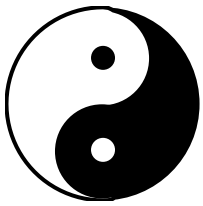
Developing Efficient Graphics Software:
The Yin and Yang of Graphics



A SIGGRAPH 2000 Course

Course Organizer

Keith Cok

SGI

Course Speakers

Keith Cok

Roger Corron

Bob Kuehne

Thomas True

SGI

Abstract

A common misconception in modern computing is that to make slow software work more quickly, you need a bigger and faster computer. This approach is expensive and often unworkable. 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 course discusses the techniques and principles involved in this approach to application development and optimization with particular emphasis on practical software development.

The course begins with a general discussion of the interaction between CPUs, bus, memory, and graphics subsystem, which provides the background necessary to understand the techniques of software optimization. After this discussion of architecture fundamentals, we present the methods used to detect performance bottlenecks and measure graphics and system performance. Next, we discuss some general optimization techniques for the C and C++ languages. Finally, we give an overview of current application-level architectures and algorithms for reducing graphics and general system overhead.

Preface

Course Schedule

1:30 PM	Introduction
1:35 PM	General Performance Overview
2:05 PM	Software and System Performance
2:55 PM	Break
3:10 PM	Profiling and Tuning Code
4:10 PM	Compiler and Language Considerations
4:55 PM	Graphics Techniques and Algorithms

About the Speakers

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

Keith Cok is a Member of Technical Staff at SGI. He currently works with software developers in 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, as an independent software developer, Keith wrote 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 received a BS in Mathematics from Calvin College, Michigan, and an MS in Computer Science from Purdue University.

Roger Corron
MTS SGI
1 Cabot Road, Hudson, MA 01749
rc@hudson.sgi.com

Roger Corron is a Member of Technical Staff in SGI Custom Engineering, where he develops custom system solutions. Previously, Roger worked in SGI Applications Engineering assisting software developers to optimize their application graphics performance. Previous to SGI, he worked at Matra Datavision optimizing and porting solid modelling software to several different graphical APIs. His interests include low level graphics APIs and extracting maximum performance from computer systems. Roger received his BS in Electrical and Computer Engineering from Clarkson University.

Bob Kuehne
MTS SGI
39001 West Twelve Mile Road, Farmington Hills, MI 48331
rpk@sgi.com

Bob Kuehne is a Member of Technical Staff at SGI. He currently assists software developers and vendors in the CAD/CAE industries in developing products to most effectively utilize the underlying hardware. His interests include object-oriented graphics toolkits, software design methodologies, creative use of graphics hardware, and human/computer interaction techniques. Prior to joining SGI, he worked for Deneb Robotics developing software for virtual reality applications. Bob received his BS and MS in Mechanical Engineering from Iowa State University and performed research on assembly techniques in virtual environments.

Thomas True
MTS SGI
2011 N. Shoreline Blvd., Mountain View, CA 94039
true@sgi.com

Thomas True is a Member of Technical Staff at SGI where he currently works assisting 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 a BS in Computer Science from the Rochester Institute of Technology and an MS in Computer Science from Brown University where 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. 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, David Campbell, Brian Thatch, Janet Matsuda, and Keith Seto, for giving us the opportunity to develop this course and the course reviewers who gave us much needed feedback.

We gratefully acknowledge Alan and Pam Thuman-Commike for their work with the proofreading, figures, and LaTeX formatting.

Course Resources On the Web

The course notes and slides are available on the SGI web site:

`http://www.sgi.com/events/siggraph00/gfxapps`

Contents

List of Figures

List of Tables

Section 1

Course Introduction

This course was developed for software graphics developers 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. In this course, you will learn how to

- Identify application and computer hardware interaction
- Introduce techniques to quantify and optimize that interaction
- Create and structure applications efficiently
- Balance the utilization of software and hardware system components

The course begins by discussing hardware systems, including CPU, bus, and memory. The course then covers graphics devices, theoretical and realized throughput, graphics hardware categorization, hardware bottlenecks, graphics performance characterization, and techniques to improve performance. Next, the course discusses application profile analysis, and compiler and language performance issues (for C and C++). The course then progresses into a discussion of application graphics rendering strategies, frameworks, and concepts for high-performance interactive applications.

This course is founded on the premise that creating high-performance graphics applications is a difficult problem that can be addressed through careful thought given to hardware and software systems interaction. The course presents a variety of techniques and methodologies for developing, analyzing, and optimizing graphics applications performance.

Section 2

Hardware Architecture and Performance

2.1 Computer System Hardware

This section describes application hardware interaction issues, specifically those encountered when writing graphics applications. The hardware upon which an application runs can vary dramatically from system to system and vendor to vendor. Understanding some of the design issues involved with hardware systems can improve overall performance and graphics performance through more effective use of hardware resources.

2.1.1 Overview

To understand why a graphics application is slow, you must first determine if the graphics are actually slow, or if the bottleneck lies elsewhere in the system. In either case, it's important to understand both the code and the system on which the code is running, how the two interact, and the strengths and weaknesses of the system.

In this section, hardware, software, and their interaction are discussed with a specific emphasis on graphics applications and graphics hardware. Also discussed is the process an application goes through to get data to the graphics hardware. Additionally, concepts for maximizing application performance are discussed throughout this section.

Bottlenecks and Yin & Yang

A key discussion throughout this portion of the course is that of *bottlenecks*. The word bottleneck refers to a point in an application that is the limiting factor in overall performance; that is, the point in an application that is the slowest and thus constrains its performance. A bottleneck in an application can be thought of much like its physical namesake, with the application trying to pour the bottle's contents (the application data and work) through the bottleneck (the slowest function, method, or subsystem). Improving performance for any application involves identifying and eliminating bottlenecks. Note, however, that once one bottleneck is removed, another often appears. For example, if in a room full of people, sorted by height, the tallest person is removed, there will still be another person remaining who is tallest. From this analogy, it would appear that an application developer's work is never done.

Fortunately, the goal in tuning an application is not to merely eliminate bottlenecks. The goal in tuning an application is to cause an application's work to be spread efficiently across all the component hardware and subsystems upon which it runs. A useful metaphor for this balance (and 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 map exactly to the main goal of application tuning, the basic concept of balance is key. If the re-purposing 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 will appear in the margin to denote a section of interest which 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.

2.1.2 Hardware Overview

Computer systems are constructed from a wide variety of components, each with their 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 is finished with 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 result of the query.

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 are used to mitigate the effects of these differences among devices. Subsequent sections discuss these issues in more detail.

2.1.3 Computer System

An abstract computer system is described in this section to motivate thinking about how a system is structured and how a program interacts with this hardware. This abstract computer system has components found on most computer systems, but it is not intended to represent any actual system. Any similarities to real computer systems, living or dead, is purely coincidental.

The architecture of a specific computer system is important to consider when designing software for that system. Specifically, it's important to consider which subsystems an application interacts with, and how that interaction occurs. There are several distinct systems on a computer, each using some interconnect fabric or "glue," (shown as a single block in Figure 2.1) to communicate with each other. Understanding this fabric and where devices live on this fabric is extremely important in determining where application bottlenecks occur, and avoiding bottlenecks when designing new software systems.

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

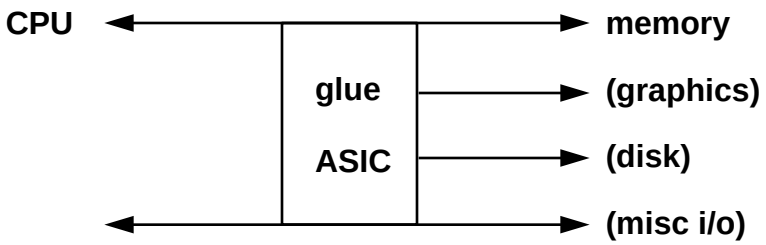


Figure 2.1: An abstract computer system fabric.

a point-to-point peering connection, which allows individual devices to communicate with preallocated guaranteed-bandwidth. In still other fabrics, some systems might live on a bus, while others in that system live on a peered 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, understanding the specifics of how graphics hardware interfaces with CPU, memory, and disk is of special importance. A diverse mix of computer systems exists on which an application might be run. This diversity ranges from systems with a shared-bus (PCI) with local texture and framebuffer, to systems with a dedicated bus to the graphics (AGP) with some local texture cache, some main memory texture cache, and local framebuffer, to systems on a dedicated bus with all texture and framebuffer allocated from main memory (SGI O2). Each of these architectures has certain advantages and disadvantages, but an application cannot be expected to fully realize the performance of these platforms without consideration of the differences among them.

As a concrete example of these differences, let's examine shared-bus systems. Graphics systems using a shared-bus architecture share bandwidth with other devices on that bus. This sharing impacts applications 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. (A complete description of the different types of graphics accelerator hardware strategies appears in later sections of the course notes.) Regardless of the type of the system being used, the key to high-performance applications is to fully utilize the entire system, balancing the workload among all the components needed so that more application work can be performed more quickly.

2.1.4 CPU

Figure 2.2 depicts a simplistic CPU to illustrate the lengthy path application data must travel before it is useful. In this figure, main memory lives on the far side of all 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 very good idea, as it can improve cache efficiencies dramatically. In fact, the premise behind caches is that data near the current data being operated upon is much more likely to be needed next. This design criterion means that memory locality affects performance, because access to cache memory is significantly faster than to main memory.

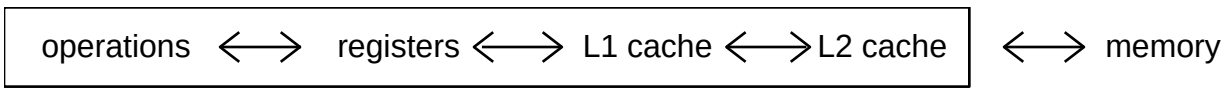


Figure 2.2: Abstract CPU.

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 (other graphics APIs have other nomenclature for this concept) can often be pre-compiled into a state such that a glCallList() simply transfers the display list directly from main memory to the graphics hardware, using a technique such as direct memory access. This technique allows large amounts of graphics data to be rendered without any complex calculations occurring on that data at run time.

2.1.5 Data Access Rates

Two other key measures of performance relevant to discussing computer hardware 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 have often 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 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 or not 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 shows where critical latencies are encountered. Latencies vary dramatically within a system. For example, network latencies can be on the order of many milliseconds (or even seconds), whereas latencies for data in L2 cache operate on the order of tens of nanoseconds (Figure 2.3).

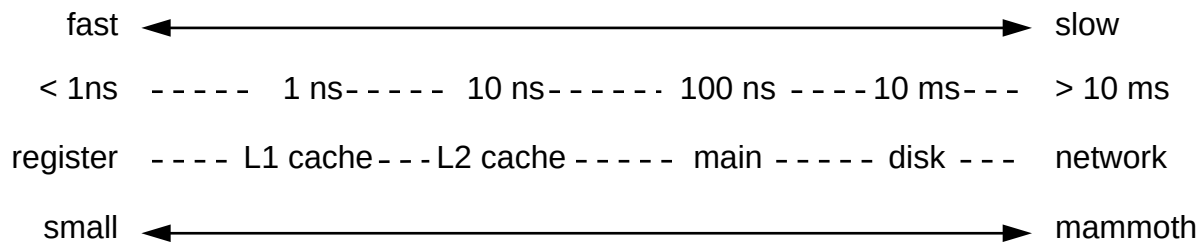


Figure 2.3: Approximate data latencies and capacities of typical system components.

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 approximately an order of magnitude slower than those of cache memories.

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) will then take 100s. As 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 1s, for a total of 100 seconds. Adding in the latency of 0.1 second per texture, we add a cumulative additional 10 seconds bringing the total transfer time up 10% to 110 seconds total. A developer aware of this issue could design methodologies such as creating a large texture with many small sub-textures within it to avoid many small data transfers that could negatively impact performance. Though contrived, this example illustrates that latency can be an issue affecting application performance, and that developers must be aware of hardware latencies so their effects (the latencies, not the developers) can be minimized.

2.1.6 Memory

Previous sections have described the effects of latencies and bandwidths on hypothetical activities. This section of the course discusses memory systems and how applications interact with data within memory. This section describes how these systems 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, etc.), and translation look-aside buffers.

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 sized significantly larger than the amount of physical RAM in a system. Addressing schemes vary, but 32-bit applications can typically address $>$1 GB of memory when only a small fraction of that is physically available. Virtual memory systems perform this task through managing a list of active memory segments known as *pages*. For details behind this operation, and that of many computer systems, see *Principles of Computer Architecture* [41] or a good introductory computer architecture book for elaboration.

Pages of memory are blocks of address space of a fixed size. The size of these blocks varies from system to system but is a constant on a specific running system. However, many hardware systems allow the page size to be changed, and some operating systems allow this to be changed 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, as will be explained more fully in a moment. Typical pages range between 1-32k in size. 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 from that application from main memory to disk. This process is known as *paging*. Each time a page of memory is requested, the memory manager checks if it already exists in main memory. If it does, no action is required, if it does not, the memory manager checks if there is space available in RAM for that page. If space is available in RAM for the needed page, no action is required, but if not, a page of resident data must be put to disk prior to writing the desired page from disk. Next, in all cases, the desired page is copied from disk, to the available page location in RAM. When an application pages, disk I/O occurs, thus impacting both the application and overall system performance. As maintaining the integrity of a running application is essential, the paging process operates in a fairly resource-intensive fashion 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. The 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 ranging from page 0 to page *n*, and a system with many physical pages of RAM. In this example application, pages 0 - 9 are active, or have data of some sort stored in them by the application, and pages 0 and 1 are physically resident in RAM. For this example, the memory manager has decreed that only two pages can be used by the application, so any application data that resides on pages other than the two in RAM are paged to and from disk.

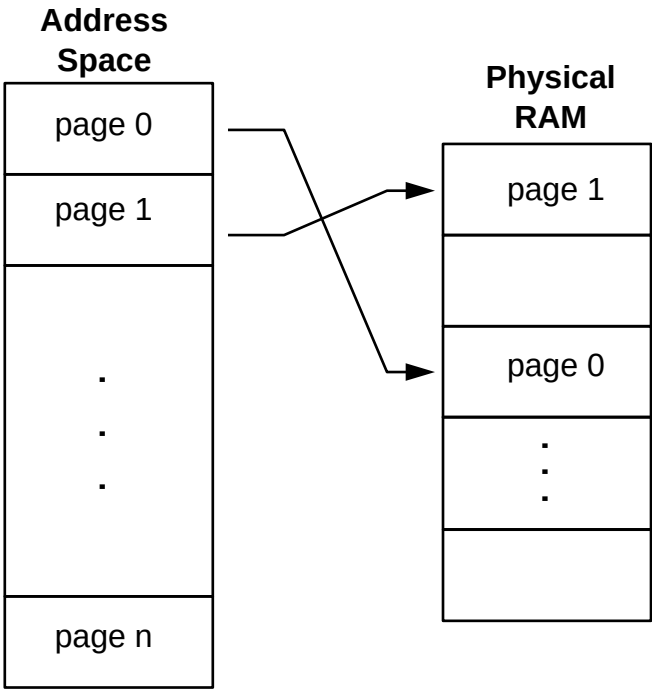


Figure 2.4: Virtual memory mapping active pages into RAM.

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 will likely require eight paging operations, which can be quite expensive, given that disk access is several orders of magnitude slower than RAM access. However, if the application could rearrange data such that it all resided on one page, no paging by the virtual memory manager would be required, and access times for this data would improve dramatically.

