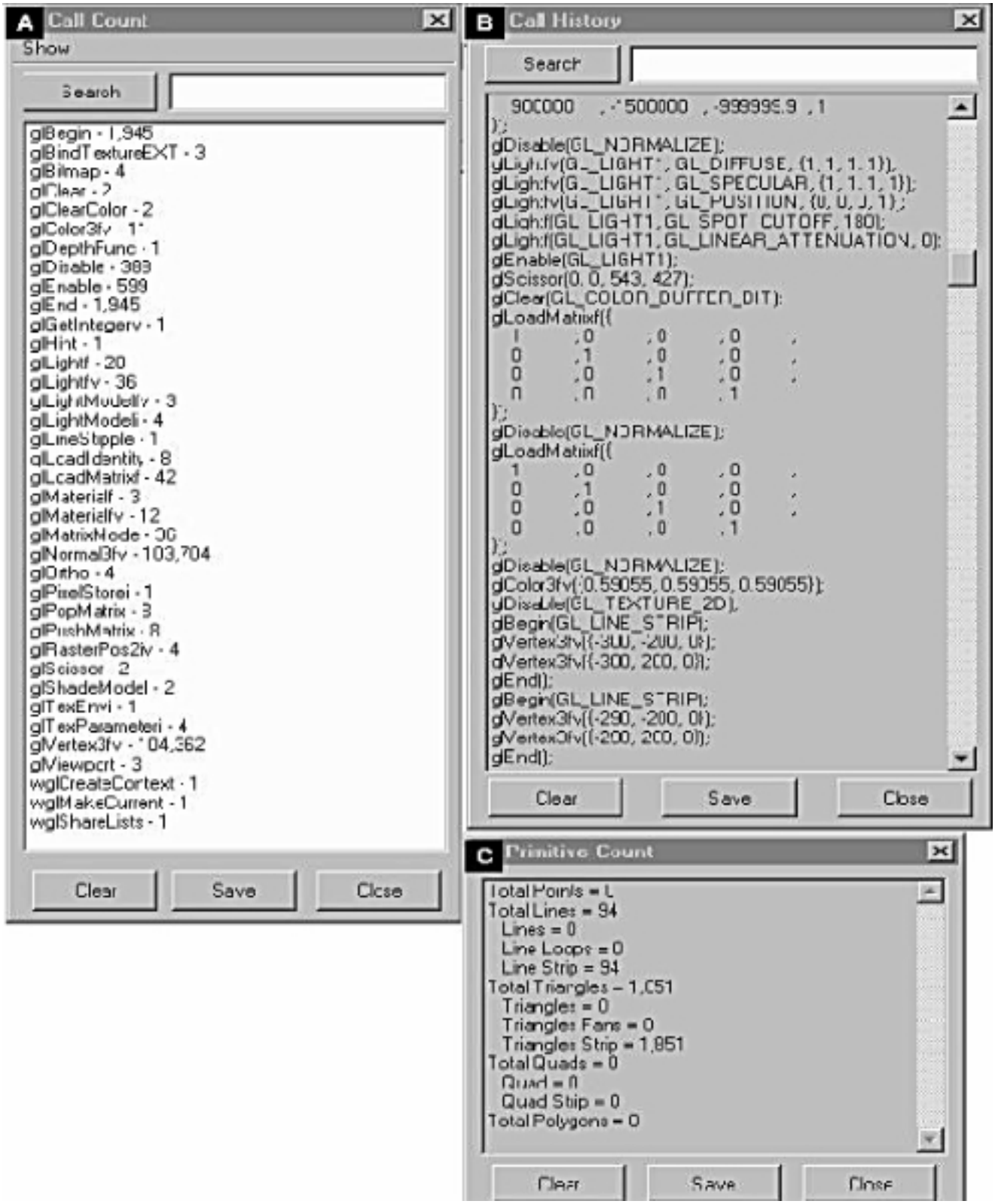


Figure 4.8: Sample Output From ogldebug, an OpenGL Tracing Tool. (A) Call count output from a running OpenGL application. (B) A call history trace from the same OpenGL application. (C) Primitive count output from the same OpenGL application.

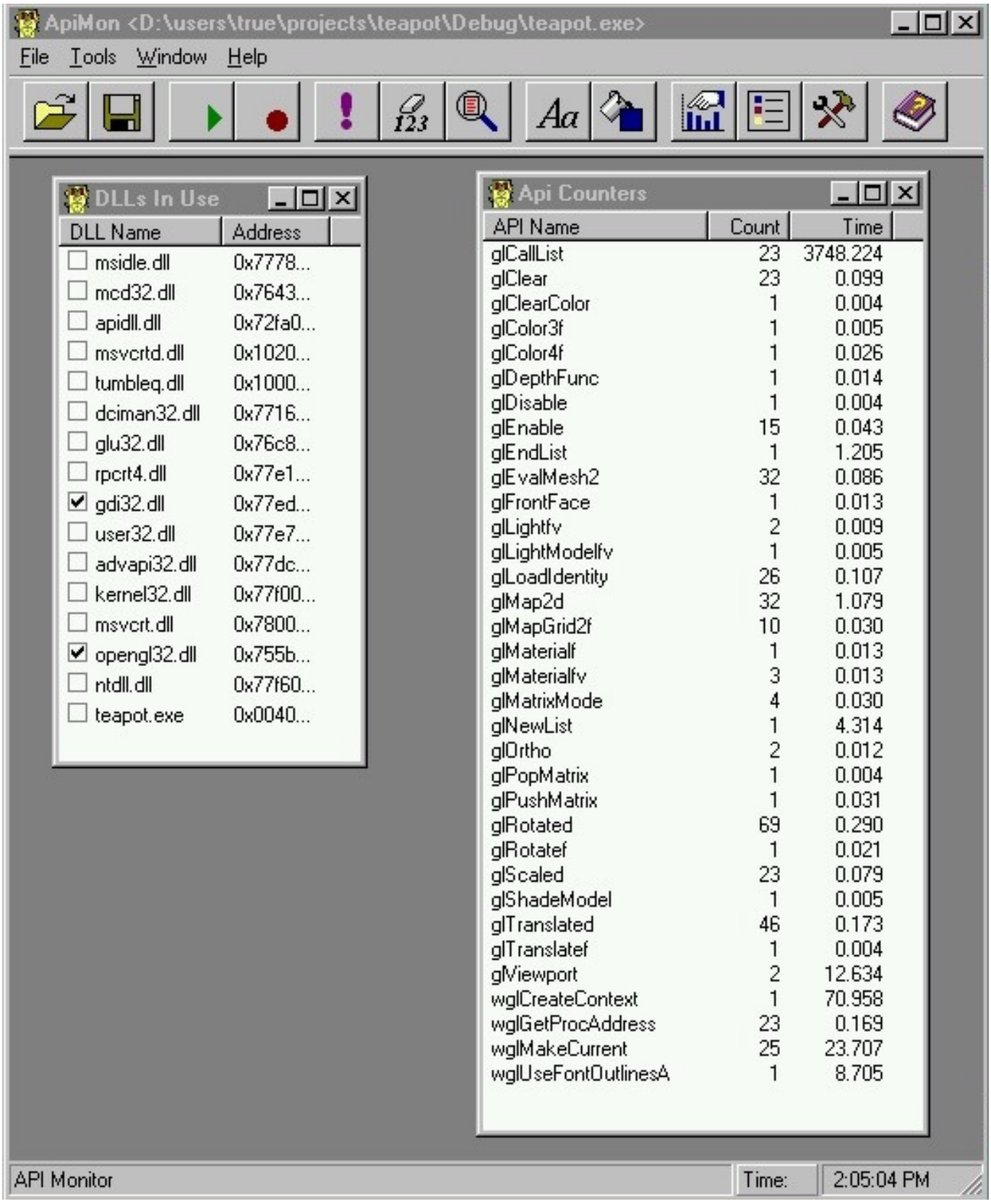


Figure 4.9: Using APIMON to Trace Graphics API Usage.

- **Page Faults**

  A large number of page faults indicates that a process is referring to a large number of virtual memory pages that are not currently in its working set in main memory and could signal a memory locality problem.

- **Disk Activity**

  A large amount of disk activity indicates that an application is exceeding the physical memory of a machine and is paging.

- **Network Activity**

  A large amount of network activity indicates that a system is being bombarded with network packets. Servicing such activity steals CPU cycles from application performance.

Because tools differ by platform, it is impossible to adequately describe them here. More detail is presented in the next section but, in general, you should familiarize yourself with the tools available on your development platform.

4.6 Conclusion

Tuning a graphics application to take advantage of the underlying hardware is an iterative process. First, basic understanding of the graphics hardware is necessary, followed by analysis of its capabilities, profiling of the application, and subsequent code changes to achieve better performance. The key concept in graphics tuning is to attain a balance among the various components involved in the rendering cycle. Balancing workload among CPU, transformation hardware, and rasterization hardware is essential to maximize the performance of a graphics application. Applying the tuning procedures and tips that are described in this section to a graphics application yields a more complete understanding of the graphics pipeline, application usage of that pipeline, and better utilization of that pipeline for faster application performance.

Section 5

Profiling and Tuning Code

By this point in the course, overall graphics application performance has been characterized and tuned and can perform at an acceptable level. The next step is to profile the code, which simply means using system tools to identify the slow parts of application software. These tools track source code lines as they are executed and measure the number of times that a line of code was executed or the number of CPU cycles that were used to execute a section of code. You can then use the results of this analysis to rewrite, or tune, code in the slowest parts of the application software.

5.1 Why Profile Software?

Even if the software is "fast enough" for current needs, it is always a good idea to know how well your code runs. Though an application runs well on a particular platform, it may not perform well on other platforms. Interactions among changing bus, memory, and CPU speeds may lead to shifted bottlenecks on different systems. For example, an application may run well on one computer configuration, but what will happen if you replace the existing CPU with a faster one that has a smaller cache? Will the program execute faster? Should you recommend that your customers replace their graphics card with the next-generation version? If your code is well-balanced, upgrading a piece of the hardware system is more likely to improve the overall application performance.

Software profiling is not difficult, but it is necessary. Although it takes time to develop the expertise to both generate and interpret profiling data, the basics are simple to master. It is well worth the effort, as profiling points out areas that need work. For example, if a graphics application is too slow, a common error might be to tune the graphics API code. However, no amount of tuning of the graphics API code will improve performance if the work that the application is doing between frames is causing the bottleneck. A quick profile run will help identify the bottleneck and direct the tuning efforts appropriately.

5.2 System and Software Interaction

Before you profile software, you should determine how it performs relative to the overall system. Does the program spend most of its time in I/O such as disk, serial, or network activity? Does the software spend an inordinate amount of time in system calls? A utility such as `time` (csh) gives you the ratio of user, system, and total time spent. If the reported system time is unexpectedly high relative to the user time,

check your system calls.[1] Not all system function calls are expensive of course, but it is important that you understand the effects of a system call before you use it. Similarly, libraries or utilities, which in turn execute system calls such as memory allocation functions, need to be handled carefully. Do not allocate memory in a time-critical graphics code. Although this is elementary, it might not be obvious that other utilities (for example, some GUI functions) may allocate memory; understand the work that the libraries in an application are doing, and use caution if these calls are in a tight loop.

Some computer systems have a FIFO queue in between the host system and the graphics system to smooth the transfer of data (see Section 4.3 for more details.) The queue can force a CPU stall if it becomes too full, which in turn stalls program execution. The state of the queue (stalled, full, or empty) during intense graphic activity can indicate if the host is flooding the graphics pipeline. Use the tools that are described in Section 4.1.1 to gather data about the FIFO usage.

Although newer chips tend to have larger cache sizes, larger caches will only temporarily mitigate the effect of poor cache usage — it is far too easy to write code that thrashes even the largest cache. Some CPUs have special hardware that monitor cache misses and points out areas of code for optimization. Data, which is then packed more densely, may reduce cache misses in these cases. Additionally, most systems have monitoring tools for paging activity. If swap activity is high, then the system needs more physical memory or it needs to better utilize the existing memory.

5.3 Software Profiling

Once the code and system interaction is understood, the code is ready to be profiled. There are two basic types of profilers: *instrumenting* and *sampling*. Instrumenting profilers count the execution of *basic blocks* and sampling profilers statistically count the number of cycles needed to execute lines of code.

A basic block is a section of code that has one entry and one exit. Basic block counting measures the best possible time (*ideal time*) that a section of code can achieve, regardless of how long an instruction might have taken to complete. Therefore, basic block profiling does not account for any memory latency issues.

Statistical sampling determines how many cycles or how much CPU time is actually spent executing a line of code and can be used to locate memory latency issues. Both instrumenting and sampling profiling reveal bottlenecks in code. However, because the two methods tend to show slightly different results, it is important that you complete both analyses.

Be careful when you profile and debug code. Code optimization by a compiler can greatly change the behavior of the software. The optimization process may change where the slow sections occur within the executable. Therefore, the profiling process must occur on optimized code (or code that is in a state identical to that which is used to ship code to customers) and not on debug code.

5.3.1 Profiling Example

It is fairly simple to profile code (Figure 5.1). Some computers require a compile-time flag to instrument the software for profiling. Other systems instrument the software after it has been compiled and linked[2]. The next step is to run the instrumented code with a relevant data set and usage scenario. Choose the data set that best represents typical customer data. When you profile, run the software when you profile

[1] UNIX® system utilities `sar` and `par` and GNU/Linux system utility `strace` report which system calls your program is calling. Corresponding utilities in the Intel VTune® profiler perform the same function for Windows® systems.

[2] On Linux, `egcs/gcov`; on IRIX®, `prof/pixie`; on Solaris™, `tcov`; on Windows NT®, Microsoft Visual Studio or `TrueTime`® from NuMega

in a manner similar to a customer's scenario. Poor choice of data and usage when profiling leads to code optimizations that are not particularly relevant. Be aware that the execution of instrumented code can take significantly longer to complete. Running the instrumented executable produces a data file with timing results that can then be interpreted as shown in the example below.

Step 1: Instrument the executable.

```
% instrument foo.exe
```

Step 2: Run the instrumented executable on carefully choosen data.

```
% instrumented.foo.exe −args
```

Step 3: Analyze the results using a profiling tool such as the Unix "prof" tool.

```
% prof foo.exe.datafile
```

Figure 5.1: The Steps Performed During Code Profiling.

As an example, a popular shareware video game was run and profiled. The results are shown in Figure 5.2. The format and content differ, of course, depending on what platform the software is profiled. But most profilers provide the time spent or the number of cycles used to execute each function. In this figure, the functions that take the most time are sorted and listed. In addition, the amount of time that each function takes is given, along with its relative percentage of execution time. The exclusive time is also listed for each function. This is the time spent by the function, not including any time spent in other functions that it calls. For a different perspective, see Figure 5.3 which shows inclusive time for the second run of this video game.