When data is resident on pages in main memory, it must then be transferred to the CPU (see Figure 2.2) in order 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 ultimately 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 32-128 bytes in length. As data is required by the CPU, data from L2 cache must be copied into a faster level-1 (L1) cache, again of a system-dependent size, typically of around 32 bytes. Finally, the actual data required from within the L1 cache is copied into the registers, and operated upon by the CPU. This is the physical path through which data must flow to be able to be operated on by the CPU.

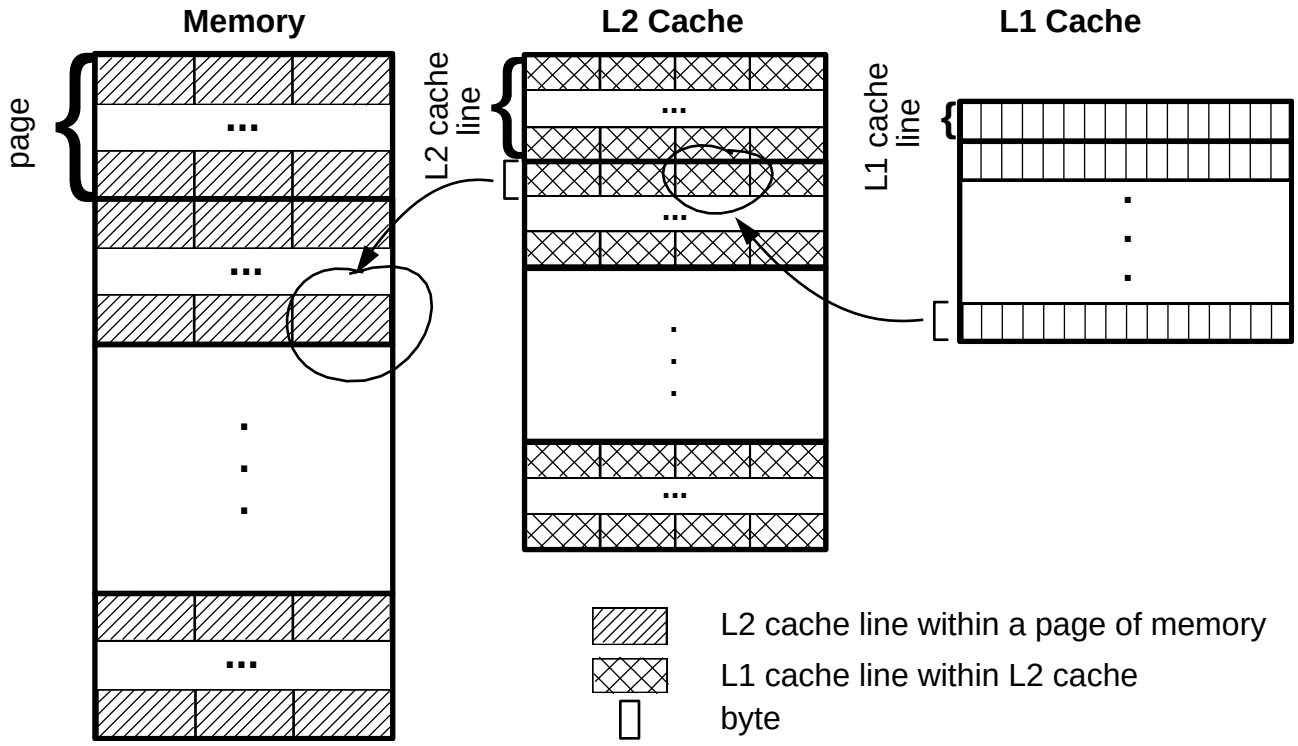


Figure 2.5: Cache line structure. Shown are pages of memory composed of multiple L2 cache lines; L2 cache composed of multiple L1 cache lines; and L1 cache composed of individual bytes.

The process by which requested data is copied into the registers will now be explained. This process is important because the consequences of its action are one of the primary factors limiting 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 again not available, the controller checks the L2 cache. Finally, if the data is still not available, a cache line containing 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 propagated through L2 and L1 cache, ultimately depositing the requested data in a register. This process is visually depicted in Figure 2.6, which shows the data request procedure as a flow chart.

Though this discussion of memory and how it works is straightforward, the relevance to application performance may not be immediately clear. Data locality, or packing frequently used data near other frequently used data in memory, is the ultimate point of any discussion of how memory works. Keeping data closer together keeps data in faster and faster memories. Conversely, data which is widely dispersed in memory is accessed by slower and slower means. The effects of data locality are best demonstrated through two examples.

In these examples, an operation is being performed in the CPU 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 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

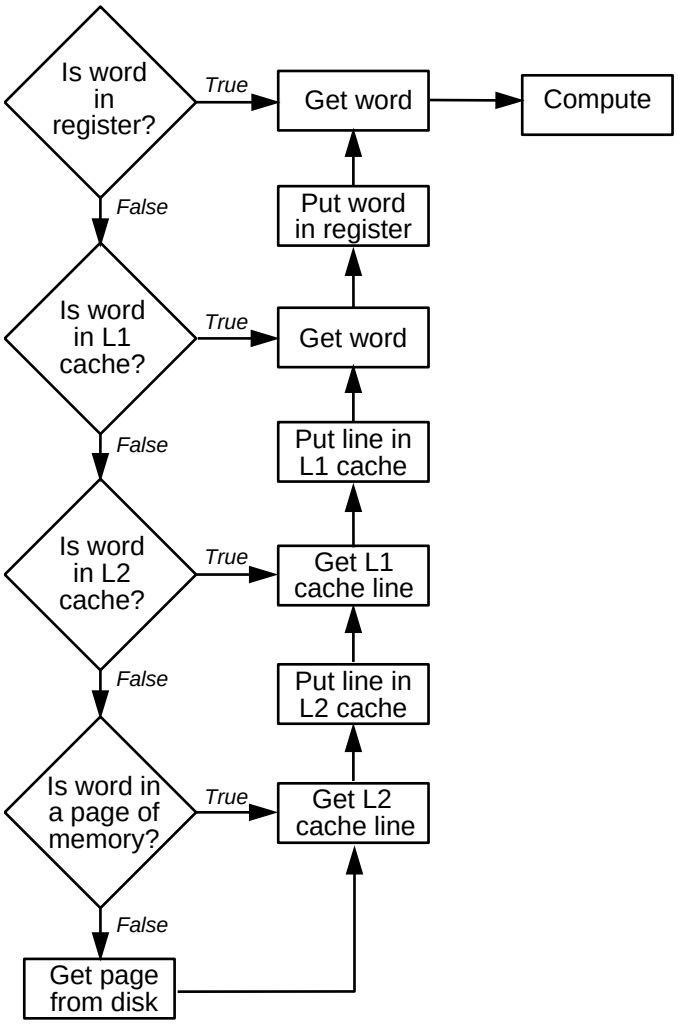


Figure 2.6: Register data request flowchart.

time to prepare memory for the operation to occur is 222 ns.

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). A much smaller time to set up this operation is needed than in the first example. Again, it takes 100 ns to access the main memory page to copy to L2 cache, 10 ns to access the L2 cache twice to copy each byte to a different L1 cache line, two 1-ns accesses to the L1 cache to load the registers. In this example, the total time to prepare the operation is 122 ns, which is nearly half the previous example's overall time. As these examples show, keeping data localized can clearly benefit application performance. Keep in mind these cache effect when designing graphics data structures to hold objects to be rendered, state information, visibility lists, etc. Some simple changes in data structure organization can possibly gain a few frames per second in the application frame rate.

Another example of how data locality can be of great advantage to a graphics application is through a graphics construct known as a *vertex array*. Vertex arrays allow graphics data to be efficiently utilized by the CPU for transformation, lighting, etc. This efficiency is primarily due to the fact that vertex arrays are arranged contiguously in memory, and 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, allowing fast access to each of them. However, because most applications do more than render flat-shaded triangles, these vertices will 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's 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. (Details are beyond this paper's scope. See a computer architecture book for more details.) A solution to this problem is the concept of interleaved vertex arrays. 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 quite likely to live in non-overlapping 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 get to the CPU is key because of the latencies involved in accessing data from various memory caches. Keeping data packed close together in memory ensures the likelihood of subsequent data accesses occurring from memory already resident in cache, and, therefore, the algorithms operating on that data will be much faster.

2.1.7 Graphics Hardware

The graphics subsystem is responsible for the actual rendering and display of 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 rendering pipeline, describes how special-purpose dedicated hardware can be used to implement it, and the relative impact different hardware implementations have on overall application performance.

Graphics Pipeline

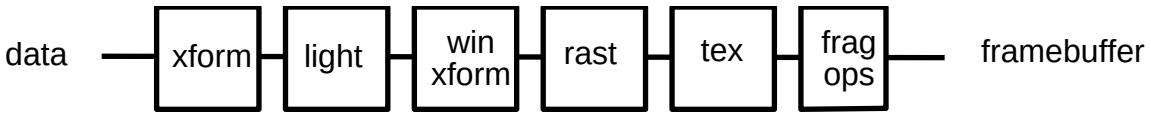
The process of rendering interactive graphics can best be described as a series of distinct operations performed on a set of input data. This data, often referred to as a *primitive*, typically takes the form of triangles, triangle strips, pixmaps, points, and lines. Each primitive enters the process as a set of user vertex data in a world coordinate system, and leaves as a set of pixels in the framebuffer. This set of stages, which performs this transformation, is known collectively as the *graphics pipeline*(Figure 2.7).

When implemented using special-purpose dedicated hardware, the graphics pipeline can conceptually be reduced to five basic stages[8].

Generation The process of creating the actual graphics data to be rendered, and organizing it into a graphics data structure. Generation includes all the work done by an application on the CPU prior to the point at which it's ready to render.

Traversal The process of walking through the internal graphics data structures and passing the appropriate data to the graphics API.

Typically, this stage of the rendering process is not implemented in dedicated hardware. Immediate mode graphics requires flexible traversal algorithms that are much easier to perform in software on the host CPU. Retained mode graphics, such as OpenGL display-lists, can be implemented in hardware and then are part of the traversal phase.



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.

Transformation The process of mapping graphics primitives from world-space coordinates into eye-space, performing lighting and shading calculations, mapping the eye-space coordinates to clip-space, clipping these coordinates, and projecting the final result into screen-space.

Graphics subsystems with hardware support for this stage do not always accelerate all geometric operations. Often, there is a limited number of paths that are fully implemented in hardware. For example, some machines may only accelerate geometric operations involving one infinite light, others may not accelerate lights at all. Some hardware accelerators may have dedicated ASICs that 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. These operations are known as *fast paths.* Determination of hardware fast paths is discussed in Sections 2.2.3 and 4.4.1.

Rasterization The process of drawing the screen-space primitives into the framebuffer, performing screen-space shading, and per-pixel operations. Per-pixel operations performed in this phase include texture lookups and depth, alpha, and stencil tests. Following this stage in the pipeline, there remain only fragments, or pixels with a variety of associated data such as depth, color, alpha, and texture.

Any (or all) rasterization operations can be incorporated into hardware, but very frequently, 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), while others may not accelerate texture at all, targeting instead markets such as CAD where texture is (as of yet, relatively) unimportant. It is important to know what is and is not implemented in hardware to construct a well performing graphics application.

Display The scanning process that transfers pixel data in the framebuffer to the display monitor.

This stage is always implemented in dedicated hardware, which provides a constant refresh rate (for example, 60 Hz, 75 Hz).

Figure 2.8 shows how these five stages overlay onto the original graphics pipeline. These five stages can be used to build a useful taxonomy that classifies graphics subsystems according to hardware implementation. This taxonomy, its mapping onto hardware, and consequent performance implications, are the subject of the next section.

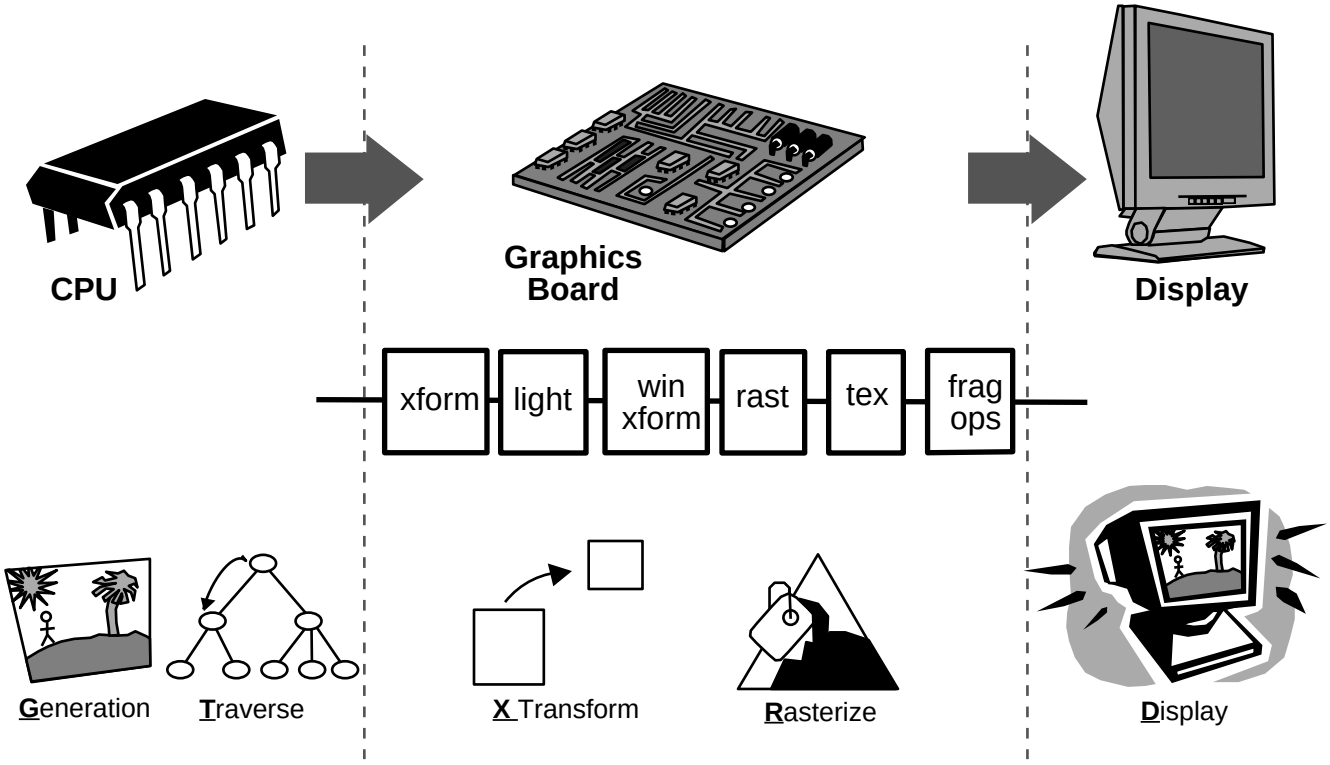


Figure 2.8: Graphics hardware pipeline and taxonomy.

2.1.8 Graphics Hardware Taxonomy

Graphics subsystems can be classified as one of four different types. Each type can be named using **G**, **T**, **X**, **R** and **D** to represent the five stages of the graphics rendering pipeline. A dash represents the division between those stages performed in dedicated graphics hardware and those stages performed in software on the host CPU. The following classification scheme is used in subsequent sections to describe how different hardware implementations impact rendering performance.

GTXR-D The sole function of a GTXR-D type graphics subsystem is to regularly update the screen at the set refresh frequency by scanning the pixel values from video memory to a display monitor. All other rendering stages are performed on the host CPU.

GTX-RD GTX-RD type graphics subsystems have a rendering engine that implements the scan conversion of screen-space objects (points, line, and polygons) into video memory and performs screen-space shading and other pixel operations (depth testing, stencil testing, etc.). Transformation and lighting are still performed on the host CPU.

GT-XRD GT-XRD type graphics subsystems go one step beyond GTX-RD with the addition of one or more transform engines that implement in hardware the transformation from object-space to eye-space, eye-space lighting and shading, and the subsequent transformation to screen-space. In this case, the CPU is left to simply generate and traverse the graphics data structures sending the object-space data to the graphics subsystem.

G-TXRD Graphics subsystems of type G-TXRD are rare because of the overwhelming demand for immediate mode graphics. Moving the traversal stage from the host CPU into dedicated hardware

imposes strict rules on user interaction, which is unacceptable in most environments. Because there are very few such systems, we will not discuss them further here.

Maximizing application performance on a particular type of graphics subsystem requires first an understanding of which portions of the graphics pipeline are used by an application, and second, which portions of the pipeline are implemented in dedicated graphics hardware. Keep both points in mind when authoring an application.

2.1.9 Bandwidth Limitations

Another important aspect of the graphics subsystem is the physical connection or fabric connecting it, 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 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 transferred between system memory and dedicated memory in the graphics subsystem must pass through the CPU, thereby increasing the requirements on the CPU and 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 using 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, there is a dedicated bus that handles the flow of data between the CPU and graphics. Current UMA machines offer a bandwidth of 3.2 GB/s.

A comparison of the various architectures can be seen in Figure 2.9. An analysis of how different graphics hardware implementations affect overall application performance can be found in Section 4.3.4.

2.1.10 Larger Computing Architectures

The previous sections described the architecture of small, single graphics-adaptor (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 cost of these systems drop, systems (either single-system-image (SSI) or cluster-of-workstations (COW)) with multiple graphics pipelines (or pipes) are increasingly common. Systems of this magnitude 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 [10]. A more general way to think of the problem is that the entire final graphics image is subdivided and farmed out to individual pipes, then recombined in the final display, independently of how that final display device is configured.

Why would an application use an SSI or COW system? An application should use an SSI or a COW system to display data sets much more complex than those displayed on a single pipe system. Through use of multiple pipelines, aggregate system performance is improved to a point where the problem can be

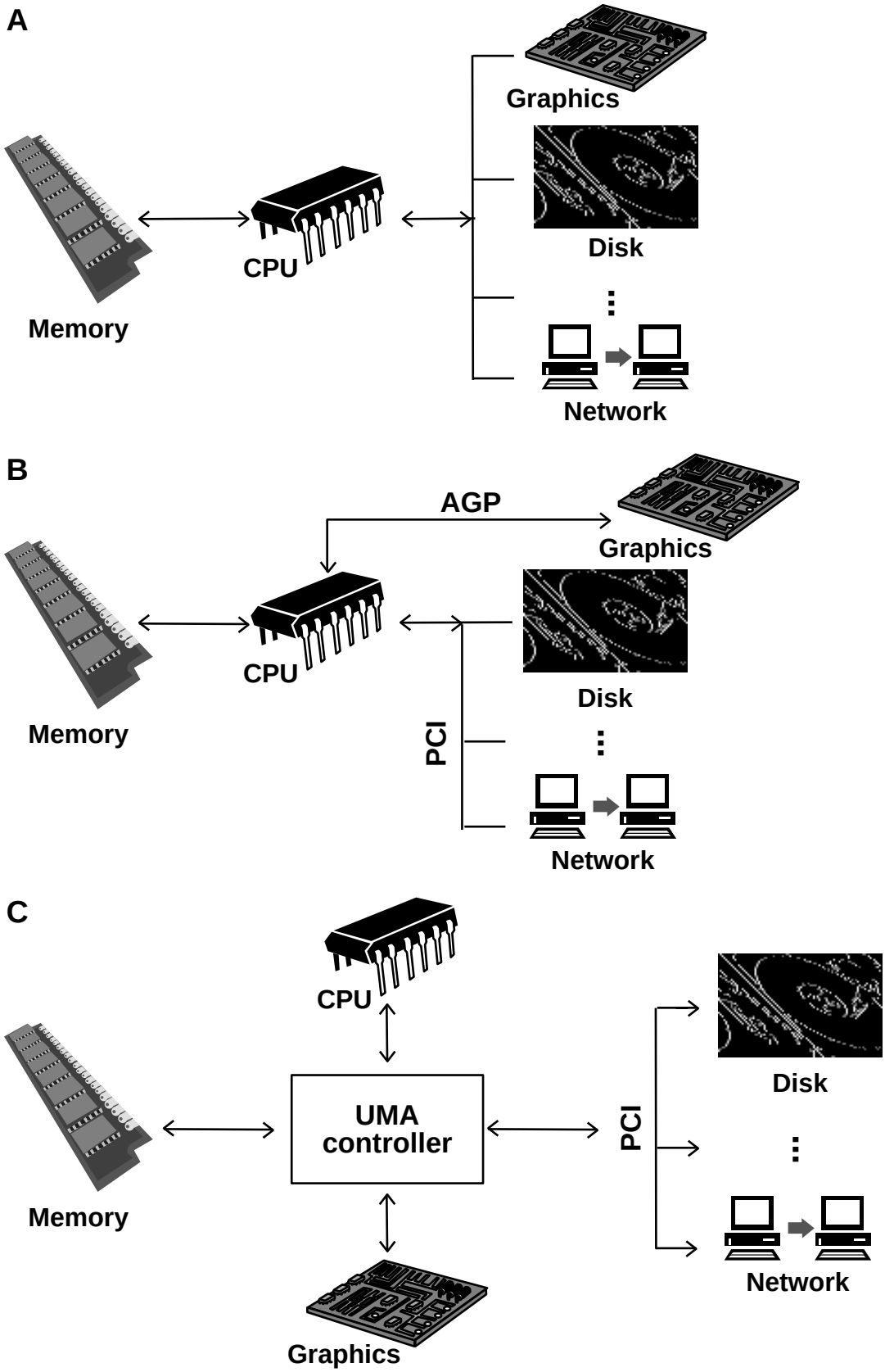


Figure 2.9: Schematic of system interconnection architectures: (A) PCI, (B) AGP, and (C) UMA.

interactively visualized. Though individual graphics adaptors are available on the desktop today which are more powerful in certain capacities than systems of a few years ago costing orders of magnitude more, users of systems always want to display more. Regardless of how fast individual systems perform, a combination of multiple graphics adaptors together will always be able to attack larger problems. Said differently, problem size is always increasing, and the path to attacking the largest problems today and tomorrow is to enlist multiple graphics adaptors to the same task.

Both SSI and COW systems address a similar problem, and in essence raise the system interface (PCI vs. AGP, etc) issues 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 will address these and other issues in large-system interactive graphics application utilization.

Single-System with Multiple Pipes

The defining features of SSI architectures are multiple graphics pipes, yet a single set of system resources, including memory, CPU, etc, all communicating 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 adaptor, for example, by specifying a specific display and screen for X-windows applications. Figure 2.11 shows how a system might be configured. In this diagram, the system consists of four dedicated rendering pipelines (indicated by monitors) which render, transfer internally across the system's bus or hub, and then are recombined on a fifth pipe. Processes are threaded (or even forked) across multiple CPUs, and functions 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 using shared-memory arenas. In both cases, explicit or implicit memory sharing, the data resides within a single logical memory subsystem, allowing easy and direct access to data across multiple process and threads.

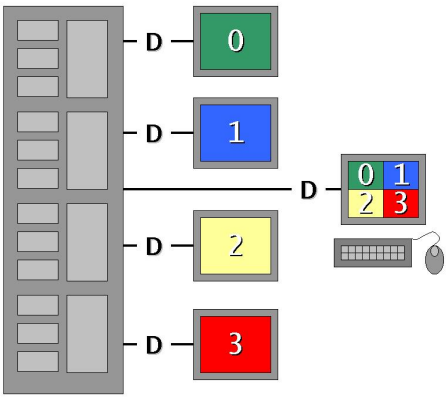


Figure 2.10: Example SSI system configuration.

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

Cluster of Workstation Pipes

The defining features of COW systems typically, 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, lower-quality graphics. In COW systems, applications must be explicitly aware of the differences among individual systems, or nodes, in the cluster, as well as the individual system capabilities, and the link performance and topology of the system connections. An example of one configuration is found in Figure 2.11. Programming interfaces are explicitly parallel, or happen through an abstraction layer such as a message-passing interface. Examples of these interfaces include OpenMP [6] and MPI [4], although many others exist. One example of other techniques include distributing objects using an object layer such as CORBA [1]. Link connection and topology are key factors in constructing and using a cluster in both determining the amount of data which can be distributed to all nodes (within the application per-frame time constraints) as well determining the latency involved (through number-of-hops in the topology) in transferring that data.

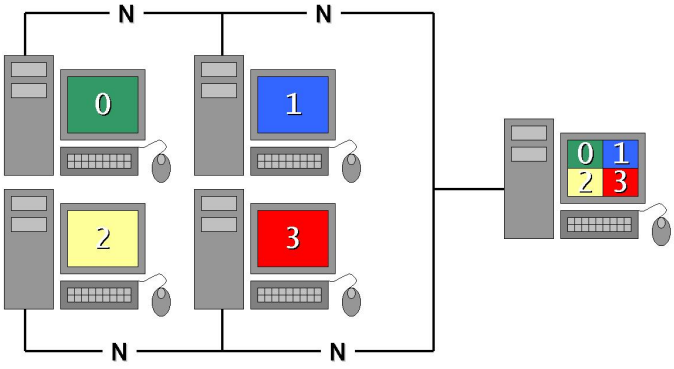


Figure 2.11: Example COW system configuration.

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 all are key to ultimately achieving good results.

First, it's necessary to understand and choose an appropriate problem decomposition. Among the choices for problem decomposition include image-space, time-based, geometry-based, and depth-based. Each of these techniques involve understanding the desired output display configuration and configuring the set of image pipelines to produce those images.

In image-space decomposition, the configuration may consist of either a single large image subdivided into a set of smaller sub-images, or a set of abutting images, perhaps not even in the same plane, such as in a CAVE. A second decomposition is geometry-based. In this configuration, each pipe views the entire view-volume, and each pipe receieves some portion of the geometry to render (a naiive approach might simply be to send each pipe 1/(number-of-pipes) objects). These resultant images are then recombined along with their depth values on the final pipeline. Another decomposition, often used in simulators, is time-based. In this tiling, 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, etc.

Yet another decomposition is depth-space tiling. In this decomposition, each pipe will handle the same screen-space area, but each will render a specific depth-section of the database (which itself is sorted by depth.) For example, on a 3-pipe system, each pipe would create a view frustum of 1/3 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 his own piece of the entire geometry, and the rendered sets of geometry are combined into the final image.

Keep in mind that several of these techinques can be used in conjunction, if enough resources are available. For example, an application might use a time-based decomposition, and then for each frame within that time-buffer, subdivide those frames spatially. Decomposition combinations such as these are extraordinarily powerful but require a significant investment in software architecture to utilize multi-pipe systems effectively.

Image-space decomposition

```
/* 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_pipe();

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.
 * this can be optimized to allow individual pipes to proceed
 * when finished. */
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 */
```

Depth-based Decomposition

```
/* each pipe gets a section of the depth-space view-volume. sort
 * so only that section of data goes to each 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 ); */

  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.
 * this can be optimized to allow individual pipes to proceed
 * when finished. */
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 */
```

Geometry-based 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.
 * this can be optimized to allow individual pipes to proceed
 * when finished. */
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 */
```

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 );
}
```

Whatever the technique chosen, the resultant images are rendered each 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 which physically abut each other, creating the illusion of a seamless image. The second, more challenging technique, involves rendering images on a number of pipes, then capturing those pixels (and potentially depth information) and sending them back across to the final graphics pipe, where they are recomposited together, then sent to the display device. Examples of all these techniques are shown in Figure 2.12.

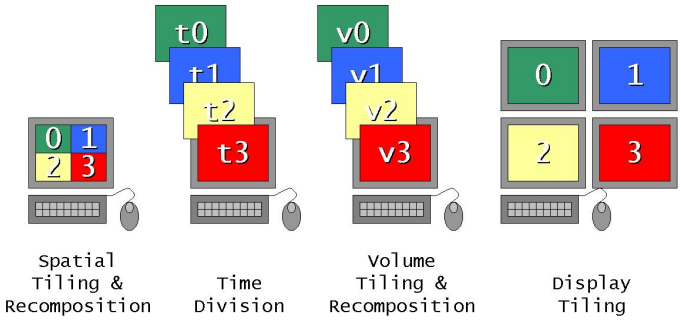


Figure 2.12: Example tiling techniques for scaling graphics performance.

The above description brings up the second and third key points to effectively utilizing a multi-pipe

system. The second key factor is to understand the system architecture. In some architectures, bandwidth may not be available to pass image sections back to a final pipe for recompositing. 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 on the order of several milliseconds, but in a SSI system, latencies may be on the order of 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, again 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 a SSI system, where latencies are lower, it's much more feasible to transmit portions of the resultant image to a single pipe for recompositing.

The last point in the previous discussion hints at the third key point in using a multi-pipe architecture. This last point is to understand the differences among individual systems, or nodes, within a system. When doing depth-space compositing, or large image reconstruction, doing this final step on a system with additional resources makes obvious sense, as the pixel demands on this system will be larger. In a more general sense, balancing the load among systems in either a COW or SSI system is absolutely essential to maximizing 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.

2.2 Graphics Hardware Specifications

2.2.1 Graphics Performance Overview

Graphics hardware vendors typically list several gross measurements of system performance when releasing new graphics hardware. Many of these measurements are benchmark data showing how hardware performs with a real set of data. Often these figures are not representative of how an application performs. Vendors typically list even more coarse measurements for their graphics hardware such as fill rate and polygon rate. Most, if not all, of these numbers should be viewed with a fair degree of skepticism and then independently verified.

2.2.2 Graphics Performance Terms

Fill rate is a measure of the speed at which primitives are converted to *fragments* and drawn into the framebuffer. Fragments are pixels in the framebuffer with color, alpha, depth and other data, not just the raw color data that appears in an image. Fill rates are reported as the number of pixels able to be drawn per second. This number is virtually meaningless without additional information about the type of pixels (and more correctly, type of fragments) that are involved in the measurement. Read literature carefully for additional information about the tests including what bit-depths each of the fragments used (32-bit RGBA, 8-bit RGB, etc.), whether or not the fragments were textured, what type of texture interpolation was used, and so on.

Fill rate consists of more than simply the number of fragments drawn to the framebuffer and transferred to the screen. While drawing 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 will be drawn completely, overwriting some of the farther back polygons. Pixels in which the two polygons intersect will have been 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. Applications with high depth-complexity are often fill-limited. Fill rate is often a bottleneck for application domains such as flight simulation.

Polygon rate is a measure of the speed at which polygons can be processed by the graphics pipeline. Polygon rates are reported as the number of triangles able to be drawn per second. Polygon rates, like fill rates, are almost meaningless without additional supporting information. Check hardware information carefully for specifics about whether or not these triangles were lit or unlit, textured or untextured, the pixel size of the triangles, etc. Polygon rates are often bottlenecks in application domains such as CAD and manufacturing simulation.