Exclusive Secs	%	Cum %	Cycles	Instructions	Calls	Function
1.076	10.5%	10.5%	209832164	241113393	15848	GL_CreateSurfaceLightmap
0.922	9.0%	19.5%	179832097	217077400	3888	S_Update_
0.868	8.5%	27.9%	169208525	187411067	420359	R_RenderBrushPoly
0.496	4.8%	32.8%	96696515	98688770	1993956	sin
0.408	4.0%	36.8%	79560696	97668367	350	GL_LoadTexture
0.347	3.4%	40.1%	67666329	76576271	20829	R_DrawAliasModel
0.343	3.3%	43.5%	66866290	62365310	1478701	glBegin
0.322	3.1%	46.6%	62838943	54407219	541646	R_CullBox
0.322	3.1%	49.8%	62785756	66456754	360125	R_RecursiveWorldNode
0.277	2.7%	52.5%	54077401	51603314	54	GL_MakeAliasModelDisplayLists
0.251	2.5%	54.9%	48995823	50654431	14768	UpdateSpaces
0.241	2.4%	57.3%	47036705	46400754	33138	EmitWaterPolys
0.210	2.0%	59.3%	40859721	70045236	5837103	glTexCoord2f
0.201	2.0%	61.3%	39164595	41458727	1923	R_DrawWorld
0.150	1.5%	62.7%	29185515	17511309	5837103	glTexCoord2f
0.144	1.4%	64.1%	2815621	27680141	64387	RecursiveLightPoint

Figure 5.2: Basic block profile example from a video game.

There are a couple of points to derive from this data. First, this profile is fairly typical. Each function takes a very small part of the overall execution time and therefore tuning this application will be difficult. A function, optimized to run at twice its original speed, will not improve performance much if it takes only 2% of the overall time. Secondly, the drop off in execution time for each function is gradual. Therefore,

there is no obvious set of functions that might be optimized. Furthermore, the first graphics API function is seventh down the list. Clearly, the game didn't spend much time drawing graphics.

This particular game runs well and it's performance is fine. However, other applications with poor performance may have similar flat profiles. They cannot be optimized easily for this reason. They are written to execute many, many functions, each only taking a very small fraction of the overall time. For a variety of reasons, software written in C++ will tend to have this problem if the authors abandon performance in the pursuit of the rich set of features that C++ offers. [3] For some helpful comments on optimizing C++ software, see Section 6.5.1.

It is probable that this set of functions, listed in this order, would not surprise the authors of this code. Some of the listed functions are used to initialize data and this is not surprising as the game player was killed off in short order. Subsequent runs, which lasted longer, show that the intialization functions are dropped down the list and are replaced by the OpenGL `glVertex3fv` function calls and the system `DMAWrite` function, as shown in Figure 5.3. [4]

In the second run, the system waited for DMA writes for about 23% of the time. The second and third functions in the list are OpenGL calls, but take much less time than the DMA writes. This data, then, seems to imply that the game would run much more efficiently with a faster graphics card. Optimizing the graphics function calls might improve performance, but only if it reduces the DMA writes. By examining the inclusive column in the data, it is evident that the function `RenderBrushPoly` spends little time executing, but a lot of time waiting for the DMA writes to occur. Perhaps this function could be rewritten to use more cycles but actually reduce the amount of DMA writes (and overall time) that occur during program execution. Finally, because the majority of the time is spent in the graphics code, the application part of the software is not a performance factor in this run.

Exlusive Secs	%	Cum %	Inclusive Secs	%	Samples	Procedure
10.710	22.6%	22.6%	10.710	22.6%	357	glWaitForDMAWrite
3.900	8.2%	30.9%	3.900	8.2%	130	glVertex3fv
3.300	7.0%	37.8%	3.300	7.0%	110	ioctl
3.000	6.3%	44.2%	3.000	6.3%	100	glTexCoord2f
2.370	5.0%	55.5%	12.030	25.4%	401	R_RenderBrushPoly
1.860	3.9%	59.4%	1.860	3.9%	62	flushRegs
1.710	3.6%	63.1%	1.710	3.6%	57	glWaitForDMARead
1.590	3.4%	66.4%	1.590	3.4%	53	glEndPolygon
1.230	2.6%	69.0%	1.230	2.6%	41	sin
1.020	2.2%	71.2%	1.020	2.2%	34	glVertex3f

Figure 5.3: PC sample profiling example from a video game, second run. This table shows inclusive sampling data.

Essentially, the task at this point is to observe the data and note any surprising placements of the functions in the execution time list. Are functions that are not intended to be there showing up? Do the graphics functions show up at all? Surprisingly, graphics functions often do not take much time — even in animation applications or so-called "graphics" applications such as CAD. These programs can spend an enormous amount of time pushing data around before drawing a single polygon; therefore, optimizing the graphics function calls can be futile.

[3] Obviously, this is a generalization, but the point still stands: always write code with an eye towards performance. It will not matter how beautiful your class structure is if it is too slow; your competition will win.

[4] It was important to exercise all portions of the code to ensure accurate results for this course. This took some time.

5.3.2 Basic Block Profiling

To illustrate basic block profiling, another example, `foo.exe`, is shown in Figure 5.4. This example has two functions of interest, `old_loop` and `new_loop`, which add and print the sum of all the values in array `x`. A third function, `setup_data`, is used only to set up the data. The function `old_loop` (Figure 5.4A) is the original function prior to profiling; `new_loop` (Figure 5.4B) is the improved function which results from application tuning.

A
```
// Code the old way
#define NUM 1024
19: void old_loop() {
20:  sum = 0;
21:  for (i = 0;i < NUM; i++)
22:    sum += x[i];
23:  printf("sum = %f\n",sum);
24: }
```

B
```
// Code the new way
27: void new_loop() {
28:  sum = 0;
29:  ii = NUM%4;
30:  for (i = 0; i < ii; i++)
31:     sum += x[i];
32:  for (i = ii; i < NUM; i += 4){
33:     sum += x[i];
34:     sum += x[i+1];
35:     sum += x[i+2];
36:     sum += x[i+3];
37:  }
38:  printf("sum = %f\n",sum);
39: }
```

Figure 5.4: Code of `foo.exe` for profiling example. (A) Original function `old_loop`. (B) Improved function `new_loop` with the loop unrolled.

What does the analysis tell us about this code segment? Figure 5.5 provides the output for the test run. The function `old_loop` took 6,168 cycles to complete. Now the fun begins — analyzing why the code is "slow" and how we can improve it. How could this be rewritten to run faster? Notice that `old_loop` (Figure 5.4A) is basically one large loop and nothing else. If you unroll the loop and call the function `new_loop`, it now looks like Figure 5.4B. (More about loop unrolling in Section 6.4.5). After re-profiling the new executable, the analysis (Figure 5.5B) shows that `new_loop` takes only 4,625 cycles, a savings of 25%.

In addition to the amount of time that each function takes, the analysis provides the lines of code that are repeated most often. The second part of the report (Figure 5.5C) provides that data. (For simplicity in this example, `old_loop` and `new_loop` are both included in the same file and both called once.) Note that lines 21 and 22 of `old_loop` were invoked 1,024 times each. (This makes sense because the code was written that way.) The loop overhead used approximately two cycles per loop invocation, and the loop body used four cycles per loop invocation. In the `new_loop` function, the loop body took 4,615 cycles ($978 + 3 * 968$) to execute — a little more than with `old_loop` (4,096). However, the loop overhead dropped from 2,061 cycles (`old_loop`) to 733 (`new_loop`) because it was executed fewer times. This is the primary source of savings from the loop-unroll optimization.

How does this savings compare on other systems? `Old_loop` and `new_loop` were combined into one file, compiled under Visual C++, and run on an Intel CPU. The results (Figure 5.6) show that `new_loop` improves on `old_loop` by about 40%.

A

	Cycles	Instructions	Calls	Function	(file, line)
[1]	6160	6168	1	old_loop	(blahdso.c, 19)
[2]	4869	8714	1	setup_data	(blahdso.c, 11)

B

	Cycles	Instructions	Calls	Function	(file, line)
[1]	4869	8714	1	setup_data	(blahdso.c, 11)
[2]	4625	4891	1	new_loop	(blahdso.c, 27)

C

Cycles	Invocations	Function	(file, line)
4096	1024	old_loop	(blahdso.c, 22)
3434	256	setup_data	(blahdso.c, 13)
2061	1024	old_loop	(blahdso.c, 21)
1435	256	setup_data	(blahdso.c, 12)
978	256	new_loop	(blahdso.c, 36)
968	256	new_loop	(blahdso.c, 35)
968	256	new_loop	(blahdso.c, 34)
968	256	new_loop	(blahdso.c, 33)
733	256	new_loop	(blahdso.c, 32)
7	1	new_loop	(blahdso.c, 29)

Figure 5.5: Results of profiling. (A) The basic profiling block of the original code. Shown is the function list in descending order by ideal time. (B) Profiling block of the modified code. Shown is the function list in descending order by ideal time. (C) Line analysis for both original and modified code. Shown is the line list in descending order by time.

Function Time (s)	Percent of Run Time	Function + Child Time	Percent of Run Time	Hit Count	Function
0.410	39.4	0.410	39.4	1	_old_loop (nt_loop.obj)
0.249	23.9	0.249	23.9	1	_new_loop (nt_loop.obj)

Figure 5.6: Profile comparison of `new_loop` and `old_loop` using Visual C++ on an Intel CPU.

5.3.3 PC Sample Profiling

Instrumenting profilers count the number of times a block of code was run, but do not record the amount of effort, or CPU cycles, that were needed to complete that block of code. Sampling profilers count the number of cycles used, which is a measurement of the amount of effort that is required to execute a line of code. These profilers, therefore, provide another useful analysis tool to determine where to tune application code.

Sampling profilers are used differently than instrumenting profilers. A sampling profiler interrupts the program at various time intervals and records the execution information, the program counter (PC), or the call stack. These profilers then provide a statistical report on the software that was executing. Because the overall system activity changes, the statistical results may vary between runs. An advantage of sampling profilers is that they do not instrument the code; therefore, the profiling run executes much faster. However, some UNIX System V platforms require that the code be relinked with different compiler flags. If an application incorporates third-party software, and relinking is not an option, sampling profilers may not be useful.

Figure 5.7 compares the PC sampling-based analysis against the basic block method and shows how these two methods differ and why both must be completed. In this example, the contents of an array are summed by using three different functions: `ijk_loop`, `kji_loop`, and `ikj_loop`. The function names denote the loop index ordering for the three-dimensional array used in Figure 5.7A.

Although the example appears simplistic, it is real code that has been extracted from volume rendering code. In this type of application, data is often viewed along the x, y, or z planes. In this application, rendering along one plane may be slower or faster than another. Why? Figure 5.7 clearly shows that the index order makes an enormous difference. Under the basic block analysis, each function takes the same number of cycles as expected (Figure 5.7B). However, under PC sampling analysis, a different behavior (Figure 5.7C) is evident. The PC sampling analysis shows that the `loop_ijk` is much more efficient than `loop_kji` because of the caching behavior of the data.

This example demonstrates the importance of using both types of profiling. PC sampling points out those areas of software that use the most CPU cycles, whereas basic block analysis points out the number of times that particular areas of software are executed. Both methods are essential for a balanced picture of application performance. If a real application has an inherent performance weakness, profiling can show you where to be especially careful when you build the data structures and code to compensate for memory latency.

5.4 Conclusion

Code profiling is critical to optimal application performance. Code profiling tools make it relatively simple to gain a basic understanding of how well different parts of the application software execute. Profiling also gives you a glimpse into the effects of instruction and data caching by comparing the basic block results to the profiling data from a statistical sampling profile.

Though profiling the application is easy, it can be difficult to find a code change that yields better performance. Initially, it may take several iterations for software changes to realize performance gains. The next section discusses some common C and C++ code changes that may increase your application's performance.

A

```
#define YNUM 100
long i,j,k;
double y[YNUM][YNUM][YNUM];
double sum = 0;

void setup_data()
{
   for (i = 0;i < YNUM;i++)
      for (j = 0;j < YNUM;j++)
         for (k = 0;k < YNUM;k++)
            y[i][j][k] = i+j+k;
}

void ijk_loop()
{
   sum = 0;
   for (i = 0;i < YNUM;i++)
      for (j = 0;j < YNUM;j++)
         for (k = 0;k <YNUM;k++)
            sum += y[i][j][k];
  printf("sum = %f\n",sum);
}
```

```
void kji_loop()
{
   sum = 0;
   for (k = 0;k < YNUM;k++)
      for (j = 0;j < YNUM;j++)
         for (i = 0;i < YNUM;i++)
            sum += y[i][j][k];
  printf("sum = %f\n",sum);
}


void ikj_loop()
{
   sum = 0;
   for (i = 0;i < YNUM;i++)
      for (k = 0;k < YNUM;k++)
         for (j = 0;j < YNUM;j++)
            sum += y[i][j][k];
  printf("sum = %f\n",sum);
}
```

B

[Index]	Cummulative Percent	Cycles	Instructions	Calls	Function	(file,line)
[1]	25.3%	51141434	37101028	1	ijk_loop	(foo.c, 47)
[2]	25.3%	51141434	37101028	1	kji_loop	(foo.c, 57)
[3]	25.3%	51141434	37101028	1	ikj_loop	(foo.c, 66)

C

[Index]	Percent	Samples	Procedure	(file, line)
[1]	38.0%	27	kji_loop	(foo.c, 57)
[2]	23.9%	17	setup_data	(foo.c, 15)
[3]	19.7%	14	ikj_loop	(foo.c, 66)
[4]	18.3%	13	ijk_loop	(foo.c, 47)

Figure 5.7: Example sampling profile showing memory latency. (A) Code for three functions that traverse a array. Each function traverses the indices in a different order. (B) Report showing basic block analysis. (C) Report showing PC sampling analysis.

Section 6

Compiler and Language Optimizations

Effective use of a compiler and linker can greatly increase the overall performance of an application. Their effective use can, however, go beyond the casual incorporation of the highest level optimization flag. This section discusses several additional optimizations that are possible with modern compilers. In addition, a programmer can use his or her knowledge of the compiler and the programming language to modify the source code specifically towards performance. Some C and C++ language examples and issues are provided.

6.1 Compilers and Optimization

Modern compilers have a large number of options that can be independently enabled or disabled to affect code performance, compiler performance, and compiler functionality. For example, performance may be increased by increasing the roundoff tolerance for calculations. Debugging features can be enabled or disabled. Numerous optimizations for loop unrolling, processor architectures, error handling, and other activities can be selectively enabled or disabled in a good compiler.

Optimizations occur within a compromise of speed, memory space, and time needed to compile and link. Therefore, there are no absolute rules about what will or will not be acceptable trade-offs within a software project. Rather, compiler optimization is usually an iterative process of discovering what is effective and what is not. Compilers may boost performance by changing the amount of arithmetic roundoff, but may be ineffective when necessary precision is lost. Compilers may gain a great deal of performance by inter-procedural analysis (IPA) and optimization, but at the expense of extended link times. (IPA is the process of rearranging code within one function based on knowledge of another function's code and structure.) In another situation, compilers may be able to optimize code if pointers are never aliased. These optimizations come, however, at the expense of compile and link time, and the possible increase of code size. Some optimizations even require multi-pass compiles on the same source code. Are they worth it? Experiment with your code and find out.

Furthermore, a developer need not use the same optimization techniques for the entire software project; certain optimizations can be used for one specific file or library, and other optimizations can be used for other files. In addition, different compilers may be used throughout the development cycle. One compiler might be used with integrated debugging tools for software development and debugging. But after code completion, another compiler with better optimization techniques may be used to produce the final shipped product binary images. One additional note is that compilers on different platforms come with different levels of quality and different types of optimizations. Study the compiler documentation carefully for

insight into how certain optimizations perform and change the way code is generated.

Discovering and working with optimizations can be well worth the effort. Consider the commonly known, although old, Dhrystones benchmark as an example. The benchmark measures how many iterations (or loops) of a specific code fragment can be executed in a given time. More loops executed means that the code performs faster. In Figure 6.1, the benchmark achieves 239,700 loops by using code that is not optimized. If the first level of optimization is used, 496,353 loops are achieved in the allotted time. Better yet, if the highest level of optimization is used and then tuned for a specific computer, 1,023,234 loops are achieved. This is nearly four times faster than the original benchmark.

Amount of optimization	**Compiler flags**	**Number of loops**
No optimization	-n32	239,700
First level	-n32 -O	496,353
Second level	-n32 -O2	512,403
Third level	-n32 -O3	484,976
Third level	-n32 -O3 -IPA[1]	1,023,234

[1]Inter-procedural analysis tuned for a specific platform

Table 6.1: Effect of optimization on the Dhrystone benchmark. All tests performed on an SGI computer.

One common complaint about compiler optimizations is that they break the application code. Generally, this happens because of a problem in the code, not in the compiler. Perhaps an inherently incorrect statement was used or one that does not adhere properly to a C or C++ standard. Or maybe the source code implicitly depends on some dubious practice. It is true, however, that the optimizations may lead to different mathematical results because of a change in arithmetic roundoff as a result of rearranged lines of code. The author has to make the final decision about each optimization by carefully weighing the advantages and disadvantages of each.

A final word on debugging code: never ship a final product with debugging enabled — it has happened! Debug code is much slower than optimized code and can be used to reverse-engineer software. This may launch a premature entry into the Open Source arena. Always ensure that executables and libraries are stripped before shipping.

6.2 32-bit and 64-bit Code

The computing industry is in the midst of a change from 32-bit to 64-bit machines which allows application writers an opportunity to port their software to the new machines. There are a variety of reasons to change, including increased memory address space, higher precision, and possible access to more machine code instructions, which potentially will lead to better performance.

None of the advantages of 64-bit applications come without potential overhead. The memory space that is required by applications increases as the data type sizes and additional alignment constraints expand. Additional performance may be elusive, and performance may actually degrade because of the additional data that is being pushed around the system.

6.3 User Memory Management

The careful placement of objects in memory can lead to efficient application operation because of the data access speed improvements that are associated with data that is accessed in the first- and second- level caches. These two caches provide data to the CPU faster than data in main memory, so keeping data cache-resident is an obvious performance improvement. This section of the course discusses what steps a developer can take to increase the likelihood that data resides in cache.

A quick survey of common data usage scenarios is the most effective means of determining what is necessary to enable data to reside in cache in those situations. One primary data structure that is used in applications is the linked list. Linked lists are used when the overall length of a set of objects is not known or when frequent reordering of those objects is necessary. This means that a set of discontinuous data structures in memory (unlike arrays that are a continuous segment of memory) is necessary. Given that, and the usage scenario of walking the list to find a particular element, how can a developer ensure that the list is as cache-resident as possible?

Many techniques exist to solve this problem, but most require that a developer manage memory explicitly. If each time a new list element is required, a new list structure is obtained via `malloc()` or `new`, the list is likely to be fragmented or spread around memory in a way such that two list elements are far apart, and unlikely to be cached. The solution to this problem is to create a routine that creates a number of list elements close together in memory, then hands them to the application when a new one is required. This allocated set of elements is known as a *pool* and is managed explicitly by a set of routines that were created expressly for that purpose. For example, in C, a suite of functions such as the following would be created:

- `void initializeList();` allocates a number of list elements and prepares them for use by the application.
- `list * createListElement();` hands an element from the set that was previously created in `initializeList()` to the application. Marks that particular list element as in-use in the pool.
- `void destroyListElement( list * );` returns the specified element to the pool of elements, and marks that new element as available for redistribution by the pool.
- `void finalizeList();` deallocates the pools and cleans up.

Similar functions can be created in C++ with class constructors and the overloading of `new` to provide the same behavior in a much more seamless fashion. A procedure like the one described above is much better than a simple `malloc`-based approach, because it increases the likelihood that list elements reside next to others in cache. It does not ensure that elements exist in cache but rather increases the *probability* that they will.

One key trade-off when doing memory management of this sort is the amount of both work and space that is allocated to doing the list management. One issue to consider is how many list elements you should preallocate. If too many are allocated, overall memory requirements for the application may be increased, yet performance improved. If too few are allocated, the store of preallocated elements will be exhausted and another segment will have to be allocated. This allocation may come at a costly and untimely performance penalty. Again, it is important to consider the balance of work in an application. Improving cache behavior definitely improves application performance if data access is an important and time-consuming task. However, it is important to pursue changes that will most affect the application being

tuned; if the application does not use linked lists, time invested in improving cache behavior of lists will not be particularly useful. Memory management techniques such as pooling are typically of most interest for data types that are used in large number and frequently allocated and deallocated. Consider memory allocation issues and usage scenarios for those data structures most commonly used by an application and then spend effort tuning those.

6.4 C Programming Optimizations

This section details some C source code considerations that may boost performance of a graphics application. These examples, while not necessarily applicable to all applications, have produced significant performance boosts in many publicly released applications. They are included as examples of issues to consider as you write C code.

6.4.1 Data Structures

Data structures are essential to any application, including graphics applications. While writing and manipulating efficient data structures does not directly affect graphics, managing data and memory effectively can lead to more efficient search and retrieval of that data. Therefore, developing, managing, and manipulating data structures efficiently is key to good graphics performance.

A
```
struct {
    str *next;
    str *prev;
    large_type foo; // lots of data

    int key; // not cached until
             // explicitly referenced
  } str;

str *ptr;

while ( ptr->key != find_this_key )
{
   ptr = ptr->next;
}
```

B
```
struct {
    str *next;
    str *prev;
    int key;   // likely to be
               // cached in already
    large_type foo; // lots of data

 } str;
```

C
```
struct {
    str *next;
    str *prev;
    int key;   // likely to be
               // cached in already
    large_type *pfoo; // pointer to foo
 } str;
```

Figure 6.1: Example of how data structure choice affects performance. (A) Typical linked list data structure with the reference locator `key` not cached with the `next` or `previous` pointers. (B) Modified version of linked list in A with `key` relocated to be cached with the `next` and `previous` pointers. (C) Third version, which uses a pointer to reference `foo`.

Consider the data structure and code shown in Figure 6.1A. This data structure is typical of a linked list with `next` and `previous` pointing to other structures in the list, and `key` used as a reference for

locating the desired data structure. In this example, all of the user data, `foo`, is cached-in when `next` or `prev` are referenced. Because `foo` is not referenced in the comparison test with `key` to locate a list element, the loading of `foo` results in potentially more cache misses and, therefore, lowered performance.

The data structure could be easily rearranged, as shown in Figure 6.1B, so that when `next` or `previous` is referenced, `key` is likely to be cached in as well. Because `next`, `previous`, and `key` probably are only several bytes each, they should all fit in most cache lines. Thus, the reference to `key` avoids a cache miss.

A further optimization can be easily made. Remember that the large `foo` data structure still exists in each of the `next` link items. When traversing the list, it is likely that the `foo` structure will be partially brought into cache. By allocating it outside the linked list and by using a pointer to reference `foo`, the linked list can be traversed in a much more cache-friendly way because multiple link data structures can reside in cache simultaneously as shown in Figure 6.1C. Naturally, the size of the cache lines changes the effectiveness of these optimizations.

6.4.2 Data Packing and Memory Alignment

Understanding how your compiler arranges data structures in memory is an important prerequisite to writing efficient code. On some platforms, compilers may attempt the optimizations that are described in this section on behalf of an application developer. However, to achieve performance in a portable fashion, it is important to consider memory issues when you develop data structures.

A computer uses one simple rule to organize data in memory: data larger or equal to a magic size must be placed on boundaries of that magic size. The magic size, referred to as the alignment size, is typically the size of the largest basic type (such as `float` or `double`). Units of data that are smaller than the alignment size can be placed on subalignment-size boundaries, and units of data that are larger than the

alignment size are placed on the next nearest alignment boundary. Armed with this rule, a developer can begin to restructure existing or new data structures in an application to maximize memory efficiency. Figure 6.2 illustrates how two structures map to physical memory, and why the word alignment of data that is equal to or larger than the word size causes padding to occur.

There are two key ramifications of keeping data structures tightly packed in memory. First, more efficient use of data structures results in a smaller “memory footprint” when the program executes. Customers like this because it allows them to work on systems with much smaller (and cheaper) physical RAM capacities. Second, your data is more likely to be cached together resulting in a higher rate of cache hits. As access to data in cache is faster than access to data in main memory, the program runs faster. Obviously, customers also like this.

6.4.3 Source Code Organization

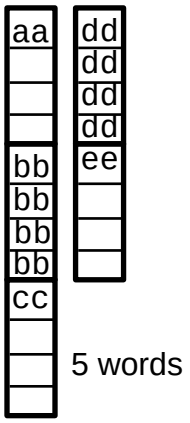
An often overlooked aspect of software development is source code organization. Which functions are put into which source files? Which object files are linked together into libraries? The performance issues surrounding code organization are not immediately obvious and are described in this section.

Source code organization is often performed by the developer according to functionality or locality. To improve performance, developers should group functions that call each other within one source file and subsequently within one library. This organization can result in reduced virtual memory paging and

reduced instruction cache misses. This improvement in efficiency can be realized because application executable code resides in the same memory page as the rest of the application data, and is therefore subject to the similar issues surrounding paging and caching (as described in Section 2.3).

A
```
typedef struct foo
{
  char  aa; // fills 8 bits,
            // pad 24 bits to get to next word
  float bb; // completely fills word
  char  cc; // fills 8 bits,
            // pad 24 bits to get to next word
  float dd; // completely fills word
  char  ee; // fills 8 bits,
            // pad 24 bits to get to next word
} foo_t;
```

B Memory space



C
```
typedef struct foo_packed
{
  char  aa; // fills 8 bits, no pad
  char  cc; // fills 8 bits, no pad
  char  ee; // fills 8 bits,
            // pad 8 bits to get to next word
  float bb; // completely fills word
  float dd; // completely fills word
} foo_packed_t;
```

D Memory space

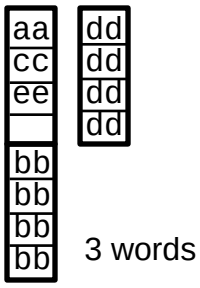


Figure 6.2: How data structure packing affects memory size on a 32-bit system. (A) A non-packed data structure `foo`. (B) Memory space used by the `foo` data structure. Each 8-bit character wastes 24 bits. Total space takes 5 words. (C) Packed version of the data structure shown in A. (D) Memory space used by the `foo_packed` data structure. The packing enables all three characters to be placed in the same word and only 8 bits of memory are wasted. Total space takes 3 words.

Tools that rearrange procedures automatically are available on some platforms. These tools can be used after the program is compiled and linked. These tools use sample data sets to create feedback files, which are then used to rearrange the procedures in an executable. However, the data sets that generate these feedback files need to be chosen carefully because they influence the overall effectiveness and relevance of these tools. Just like when profiling applications, choosing representative data is the most important factor. If sample data is chosen poorly, the rearrangement of procedures in the executable might be slower for a more common usage scenario. Contact specific hardware vendors for more information about their tools.

6.4.4 Software Pipelining

Many modern computers use superscalar CPUs. These include the Pentium®, Itanium™, PA-8000, and the R10000® series chips. These processors are capable of handling more than one instruction at a time. A compiler may use this feature to restructure statements within the body of a loop so that one iteration of the

loop can start before the prior iteration finishes. This technique is called *software pipelining*. Although the compiler cannot be directly instructed to pipeline a loop, some source code changes may enable software pipelining.

The compiler will not bother to software pipeline a loop if the number of iterations are low or if the work accomplished within the loop is too small. Pipelining may be enabled by unrolling the loop instead. Typically, inner loops are software pipelined, so concentrate efforts there. Ensure that no function calls or conditionals exist within the loop. If possible, rewrite the loop so that no flow dependencies exist between iterations. (A flow dependency occurs when one loop iteration depends on the computational results of the prior iteration.) Finally, remove any potential for pointer aliasing (Section 6.4.9) within a loop as that may prevent software pipelining.

6.4.5 Unrolling Loop Structures

Another common optimization technique is known as *loop unrolling*. Consider the function `old_loop` shown in Figure 6.3A, which demonstrates a conventional loop that consists of the loop overhead (line 21) and the loop body (line 22). (This example is profiled in Figure 5.4.) Execution speed can be improved if the loop setup overhead can be better amortized by completing more work in the loop body. Remember that `i` is incremented and compared to `NUM` for every loop iteration. The resulting modified loop, `new_loop`, is shown in Figure 6.3B; it consists of four statements in the loop body (lines 33-36). This function completes four times the amount of original work for the same amount of loop overhead. In addition, this method may expose the loop's parallelism to a processor that can take advantage of software pipelining. The performance gain from this technique can be substantial and may far exceed the performance gained by simply reducing the loop overhead.

Of course, because `NUM` is unlikely to always be a multiple of 4, the software first needs to find the remainder of `NUM` divided by 4 and sum those array entries as well. This extra loop, known as a *preconditioning loop* is shown in lines 30 and 31. However, for some applications the loop size, `NUM` in this case, is known and finding the remainder is not necessary. For example, the code might be running through an array that is known to be dimensioned by 1,024. Other applications may not be so fortunate.

A
```
// Code the old way
#define NUM 1024
19: void old_loop() {
20:  sum = 0;
21:  for (i = 0;i < NUM; i++)
22:    sum += x[i];
23:  printf("sum = %f\n",sum);
24: }
```

B
```
// Code the new way
27: void new_loop() {
28:  sum = 0;
29:  ii = NUM%4;
30:  for (i = 0; i < ii; i++)
31:     sum += x[i];
32:  for (i = ii; i < NUM; i += 4){
33:     sum += x[i];
34:     sum += x[i+1];
35:     sum += x[i+2];
36:     sum += x[i+3];
37:  }
38:  printf("sum = %f\n",sum);
39: }
```

Figure 6.3: Example of loop unrolling. (A) Original function `old_loop`. (B) Improved function `new_loop` with the loop unrolled.

In the example shown in Figure 5.5, the amount of work that the code segment completed took 6,168 cycles. By reducing the loop overhead relative to the amount of work accomplished, the improved code took 4,891 cycles, which results in a savings of approximately 25%. Of course, the size of `NUM` and a choice of value other than 4 affects the total savings achieved.

Which loops are good candidates for loop unrolling? "Fat" loops, those that complete a lot of work relative to the overhead, are poor candidates. If the loop iteration, or trip count, is small, the amount of savings is likely to be negligible. Loops that contain function calls also should be ignored as they are likely to be expensive. In addition, loops that contain branches probably will not yield much additional performance when unrolled. If possible, consider removing branches or inlining functions. The compiler may then be able to automatically unroll the loop.

Note, however, that there are drawbacks to loop unrolling. First, it adds visual clutter and complexity to the code because the loop operations are duplicated. Second, because code is duplicated, loop unrolling can increase the code size. Third, the compiler may already optimize by loop unrolling and it may do a better job than manual attempts. Furthermore, manual unrolling can actually prevent a number of optimizations.