Fill rates directly correspond to the rasterization phase of the pipeline referred to in Figure 2.8, whereas polygon rates directly correspond to the transformation phase of the pipeline. Because these are the two phases of the pipeline most commonly implemented in hardware, achieving a good balance between these two is essential to good application performance. For example, if a graphics vendor claims a high polygon rate, but low fill rate, the card will be of little use in the flight simulation space because of the type of graphics data typically used in flight simulations. Flight simulations typically draw relatively few polygons, but most are large, and are textured and fogged, and often overlap (think of trees in front of buildings in front of layers of ground terrain), thus having high depth-complexity. In another example, if graphics hardware claims a high fill rate, but low polygon 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 more fully utilized, more complex or more 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 or not an application will be able to perform up to these published standards. However, even armed with all this information, there are still many variables on which hardware vendors do not provide data, which affect an application's performance. Ultimately, to measure the real performance of a system's graphics, you must test the hardware.

2.2.3 Graphics Performance Techniques

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 similar to that used in the vendor's tests, or you can use a focused test application such as GLPerf [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 can potentially modify the measured applications behavior. Kill all unnecessary services/daemons before benchmarking. A second issue to be aware of when benchmarking is that of /em frame quantization. Frame quantization is the characteristic of a graphics system to only swapbuffers (draw the back-buffer to the visible screen area) at the next vertical retrace interval. This implies that for benchmarking, single-buffer mode should be used, as single-buffer rendering does not wait for the next vertical refresh before swapping buffers. Though single-buffered rendering introduces a visual artifact known as *tearing*, the application will be drawing as fast as possible, ensuring accurate frame-rate measurement.

There are several design parameters to keep in mind when writing 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 (is lighting enabled?, is depth buffering enabled?, etc). 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 state as possible, paying particular attention to the state 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 being accurately measured, and not setup/shutdown costs associated with each frame of drawing.

Another test technique is to use the application GLPerf, available through the OPC web site [2]. GLPerf is designed to allow testing of most of the OpenGL pipeline within a simple script-based test framework. GLPerf scripts can test a variety of parameters in many combinations, thus providing an automated way of gathering performance data across a set of rendering conditions.

Upon testing graphics hardware with either a test framework, GLPerf, or some other tool, performance may still not be as high as expected from the graphics hardware. Many hardware accelerators are only "fast" when using very specific data formats or state settings. Try changing vertex data formats to vertex arrays, compiled arrays, tristrips, quadstrips, etc. Change pixel format data among RGBA, RGB, AGBR, BGR, etc. Change light types from directional to local, change lighting modes, change texture modes, etc. Vary all the important parameters in an application space to see which yield both the highest performance and desired quality for that application.

Once application performance on specific graphics hardware has been characterized and hardware bottlenecks eliminated through profiling, analysis, and redesign, how can rated graphics hardware performance be realized for an application? Unfortunately, it's 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 differences among hardware platforms, steps can be taken in the design and implementation of graphics applications to mitigate these differences.

2.3 Hardware Conclusion

There are only a few simple steps to ensuring quality application performance.

- Know the target computing platform.
  - Understand its capabilities for data I/O, cache and CPU architecture, and primarily the data paths to and through the graphics hardware.
- Learn the fast paths within a computing system.
  - Read the documentation provided by hardware vendors.
  - Contact developer representatives of these hardware vendors for further information.
  - Write test cases that tax the specific paths important to the application.
  - Provide feedback to hardware vendors.

- Ensure maximum performance of a computing system by using provided hardware features and extensions. Graphics system performance can be most dramatically affected through use of vendor-provided extensions.
  - Use run-time queries to determine which extensions are available and then use them fully.
  - Use both the fill and transformation portions of the graphics hardware to maximize use of the available resources.
  - Balance the workload among all system components.

Though hardware systems are continually improving in performance, features, and overall capability, applications and users of those applications are similarly increasing their demands and workloads. It is essential for application writers to understand the capabilities of their target hardware platforms in order to meet these demands. Understanding these capabilities and writing software with hardware in mind will afford the best possible performance across a wide variety of computing architectures.

Section 3

Graphics Hardware Pipeline

3.1 Introduction

To achieve the highest performance from a computer graphics application a delicate balance must be maintained between the requirements of the software application and the capabilties of the system hardware. An out-of-balance system does not perform optimally.

In order to achieve this balance, an understanding of the underlying system hardware is required, specifically the hardware which implements the graphics rendering pipeline. This has become even more important in recent years as graphics architectures continue to evolve at a rapid pace with different parts of the graphics rendering pipeline implemented in dedicated hardware within the graphics subsystem.

Equally important is an understanding of the software side of the equation, not only the application software and the requirements that it places on the computer system hardware but also any software which complements the graphics hardware to implement the complete graphics rendering pipeline.

This section will build upon Section 2.1.7 which introduced the 5 basic stages of the graphics rendering pipeline. Here, we will examine in detail **T**ransformation and **R**asterization. These stages form the core of the pipeline and are typically implemented using special purpose hardware within the graphics subsystem. Breaking each of these stages into function blocks produces the more detailed diagram in Figure 3.1. It is at this level that we will examine the functionality of the graphics rendering pipeline and how it is implemented as a combination of hardware and software.

This section will serve as an introduction to the details of the graphics rendering pipeline. However, this section is by no means meant as a complete guide to the graphics rendering pipeline, but simply a collection of useful information to be considered by the application developer when evaluating the performance of a graphics application. For more specific information on the graphics rendering pipeline see more detailed reference documentation [42]. To see how this pipeline is actually be implemented in software, check out the OpenGL^{TM} Sample Implementation [5] or Mesa^{TM} [3].

Except in the case of extremely low-end graphics adapters which implement most of the graphics rendering pipeline in software, the graphics rendering pipeline is typically implemented in hardware by one or more special purpose ASICs. The first ASIC or group of ASICs is commonly referred to as the rendering or rasterization engine. This chip excels at 2D screen coordinate integer-based computations and as such implements the functionality of the **R**asterization stage. This chip executes *microcode* rendering instructions on 2D primitives to produce pixel fragments. Mid to high-level graphics accelerator cards feature an additional ASIC or set of ASICS for geometry processing, commonly referred to as a transform engine. The transform engine is like a special purpose CPU that executes microcode instructions to efficiently perform the floating-point calculations required in matrix multiplication and the evaluation of

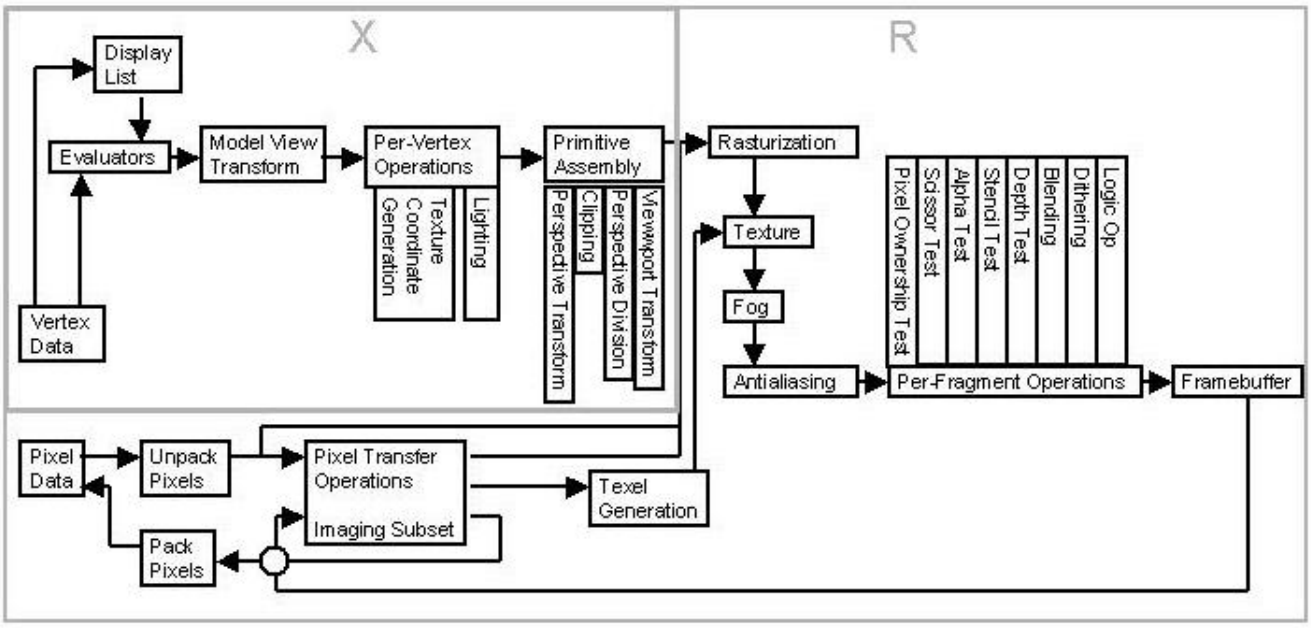


Figure 3.1: Graphics Pipeline.

plane equations, the **T**ransform functionality within the graphics rendering pipeline.

The rendering process follows a set of paths through the rendering pipeline. This set of paths is composed of a single geometry path and multiple imaging paths. Understanding how each path is implemented and functions within the graphics pipeline will provide insight for application design later performance tuning.

3.2 Geometry Path

In the geometry the graphics rendering pipeline operates on vertices, colors, and normals to produce pixel values that are then drawn into the framebuffer. The general data flow in the geometry path is from memory, through the CPU, through the graphics subsystem, and into the framebuffer. The amount of involvement of the CPU typically depends upon the capabilties of the graphics hardware subsystem and is typically inversely proportional to the hardware functionality of the graphics subsystem. In other words, the more work performed by the graphic subsystem, the less work performed by the host CPU.

A variation of this basic data flow occurs in the case when display lists are utilized. Some graphics hardware has the capability to store display lists in an optimized form in dedicated memory such that the vertex data no longer needs to be retreived from system memory each time it is rendered.

Figure 3.2 illustrates the basic path that geometry takes through the rendering pipeline. In this case, 3D vertex data is rendered either directly or from a display list into the framebuffer. The geometry passes through a series of specific stages each of which performs an operation on the data before passing it to the next stage. Geometry data can also be only 2D, but the data flow and operations remain basically the same as the 3D case. Each of the stages in the geometry path is described below.

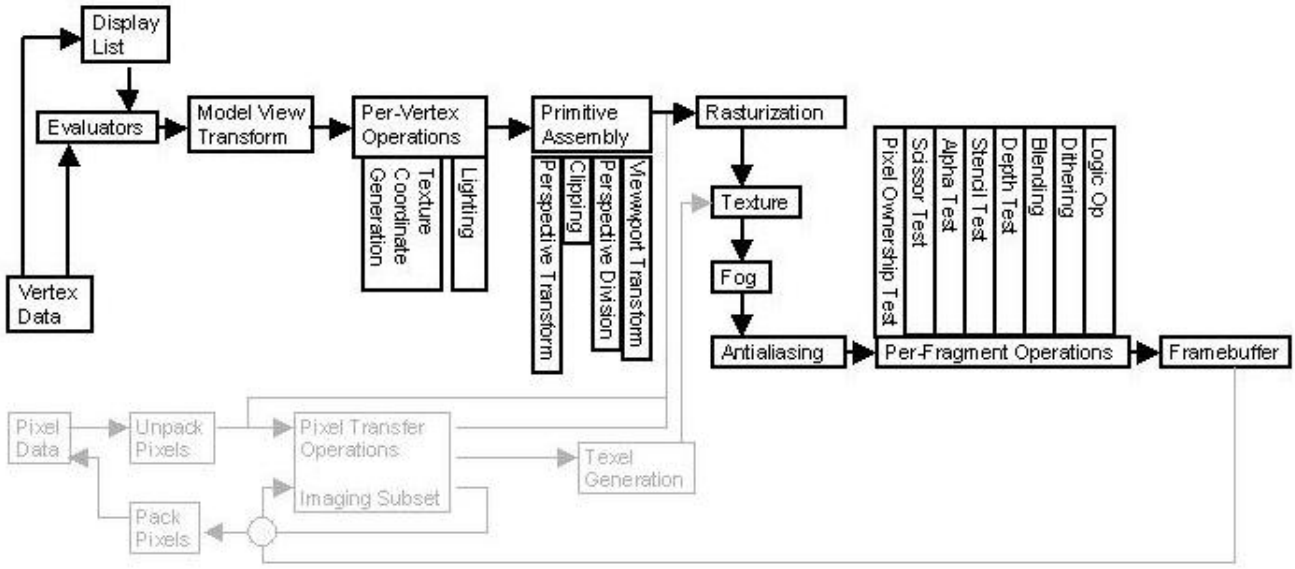


Figure 3.2: 3D Path no Texture.

3.2.1 Model-View Transform

The model-view transform maps the vertex coordinates for each graphics primitive from object space into eye space by multiplying the x_o, y_o, z_o and w_o coordinates for each vertex by the 4 x 4 model-view matrix M. Surface normals are also transformed by the model-view matrix to preserve the relationship between each normal and its associated vertex data. The model-view matrix is a combination of the viewing and modeling transforms specified by the application.

$$\begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} = M \begin{pmatrix} x_o \\ y_o \\ z_o \\ w_o \end{pmatrix}$$

The model-view transform is an example of a floating-point operation typically performed by the geometry engine within the graphics subsystem. Graphics subsystems without dedicated geometry processing hardware implement the model-view transform using software executing on the host CPU. In eye space, also known as world coordinates, an object is positioned with respect to the viewer so that it can be properly lit, shaded and rendered in relation to all the other objects and light sources in the scene.

3.2.2 Per-Vertex Operations

Once objects have been transformed into the world coordinate system per-vertex operations can be performed. Per-vertex operations are texture coordinate generation and lighting.

3.2.3 Texture Coordinate Generation

Although most applications send down texture coordinates, the rendering pipeline can automatically generate texture coordinates based on input parameters from the application. This functionality is useful for environment mapping applications, projected textures, etc.

3.2.4 Lighting

The lighting state computes RGBA color values for each vertex in a scene. As a result, lighting is the most computationally complex stage of the 3D rendering pipeline. Before exploring some specific details which should be considered in the context of an application, it's beneficial to review the basic lighting calculations.

The lighting stage computes RGBA color values by applying a light model equation to a set of input paramters. These input parameters include:

- Vertex coordinates.
- Vertex normal.
- Light source coordinates.
- Light source characteristics.
- Material characteristics.

The color for each vertex is then calculated as the sum of the vertex material emission, the global ambient light scaled by the vertex ambient property, and the attenuated ambient, diffuse and specular contributions from all light sources. Examination or each term in the lighting equation will demonstrate how the values of particular parameters can influence work performed by the graphics hardware.

Material Emission

The material emission is simply the RGB value which indicates the color of an object that is not attributed to any external light source.

$$color_{emissive} \tag{3.1}$$

Scaled Global Ambient Light

This term is computed simply by mulitplying the global ambient light in the scene by the material ambient value for an object. Each of the R, G, and B values is multiplied separately to compute the final RGB values.

$$color_{global} = ambient_{scene} * ambient_{material} \tag{3.2}$$

Light Source Contributions

Each light source within a scene may contribute to the final vertex color. The contributions from each light source are scaled by the attenuation and spot effect and then added to the contributions from other light sources to obtain the final value for all light source contributions. The equation to compute the contribution for each light source is:

$$\begin{aligned} contribution &= att * spot * \\ &\quad (color_{ambient} + color_{diffuse} + color_{specular}) \end{aligned} \tag{3.3}$$

The attenuation factor (att) is calculated in relation to the distance (d) between the light's position and the vertex as:

$$att = \frac{1}{att_{constant} + att_{linear} d + att_{quadratic} d^2} \tag{3.4}$$

Except in the case of a directional light where the attenuation factor is simply 1.0. In this case, the light rays entering the scene are parallel because a direction light is considered to be infinitely far away from the scene.