6.4.6 Memory Reference Optimizations

Large data arrays may cause poor cache behavior when a loop strides through the data. For example, in image processing where array sizes are often large, it is frequently more efficient to break up the array into smaller subarrays. The size of these subarrays can be designed to reside within either the first or second level cache. This technique is often called *cache blocking*.

A second example is a loop that walks down columns in an array. If each row is aligned so that elements along the row-axis are cached in with each access, then walking through each column of data involves caching a new row of data with each loop iteration. However, if the array is accessed across rows instead of down columns, the data is in cache and is accessed much more quickly. Section 2.1 points out that data access to array elements in cache is far faster than those from main memory.

A

```
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      A[i,j] = A[j,i] + B[i,j]
```

B

```
for (i = 0; i < N; i += 2)
   for (j = 0; j < N; j += 2) {
      A[j  ,i  ] = A[j  ,i  ] + B[i  ,j  ];
      A[j+1,i  ] = A[j+1,i  ] + B[i  ,j+1];
      A[j  ,i+1] = A[j  ,i+1] + B[i+1,j  ];
      A[j+1,i+1] = A[j+1,i+1] + B[i+1,j+1];
   }
```

Figure 6.4: Example of optimization using cache blocking within a vector sum. (A) Original code. (B) Optimized version using cache blocking.

Dowd [16] combines these two concepts in an excellent example. Figure 6.4A shows a vector sum computation in which one array is referenced with a unit stride and the other with a stride of `N`. A first intuitive optimization might be to reorder the indices, but the algorithm still strides through either array `A` or `B` by `N`. Dowd provides a cache blocking algorithm in Figure 6.4B, which references a few elements of A, then B, in neighborhoods. This method unrolls the outer and inner loop to reuse the cache entries as much as possible to improve the caching behavior.[1] This performance optimization easily can lead to 100% performance improvement.

[1]This book is highly recommended. Although its emphasis is on high-performance computing algorithms and the examples are written in FORTRAN, the book provides an excellent overview of code optimization.

6.4.7 Inlining and Macros

Many functions can be written in several lines of code. Much like loops, the overhead in accessing a function must be offset by the work that is done by that function. For small functions, the overhead of calling that function may be more expensive than actually performing the commands in place. A good compiler optimizes these inefficiencies away through the use of *inlining*, the technique of replacing the call to a function with an in-place copy of the functions contents. Macros can take the place of inlining if the function is too large to be optimized in this way. Also consider using the keyword `inline` wherever possible. When you use inlining, be sure to watch the overall code size because heavy use of inlining and macro-expansion can increase the size of the code dramatically and impair performance due to instruction cache misses.

6.4.8 Temporary Variables

Another common optimization technique is local temporary variables. You can use temporary variables in place of references to global pointers within a function or to avoid repeatedly dereferencing a pointer structure, as shown in Figure 6.5. As with other compiler optimizations, some compilers may have the ability to perform this optimization and others may not. In the interest of better performing cross-platform code, modify the source to avoid this performance pitfall.

A
```
x = global_ptr->record_str->a;
y = global_ptr->record_str->b;
```

B
```
tmp = global_ptr->record_str;
x = tmp->a;
y = tmp->b;
```

Figure 6.5: Example of optimization using temporary variables. (A) Original code. (B) Optimized version.

Figure 6.6 demonstrates how within a function, a temporary variable, `tmp`, can replace several references to a global pointer, `newPnt`. In Figure 6.6A, a matrix multiply function is used to transform a point (`oldPnt`). Repeated dereferences of the global variable `newPnt` occur within the loop. Removal of this unnecessary step, shown in Figure 6.6B, results in better cache behavior, increased performance, and an up to 50% faster loop with some compilers.

6.4.9 Pointer Aliasing

In C and C++, pointers reference and perform various data operations on sections of memory. If two pointers point to potentially overlapping regions of memory, those pointers are said to be *aliases* [12]. To be safe, the compiler must assume that two pointers with the potential to overlap may be aliased, and this may severely restrict its ability to optimize those pointers by reordering or parallelizing the code. However, if the compiler knows that the two pointers never overlap, significant optimization can be accomplished.

Consider the code example from Cook [12] (Figure 6.7A). This code is excerpted from an audio application, but the problems of aliasing are common to graphics applications as well. In this example, `p1` may point to memory that overlaps memory that is referenced by `p2`. Therefore, any store through `p1` can potentially affect memory pointed to by `p2`. This problem prevents the compiler from taking advantage of instruction pipelining or parallelism that is inherent in the CPU. Loop unrolling may help solve the problem, but in this case a simpler solution exists.

Optimally, the compiler would recognize aliasing and optimize accordingly. This is unrealistic in any large software project. Furthermore, there is no way to indicate which pointers are aliased and which are

A
```
void tr_point1(float *oldPnt,
               float *m,
               float *newPnt)

  float *c1, *c2, *c3, *c4,
        *op, *np;

  c1 = m; c2 = m + 4;
  c3 = m + 8; c4 = m + 12;

  for (j=0, np=newPnt; j<4; ++j)
  {
     op = oldPnt;
     *np    = *op++ * *c1++;
     *np   += *op++ * *c2++;
     *np   += *op++ * *c3++;
     *np++ += *op++ * *c4++;
  }
```

B
```
void tr_point2(float *oldPnt,
               float *m,
               float *newPnt)

  float *c1, *c2, *c3, *c4,
        *op, *np, tmp;

  c1 = m; c2 = m + 4;
  c3 = m + 8; c4 = m + 12;

  for (j=0, np=newPnt; j<4; ++j)
  {
     op = oldPnt;
     tmp    = *op++ * *c1++;
     tmp   += *op++ * *c2++;
     tmp   += *op++ * *c3++;
     *np++  = tmp + (*op * *c4++);
  }
```

Figure 6.6: Example of optimization using temporary variables with a function. (A) Original code. (B) Optimized version.

not. However, the C99 standard uses the keyword `restrict` for the C language to solve this problem. The `restrict` keyword is used to indicate which pointers are aliased and which are not. Using `restrict`, the code in Figure 6.7A would be rewritten as shown in Figure 6.7B. Cook [12] states that a 300% performance improvement occurred by using this technique compared to the original code. In addition, adding this keyword to the code and recompiling is a much simpler and faster change than unrolling the loop.

A
```
void add_gain(float *p1,
              float* p2,
              float gain)
{
  int i;
  for (i = 0; i < NUM; i++)
  p1[i] = p2[i] * gain;
}
```

B
```
void add_gain(float * restrict p1,
              float * restrict p2,
              float gain)
{
  int i;
  for (i=0; i< NUM; i++)
  p1[i] = p2[i] * gain;
}
```

Figure 6.7: An example of pointer aliasing. (A) Function with pointer aliasing. (B) Revised function using the `restrict` keyword to optimize pointer aliasing.

6.5 C++ Programming Optimizations

This section describes a few performance issues to be aware of when you design and code in C++. C++ provides many efficiencies in design, architecture, and reuse aspects of software development, but it also has associated performance implications that you need to consider when you implement your designs. Ensure that your software abstraction does not impair the application performance.

6.5.1 General C++ Issues

There are only a few major issues to consider when you write C++ software, but many little issues exist that can add up to slow performance. These smaller issues are of two general sorts and can be summarized rather simply. First, be aware of what the compiler does with expressions of various types; and second, avoid expensive operations either explicitly in code or through compiler flags. A few specific issues follow.

When objects are constructed in C++, the specific instance that is created has its constructor invoked. This constructor can be written to do much work, but even in some simple cases, such as where only initial values are set, there is the overhead of a function call for each object constructed. Because of the invocation of the constructor on each instance of an object, certain situations such as static array creation can be very expensive. In other cases, if objects are passed by value across functions, the compiler instructs that a complete copy of the object be created, which invokes a copy constructor. This is potentially very expensive. When passing arguments to functions, pass arguments by reference instead of by value.

There are many other minor C++ issues that developers should consider when writing software. Some are subtle and insidious, some are not, but the main point of any of the problems listed in this section is to understand how the compiler operates, its warnings, and what can be done in code to avoid these issues.

- Use the `const` keyword wherever possible to ensure that a compiler detects a write to read-only objects. Some compilers can also perform some optimizations on `const` objects to avoid aliasing.

- Understand how temporary classes are created. As objects are transformed from one type to another (through type conversion and coercion), temporary copies of these classes can be created, invoking some constructor code and causing allocation of extra memory. Compilers sometimes warn of this issue.

- Understand what overloaded operators exist for objects in an application. Overloaded operators offer another path into user-written code that can be of arbitrary complexity. Despite the visual readability of overloading an operator to perform vector addition, for example, problems can occur when types differ and the compiler attempts to reconcile this through type conversion and coercion, incurring problems associated with temporary classes.

- Inline functions as a compiler hint wherever possible. Inlining can replace small functions with in-place code, speeding execution.

- Understand how a compiler behaves you when use C++ keywords such as `inline`, `mutable`, and `volatile`. Use of these keywords can affect how data is accessed and how compiler optimization is performed.

- Profile how run-time type identification (RTTI) performs on the systems on which an application will run. In some cases, adopting an application-specific type methodology may be more efficient, even though RTTI is part of the ANSI standard.

- Function call overhead can be significant. When structuring an application, ensure that a balance exists between the number and size of functions and overall performance. Inlining functions may help; condensing small functions into fewer larger ones can yield new tuning opportunities.

6.5.2 Virtual Function Tables

One of the core features of an object-oriented language is inheritance, and one aspect of inheritance in C++ is virtual functions. Understand where virtual functions are necessary and use them there only. Virtual functions are implemented essentially as function tables that are stored within a class instance; this class instance defines which virtual function to call when a specific instance of a class has a virtual method invoked. There are several performance issues to keep in mind when you use virtual functions.

Because the virtual function table is stored within a class instance in memory, there is associated memory overhead for this table. The increase in the size of an instance means that an instance takes up more space in main memory and requires more space when cached. Using more space when cached implies that less data overall can be in the cache. Therefore, the application is more likely to have to fetch data from main memory, thus affecting performance.

A second implication of using virtual functions is that an additional memory dereference is required when a virtual function is invoked. For more information about memory issues see Section 2.3. This overhead is relatively minor in the grand scheme, but many little things add up quickly to slow an application. Balance the costs of virtual function (and function table) invocation with a larger amount of work performed in that function. Using a method that is implemented as a virtual function (or function table in a C application) to retrieve individual vertices in a rendering loop would be a poor amortization of the startup costs.

6.5.3 Exception Handling

Exception handling is a powerful feature of the C++ language, yet it has some undesirable performance characteristics. Exceptions can be thrown from within any function at any time. Compilers must keep track of additional state data (typically with each stack frame) to preserve state in such a way that useful information can be retrieved when an exception is thrown. Tracking this additional data can cause applications that are compiled with exceptions, but perhaps not even using them, to be slower. Compilers also may not be able to optimize code as significantly with exceptions enabled. To deal with these undesirable characteristics of exceptions, follow this advice: catch exceptions that are not basic types by reference to reduce the number of copies made of exception objects; use exceptions to handle only abnormal conditions — their overhead is too great for common error handling. Understand the implications of exception use for the operating systems and compilers that are used to build an application.

6.5.4 Templates

Templates are another language feature of C++ that enables high levels of code reuse. Templates preserve type safety while they enable the same code to operate on multiple data types. The efficiency of reusing the same code for performing a certain operation for all data types stems from having to implement efficient code only once. Templates can be difficult to debug, but are easily implemented as a concrete class first, then as a template after they have been debugged. Another solution to efficient template usage is to use commercial libraries or the Standard Template Library (STL[2]), which is now part of the ANSI language

[2]The Standard Template Library — http://www.cs.rpi.edu/∼musser/stl.html

specification. Extensive use of templates may cause code expansion due to techniques that compilers use to instantiate template code. Read compiler documentation to learn how templates are instantiated on a particular system.

6.6 Conclusion

This section has covered a number of topics and issues related to high-performance C and C++ software. Software tuning can increase an application's performance, and knowledge of the programming language can further increase that performance. However, it should be obvious that the possible performance improvement is limited with these techniques. As Commike [11] writes, "the most highly tuned bubble sort in the world is still a bubble sort and will be left in the dust by any decent quicksort implementation." A need for good algorithms is evident.

Fortunately, the basics of good algorithms are taught in the early fundamentals of programming classes. Usually these topics cover general ideas such as sorting and searching. Other sources [21] offer a collection of algorithms, programs, and mathematical techniques specifically for the computer graphics programmer. These "gems" are general purpose and fit into any desired application domain.

This course has mentioned the need to reduce the amount of information rendered if an application is geometry bound. Level of detail (LOD) and culling algorithms are techniques that can be used to reduce the amount of information and complexity in a particular scene. Past SIGGRAPH conferences have offered research into these higher-level algorithms and an excellent overview of these algorithms is given in Appendix A.

Conclusion

Efficient graphics software is built on three foundations. These foundations, of course, rely on your knowledge of how software, graphics function calls, and the computer system interact with one another.

Because many graphics applications spend much of their time processing information that is not directly related to calling any graphics API, the first foundation is based on well-written application software. This software is distinct from software that is used to call any graphics API; instead, it is used to process data, take user input, or store data. Delays in the execution of this part of the code decrease overall performance. Fortunately, a host of tools are available that can clearly define any existing inefficiency in the application software.

The second foundation rests on an efficient graphics structure and how that structure interplays with the system hardware. Graphics API calls can be implemented poorly, and no amount of code analysis or restructuring will change that fact. Fortunately, most graphics hardware suppliers provide key pointers that demonstrate how to improve graphics API and hardware interaction.

Unfortunately, well-written code and graphics function calls do not compensate for a poor choice of graphics algorithms. Efficient algorithms, then, are the third foundation on which graphics performance rests. As the course pointed out, a poor algorithm can effectively kill any performance gained by clever coding or graphics hack. Fortunately, SIGGRAPH conferences are replete with examples of such algorithms, and some of them are captured here.

Creating high-performance graphics software can be difficult. The purchase of a bigger-faster-cheaper computer may be a solution, but this is a temporary solution that does not fit many situations. It is far easier – and less expensive in the long run – to examine how the software and system interact, and then modify the application software accordingly. This effort can be one of the most challenging and satisfying aspects of developing efficient graphics software.

Appendix A: Graphics Techniques and Algorithms

A-1 Introduction

In this course[1], you have learned tools and techniques to determine how well an application is running and how to improve performance. Although tuning the individual parts of an application increases performance, tuning can only go so far. The metaphor for this section is as follows: "The most highly tuned bubble sort in the world is still a bubble sort and will be left in the dust by any decent quicksort implementation." The goal of this section is to describe additional techniques that improve application performance and demonstrate how these techniques can be combined with knowledge of the application domain and system architecture to produce high-performance applications.

Each application is written to solve a specific domain problem, and each problem domain comes with a set of requirements to which the application must adhere. These requirements sometimes differ drastically among domains. For example, a visual simulation application might be required to run at a 30-Hz or even 60-Hz constant frame rate; the frame rate in a scientific visualization application might be measured not in frames per second, but seconds per frame; and, an interactive modeling application might require a delicate balance between interactive user response and image quality. Many more domains exist, each with its own set of requirements. An application writer needs to look at these requirements to determine how the application as a whole fits together to solve the user's problem. Furthermore, these requirements are usually not mutually exclusive. An application typically does not need to achieve a high constant frame rate *and* a high-fidelity scene, but a balance of both.

This section covers both idioms that are used to increase perceived graphics performance and application-level architectures that use these idioms to achieve the best possible application performance. This section primarily emphasizes interactive applications. Therefore, many of the techniques described do not fit well into an application where the end result is only a generated image, but rather are appropriate for applications where the goal is user-interactivity in generating images.

A-2 Idioms

idiom: The syntactical, grammatical, or structural form peculiar to a language [58].

The language of the computer is very specific — one misplaced symbol, and the computer no longer

[1]This section comes from this course given at SIGGRAPH 2000 and SIGGRAPH 1999. It was originally written by Alan Commike in 1999 and modified by Roger Corron in 2000.

does what is expected. When that language is used for a graphics application, similar although not as catastrophic results can occur. For example, an application might not meet the needs of the users if it is not architected properly. Many idioms help in architecting a graphics application, and these generally take the form of reducing the information that needs to be rendered. The basic premise of these idioms is that an application needs only to render what the user sees and that rendering needs to be only as detailed as the user can perceive. This may seem obvious, but few applications exist that are effective at applying all the techniques described.

The following sections outline some useful idioms for reducing the information that needs to be rendered (culling), reducing the complexity of the information that gets rendered (level of detail), and reducing the amount of data that has to be transferred at a given stage of the pipeline (caching).

Effective use of these idioms reduces both the geometry load and the pixel fill load of an application, which enables applications to render scenes that are much more complex in a shorter amount of time. Unfortunately, this effective speedup can introduce a feedback loop that can cause swings in frame rate and a reduction in the amount of time that can be spent calculating versus drawing. This feedback loop begins by reducing the graphics load, thereby increasing the effective frame rate. The increase in frame rate reduces the amount of time that is available for non-rendering tasks, which adds more geometry load to the graphics system due to less time to cull and calculate proper level of detail, and so on, creating the feedback loop. Therefore, when you use culling and multiple levels of detail, it is necessary to have a frame-rate control mechanism that can balance the graphics and CPU load.

A-2.1 Caching

Caching is the well-known technique of locally storing data that is expensive to recompute or fetch from remote storage. Caching reduces data transfer by storing graphics information in one part of the graphics pipeline so that it does not have to be retransmitted. Applied to graphics applications, caching can minimize data generation, accelerate traversal, and possibly avoid rendering altogether.

Geometry Caching - Display Lists

A *display list* is a data structure that stores graphics commands in a format that is optimized for fast traversal and transfer to the graphics system. Display lists may be provided by the graphics vendor or may be implemented within an application. Vendor-supplied display lists optimize traversal by precompiling graphics API calls into graphics commands and data structures in a format that is native to the graphics system. This format is aligned for rapid transfer to the graphics hardware by the CPU and may, depending on the system, be transferred by DMA. If your graphics vendor does not provide native display lists, it is often advantageous to implement a display list within your application. For example, if your application edits and displays NURBS or other parametric surfaces, a display list can store the surface tesselations as triangle strips, which removes the need to retesselate. Both types of display list can contain meta-information such as bounding boxes to enable other optimizations. Because display list generation and editing take time, display lists are best for caching static geometry that will be displayed more than changed. Display lists are stored in system memory, and their memory requirements need to be balanced against the performance acceleration they supply. In most cases, display lists provide a useful performance boost at a reasonable cost.

Data Caching - Paging, Tiling, and Bricking

Many applications roam through a large database that is stored on disk. The database may contain geometry, or in other cases, image or volume data that is formatted as texture. It can be worthwhile to organize the database spatially, and create a cache for the data that is most likely to displayed next. If multithreading is used, the cache can be loaded by prefetching instead of on demand.

For imaging and volume visualization applications, the data is easy to organize as tiles or bricks, with the nearest spatial neighbor implicit in the data definition. These applications are particularly amenable to data caching and prefetching. Applications that display 3D geometry are harder to organize in this way, because the natural hierarchy created by the user is not always as spatially coherent as in the imaging or volume cases.

Image Caching - Backing Store

Caching the final image can be used to avoid rendering altogether in cases where the geometry has not changed, but the image has been disturbed by outside events such as the superposition of GUI elements or the windows of other applications. The image may be saved in backing store and restored to avoid redrawing. Image saving may be done by performing a copy after rendering is complete, although this has the disadvantage of not working if the graphics window is already obscured. Also, it may be necessary to preserve other buffers than the visible part of the framebuffer, such as the depth buffer or auxiliary buffers. If your target graphics system has sufficient offscreen memory, it may be possible to perform all rendering to offscreen memory and then copy the final image to the onscreen window. This has the advantage of automatically keeping all graphics buffers offscreen and ”unobscured” at all times. Not all graphics systems have sufficient offscreen memory, but some, including some low-cost UMA systems, do. The cost in memory of backing store must be weighed against the usability cost of not providing it. The alternative technique of placing the application GUI in the system overlay planes often does an adequate job of preventing excessive rendering. Backing store works best for environments where overlay planes are unavailable or where the framebuffer must be shared with other applications that do not use the system overlay planes.

A-2.2 Culling

One of the most effective ways of improving graphics rendering performance of a scene is to not render all the objects in that scene. *Culling* is the process of determining which objects in a scene need to be drawn and which objects can safely be elided. In other words, the objects of the scene that can safely be elided are those that are not visible in the final rendered scene. This concept has fostered years of research work [19, 60, 59, 9, 23] and many useful techniques.

The premise behind culling is to determine if a geometric object needs to be drawn before actually drawing it. Therefore, the first step is to define the objects to test. In most cases, it is not computationally feasible to test the actual, perhaps very complex geometric object, so a simpler representation of the object is used: the *bounding volume*. This representation can take the form of a bounding sphere, a bounding box, or even a more complex bounding convex hull.

A bounding sphere is a point and a radius, defined to completely encompass the extents of the geometry that it represents. A bounding sphere is very fast and efficient to test against, but not very accurate in determining the extents of the object. Bounding sphere extents are fairly accurate when the dimensions of an object are similar. For example, box-shaped objects such as buildings, cars, and engines are usually well represented by bounding spheres. However, bounding spheres are a poor representation in many cases,

particularly when a single dimension is much larger than another. For example, the bounding sphere of an elongated object in a scene is much larger than the true extents of the object. Objects such as pens, trees, missiles, and railroad cars are not particularly well-represented by bounding spheres.

Significant efficiency is gained by grouping objects spatially and testing the bounding sphere of the larger group instead of testing each individual object in that group. For this to be effective, the geometry for the scene needs to be grouped hierarchically with bounding sphere information determined at the lowest levels and propagated up the tree. A bounding sphere test of a large group of geometry can quickly determine that none of its contained geometry needs to be tested, thus avoiding the test of each geometric object.

The process of recursively testing a bounding sphere and, if needed, the child geometry contained in the bounding sphere, can continue all the way down to individual geometric objects. You can use bounding boxes of the actual geometry when you need a more accurate test of the geometric extents. The level at which the bounding sphere test stops and the point at which bounding box tests are started can be based on the amount of time that is either allotted to culling the scene or set to a fixed threshold. The cull time needs to be balanced with the draw time. A very accurate cull that takes more time than the allotted frame time is not very useful. On the other hand, an early termination of the cull that causes excess geometry to be drawn slows down the overall frame rate.

Bounding boxes also suffer from some of the same problems as bounding spheres. In particular, a poorly oriented bounding box has the same problems as a bounding sphere representing an elongated object — poor representation of an object leading to inaccurate culling. Ionesˆ, *et al.*, have recently published a paper on the determination of the optimal bounding box orientation [33].

View Frustum Culling

One of the easiest forms of culling is *view frustum culling*. Geometry is identified as *full-in*, *full-out*, or *partial* with respect to the view frustum. Geometric objects that lie fully outside the view frustum can safely be elided. Geometric objects that lie fully within the view frustum must be drawn (unless elided in another culling step). Geometric objects that lie partially inside and partially outside the view frustum can either be split into the full-in portion and the full-out portion, or added to the full-in list to be clipped by the hardware when rendered.

An advantage of differentiating between full-in and partial can come with systems that implement software clipping. In some cases, the graphics library implementation allows the application to turn off clip testing when all geometry lies fully within the view frustum. In these cases, there is a contract between the application and the graphics library: the application agrees not to send geometry that lies outside of the view frustum, and the graphics library agrees to speed processing of the geometry. Rendering results are undefined if this contract is broken by sending down geometry outside of the view frustum. Usually, the undefined results manifest in the form of an application crash or an improperly rendered scene.

Like many operations that change graphics state, notifying the graphics system that geometry does not need to be clipped is not a computationally free operation. This means that the application should be structured so that it does not have to repeatedly turn on and off the clipping state when rendering partial and full-in geometry.

Backface Culling

Manifold surfaces always have some polygons that are facing the viewer and others that are facing away from the viewer. Polygons that are facing away from the viewer are not visible and do not need to be

rendered. The process of determining which polygons are frontfacing (visible)and which are backfacing (not visible), and then eliding those that are backfacing is called *backface culling* [60]. Backface culling is done on a per-object, and sometimes per-primitive, basis.

OpenGL performs backface culling as the first step of rasterization after clipping, transformation, and lighting. This only eliminates rasterization, which is not helpful for applications bound by transformation and lighting. In such graphics systems, it can be worthwhile to perform backface culling explicitly, before transformation takes place.

A simple approach to calculating the face of a polygon is to take the dot product of the polygon normal and a ray from the camera (or eye-point). If the dot product is negative, the polygon is facing toward the user and needs to be drawn. If the dot product is positive, the polygon is facing away from the user and can safely be elided. One aspect of dot products that needs attention is the meaning of the dot product sign. When the user is inside the object, the meaning of the positive and negative dot product is reversed. The possibility of the eye point entering an object needs to be handled in all cases where the direction of the normal is important, such as lighting. Backface culling adds an additional case to the handling of flipped normals.

Before implementing your own backface culling, test your application performance and check your vendor's documentation. OpenGL backface culling may be adequate and if not, your vendor may provide an extension to perform camera-space backface culling.

Occlusion Culling

A more complex form of culling, *occlusion culling*, is the process of determining which objects within the view frustum are visible. Only objects that are not behind other objects or are not seen through those objects from the current viewpoint are visible in the final rendered scene. The objects that are visible are known as *occluders*, and those that are blocked are known as *occludees*. The determination of the optimal set of occluders is the goal of an occlusion culling algorithm. The objects in this optimal occluder set are the only objects that need to be drawn; all other objects can safely be elided.

The key to an effective occlusion culling algorithm is to determine which objects in a scene are occluders. In many cases, you can use the information that is available in the application domain as a means to help determine the occluders. In domains such as architectural walkthroughs or certain classes of games, the world is naturally made up of *cells* and *portals* between the cells. In this case, you can use a cell and portal [55] culling algorithm to make a map of the visibility between cells. Only cells that are visible from the current cell need to be rendered.

When knowledge about the underlying spatial organization does not lead to the use of a specialized algorithm to determine occluders, you can use a general occlusion algorithm [61, 23]. One method of occlusion culling is to use the hierarchical bounding-box or bounding-sphere information in conjunction with a typical hardware depth buffer. The scene is sorted in a rough front-to-back ordering, and all geometry in the scene is marked as a possible occluder, meaning that all geometry needs to be drawn. The depth sort is necessary to take advantage of the natural visibility effects where a closer object generally obstructs the view of a further object. The bounds of each object are rendered in turn, and the depth buffer is compared to the previous depth buffer. If the depth buffer changes between drawing one object and the next, the object is visible and is not occluded. If the depth buffer did not change, the object is not visible and can safely be elided. It is possible that the hardware can efficiently feed back the depth buffer hit information outside of reading the full depth buffer. Check with your hardware vendor when you implement an occlusion culling algorithm to find out if there are extensions that enable efficient occlusion culling algorithms.

More detail on occlusion culling can be found in Zhang [59], which covers occlusion culling background material and an extensive algorithm for choosing the optimal occlusion set.

Contribution Culling

You can also use culling to elide objects that are small enough not to be noticed if they are missing from the scene. This form of culling, called *contribution culling* [59], makes a binary decision to draw or not draw an object depending on its pixel coverage in screen space. An object that only occupies a few pixels in screen space can be safely elided with very little impact in the overall scene. Examples of situations in which contribution culling can be applied include objects that are a large distance from the eye, such as trees when flying at altitude in a flight-simulator, or objects that are very small in comparison to the entire scene, such as bolts on an engine when designing a truck. Contribution culling can also assist occluder selection for occlusion culling, because objects with low pixel coverage are not good occluders.

The screen space size of an object can be determined either computationally or in a preliminary rendering pass. In either method, the bounding representation is used instead of the actual geometry that is associated with the object. Check with your hardware vendor when you implement a contribution culling algorithm. It is possible that the hardware can efficiently feed back the pixel coverage information much easier and faster than a computational approach or straightforward graphics language implementation.

A-2.3 Application-specific Heuristics and Combinations of Idioms

Typically, these idioms are combined in a pipelined fashion. First, an appropriate level of detail is selected; then, multiple stages of culling are applied to reduce the geometry load on the graphics system. Caching is used at various stages of the pipeline to minimize and optimize data transfer. However, knowledge of your application domain may enable you to invent combinations of these idioms or heuristics that accelerate your application more than generic techniques. Some examples are given below.

Accelerated Panning

> If users of your application typically spend a lot of time panning complex 3D images, you can combine image caching and view frustum culling to accelerate classic translation. Blit the image to translate the part of the image that will remain visible after translation; then, use view frustum culling on the exposed region to render only new exposed geometry.

Accelerated Dynamics

> If you must render complex geometry of which only a small part is dynamic, the following technique can be useful. The static geometry is rendered first to form the background, and all the output buffers (including depth and auxiliary buffers) are saved. Then, instead of clearing the frame buffer and redrawing everything, the bounding box of the dynamic geometry is restored in all buffers at the beginning of each frame and only the dynamic geometry is redrawn. Again, system architecture and cost will affect your design. [22]

Oversampled Antialiasing

> Antialiasing has classically been done by either blending, which in the general case requires depth sorting of polygons to avoid blending artifacts, or by accumulating several renderings of the image, each displaced by a small amount. Each method has disadvantages. The depth sorting that is required for blending can add large amounts of data generation time to the graphics pipeline. The

multiple renderings that are required for accumulation buffer antialiasing adds very large amounts of traversal, transformation, and rasterization. If your target graphics system has fast imaging operations and sufficient framebuffer available, it can be worthwhile to implement antialiasing by *oversampling*. In this technique, the image is rendered once in a large offscreen buffer, and then filtered into the smaller visible window with an appropriate (and fast) kernel. When compared to blending, oversampling trades off framebuffer use against the CPU and main memory requirements of a depth sort. When compared to accumulation, oversampling has a fraction of the traversal and transformation requirements, comparable rasterization requirements, and a higher framebuffer requirement.