The spotlight effect ($spot$) for a light source evaluates to one of three possible values specified below. In the case that the light is a spotlight source and the vertex does fall within the illumination cone, the spot effect is the dot product of the unit vector (v_x, v_y, v_z) that points from the spotlight to the vertex and the vector (d_x, d_y, d_z) that describes the spotlight's direction. This dot product varies with the cosine of the angle between the two vectors. As a result, vertices directly in line get maximum illumination while vertices off the axis receive less illumination as the cosine of the angle gets smaller.

$$spot = \begin{cases} 1, & light\, not\, spotlight \\ 0, & light\, is\, spotlight\, but\, vertex\, outside\, illumination\, cone \\ V \odot d, & otherwise \end{cases}$$

The ambient term ($color_{ambient}$) is simply the ambient color of the light source scaled by the ambient material property:

$$color_{ambient} = ambient_{light} * ambient_{material} \tag{3.5}$$

The diffuse term ($color_{diffuse}$) is calculated by scaling the diffuse color of the light source by the diffuse material property and then scaling the result by the dot product of the unit vector $(L_x.L_y, L_z)$ that points from the vertex to the light and the normal vector (n_x, n_y, n_z). Once again, as in the case for the factors in the angle between the light source and the vertex normal. If the dot product is less than or equal to 0, then there is no diffuse component contribution at that vertex. If the result is less than 0, then the light is on the wrong side of the surface.

$$color_{diffuse} = \lceil L \odot n, 0 \rceil * diffuse_{light} * diffuse_{material} \tag{3.6}$$

The final term is the specular term ($color_{specular}$). Before calculating the specular term, it must be determined that there is a specular component contribution at the vertex. Once again, like in the diffuse case, the dot product between **L** and **n** is used to account for the position of the vertex in relation to the direction of the light source. If the result of the dot product is less than or equal to 0, the light is on the wrong side of the surface and there is no specular component contribution at that vertex.

If the dot product is positive, then there is a specular component contribution. This contribution is then calculated using the normalized sum (s_x, s_y, s_z) of the two unit vectors that point between the vertex and the light position and vertex and the viewpoint. In the case that the viewpoint is not considered local to the scene, the second vector is then simply (0, 0, 1). The complete equation for calculating the specular component contribution is

$$color_{specular} = (\lceil s \odot n \rceil)^{shininess} * specular_{light} * specular_{material} \tag{3.7}$$

With all the various terms calculated, the complete equation for calculating the vertex color given n light sources can be expressed as

$$\begin{aligned} color &= color_{emissive} \\ &+ color_{global} \\ &+ \sum_{i=0}^{n-1} (att_i)(spot_i)[color_{ambient} + color_{diffuse} + color_{specular}]_i \end{aligned} \tag{3.8}$$

One final wrinkle to consider is that some graphics systems simply sum the ambient, diffuse, specular and emissive contributions as done in Equation 3.8. In the case of texture mapping, however, this can produce muted or undersireable specular hightlights as textures are applied after lighting. To eliminate this unwanted effect, the specular component can be computed separately, in effect, causing two color values to be calculated per vertex as in the following equation.

$$\begin{aligned} color_{primary} &= color_{emissive} \\ &+ color_{global} \\ &+ \sum_{i=0}^{n-1} (att_i)(spot_i)[color_{ambient} + color_{diffuse}]_i \end{aligned} \tag{3.9}$$

$$color_{secondary} = \sum_{i=0}^{n-1} (att_i)(spot_i)[color_{specular}]_i \tag{3.10}$$

In this case, during texture mapping, $color_{primary}$ is combined with the texture color. The specular color ($color_{secondary}$) value is then added to the result to produce more visible and realistic specular hightlights.

Hardware support for lighting varies widely amoung graphics vendors. A graphics subsystem with a geometry engine will generally implement lighting calculations in hardware, but the extent to which such lighting calculations are performed is highly dependent on the sophistication of the geometry processor and the types of lighting in the scene. Some vendors even go as far as implementing a lighting engine separate from the transform engine.

Some graphics subsystems will only implement infinite (also known as directional) lights in hardware, while others will also implement local lights. In the case of infinite or directional light sources, the light rays are considered parallel and as a result, the attenuation factor in Equation 3.4 is simply 1.0. This simplifies the overall lighting model equation by removing the per-vertex calculation of the attenuation factor. In OpenGL^{TM}, infinite or directional lights are specified with the fourth coordinate of `GL_POSITION` set to 0.0.

The specification of a local viewer also adds to the complexity of the lighting model equation. When a local view is specified, the calculation of the specular term in Equation 3.7 requires that the angle between the viewpoint and each object must be calculated. With an infinite viewer, this angle is not used producing slightly less realistic results but at a reduced computational cost.

The number of lights actually implemented in hardware also varies widely between graphics subsystems. Lights in excess of the number implemented in hardware will be implemented in software on the host CPU. If vendor supplied documentation does not specify hardware implementation, do some testing as outlined in Section 2.2.3 to determine the hardware support for lighting. If the performance decreases linearly as additional lights are added, then lighting is implemented in software. If performance only decreases slightly up to a point where it falls of dramatically, then lights are implemented in hardware with the cutoff point for the number of lights supported being the point at which the performance drops off significantly.

A graphics card without geometry acceleration will not implement lighting. This forces all lighting calculations to be performed in software on the host CPU. Independent of hardware or software implementation, typical lighting models are approximations which do not take into consideration effects such as shadows or objects which reflect or radiate light. More sophisticated lighting models must be implemented in software within the application or by the creative use of texture mapping.

Since lighting is the most compute intensive part of the rendering pipeline, lighting code within the graphics subsystems is typically tuned by the vendor to provide the best performance for the most common cases. Lighting code is often the first code tuned because tuning the lighting code offers the most return on investment or bang for the buck. One aspect of this tuning is that values which do not change from vertex to vertex are typically cached to reduce the per-vertex computational overhead. As such, changing properties unnecessarily between vertices can force unnecessary calculations and impact performance. One example in OpenGL^{TM} is the use of `glColorMaterial()` which specifies that material parameters track the vertex color. When `glColorMaterial()` is used, calculations which utilize material parameters must be recalculated between vertices.

3.2.5 Primitive Assembly

Primitive assembly consists of several suboperations which map 3D vertex data from eye coordinates into window coordinates. The resulting 2D primitives are then ready for rasturization. Graphics subsystems with geometry processing capabilities perform these operations in hardware.

Application-Defined Clip Planes

Before the perspective transform, objects are clipped against application-defined clip planes. One useful application of such clipping planes is the removal of extraneous objects. Clipping undesireable objects from the scene at this point in the rendering pipeline can improve performance by reducing the amount of work which must be performed at subsequent stages of the pipeline. Application-defined clipping planes can also be used to display cut-away views.

Unlike clipping against the view volume which is done in clip coordinates after the perspective transform, clipping here is done in eye space. As such, this operation always takes place in three dimensions. Specified in object space as the coefficients of the plane equation Ax + By + Cz + D = 0, these coefficients are multiplied by the inverse of the model-view matrix M to obtain the plane equation coefficients in eye coordinates.

$$\left(A^{'}B^{'}C^{'}D^{'}\right) = (ABCD)\,M^{-1}$$

All vertices in eye space with coordinates $(x_e\ y_e\ z_e\ w_e)$ that satisfy

$$\left(A^{'}B^{'}C^{'}D^{'}\right)\begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} \geq 0$$

lie in the half-space defined by the plane equation. Vertices that do not statisfy this condition do not lie in the half-space and are clipped away.

Note that when objects are clipped, while some vertices are clipped away, new vertices are added where an object intersects with the clip volume. As a results, there can be a net increase, a net decrease, or no change in the number of vertices that will be processed in subsequent stages of the pipeline. When new vertices are added, the color, texture and depth values at those vertices are calculated by interpolating between the values at the original vertices.

An optimization performed in some implementations is to implement the clipping against the application-defined clip planes before the lighting stage. In cases where a significant amount of geometry may be clipped away, this significantly reduces the amount of work required during the lighting stage at the expense of having to track clipped vertices to access per-vertex data required by the lighting model.

Perspective Transform

The **perspective transform** maps graphics primitives from eye space into clip space by multiplying the x_e, y_e, z_e, and w_e coordinates for each vertex in eye space by the 4 x 4 projection matrix P. Again, the perspective transform, like the model-view transform is a floating point operation handled efficiently by the geometry engine within the graphics subsystem hardware.

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = P\begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix}$$

Clipping

Once in clip space, objects are *clipped* against the view volume specified by

$$\begin{pmatrix} -w_c \leq x_c \leq w_c \\ -w_c \leq y_c \leq w_c \\ -w_c \leq z_c \leq w_c \end{pmatrix}$$

Like in the case for application-defined clip planes in Section 3.2.5, clipping is performed at this point using the same method used previously. In this case, however, the plane equations are the six plane equations which define the view volume.

Clipping can also be implemented by combining clipping against the application-defined clip planes and view-volume clipping into a single operation. In this case, the clip volume and resulting plane equations become the intersection of the half-spaces defined by the application-defined clip planes with the view volume.

Perspective Division

Perspective division divides the x_c, y_c, and z_c coordinates by w_c to map each of the clip space coordinates into normalized device coordinates x_d, y_d, z_d.

$$\begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} = \begin{pmatrix} x_c/w_c \\ y_c/w_c \\ z_c/w_c \end{pmatrix}$$

Viewport Transform

Finally, the viewport transform uses the viewport parameters of the application to map each of the normalized device coordinates into window coordinates. Given a viewport's width (w), height (h), and center (o_x, o_y), the window coordinates (x_w, y_w, z_w) for a vertex are calculated

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} (w/2)x_d + o_x \\ (h/2)y_d + o_y \\ ((f-n)/2)z_d + (n+f)/2 \end{pmatrix}$$

Where n and f represent the near and far clipping planes specified by the application.

3.2.6 Rasterization

Rasterization is the process of converting a geometric primitive into an image. Rasterization determines which pixels are occupied by a primitive and then calculates and assigns the appropriate color, depth, and stencil values for each pixel. Graphics subsystems with rasterization, also known as rendering engines perform this operation in dedicated hardware. However, there are typically special cases where a software implementation might need to be used when a specific operation or rendering mode is not implemented in hardware. As described in Section 4.4.1, this is known as falling off the fast path.

The process of rasterization can be broken down into two steps. The first step determines which grid squares in a 2D integer grid in window coordinates are occupied by a primitive. The second step assigns a color and depth value to each square. Each grid square and it's associated data is called a *fragment.*

Nearly all graphics subsystems today implement rasterization and at least some of the subsequent per-fragment operations in dedicated hardware. It's at this stage that primitives are matched to those supported natively by the rendering engine and sequences of microcode-like rendering instructions are executed by the rendering engine to perform rasterization and all subsequent per-fragment operations before a primitive is display in the framebuffer. Rendering engines perform rasterization by executing a series of rendering instructions. These instructions are commonly implemented in such a way that all subsequent per-fragment required to render a primitive in the framebuffer are chained together and performed in sequence with rasterization. However, rasterization is presented here with subsequent stages and per-fragment operations discussed later.

Rasterization engines may not support all primitives natively. For example, triangle strips may be actually rasterized by duplicating vertices and rasterizing individual triangles. Quads may be rendered as two triangles, etc. Non-native primitives are supported transparently to applications excepting the speed at which these primitives are rasterized.

Points are typically rasterized by truncating their x_w and y_w window coordinates to integers. The resulting (x, y) then specify an address on the integer grid. Hardware that does not support points with a width greater than 1.0 grid units, may convert such points into quads.

Lines are rasterized by rendering instructions which typically implement Bresenham's classic algorithm [18]. Support for lines with a width greater than 1.0 is implementation dependent with some hardware rendering them as quads while others might fallback to a software rendering path.

Polygon rasterization includes the rasterization of polygons, independent triangles, triangle strips, triangle fans and quadralaterals. The first step in polygon rasterization is *back face culling*. If it is enabled, only polygons which are determined to be front facing are rasterized. Because rasterization engines typically operate on *spans*, higher-order polygons are typically decomposed into triangles for easy rasterization using a scan line algorithm that linearly interpolates triangles along edges and linearly interpolates data across horizontal spans.

Other rasterization problems that may cause an application to force software rasterization include:

- Stippling.
- Antialiasing (See Section 3.2.9).
- Additional polygon modes (edge flags, wireframe, point).

3.2.7 Texture

When texturing is enabled a post-rasterization texture application stage is added to the basic 3D geometry path. This new stage is combined with a 2D path which acts upon the image data to place it in texture memory prior to application. The 2D texture path will be described in Section 3.3

When texture mapping is enabled the rasterizer will, when executing the scanline algorithm to rasterize the triangle, interpolate between texture coordinates to compute an offset into texture memory. This texture data is then used to color the fragment. This coloring occurs in different ways depending on the texturing mode. For example, if a 'replace' mode is selected, then the texture color value will be used as the color for a particular fragment. In another example, if 'blend' mode is selected, then the texture color value will be blended with the triangle color value to determine the final fragment color.

3.2.8 Fog

Fog is also often commonly referred to as depth cueing. When a fog operation is enabled, objects appear to fade into the distance. Fog color is computed by blending an incoming fragment's post-texturing color with a fog blending factor to calculate the final fogged color:

$$color_{final} = (f)color_{fragment} = (1 - f)color_{fog}$$

where

$$f = e^{-(density*z)} \tag{3.11}$$

$$f = e^{-(density*z)^2} \tag{3.12}$$

$$f = \frac{end - z}{end - start} \tag{3.13}$$

and z is the eye-coordinate distance between the viewpoint and the fragment center. Depending upon the graphics pipeline implementation f may be computed at each fragment, or computed only at vertices and interpolated. The fog operation is typically not implemented as a test and therefore fragments which are completely fogged are not discarded.

3.2.9 Antialiasing

Antialiasing, the processing of removing the "jaggies" from an object, if enabled, is one stage in the rendering pipeline where your mileage will vary depending upon the graphics pipeline implementation. Antialiasing at this stage is performed on a per-primitive basis.

In RGBA mode, antialiasing is implemented by multiplying the A values for a fragment by the percentage of coverage by a primitive. This A value is then used during the blending per-fragment operation stage of the pipeline, to blend the fragment's color with the color of the corresponding pixel already in the framebuffer. In CI mode, the least significant bits of the color index for a pixel are set according to the percentage of coverage. All ones represents complete coverage while all zeros represents no coverage.

Antialiasing of primitives, especially those other than lines of width 1.0 (wide lines, triangles and, quads), are often implemented in software even on relatively high-end graphics systems with dedicated rendering engines. This is due to the fact that the prodecure for computing the coverage values for various fragment types is difficult to implement in dedicated hardware. Therefore, enabling antialiasing for primitives other than lines will have a significant impact on performance. There may however be other ways to achieve the desired effect by using multisampling or another full-scene antialiasing technique such as the accumulation buffer [25] or oversampling as described in Section [**?**].

3.2.10 Per-Fragment Operations

Per-fragment operations are a series of tests and operations performed upon each fragment before the data for that fragment can influence pixel locations in the framebuffer. These operations are typically implemented by the rasterization engine as additional rendering instructions executed during rasterization.

Pixel Ownership Test

The pixel ownership test determines if the pixel at location (x_w,y_w) is owned by the current rendering context. As a result of this test, a fragment may be discarded completely, or may continue to be rendered with some subset of the subsequent per-fragment operations being applied.