Substitute Texture for Geometry

If your target systems have texturing operations, substitution of texture for detailed geometry can accelerate performance by reducing the amount of data that is sent down the graphics pipeline and transformed. A specific example of this is the acceleration of high-quality shading by substituting a sphere mapped texture of the shading model and coarsely tessellated geometry for finely tessellated geometry that is lit and shaded by the graphics system. This technique can also be used to support shading models that are not implemented in the target graphics API. Because the texture must be computed in software, this technique is most useful for the combination of dynamic geometry and static lighting conditions.

Accelerated panning and accelerated dynamics utilize *frame coherence*. Each frame of the output image contains much of the data from the previous frame, so much so that it is worthwhile to cache it and rerender only a small part of the image. Oversampled antialiasing substitutes a fast 2D operation that requires a large amount of framebuffer for 3D operations that conserve framebuffer but use large amounts of time. The substitution of texture for geometry also substitutes fast 2D operations for slow 3D operations. There may be other ways to accelerate your application. Is your geometry laid out in such a way that some objects will always make better occluders than other objects? (For example, sheet metal vs. rivets.) Do you typically draw large arrays of coplanar or parallel surfaces that can be backface removed together at a small computational cost? Step back and cast a critical eye at your application, the problems it solves, and its usage.

A-2.4 Level of Detail

As the viewpoint of a scene changes, more or fewer pixels are devoted to rendering each object in the scene. By taking advantage of reductions in the pixel area of a full-fidelity image, a corresponding reduction of the geometric complexity can be introduced. The idea is to introduce a *level of detail* (LOD, pronounced läd) for each object in a scene [9, 26, 41]. When an object is far from the viewer, fewer triangles need to be devoted to rendering the object to retain the same image fidelity.

Many types of models exist that you can simplify with multiple levels of detail. Two of the larger classes are large, relatively flat *terrain* or *height field* models that stretch into the horizon and general 3D object models such as cars, buildings, and the associated parts of each. These two classes require different techniques for LOD manipulation. A continuous terrain model needs to have a higher level of detail close to the user and a lower level further back, where both levels are active in the same object at the same time. You can use specialized terrain LOD algorithms [39] or general adaptive algorithms if they allow the decimation factor to vary over the model in a view-dependent fashion [10, 29]. In most cases, a general 3D object, where the size of the object is small compared to the full scene, has a constant LOD at any point in

time. As the eye-point moves closer to the object, more detail is displayed. As the eye-point moves further from the object, less detail is displayed. The LOD can be calculated prior to rendering [10] or calculated "on the fly" as a progressive refinement of the object [29].

As the user moves through a scene, the LOD for each object or group of objects changes. Each new LOD is potentially a new geometric representation. Simply rendering the new representation instead of the old is considered a *hard change* in the scene. In other words, as the user transitions from one LOD to another, the transition is noticed by the user as a "popping" effect. You can minimize this effect by using softer methods of LOD transitions such as geometry morphing (geomorph) or blending. A good LOD implementation should present few visual artifacts to the user.

Creating the LOD objects is only part of the full LOD idiom. To effectively use multiple LOD objects in a scene, you must determine the correct LOD for each object. Properly determining the correct LOD can greatly increase frame rate [19, 46, 50]. The LOD can be based not only on the distance from the eye, but also on the cost of rendering the object and the perceived importance within the scene [19]. In many cases, the geometry can be totally replaced by a textured image [50], thereby reducing the geometry load to a single polygon.

Creating the LOD Models

Geometric models come from many sources and can vary greatly in their complexity. Very dense models arise from 3D scans of real-world models, from surface extraction of volumetric data, terrain acquired by satellite, and parametric surfaces generated from a modeling package. Clark [9] first proposed the use of simplified models to increase frame rate while rendering interactive applications. Since then, geometric surface simplification has been a strong research topic. Heckbert and Garland [27] provide a complete survey of geometry surface simplification along with a taxonomy of algorithms that span multiple disciplines.

Using multiple LODs within the same scene is also known as *multiresolution modeling*. With multiresolution modeling, there is no need to render a highly tessellated model when the tessellation detail is not visible in the final scene. Heckbert and Garland classify surface simplification algorithms into three classes: height fields, manifold surfaces, and non-manifold surfaces. A simplistic definition of a manifold surface is one where an edge is shared between only two triangles or not shared at all.

Height Fields

Heckbert and Garland further subdivided height fields into six subclasses: regular grid methods [36, 32], hierarchical subdivision methods [54, 47, 15], feature methods [52], refinement methods [18, 28, 45, 20], decimation methods [37, 48], and optimal methods [5]. Many of these algorithms are very computational and therefore, can be used only to preprocess the LODs that are used during rendering. These preprocessed LODs are generally not sufficient for an interactive application where the user controls the eye-point and viewing parameters. This is especially true for surfaces that are very large, as in terrain models, where a single LOD is not sufficient over the whole surface. In fully interactive applications, the LOD across the height field needs to be what Hoppe refers to as "view-dependent" [30]: the LOD across the height field varies as the eye-point and view frustum changes. This entails a real-time algorithm with a continuously variable LOD that enables more detail close to the eye-point and less further away.

The number of algorithms that allow view-dependent, real-time height field LOD calculations is small. For the algorithm to be effective, Lindstrom *et al.*[39] defines five properties that are important for a height field LOD algorithm:

- At any instant, the mesh geometry and the components that describe it should be directly and efficiently queryable, allowing for surface following and fast spatial indexing of both polygons and vertices.
- Dynamic changes to the geometry of the mesh, that lead to recomputation of surface parameters or geometry should not significantly impact the performance of the system.
- High frequency data such as localized convexities and concavities and local changes to the geometry should not have a widespread global effect on the complexity of the model.
- Small changes to the view parameters (for example, viewpoint, view direction, field of view) should lead only to small changes in complexity to minimize uncertainties in prediction and allow maintenance of (near) constant frame rates.
- The algorithm should provide a means of bounding the loss in image quality that is incurred by the approximated geometry of the mesh. That is, a consistent and direct relationship should exist between the input parameters to the LOD algorithm and the resulting image quality.

A single algorithm that fulfills all of these properties, runs in real time, and handles very large surfaces is difficult to achieve. The IRIS Performer [46] library's Active Surface Definition (ASD), Linstrom's [39] algorithm, and Hoppe's view-dependent progressive mesh [31] are some examples of algorithms that fulfill all properties. These algorithms depend on a hierarchical surface definition but take different approaches to achieve a similar result. Lindstrom and Hoppe work with the original height field breaking the surface into LOD blocks. They simplify each block with a continuous LOD function that is based on eye position, height, and an error tolerance. The ASD algorithm starts with a triangulated irregular network (TIN) and precomputes the LOD blocks. Lindstrom works with the entire surface but limits the maximum size that can be rendered to what can fit in memory. In addition, even though the LOD is continuous, Lindstrom does not geomorph the surface when changing from one level to another, which can cause a noticeable popping effect. In contrast, ASD and Hoppe store the hierarchical LOD blocks on disk and load the appropriate block as needed, depending on the viewer velocity and direction. This action enables an infinite surface to be convincingly rendered. Furthermore, both ASD and Hoppe geomorph the vertices as the LOD level changes. This step produces a smooth-looking surface representation even when the error tolerance becomes high.

Manifold and Non-Manifold Surfaces

Manifold and non-manifold surfaces are a more general simplification problem than height fields because the surface does not fall into a simple 2D parameterization. Many methods have been constructed [56, 49, 17, 29, 10, 30, 40] to solve this problem, each having advantages and disadvantages. Recently, these simplification algorithms have expanded the domain coverage to include real-time algorithms [34, 30, 40] and view-dependent information [10, 30, 40], and to attain higher compression rates for low-bandwidth transmission of data [8, 53, 24].

Determining Which LOD to Use

Generating multiresolution models is only the first step to effectively using LODs in an application. The second step of the problem is to decide when to use which LOD level [19, 46]. This is a very important consideration with little formal information published. The generation of LOD models is rooted in computational geometry and statistical error measurements, whereas determination of which LOD model to use is purely heuristic.

The goal with interactive applications is to keep the system in *hysteresis*, meaning that the changes due to user viewpoint and scene complexity should have a minimal effect on frame rate. The first step in achieving this goal is to decide on a frame rate. The desired frame rate depends on the application domain. A visual simulation may need to run at 30 Hz or 60 Hz, whereas a scientific visualization of hundreds of megabytes of data may require only a 1-Hz frame rate. Most applications are somewhere in the middle and in many cases, do not target a specific frame rate. This target frame rate helps determine which LODs need to be used and without it, the graphics pipeline may be under-utilized or overloaded. A target frame rate sets a bound on the minimum frame rate without which the frame rate is unbounded, which enables an application to become arbitrarily slow.

Often, application developers who have not incorporated frame rate control into their applications, rationalize the decision by saying that they always want the fastest frame rate; hence, they do not need to set a target frame rate. This viewpoint is always countered by the fact that a frame-rate control mechanism combined with LODs enables the fastest frame rate to be increased by using less complex LODs. For example, if an application is running slower than the target frame rate, it can decrease the LOD complexity, thereby reducing the geometry load on the system and increasing the overall frame rate. Without a target frame rate and associated frame-control mechanism, increasing the frame rate cannot happen reliably. Adjusting the geometry load based on the difference between current and target frame rate is known as *stress management*. Stress is a multiplier, which is calculated on this difference and incorporated into the LOD selection function. One method of determining which LODs to render is to determine the cost in frame time it takes to render each object and the benefit of having that object at a certain LOD level.

Funkhouser *et al.* [19] defines cost and benefit functions for each object in a scene. The cost of rendering an object O at level of detail L with rendering method R is defined as $Cost(O, L, R)$, and the benefit of having object O in the scene is defined as $Benefit(O, L, R)$. Therefore, to determine the LOD levels for all objects in a scene, S, maximize

$$\sum_{S} Benefit(O, L, R)$$

subject to

$$\sum_{S} Cost(O, L, R) \leq TargetFrameRate.$$

Generating the cost functions can be done experimentally as the application starts by running a small benchmark to determine the rendering cost. This benchmark can render some of the basic graphics primitives in different sizes by using multiple graphics states to determine the characteristics of the underlying system. The cost of rendering certain primitives is useful not only for LOD control, but also for the general case of determining some of the fast paths on given hardware. Of course, though the benchmark is not a substitute for detailed system analysis, you can use it to fine-tune for a particular platform. It is up to the application writer to first determine which modes and rendering types are fastest separately and in combination for a particular platform and then to code those into the benchmark.

The $Benefit$ function is a heuristic based on rasterized object size, accuracy of the LOD model compared to the original, importance in the scene, position or *focus* in the scene, perceived motion of the object in the scene, and hysteresis through frame-to-frame coherence. Unfortunately, optimizing the above for all objects in the scene is NP-complete and therefore too computationally expensive to attempt for any real data set size. Funkhouser *et al.* uses a greedy approximation algorithm to select the objects with the highest $Benefit/Cost$ ratio. They take advantage of frame-to-frame coherency to incrementally update the LOD for each object starting with the LOD from the previous frame. The $Benefit$ and $Cost$ functions

can be simplified to reduce the computational complexity of calculating the LODs. This computational complexity can become the overriding frame time factor for complex scenes, because the LOD calculations increase with the number of objects in the scene. Similar to using LODs to reduce geometry load, it is necessary to measure the computational load and reduce computation when the calculations begin to take more time than the rendering.

Using a predictive model such as the one described above, you can control the frame rate with higher accuracy than with purely static or feedback methods. The accuracy of the predictions highly depend on the $Cost$ function accuracy. To minimize the divergence of actual frame rate to calculated cost, you can introduce a stress factor to artificially increase the LOD levels as the graphics load increases. This stress factor is a feedback loop that depends on the true frame rate.

Using Billboards

Another approach to controlling the level of geometric detail in a scene is to substitute an *impostor* or a *billboard* for the real geometry [46, 42, 50, 51]. In this idiom, the geometry is pre-rendered into a texture and then texture mapped onto a single polygon or simple polygon mesh during rendering. This is an advanced form of the texture for the geometry substitution that is described in section A-2.3. IRIS Performer [46] has a built-in billboard (sometimes known as *sprite*) data type that can be explicitly used. The billboard follows the eye-point with two or three degrees of freedom, which appear to the user as if the original geometry is being rendered. Billboards are used extensively for trees, buildings, and other static scene objects.

Shade *et al.* [50] creates a BSP tree of the scene and renders via a two-pass algorithm. The first pass caches images of the nodes and uses a cost function and error metric to determine the projected lifespan of the image and the cost to simply render the geometry. The projected lifespan of the image alleviates the problem of the algorithm trying to cache only the top-level node. A second pass renders the BSP nodes back to front by using either geometry or the cached images. This algorithm works well for sparsely occluded scenes. In dense scenes, the parallax due to the perspective projection shortens the lifetime of the image cache, thereby making the technique less effective.

Sillion *et al.* [51] have a similar approach, but instead of using only textures mapped to simple polygons, they create a simplified 3D mesh to go along with the texture image. The 3D mesh is created through feature extraction on the image that is followed by a re-projection into 3D space with the use of the depth buffer. This 3D mesh has a much longer lifetime than 2D texture techniques, but at the expense of much higher computational complexity in the creation of the image cache.

A-3 Application Architectures

There are many techniques that have wide ranging ramifications on the whole or part of the application architecture. Applying these techniques along with the above idioms, efficient coding practices, and some platform-dependent tuning helps ensure that the underlying application performs as well as possible on the target platform.

A-3.1 Multithreading

Multithreading is the general ability to have more than one thread of control that shares a work load for a single application. These threads run concurrently on multiprocessor machines or are scheduled in some

manner on single-processor machines. Threads also may all reside within the same address space or may be split across separate exclusive address spaces. In a cluster of workstations, threads execute on separate machines and communicate by a message passing interface. The mechanism of thread control is not as important as the need to use multiple threads within an application.

Even when using only a single processor, multithreading can still improve application performance. Additional threads can accomplish work while the main thread is waiting for something to happen, which is quite often. The main thread may be waiting for a graphics operation to complete before issuing another command; waiting for an I/O operation to complete and block in the I/O call; or waiting for memory to be copied from main memory into the caches. In addition, when multiple processors are available, the threads can run free on those processors and do not have to wait for the main thread to stall or to context swap to get work done.

Multiple threads can do many of the computational tasks that are involved in deciding what to draw, such as LOD control, culling, and intersection testing. Threads can be used to page data to and from disk or to pipeline the rendering across multiple frames. Again, an added benefit comes when running the application on multiprocessing machines. In this case, the rendering thread can spend 100% of its time rendering while the other threads are dedicated to their tasks 100% of the time.

A few issues are associated with using multiple threads. The primary concern becomes data exclusion and data synchronization. When multiple threads act on the same data, only one thread can change the data at a time. That change then needs to be propagated to all other threads so that they see the same consistent view of the data. It is possible to use standard thread-locking mechanisms such as semaphores and mutexes to minimize these multiprocessing data management issues. This approach is not optimal, because as the number of objects in the scene increases, the corresponding locking overhead also increases. A more elaborate approach that is based on multiple memory buffers is described in [46]. Another issue is the time consumed by thread creation. It may be worthwhile to cache and reuse threads instead of creating and destroying them freely. As in all other aspects of graphics, performance measurement is an essential part of threaded architecture design.