It's at this stage that window clipping takes place. Fragments which are obscured or are not visible are discarded. Some graphics pipeline implementations use extra bit planes, called clip planes, to store clip IDs at each pixel. In this case, pixel ownership is determined by comparing the clip ID at each pixel in the framebuffer with the ID of the current rendering context. This technique operates up until the point atwhich the number of clip planes is insufficient to store the number of clip IDs. At this point, a pure software implementation is used. In this case, and in systems that do not support clip planes, when a window is partially obscured, a set of clip lists is defined. The clip list is a list of rectangles within the window that are either visible or obscurred. Each fragment must be checked against all the rectangles in this list using an inside/outside test. Because this operation involves interaction with the windowing system this test has a severe impact on performance. This implies that windows will render fastest unobscured by other windows.

Scissor Test

The scissor test determines if the fragment at location $(x_w,\ y_w)$ lies within the rectangle defined by $left, width, bottom and top$. If $left <= x_w < left+width$ and $bottom <= y_w < bottom+height$, then the scissor test passes. This test is a simple inside/outside test commonly implemented by rasterization hardware.

Alpha Test

The alpha test compares the alpha value of the current fragment against a constant value. Fragments which do not pass the test are discarded. Again, this test is commonly implemented by rasterization hardware.

Stencil Test

The stencil test compares the value in the stencil buffer at location (x_w, y_w) with a constant and discards fragments which don't pass the test. The performance of this test is slow on systems which do not have a hardware stencil buffer.

Depth Test

The depth test compares the depth value for an incoming fragment (x_w, y_w) with the value currently in the depth value at that location. If the test fails, the incoming fragment is discarded. This test is slow when a hardware z-buffer is not available.

Blending

The blending operation combines the R, G, B and A values for an incoming fragment with the R, G, B and A values stored at the incoming fragments (x_w, y_w) location within the framebuffer. The blending equations are implemented by rasterization hardware.

Dithering

Dithering is not typically implemented by rasterization hardware and should be avoided for optimum performance.

Logic Op

The logic op test performs a logical operation between the color of the incoming fragment and the color in the framebufffer at the incoming fragments location. The result replaces the values in the framebuffer at the fragment's (x, y) coordinates. Logic operations are implemented by the rasterization engine.

3.3 Image Paths

The graphics pipeline contains a set of image paths which perform a series of imaging operations on pixel data. Pixel or image data is a set of data values sampled at discrete points. Although image data is most commonly 2D, such data can also be 1D or 3D. The data values within an image most often contain color data, but can contain other types of data as well. This adds to the overall functionality for which the imaging paths can be used.

The imaging paths perform **Draw**, **Read**. **Copy**, **Texture**, and **Bitmap** operations. Though these paths are functionally different, they share many operational pipeline stages. This section will describe each of these paths, the stages through which data passes along each path, and how these paths connect to the same rasterization backend used by the 3D geometry path.

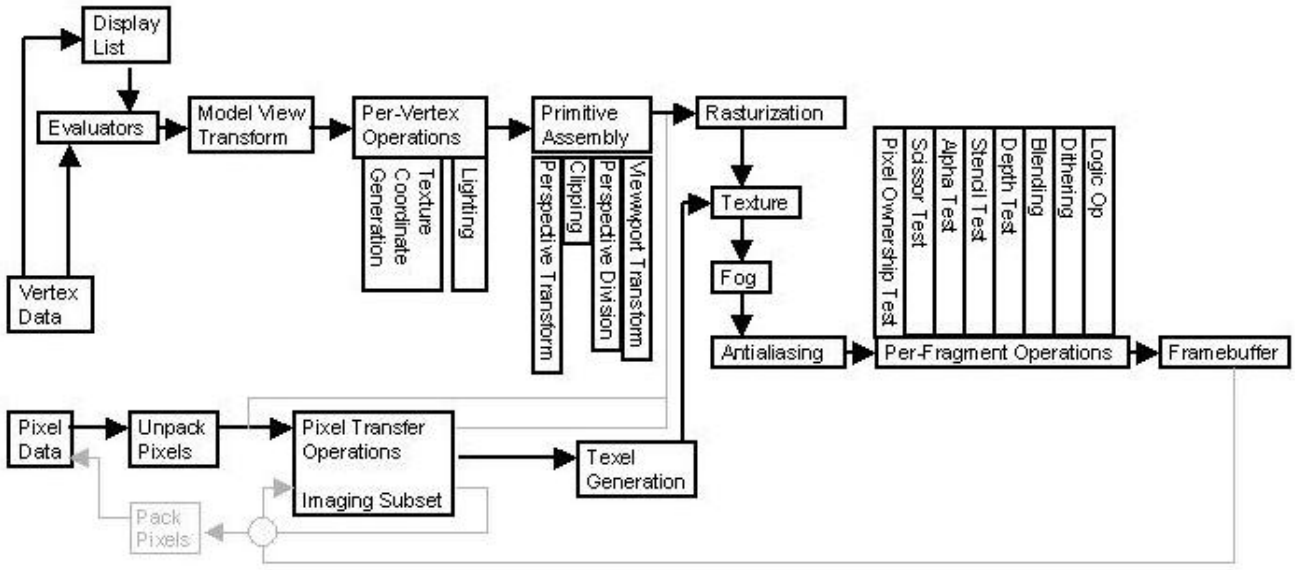


Figure 3.3: 3D Path with Texture.

Unlike the 3D path, inwhich data moves in a single direction, from memory to the framebuffer, data in the image paths can move in the reverse direction as well. This "readback" from the framebuffer back to graphics or system memory is a primary "raison d'être" for the imaging side of the graphics pipeline.

3.3.1 Draw Pixels

Figure 3.4 illustrates the Draw Pixels path through the graphics pipeline. This path employs a series of stages to render image data in the framebuffer.

Unpack Pixels

This stage serves to convert the data from the format inwhich it is stored in memory, into an internal storage format according to parameters supplied by the application. This stage also performs byte- swapping and data alignment operations.

For best application performance, pixel data should be stored within the application in a format that is native to the graphics hardware. This mitigates the unpacking that is required and as a result improves performance. Image data that is in a native format to the graphics hardware is typically transferred from system memory to graphics using a DMA operation. Non-native data will require conversion by the host CPU.

Pixel Transfer Operations

The pixel transfer stage performs operations on pixel data as it is transferred between system memory and the graphics subsystem. Note that this is different than the unpack operation which basically just rearranges the data without altering it in any way. Some typical pixel transfer operations are scale, bias, and color-mapping functions.

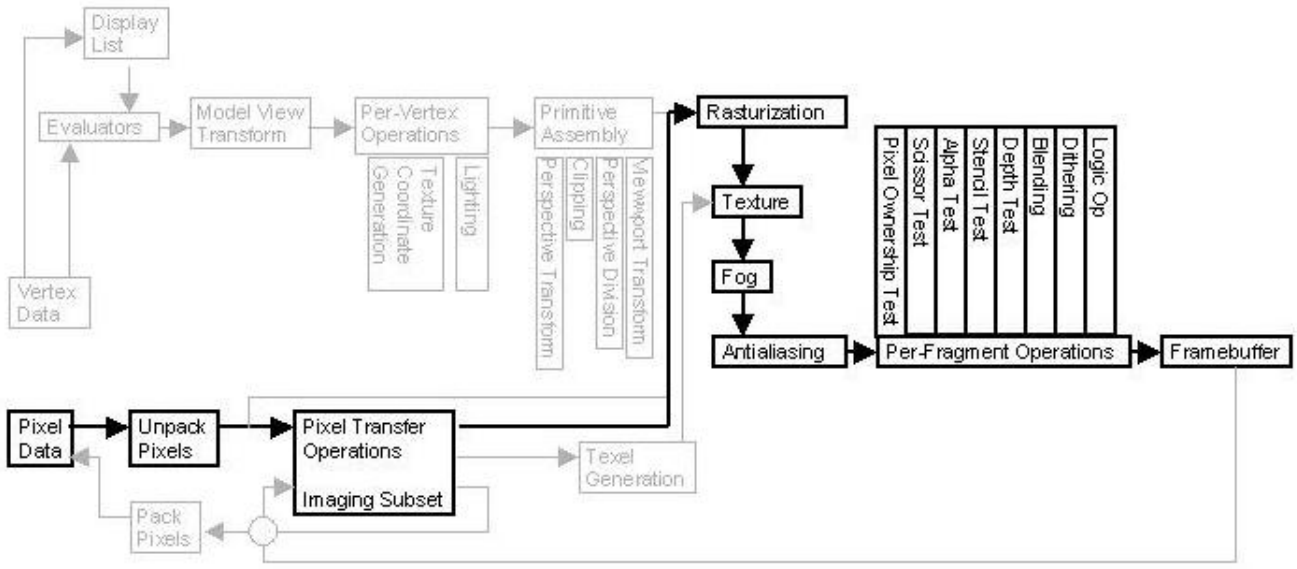


Figure 3.4: 2D Draw Pixels Path.

Imaging Subset

A new feature extension in OpenGL^{TM} 1.2 is the Imaging Subset, a collection of additional pipeline stages that provides pixel processing capabilities. Figure 3.5 illustrates how these additional stages fit within the legacy 2D pipeline.

Each stage of the Imaging Subset operates on pixel data that results from the unpack stage in the pipeline. Varying degrees of each function are typically implemented in hardware within the graphics subsystem.

Color Table Lookup Color table lookup is used to replace a pixels color value with previously defined in a lookup table (LUT). Typically, color LUTs can be used in three placed within the imaging pipeline.

- As pixel data enters the rendering pipeline.
- After convolution.
- After color-matrix transformation.

Convolution The convolution stage implements pixel filter kernels which replace each pixel in an image by some weighted average of it and its neighboring pixels. This functionality is useful for sharpening or blurring, and other image filtering operations. As the number of pixels used in the average increases, thus increasing the size of the convolution filter kernel, the number of required calculations increases. This additional complexity makes hardware implementation of convolution extremely difficult, except for small filter kernels. Hardware convolution implementations typically only support kernels of 3x3 or 5x5 although some higher-end graphics hardware can support hardware convolution with kernels of up to size 7x7. Vendor documentation should indicate the maximum filter kernel size supported by a graphics subsystem. If it does not, write a simple test program which performs convolution and compare the results for different kernel sizes. A sudden increase in the time required to perform the convolution is a good indication that the maximum allowable hardware kernel size has been exceeded.

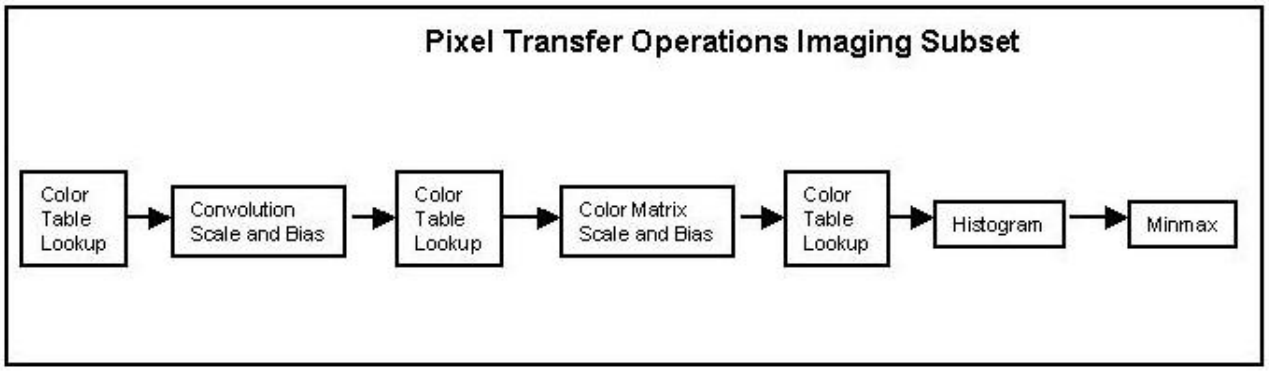


Figure 3.5: 2D Read Pixels Path.

Color Matrix This stage performs color matrix operations and linear transformation on pixel values. The functionality is used to perform color space conversions within the graphics subsystems without using the CPU. However, because operations are performed with fewer bits of precision in the graphics subsystem than if they were performed on the host CPU, the precision may not be of acceptible quality for certain applications.

Histogram The histogram function calculates the histogram for a specified pixel rectangle.

Minmax Given a pixel rectangle, the minmax function computes the minimum and maximum pixel component values for a particular pixel rectangle. Rendering engines can be programmed to efficiently implement this functionality.

Be aware that hardware which was not designed for OpenGL^{TM} 1.2 may implement fewer if any of these operations in dedicated hardware.

3.3.2 Texture Path

A variation of the Draw Pixels path is the Texture path in Figure 3.6. This path shares many of the same stages as the Draw Pixels path but the difference in the the case of texture is that image data resulting from the Pixel Transfer Operations and Imaging Subset is passed to the texel generator rather than the rasterizer. Texel values are then placed in texture memory to be applied during the texture application stage of the geometry pipeline as described in Section 3.2.7.

Data is transferred from host memory to graphics using a DMA operation if possible. If the pixel data within the application is in a non-native format then it can't be directly downloaded and must be reformatted by software executing on the CPU, as described earlier for the draw-pixels case.

3.3.3 Read Pixels

The Read Pixels path in Figure 3.7 is another 2D path. This path is unique in that it works in reverse taking data from the framebuffer and putting it in system memory. The most common path in computer graphics to simply to draw an object to the framebuffer, but it is quite common in fact to read rendered images from the framebuffer back into system memory for storage or additional graphics processing.

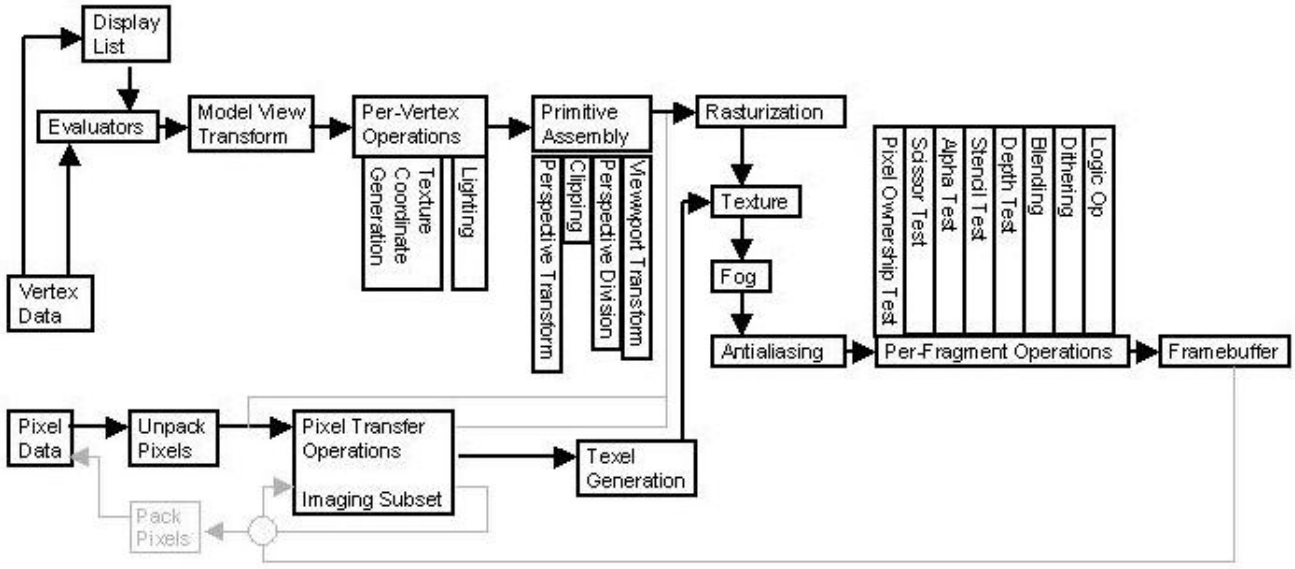


Figure 3.6: Texture Path.

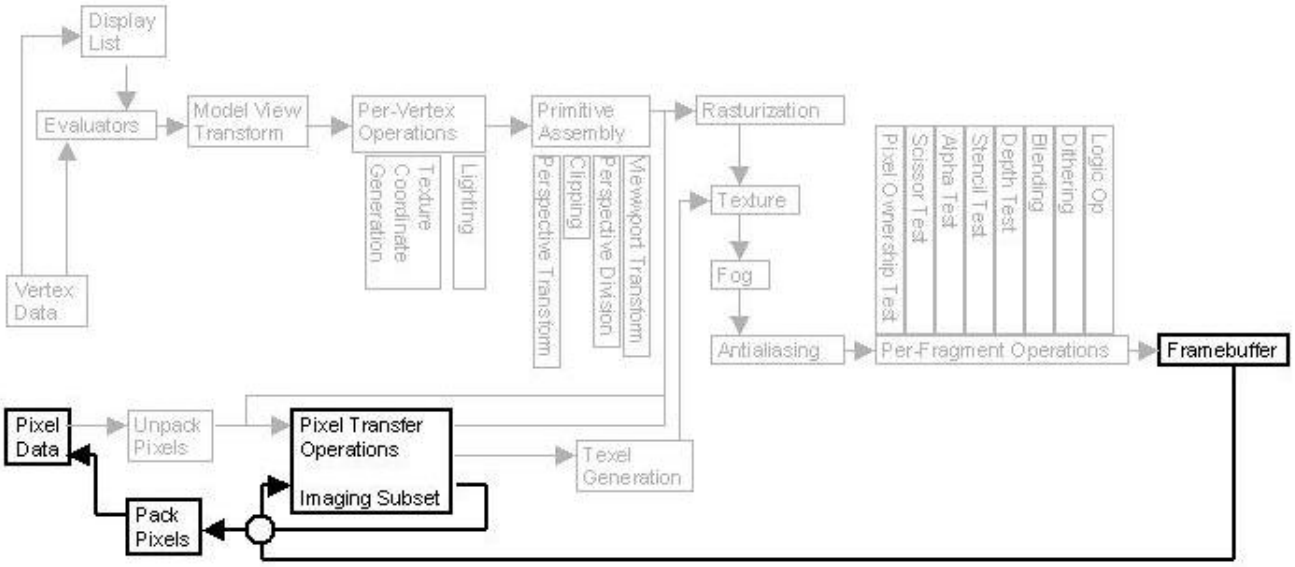


Figure 3.7: 2D Read Pixels Path.

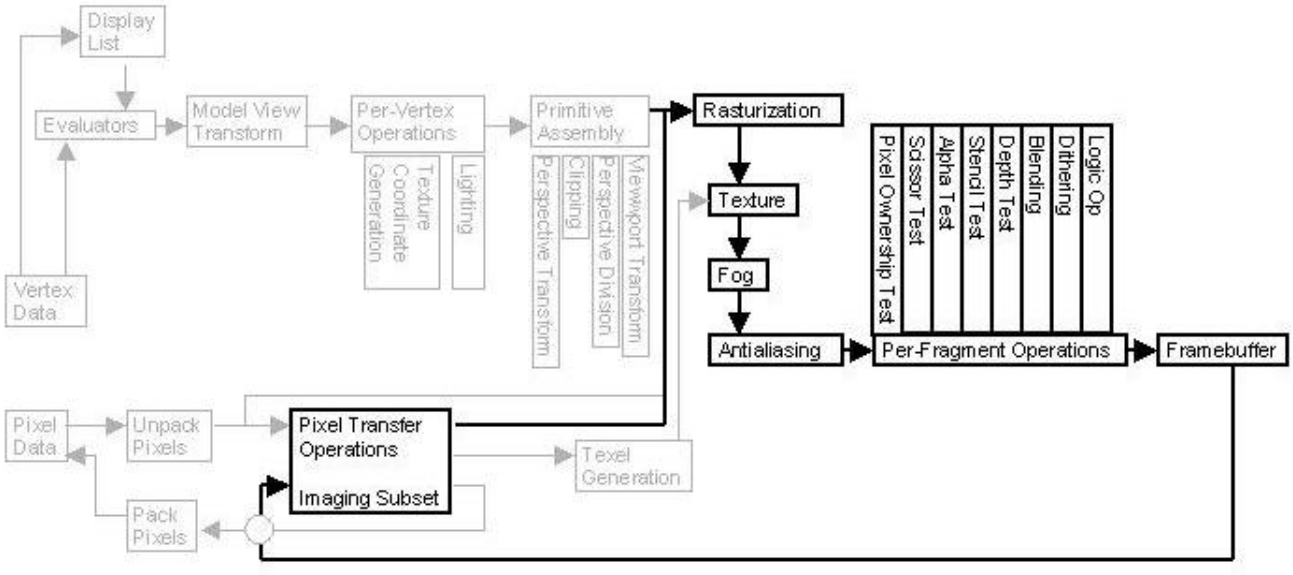


Figure 3.8: 2D Copy Pixels Path.

In the Read Pixels case, pixel data which has been previously rasterized to the framebuffer is read and passes through the same Pixel Transfer Operations and Imaging Subset stages described above. The pixel data is then packed as specified by the application and placed into system or graphics memory.

Historically, the focus of graphics hardware architecture has been on high-performance geometry processing and pixel fill-rates at the expense of other operations, specifically the ability to ready pixel data back from the framebuffer. As a consequence, the efficient implementation of the read pixel operation has been neglected and as a result performs poorly on many graphics subsystems. A common problem on AGP based graphics subsystems is that data will transfer to the graphics at AGP speeds, with data being transferred on both the leading and trailing edges of the clock, but data is only read back from the graphics subsystem at half that data rate because the interface on the graphics card can only send data on the leading edges.

Another common problem is that pixel read operations break the inherent pipelining in a graphics subsystem. A problem commonly encountered is that the read occurs before the graphics have finished rendering. This is solved by ensuring that all drawing is complete before trying to readback. Another solution is to time the readback to take place when the application is performing other CPU operations.

3.3.4 Copy Pixels

Figure 3.8 illustrates the Copy Pixels path. This path is a hybrid of the Draw Pixel and Read Pixel paths previously discussed. As in the Read Pixels case, pixel data which has been rendered is read from the frame buffer or other auxiliary buffer and passes through the Pixel Transfer Operation and Imaging Subset stages before being re-rasterized and redisplayed in the framebuffer.

This path suffers the same performance pitfalls as in the Read Pixels and Draw Pixels paths as it is a combination of the two pixel operations. One example of this as described earlier is that some AGP graphics adapters during the readback operation cannot send data back at full AGP rates.

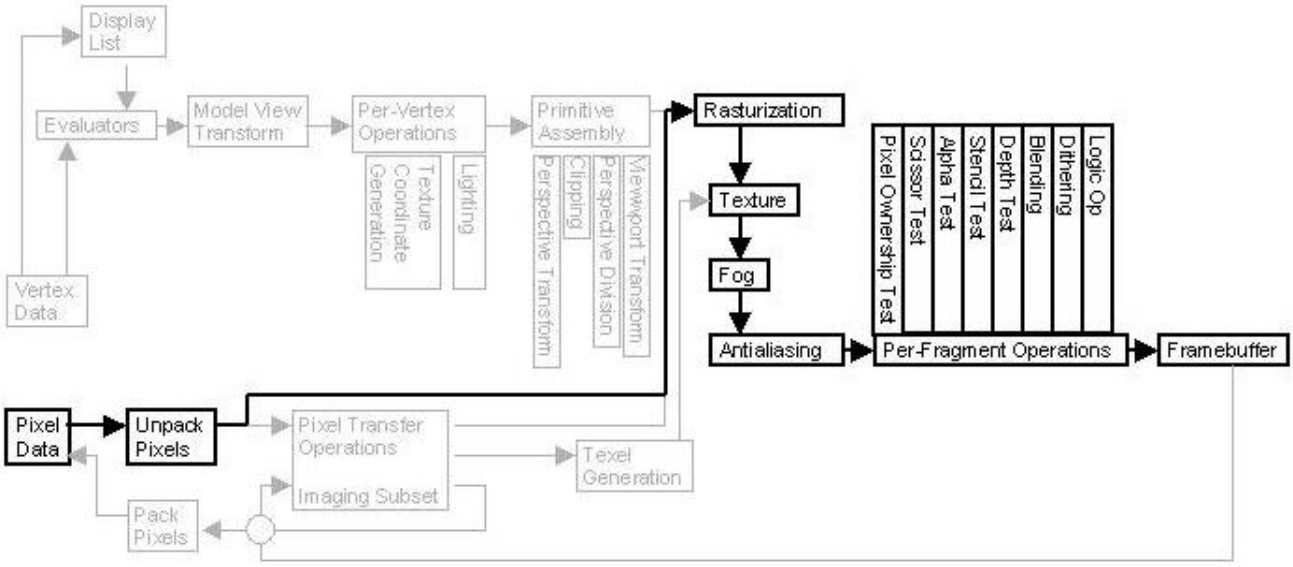


Figure 3.9: 2D Bitmap Path.

3.3.5 Bitmap

Figure 3.9 illustrates one final 2D rendering path, the path for bitmap operations. This path is similiar to that for the Draw Pixels case, but in this case bitmap data is obtained from system memory without Pixel Transfer or Imaging Operations being performed. The bitmap data simply passes directly to rasterization.

This path is typically implemented in a graphics subsystem as a special case of the DrawPpixels path where the data processed is simply a single bit per pixel. Low performance bitmap rendering is typically due to the fact that the hardware was designed to render pixels as a combination of color components, but when each color component is packed differently, as in a bitmap, performance suffers.

3.4 Conclusion

The graphics rendering pipeline consists of geometry and imaging paths implemented as a combination of hardware and software. Graphics operations not implemented in graphics hardware are performed in software executing on the host CPU. This section described the operations of the 2D and 3D paths through the graphics pipeline and the stages within these paths. Effective performance analysis and tuning of a graphics application can be better achieved with a complete understanding of these paths and their potential performance implications.

Section 4

Software and System Performance

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

Before digging in and examining each stage of the process in detail, it should be stressed that the process described here is iterative and is never really complete. Once a bottleneck or application performance problem and been identified and addressed, the tuning process should start anew in search of the next performance bottleneck. Code changes as well as hardware changes can cause performance bottlenecks to shift among the different stages of the rendering process and also shift between the CPU and graphics subsystem. As a result, performance tuning is an ongoing process.

4.1 Quantify: Characterize and Compare

To achieve a balance between the demands of an application program and the computer graphics hardware, examine the application to access the actual graphics requirements. The goal is to collect some basic information to determine what the application is doing without regard for the underlying computer graphics hardware. With this information, you can compare it to the ideal performance of the graphics hardware to learn the relative performance of the application.

4.1.1 Characterize Application

Application Space

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

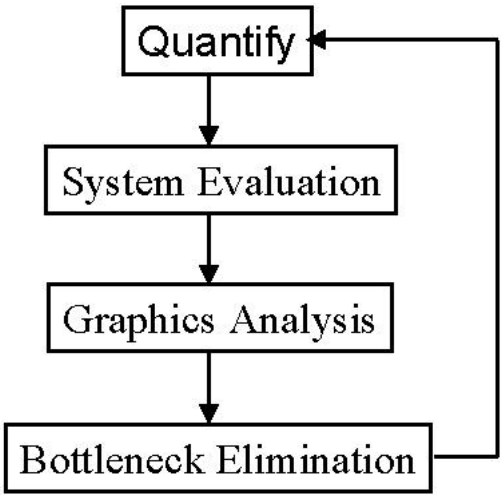


Figure 4.1: A Four Step Process.

Primitive Types

Determine the primitive types (triangles, quads, pixel rectangles, etc.) being used by the application and if there is a predominant primitive type. 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 rendered per frame by instrumenting the code to count the number of primitives between buffer swaps or screen updates. For primitives sent in lists, report the number of lists and the number of primitives per list. Add instrumentation such 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 to use the application and exercise code paths as a true user would. The work process that a bona fide user encounters day in and day out is the most useful to consider. It is also important to exercise multiple code paths when gathering data about performance.

After determining 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, inspiring thought about bandwidth saving alternatives. For example, consider the worst case as illustrated in Figure 4.2. To render a triangle with per-vertex color, normal and texture data requires 56 bytes of data per vertex, 168 bytes per triangle. Rendering the three triangles individually requires 504 bytes of data (Figure 4.2A); rendering the triangles as a strip only requires 280 bytes of data (Figure 4.2A), which saves 224 bytes. In a real application, this

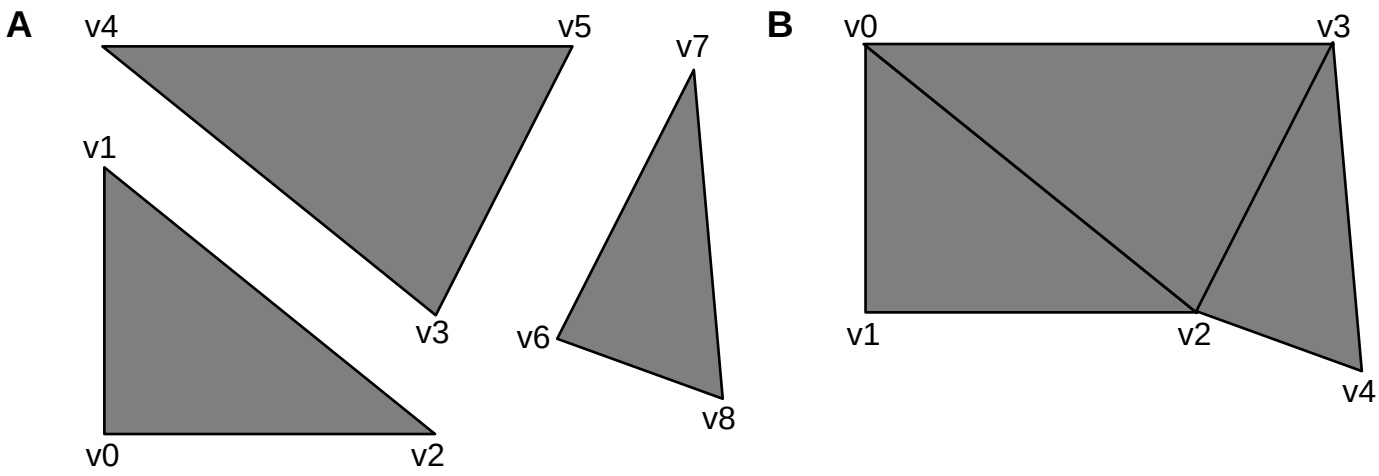


Figure 4.2: Worst case per-vertex data for triangles. (A) Shown are three triangles, each vertex containing position (XYZW), color (RGBA), normal (XYZ), and texture (STR). Rendering a single triangle requires 56 bytes of data per vertex, resulting in a total of 168 bytes of data. The set of triangles 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.

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

Lighting Requirements

Lighting requirements are a critical consideration in order to fully quantify the graphics requirements of an application:

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

Lighting information is easily discovered by looking at the graphics API calls in the application source code.