Threads can be used in a pipelined fashion or in a parallel fashion for rendering. In many cases, combining the two techniques produces the greatest performance benefit. In a pipelined renderer, each stage of the pipeline works on an independent frame with its own view of the data. Here the latency is increased by the number of stages in the pipeline, but the throughput is also increased. Parallel concurrent processes all work on the same frame at the same time, perhaps by using multiple hardware graphics pipelines (see 2.5). The synchronization overhead is higher, but latency is reduced. A combination of the two approaches can have a pipelined renderer that has asynchronous concurrent threads handle non-frame-critical aspects of the application such as I/O. The target system architecture determines what is possible, whereas the application requirements determine what is useful. The following operations are some areas where a separate thread can work either as a stage in a pipeline or as a parallel concurrent thread.

Culling

The process of culling determines which geometric objects need to be drawn and which geometric objects can be safely elided from the scene (see section A-2). Culling is traditionally done early in the rendering process to reduce the amount of data that later stages need to process. As one of the first stages in a multithreaded application, the culler thread can traverse the scene doing view frustum, backface, contribution, and occlusion culling. Each of these culling algorithms can be done in a pipelined fashion spread over multiple threads. The resulting output of the culling threads can be incorporated into a new second-stage scene structure, which is passed to the remaining parts of the application.

Level of Detail Control

Multiple levels of detail (LOD) per object is one of the most effective ways of reducing geometric complexity (see A-2.4). The determination of the correct LOD for each object can be a time-consuming task and is perfectly suited to run in a separate thread. LOD threads should run after the culling stage, or be pipelined with early results from the culling stage to prevent calculation of LOD values for objects that are not rendered.

Intersection

Most applications do more than just render and they enable the user to interact with the scene. This interaction entails calculating intersections either on an object-to-object basis or as a ray cast from a viewing position to an object. An intersection thread can be run concurrently with LOD calculations to generate a hit list that is passed to the application before rendering.

I/O

In applications where all data is generally not all visible simultaneously, it is beneficial to load only the portion of the data that is currently being used. Complex visual simulations or architectural walkthroughs are two of the many types of applications that have large *databases* in which the data is *paged* off the disk as the user moves through the world. As the user approaches an area where the data has not yet been loaded, the required data is read off the disk or a network interface to be ready to use when the user arrives at the new area. One or more asynchronous threads are generally allocated to I/O operations such as paging database data from external storage or tracking information from input devices. These threads can be asynchronous because they do not need to complete to generate data for the next frame of the rendering process. An additional benefit of an asynchronous I/O thread is that an application is not tied to the variable read rates that are inherent in disk, network, or other external interfaces. The maximum frame rate of an application is gated by the I/O device when I/O is done inline as part of the rendering loop. This point is especially apparent with input devices that have a very high data latency that put a bounds on the frame rate.

Because I/O threads are asynchronous and may not have completed their operation before the data that they are responsible for is needed, the application needs to have a fallback to replace the missing data. Database paging operations can first bring in small, low-resolution data that is quick to read to ensure that some data is ready to be rendered if needed. Similarly, missing tracking information can simply reuse previous data or interpolate where the new position may be based on the previous heading, velocity, and acceleration.

A-3.2 Memory vs. Time vs. Quality Trade-offs

There are many trade-offs between memory, time, and quality that need to be considered. Depending on the target audience and application type, memory utilization may be a higher priority than frame rate, or frame rate may be most important regardless of the amount of memory needed. Quality has similar issues: higher quality may mean more memory or slower frame rate.

Level of Detail

Changing between appropriate LODs for a given object should be almost invisible to a user. When LOD levels are artificially changed because of the need to increase frame rate, users begin to notice changes in the scene. Here, frame rate and image quality need to be balanced. Similarly, if a proper blend or morph between two LOD levels is not done, the switch between the two LODs is apparent and distracting. In either case, the use of LODs is important for an application. Memory considerations for generating LODs should be a concern only for very memory-conscious applications. If memory becomes a concern, consider paging the LOD levels from disk when needed.

Mipmapping

Textures can be pre-filtered into a multiple power of two levels that form a pyramid of texture levels. During texture interpolation, the two best mipmap levels are chosen, and texel values are interpolated between those levels. This process reduces texturing complexity when the ratio of screen space to texture dimension gets very small. Interpolation between smaller levels produces a better image at the cost of memory to store the texture levels and a possible performance hit on some graphics systems that do not have hardware support for mipmapping. The memory bloat that is associated with mipmapping is minimal in fact, adding only one-third the original image size. This memory bloat is usually outweighed by the increase in image quality and performance for hardware that accelerates mipmapping.

Paging

For very large databases or other types of applications that work with large data sets, all of the data does not have to be loaded upfront. An application should be able to roam through an infinitely large scene if it is supplied with an infinitely large disk array.

Lower Fidelity Scenes

The full-fidelity scene does not always need to be drawn in interactive applications. Draw a more coarse approximation of the scene if the render time falls below interactive rates. As more time becomes available, draw higher fidelity scenes. Infinite time is available when the user is not moving, so you can use advanced rendering techniques to further improve the quality of a static scene.

A-3.3 Scene Graphs

All graphics applications have some sort of scene graph. A scene graph is the basic data structures and traversal algorithms that render from those data structures. Some small changes exist that you can make in the scene graph and use throughout the application to make a large impact on the overall usability of the application. Be aware that a scene graph API can get very complex with more time spent on creating the scene graph API than the domain-specific application. It is often more efficient both in terms of the time required and scene graph performance to use an off-the-shelf scene graph API.

Bounding Information

One of the easiest pieces of information to use and most beneficial to store in the scene graph is bounding information for objects in a scene. Both bounding spheres and bounding boxes may be stored, each used where appropriate.

Pre-Calculations

Many times objects in a scene have static transformations that are associated with them; for example, wheels of a car are always positioned relative to the center of the car, offset by some transformation. These extra transformations can quickly add up with complex scenes. A pass through the scene graph can be done before rendering begins to collapse static transformations by recalculating the vertices of the objects, physically moving the vertices to their transformed locations. You can similarly concatenate other states in the scene, namely rendering modes, colors, and you can even pre-calculate lighting in some situations.

State Changes

State changes are generally an expensive operation for most graphics systems. Try to render all items with the same state to minimize the number of times state needs to be changed in a scene. Rendering a geometric checkerboard is much faster if you render all black squares first followed by all white squares, instead of rendering alternate black and white squares. If each object is able to keep track of the state settings it uses, then sort the scene by state becomes possible and rendering becomes more efficient. This sorting creates lists of renderable items that have multiple levels of sorting from most expensive to least expensive.

Performance Monitoring and Timing

Obtaining accurate timing is useful when you decide how much can be drawn per frame. This timing information can be supplemented with information about the number and types of primitives that are being drawn, how many state changes are taking place, the relative time each thread of control takes to do its job, measured threading overhead, and many other interesting pieces of information.

For debugging purposes, it is useful to know what is actually being drawn, especially when trying to fix a fill-limited or geometry-limited application to see how the state changes affect what is actually rendered. Besides timing information, the depth complexity of a scene should be viewable as an image of the depth buffer to see how many times each pixel is filled. This is a measure of how well the culling process is performing. It is also useful to be able to turn off certain modes to see their effect. For example, turning off texturing or drawing the scene in wire frame can be useful for debugging.

Static vs. Interactive Scenes

Many applications present a scene to the user, enable the user to modify the scene in some way, and then present the updated scene to the user. A scene presented in this fashion can be considered a *static scene* because it needs to be of high quality but not interactive. Scenes that users interact with should be of high quality, but primarily should be rendered with interactivity of a higher priority than higher quality.

An interactive scene needs to use many of the previous techniques (such as culling and LODs), but may have to go even further to reduce complexity to achieve responsive user interaction. This process may include removing specific object representations by substituting bounding boxes for them.

Appendix B: Multipipe Decomposition Algorithms

This appendix presents pseudo-code for each of the decomposition methods described in Section 2.5.3.

B-1 Image-space Decomposition

```
// In image-space decomposition, the image is subdivided by
// screen space.  Each pipe gets a section of the image-space
// view-volume. Sort so only that section of data goes to each pipe.

sort_geometry_by_screen_space();

for( pipe_num < num_pipes; pipe_num++ ) {
  set_graphics_context_to_window_on_pipe( pipe_num );
     /* OpenGL/glX: glXMakeCurrent( pipe_num ); */

  render_individual_pipe_data( pipe_num );
     /* OpenGL: glBegin/End */

  save_image_buffer( color_buffer, pipe_num );
     /* OpenGL: glReadPixels( ... GL_RGB ... ); */
}

//* ensure all pipes have finished rendering before proceeding.
barrier_wait_for_all_pipes_to_finish();

set_graphics_context_to_window_on_pipe( output_pipe );
     /* OpenGL/glX: glXMakeCurrent( output_pipe ); */

for( pipe_num < num_pipes; pipe_num++ ) {
  restore_image_buffer( color_buffer, pipe_num );
      /* OpenGL: glDrawPixels( ... GL_RGB ... ); */
}
/* image recomposition complete: display final image */
```

B-2 Depth-space Decomposition

```
// In depth-based decomposition, each pipe gets a section of the
// depth-space view-volume.

sort_geometry_by_depth();

for( pipe_num < num_pipes; pipe_num++ )
{
  set_graphics_context_to_window_on_pipe( pipe_num );
     /* OpenGL/glX: glXMakeCurrent( pipe_num ); */

  render_individual_pipe_data( pipe_num );
     /* OpenGL: glBegin/End */

  save_image_buffer( color_buffer, pipe_num );
     /* OpenGL: glReadPixels( ... GL_RGB ... ); */

  save_image_buffer( depth_buffer, pipe_num );
     /* OpenGL: glReadPixels( ... GL_DEPTH ... ); */
}

// ensure all pipes have finished rendering before proceeding.
barrier_wait_for_all_pipes_to_finish();

set_graphics_context_to_window_on_pipe( output_pipe );
     /* OpenGL/glX: glXMakeCurrent( output_pipe ); */

for( pipe_num < num_pipes; pipe_num++ )
{
  enable( DEPTH_TEST & STENCIL_WRITE );

  restore_image_buffer( depth_buffer, pipe_num );
      /* OpenGL: glDrawPixels( ... GL_DEPTH ... ); */

  disable( DEPTH_TEST & STENCIL_WRITE );
  enable( STENCIL_TEST );

  restore_image_buffer( color_buffer, pipe_num );
      /* OpenGL: glDrawPixels( ... GL_RGB ... ); */
}

/* image recomposition complete: display final image */
```

B-3 Geometry-space Decomposition

```
// Each pipe gets a fraction of the total geometric objects. Each
// pipe views the entire view-volume.

divide_geometry_among_pipes();

for( pipe_num < num_pipes; pipe_num++ )
{
  set_graphics_context_to_window_on_pipe( pipe_num );
     /* OpenGL/glX: glXMakeCurrent( pipe_num ); */

  render_individual_pipe_data( pipe_num );
     /* OpenGL: glBegin/End */

  save_image_buffer( color_buffer, pipe_num );
     /* OpenGL: glReadPixels( ... GL_RGB ... ); */

  save_image_buffer( depth_buffer, pipe_num );
     /* OpenGL: glReadPixels( ... GL_DEPTH ... ); */
}

// ensure all pipes have finished rendering before proceeding.

barrier_wait_for_all_pipes_to_finish();

set_graphics_context_to_window_on_pipe( output_pipe );
     /* OpenGL/glX: glXMakeCurrent( output_pipe ); */

for( pipe_num < num_pipes; pipe_num++ )
{
  restore_image_buffer( depth_buffer, pipe_num );
      /* OpenGL: glEnable( DEPTH_TEST );
       * OpenGL: glDrawPixels( ... GL_DEPTH ... ); */

  restore_image_buffer( color_buffer, pipe_num );
      /* OpenGL: glDrawPixels( ... GL_RGB ... ); */
}

/* image recomposition complete: display final image */
```

B-4 Time-based Decomposition

```
// Each pipe gets the 'next' frame. The next frame is computed by
// either continuously sampling input devices, or by extrapolating
// along some smoothed previous 'n' input steps. In either case,
// each subsequent new view is rendered on another pipe, then
// back to the main pipe.

sort_geometry_by_pipe();

for( pipe_num < num_pipes; pipe_num++ )
{
  set_graphics_context_to_window_on_pipe( pipe_num );
     /* OpenGL/glX: glXMakeCurrent( pipe_num ); */

  set_view( new_view_matrix );
  render_all_data();

  save_image_buffer( color_buffer, pipe_num );
     /* OpenGL: glReadPixels( ... GL_RGB ... ); */

  set_graphics_context_to_window_on_pipe( output_pipe );
     /* OpenGL/glX: glXMakeCurrent( output_pipe ); */

  restore_image_buffer( color_buffer, pipe_num );
}
```

Glossary

API: See Application Programming Interface.

Application Programming Interface: A collection of functions and data that together define an interface to a programming library.

ASIC: Application Specific Integrated Circuit. Examples of ASICs include chips that perform texture-mapping, lighting calculations, or geometric transformations.

Asynchronous: An event or operation that is not synchronized. Asynchronous function calls are those that can occur at any time and do not wait for other input to complete before returning.

Bandwidth: A measure of the amount of data per time unit that can be transmitted to a device.

Basic Block: A section of code that has one entry and one exit.

Basic Block Counting: Indicates how many times a section of code has been executed (the hot spot), regardless of how long an instruction might have taken.

Billboard: A texture, or multiple textures, that represent complex geometry. The texture is mapped to a single polygon that follows the eye-point.

Binary Space Partitioning: Usually referred to as a BSP tree. This is a data structure that represents a recursive, hierarchical subdivision of space. The tree can be traversed to quickly find the locations of items in a scene.

Block: The process of not allowing the controlling program to proceed any further in its current thread of execution until the device that is being communicated with is finished with its operation.

Bottleneck: A point in an application that is the limiting factor in overall performance.

Bounding Box: The extents of an object that are defined by the smallest box that fits around the object. A bounding box can be axis-aligned or oriented in some way to better fit the object extents.

Bounding Sphere: The extents of an object that are defined by the smallest sphere that fits around the object.

Bounding Volume: The extents of an object or group of objects that can be defined by using a bounding box, bounding sphere, or other method.

BSP Tree: See Binary Space Partitioning.

Cache Blocking: Memory reference optimization that reorders the memory accesses in a loop nest so that data are worked on in small neighborhoods that fit in cache. Also known as tiling.

Cache Line: The smallest unit of transfer into a cache.

Callstack Profiling: See Program counter profiling.

Contribution Culling: A binary decision to draw or not draw depending on the pixel coverage in screen space.

COW: See Cluster of Workstations.

CPU: Central Processing Unit.

Cluster-of-Workstations: A collection of workstations that is designed to be used to produce a single computational or graphical result.

Culling: The process of determining which objects in a scene need to be drawn and which objects can safely be elided.

Data Locality: The property of data to reside 'near' other data in memory. One way to achieve data locality is to use a vertex array, which stores vertices linearly in memory - subsequently accessed vertices will be adjacent to just-used vertices, and likely have better cache behavior.

Database: The application one buys from Oracle or Sybase. Also, the store of data that can be rendered. Usually used in the visual simulation domains.

Depth Complexity: The measure of how many times a single pixel on the screen is filled. Depth complexity can be reduced by using Culling.

Direct Memory Access: A way for a piece of hardware in a system to bypass the CPU and read directly from the memory. This is generally faster than the PIO, but there is a constant setup time that makes DMA useful only for large data transfers.

Display: The output device.

DMA: Direct Memory Access.

FIFO Buffer: A mechanism that mitigates the effects of the differing rates of graphics data generation and graphics data processing.

Fill Rate: A measure of the speed at which pixels can be drawn into the frame buffer. Fill rates are reported as a number of pixels that can be drawn per second.

Full-in: A geometric object that lies fully inside the view frustum.

Full-out: A geometric object that lies fully outside the view frustum.

Fragment: A fragment is an OpenGL rasterized piece of geometry or image data that contains coordinate, color, and depth information.

Frustum: The perspective corrected view volume.

Frustum Culling: Removing all geometry that lies outside of the frustum.

Generation: All of the work done by an application prior to the point at which it is nearly ready to render.

Graphics Pipeline: The stages through which a primitive is operated on to transform it into an image.

Height Field: A mapping of a data value to a height that is relative to the image plane. One common mapping is to take a grid of elevation data (terrain) and map it to a trianglated surface.

Host: A synonym for CPU. See CPU.

Hysteresis: Minimizing the effect of a changing scene to keep a constant frame rate.

Impostor: A billboard with depth information.

Inlining: The technique of replacing the call to a function with an in-place copy of the functions contents.

Interprocedural Analysis: The process of rearranging code within one function based on knowledge of another function's code and structure.

LOD: See Level of Detail

Latency: A measure of the amount of time it takes to fully transfer a single unit of data to a device.

Level of Detail: Alternate representations of geometric objects in which successive levels have less geometric complexity.

Manifold Surface: A closed surface that can be topologically mapped to a sphere.

Microcode: Instructions that implement the instruction set of a processing unit. Typically composed of bit fields which control specific low-level processor operations. Several microcode instructions or microinstructions are required to decode and implement higher-level operations.

Native data formats: Data formatted in the same fashion that is used internally by a graphics subsystem. Pixels, vertices, normals, and a number of other basic data types have preferred, or native data formats. Example: AGBR may be native but RGBA may not.

Node: Description of a single computing element in a cluster or a single-system-image workstation. A computing element typically consists of at least one CPU, memory, and some I/O capability. In an SSI system, a node is typically a board within the system; in a cluster, a node is a single system within the cluster.

Occlusion Culling: Determination of the visible objects from the current viewpoint.

Page: A unit of virtual memory.

Paging: Copying data to and from one device to another, usually disk to memory.

Pipeline: See graphics pipeline.

PIO: Programmed I/O.

Pixel: A picture element. All the bits at location (x, y) in all the bitplanes of the framebuffer that form the single pixel (x, y). In OpenGL window coordinates, a pixel corresponds to a 1.0 x 1.0 screen area.

Polygon Rate: A measure of the speed at which polygons can be processed by the graphics pipeline. Polygon rates are reported as the number of triangles that can be drawn per second.

Primitive: Basic graphic input data such as triangles, triangle strips, pixmaps, points, and lines.

Profile: To measure quantitatively the performance of individual functions, components, or modules of an executing program.

Program Counter Profiling: Uses statistical callstack or program counter (PC) sampling to determine how many cycles or CPU time is spent in a line of code.

Programmed I/O: Transferring data from one device in a system to another by having the CPU read from the first and write to the second. See DMA for another approach.

Rasterization: Process that renders window-space primitives into a frame buffer.

SSI: Single-system-image. Refers to a type of multiple-graphics pipeline-based system that is running a single copy of an operating system.

Scene Graph: The data structure that holds the items that will be rendered.

Single-System-Image: A collection of graphics pipes within a system that produce a single computational or graphical result via traditional programming models.

Span: Segment of a scanline inside a polygon upon which a scanline algorithm operates to rasterize a primitive.

Stall: A condition where further progress cannot be made due to the unavailability of a required resource.

Static Scene: A scene that needs to be of high quality but not interactive.

Stress Factor: A computed value for a scene such that the further behind the scene gets from its target frame rate the higher the stress factor becomes.

Synchronous: The opposite of asynchronous. Synchronous function calls are those that do not return until they have finished performing whatever action is requested of them. For example, a synchronous texture download function waits until the texture has been completely downloaded before returning, while an asynchronous download function simply queues the texture for download and returns immediately.

Tearing: The effect that happens when a rendering is not synchronized to the monitor refresh rate in single buffered mode. Parts of more than one frame can be visible at one time, which gives a "tearing" look to a moving scene.

Transformation: Usually used as the process of multiplying a vertex by a matrix, thereby changing the location of the vertex in space.

Traversal: The portion of an application that walks through internal data structures to extract data and call specific graphics API calls (in OpenGL things such as glBegin(), glVertex3f(), and glEnable(foo)).

Virtual Memory: Addressing memory space that is larger than the physical memory on a system.

Word: The “natural” data size of a specific computer. 64-bit computers operate on 64-bit words, 32-bit computers operate on 32-bit words.

Bibliography

[1] CORBA Website. *http://www.corba.org*.

[2] GLperf Repository. *ftp://ftp.specbench.org/dist/gpc/opc/glperf/*.

[3] MPI Website. *http://www.mpi-forum.org*.

[4] OpenMP Website. *http://www.openmp.org*.

[5] Pankaj K. Agarwal and Subhash Suri. Surface approximation and geometric partitions. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 24–33, 1994. (Also available as Duke U. CS tech report, ftp://ftp.cs.duke.edu/dist/techreport/1994/1994-21.ps.Z).

[6] Drew Card and Jason L. Mitchell. Fast geometry processing with ati_vertex_array_object and ati_element_array. *http://www.ati.com*.

[7] Et al. Carolina Cruz-Neira. The cave: audio visual experience automatic virtual environment. *Communications of the ACM*, 35(6):64–72, 1992.

[8] Andrew Certain, Jovan Popović, Tony DeRose, Tom Duchamp, David Salesin, and Werner Stuetzle. Interactive multiresolution surface viewing. In *SIGGRAPH 96 Conference Proceedings*, pages 91–98. ACM SIGGRAPH, 1996.

[9] James H. Clark. Hierarchical geometric models for visible surface algorithms. *CACM*, 19(10):547–554, Oct. 1976.

[10] Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, and William Wright. Simplification envelopes. In *SIGGRAPH '96 Proc.*, pages 119–128, Aug. 1996. http://www.cs.unc.edu/∼geom/envelope.html.

[11] Keith Cok, Alan Commike, Bob Kuehne, Tom True, and Roger Corron. Developing Efficient Graphics Software. In *SIGGRAPH 99 and SIGGRAPH 2000 Conference Course Notes*. ACM SIGGRAPH, August 1999, 2000.

[12] Doug Cook. Performance implications of pointer aliasing. *SGI Tech Focus FAQ, http://www.sgi.com/tech/faq/audio/aliasing.html*, 1997.

[13] ATI Corporation. TRUEFORM. *http://www.ati.com/*, 2001.

[14] NVIDIA Corporation. OpenGL Extension Specification. *http://www.nvidia.com/developer*, 2001.

[15] Leila De Floriani and Enrico Puppo. A hierarchical triangle-based model for terrain description. In A. U. Frank et al., editors, *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*, pages 236–251, Berlin, 1992. Springer-Verlag.

[16] Kevin Dowd. *High Performance Computing*. O'Reilly & Associates, Inc., first edition, 1993.

[17] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. In *SIGGRAPH '95 Proc.*, pages 173–182. ACM, Aug. 1995. http://www.cs.washington.edu/homes/derose/grail/treasure_bags.html.

[18] Robert J. Fowler and James J. Little. Automatic extraction of irregular network digital terrain models. *Computer Graphics (SIGGRAPH '79 Proc.)*, 13(2):199–207, Aug. 1979.

[19] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics (SIGGRAPH '93 Proc.)*, 1993.

[20] Michael Garland and Paul S. Heckbert. Fast polygonal approximation of terrains and height fields. Technical report, CS Dept., Carnegie Mellon U., Sept. 1995. CMU-CS-95-181, http://www.cs.cmu.edu/∼garland/scape.

[21] Andrew S. Glassner. *Graphics Gems*. Academic Press, 1990.

[22] Anatole Gordon, Keith Cok, Paul Ho, John Rosasco, John Spitzer, Peter Shafton, Paula Womack, and Ian Williams. Optimizing OpenGL coding and performance. *Silicon Graphics Computer Systems Developer News*, pages 2–8, 1997.

[23] Ned Greene. Hierarchical polygon tiling with coverage masks. In *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 65–74. ACM SIGGRAPH, 1996.

[24] Stefan Gumhold and Wolfgang Straßer. Real time compression of triangle mesh connectivity. In *SIGGRAPH 98 Conference Proceedings*, pages 133–140. ACM SIGGRAPH, 1998.

[25] Evan Hart and Jason L. Mitchell. Hardware shading with ext_vertex_shader and ati_fragment_shader. *http://www.ati.com*.

[26] Paul S. Heckbert and Michael Garland. Multiresolution modeling for fast rendering. In *Proc. Graphics Interface '94*, pages 43–50, Banff, Canada, May 1994. Canadian Inf. Proc. Soc. http://www.cs.cmu.edu/∼ph.

[27] Paul S. Heckbert and Michael Garland. Survey of polygonal surface simplification algorithms. Technical report, CS Dept., Carnegie Mellon U., to appear. http://www.cs.cmu.edu/∼ph.

[28] Martin Heller. Triangulation algorithms for adaptive terrain modeling. In *Proc. 4th Intl. Symp. on Spatial Data Handling*, volume 1, pages 163–174, Zürich, 1990.

[29] Hugues Hoppe. Progressive meshes. In *SIGGRAPH '96 Proc.*, pages 99–108, Aug. 1996. http://research.microsoft.com/∼hoppe.

[30] Hugues Hoppe. View-dependent refinement of progressive meshes. In *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 189–198. ACM SIGGRAPH, 1997. http://research.microsoft.com/∼hoppe.

[31] Hugues Hoppe. Smooth view-dependent level-of-detail control and its applicatio to terrain rendering. In *IEEE Visualization '98*, pages 35–42, 1998. Available at http://research.microsoft.com/∼hoppe.

[32] Peter Hughes. Building a terrain renderer. *Computers in Physics*, pages 434–437, July/August 1991.

[33] Andrey Iones, Sergei Zhukov, and Anton Krupkin. On optimality of obbs for visibility tests for frustum culling, ray shooting and collision detection. In *Computer Graphics International 1998*. IEEE, 1998.

[34] Leif Kobbelt, Swen Campagna, Jens Vorsatz, and Hans-Peter Seidel. Interactive multi-resolution modeling of arbitrary meshes. In *SIGGRAPH 98 Conference Proceedings*, pages 105–113. ACM SIGGRAPH, 1998.

[35] Bob Kuehne. Displaying surface data with 1-d textures. *Silicon Graphics Computer Systems Developer News*, March/April 1997.

[36] Mark P. Kumler. An intensive comparison of triangulated irregular networks (TINs) and digital elevation models (DEMs). *Cartographica*, 31(2), Summer 1994. Monograph 45.

[37] Jay Lee. A drop heuristic conversion method for extracting irregular network for digital elevation models. In *GIS/LIS '89 Proc.*, volume 1, pages 30–39. American Congress on Surveying and Mapping, Nov. 1989.

[38] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *SIGGRAPH 01 Conference Proceedings*, Annual Conference Series, pages 149–158. ACM SIGGRAPH, 2001.

[39] Peter Lindstrom, Devid Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 109–118. ACM SIGGRAPH, 1996.

[40] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series. ACM SIGGRAPH, 1997.

[41] David P. Luebke, Martin Reddy, Benjamin A. Watson, Jonathan Cohen, Amitabh Varshney, and Robert E. Huebner. *Level of Detail for 3D Graphics: Application and Theory*. Morgan Kaufmann, 2002.

[42] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. In *1995 Symposium on Interactive 3D Graphics*, pages 95–102, 1995.

[43] Miles J. Murdocca and Vincent P. Heuring. *Principles Of Computer Architecture*. Addison-Wesley, 1998.

[44] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley, third edition, 1999.

[45] Shmuel Rippa. Adaptive approximation by piecewise linear polynomials on triangulations of subsets of scattered data. *SIAM J. Sci. Stat. Comput.*, 13(5):1123–1141, Sept. 1992.

[46] John Rohlf and James Helman. IRIS performer: A high performance multiprocessing toolkit for real-time 3d graphics. In *SIGGRAPH 94 Conference Proceedings*, Annual Conference Series, pages 381–394. ACM SIGGRAPH, 1994.

[47] Lori Scarlatos and Theo Pavlidis. Hierarchical triangulation using cartographic coherence. *CVGIP: Graphical Models and Image Processing*, 54(2):147–161, March 1992.

[48] Lori L. Scarlatos and Theo Pavlidis. Optimizing triangulations by curvature equalization. In *Proc. Visualization '92*, pages 333–339. IEEE Comput. Soc. Press, 1992.

[49] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH '92 Proc.)*, 26(2):65–70, July 1992.

[50] Jonathan Shade, Dani Lischinski, David H. Salesin, Tony DeRose, and John Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 75–82. ACM SIGGRAPH, 1996.

[51] François Sillion, George Drettakis, and Benoit Bodelet. Efficient impostor manipulation for real-time visualization of urban scenery. In *EUROGRAPHICS '97*, volume 16, 1997.

[52] David A. Southard. Piecewise planar surface models from sampled data. In N. M. Patrikalakis, editor, *Scientific Visualization of Physical Phenomena*, pages 667–680, Tokyo, 1991. Springer-Verlag.

[53] Gabriel Taubin, André Guéziec, William Horn, and Francis Lazarus. Progressive forest split compression. In *SIGGRAPH 98 Conference Proceedings*. ACM SIGGRAPH, 1998.

[54] David C. Taylor and William A. Barrett. An algorithm for continuous resolution polygonalizations of a discrete surface. In *Proc. Graphics Interface '94*, pages 33–42, Banff, Canada, May 1994. Canadian Inf. Proc. Soc.

[55] Seth Teller and Pat Hanrahan. Global visibility algorithms for illumination computations. In *SIGGRAPH 93 Conference Proceedings*, Annual Conference Series, pages 239–246. ACM SIGGRAPH, 1993.

[56] Greg Turk. Re-tiling polygonal surfaces. *Computer Graphics (SIGGRAPH '92 Proc.)*, 26(2):55–64, July 1992.

[57] Alex Vlachos, Jorg Peters, Chas Boyd, and Jason L. Mitchell. Curved pn triangls.

[58] Merriam Webster. *The Merriam Webster Dictionary*. Merriam Webster Mass Market, 1994.

[59] Hansong Zhang. *Effective Occlusion Culling for the Interactive Display of Arbitrary Models*. PhD thesis, The University of North Carolina at Chapel Hill, 1998. Also available at http://www.cs.unc.edu/∼zhangh/research.html.

[60] Hansong Zhang and Kenneth E. Hoff. Fast backface culling using normal mask. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, 1997.

[61] Hansong Zhang, Dinesh Manocha, Tom Hudson, and Kenneth E. Hoff. Visibility culling using hierarchical occlusion map. In *SIGGRAPH 96 Conference Proceedings*, 1997. Also available at http://www.cs.unc.edu/∼zhangh/research.html.