All the listed lighting variables affect the number and complexity of calculations that must be performed in the lighting equations. For example, the use of local lights requires the calculation of an attenuation factor, which makes them more expensive than use of infinite light sources. Furthermore, the use of 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. When two-sided lighting is used, lighting is done twice, once for the front face of a polygon and a second time for the back face. Please review section 3.2.4 for more information on 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 currently attainable number of frames 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 single-buffer mode can eliminate some potential measurement artifacts here, as described in more detail in Section 2.2.3. Some systems provide hooks (and tools, such as `osview`) into the hardware, which can measure framebuffer swaps for any application.

4.1.2 Compare Results

After collecting the above data, you can make a full comparison between the current performance of the application and the ideal performance of a particular system. Use this comparison to determine if the application performance is what you expect given the capabilities of the available hardware. Methods for obtaining ideal performance data for a system are described in Section 2.2.3.

Compare how the application data gathered earlier compares with data supplied by the manufacturer or obtained using a test program. Don't forget to consider that data supplied by the manufacturer is optimal and may not be realistic. Is the application using primitives recommended (and accelerated) by the hardware vendor? Also, consider the fact that the application may need to take time to generate the data to render for each frame. This is time which 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 that quoted by the manufacturer or obtained from a test program? If this is the case, then no amount of tuning of the graphics code will improve the user experience. (See Section 5 for a performance boost.) If this is not the case, then the application's graphics are performing poorly and can benefit from tuning 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 impacts overall performance.

4.2 Examine the System Configuration

Often the first system component to be considered when examining the graphics performance is the graphics hardware. Yet it is better to first examine the other system components first to see how they are configured and how they might affect rendering performance. Eliminate other system components from the performance tuning equation before examining 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 setup and configured.

4.2.1 Resources

Memory

Insufficient memory in a system can cause excessive paging. Understand the memory requirements of your application and measure them against the available memory in the system. Large amounts of disk activity while an application is running indicates memory page swapping, 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. Keep in mind that large models and databases add to the overall memory footprint of an application.

Consider how system memory is used to store graphics data. Some systems implement a UMA where the frame buffer resides in system memory, and other systems might use AGP where some textures and most graphics data are stored in system memory before a high-speed transfer to the frame buffer. 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 frame buffer 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 frame buffer, which removes the cost of the additional copy from system memory to dedicated video memory, as found in more traditional hardware. One caveat of this approach is that this memory is never available to an application. As a result, if this effective loss of system memory is not taken into account by boosting the physical system memory accordingly when configuring the system, an application that fits nicely on a traditional system may swap on a UMA system.

On a system built around AGP, system memory is used to hold graphics data, but this memory is not reserved for the frame buffer and can be allocated and freed as necessary so that it may be used by the application. The use of 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 non-resident textures requires a full fetch from main memory, with all the attendant performance implications of main-memory access.

Know the memory access times and bus speeds of the system. Examine these in respect to the amount of data that the application is moving around when rendering. Consider if an application's optimal data transfer per unit time will exceed that which can be provided by the memory and bus. No matter how fast the CPUs in a system, the overall performance in some applications domains will be limited by the bus speeds on which the CPUs sit. 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 (IDE, SCSI, fibre channel, etc.), consider the actual location of the disks and the application requirements. Streaming video to the screen from a slow disk will always be physically impossible, regardless of the speed of the graphics hardware. Store data and textures on local disks, as 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 from the graphics hardware manufacturer is installed on the system before examining the display configuration. Manufacturers are constantly fixing bugs and tuning their drivers and the latest driver will typically perform best. If it is unclear if a new driver will offer the best performance,

compare the performance between 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 windowing 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 2.1.7. The display configuration should be set to take full advantage of the features implemented in hardware and necessary for the application.

When display information is queried by an application, the windowing 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 considering the display properties 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. There are often a number of visuals which match an applications needs and 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 ensure that it is using an appropriate configuration for the underlying graphics hardware. One way to do this might be to have an application do some simple benchmarks tests at startup which exercise frequently used functionality. Use the results of these tests to decide upon an optimum display configuration. The following are important display parameters to consider:

Pixel Formats / Visuals The pixel formats/visuals 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 required by an application. Have a fall-back strategy if the application can't get the desired pixel format. For example, if the display is configured such that there are no pixel formats or visuals available with hardware alpha-blending, an application that draws alpha-blended shapes forces the graphics driver to perform alpha-blending in software. A fall-back for this scenario might be to use stippled-alpha rather than blended-alpha.

Color Buffer Choose a visual which matches your application's needs for color precision. Systems often contain have visuals with 12-bits of precision available per color-component, but may have no alpha planes available in this configuration. A second consideration is to choose visuals which match, and only just match, the requirements for the applicatoin. Visuals with more precision per-pixel induce extra fill work, and can potentially be a 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, using 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. Using 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, while too few bits of precision creates visual artifacts known as “Z-fighting” or “flimmering”.

Auxiliary Buffers Several auxiliary buffers typically exist for a particular visual and selection of a visual with appropriate auxiliary is essential. Typical additional buffers available include stencil, accumulation, stereo, and others. Certain combinations of auxiliary buffers may force the rendering driver off the fastpath.

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 7.3.2. Many hardware graphics drivers now let users disconnect buffer swaps from the vertical retrace to improve performance by allowing an application to render to the back buffer as quickly as possible. Be aware 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 loading data and textures from a remote file system during rendering as network traffic and latencies will 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, doing a DNS lookup, or redrawing a simple animated-gif on a web page causes CPU usage 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 an understanding of how the performance of an application oscillates between the CPU and the graphics hardware subsystem. It’s also important to 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 it’s execution. An application oscillates like a pendulum between varying degrees of these two states as 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 (discussed in Section 2.1.1), 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 being out-of-balance.

As application data is traversed by an application, it is passed through a graphics library that prepares it for passing over the interconnect fabric(see Section 2.1.3). At this point, the graphics commands enter a command buffer, often a first-in-first-out buffer known as a *FIFO*. 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 fully using both the CPU and 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, the world is rarely perfect, and ideal performance is rarely achieved. Instead, application performance falls into one of the two categories below.

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 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

On the other hand, if the graphics subsystem is processing data slower than the FIFO is being filled, the FIFO will issue an interrupt causing the CPU to stall 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, awaiting a time when the hardware can again begin processing data. An application in this state is graphics-bound such that the overall performance is governed by how fast the graphics hardware can render the data that the CPU is sending it. 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 frame buffer, which is common in applications that draw large polygons. In the context of the graphics pipeline as described in Section 3, an application which is fill limited is limited by the speed at which rasterization and subsequent pipeline stages can be executed. The fill limit, specified in megapixels/sec, is determined by the rasterization capabilities of the graphics accelerator card. As the fill limit is reached, consider increasing application geometry load by more finely tesselating objects to improve the balance between fill rate and geometry transfer. This has the advantage of using otherwise idle geometric processing capability and can potentially reduce fill requirements because geometry more closely matches the shape being rendered.

Geometry Limited

An application that is geometry-limited is limited by the speed at which vertices can be lit, transformed, and clipped. Programs containing large amounts of geometry, or geometric primitives that are highly tessellated can easily become geometry-limited or transform-limited. An application which 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 upon the hardware capabilities of the graphics subsystem and where the geometric calculations are performed. Consider replacing smaller polygons with larger ones or reducing lighting requirements to balance the pipeline between geometry transfer and fill rate in this case.

4.3.4 Architectural Considerations

Thinking again about the different types of graphics subsystems outlined in Section 2.1.7, consider how an application can be CPU-bound or graphics-bound.

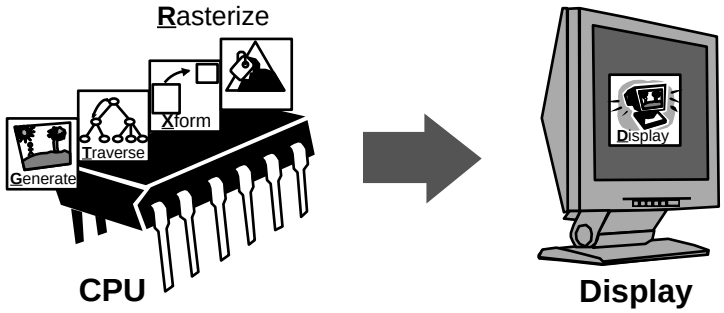


Figure 4.3: Schematic of the **GTXR-D** graphics subsystem.

GTXR-D As shown in Figure 4.3 all rendering stages are performed on the host CPU. If the CPU renders pixel values into the frame buffer faster than the screen is refreshed, and swap buffer calls are tied to the vertical retrace of the screen, the CPU stalls before rendering the next frame. In this case, an application is graphics-bound. However, since the CPU is calculating all other rendering stages, on this type of hardware, an application is much more likely to be CPU-bound.

Ultimately, CPU speed, scene complexity, and monitor refresh rate dictate the balancing point. If an application proves to be graphics-bound, increase the monitor refresh rate or disconnect buffer swaps from the vertical retrace. For a CPU-bound application, reduce the scene complexity by eliminating geometry. Another option is the use of more efficient graphics algorithms for rasterization, depth, stencil testing, etc. Also, consider potential code optimizations described in Sections 5 and 6.

As more and more parts of the graphics rendering pipeline as described in Section 3 have been implemented in special purpose hardware, this type of graphics system is becoming less prevalent in 3D graphics workstations and 3D game systems. It's for this reason that it is not discussed in more detail.

GTX-RD As shown in Figure 4.4 screen-space operations are performed in hardware while geometric operations are performed on the host CPU. If the CPU can generate and send the screen-space primitives and the associated rendering commands faster than the rendering subsystem can process them,

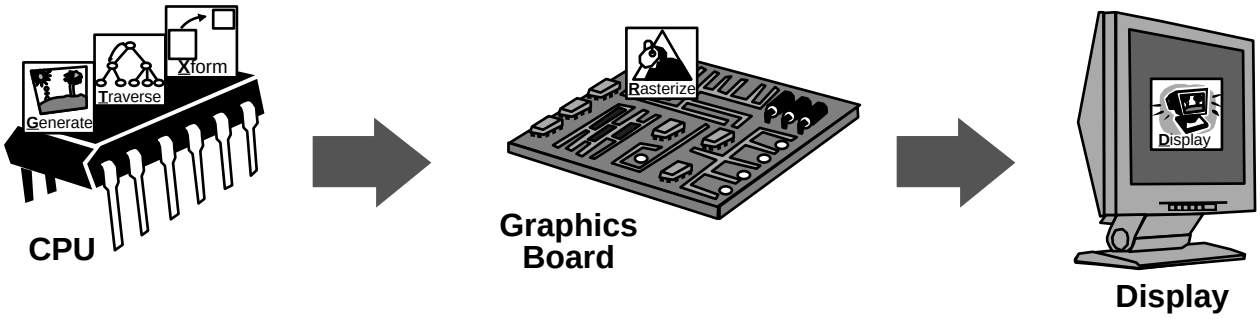


Figure 4.4: Schematic of the **GTX-RD** graphics subsystem.

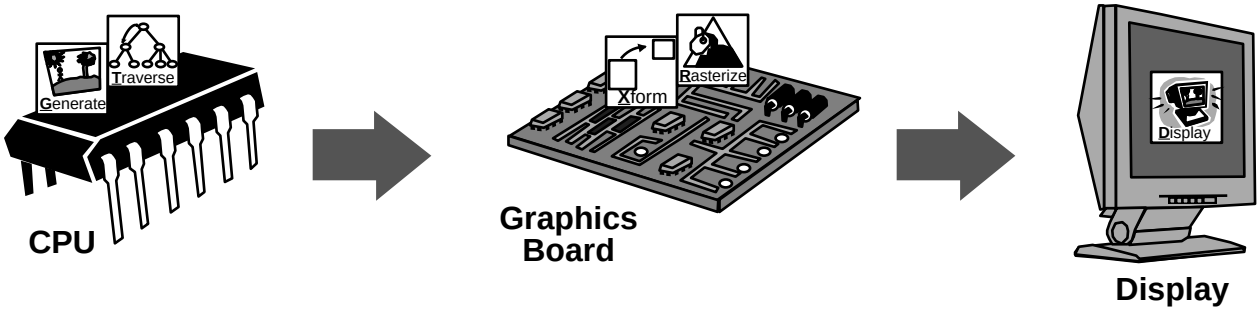


Figure 4.5: Schematic of the **GT-XRD** graphics subsystem.

the graphics FIFO fills causing the CPU to stall. When this happens, the application is graphics-bound. However, with this type of graphics subsystem, it's much more likely that an application will be CPU bound as the CPU is required to perform the approximately 100 single-precision, floating-point per-vertex operations required to transform, light, clip test, project, and map vertices from object-space to screen-space [9].

Whether graphics-bound or CPU-bound there is a balance between the CPU speed, the scene complexity, and the graphics hardware. For an application that is CPU-limited, reduce the number of calculations required by reducing the scene complexity. If an application is graphics-bound, in addition to the options given above for the GTXR-D adapter, one way to improve performance would be to use a smaller graphics window, reducing the scan conversion area. Another improvement would be to reduce depth complexity and the number of times a pixel is drawn. This is accomplished by not rendering objects that are occluded in the final image (see Section 7.2).

GT-XRD As shown in Figure 4.5 even though much of the rendering burden has been removed from the CPU, an application can still be CPU-bound if the CPU is totally consumed packaging and sending down data to render. A more common scenario is the graphics FIFO fills up causing the CPU to stall while the graphics subsystem performs all the transformation, lighting, and rasterization.

An application that is CPU-bound might be performing some calculations that could be performed more efficiently on the specifically designed graphics hardware. If so, offload these tasks to the graphics subsystem. For example, ensure that the application is using a lighting model implemented in the graphics hardware. Another example is to use the graphics hardware to perform matrix operations as GT-XRD hardware is designed to efficiently perform matrix multiplication operations. Do

this by specifying these operations with the graphics API, and let the dedicated graphics hardware do the required calculations, thereby freeing the CPU to perform other tasks.

For graphics-bound applications, consider moving some of the eye-space lighting or shading calculations back to the host CPU, or packaging the data into formats that are more easily processed by the graphics subsystem. Try using display lists or compiled vertex arrays to limit setup time required by the graphics hardware. Also, try reducing lighting requirements may reduce the computational complexity of the lighting calculations.

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

Numerous techniques can be used to determine if the performance of an application is bound by the CPU or by the graphics subsystem. Use these techniques before trying more complicated tools.

- Shrink the graphics window. If the framerate improves then the application is fill-limited as the overall performance is limited by the time required to update the graphics window. Shrinking the graphics window effectively shrinks the viewport. This shinks the size of primitives and reduces the fill requirements. Before doing this test, ensure that the behaviour of the application does not change. Some applications will change their behaviour and send down less polygons when the graphics window is made smaller. This behaviour invalidates the test as not only are the fill requirements reduced, but the geometry requirements are reduced as well.

- Reduce geometry processing requirements. Render using fewer/no lights, materials properties, pixel transfers, clip planes, etc. to reduce 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 lighting and geometric processing is performed on the host, then an increase in frame rate in this case is typical of an application which is CPU-limited.

- Remove all graphics API calls. This establishes a theoretical upper limit on the performance of an application. The quickest way to do this 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 and in either the generation or traversal phases. Keep 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 fallen off the fast path and has become CPU-bound doing software rendering. Often, a simple state change can cause this. This is actually a common problem on GTX-RD and GT-XRD subsystems where not all rendering modes are implemented in hardware.

Figure 4.6 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.

4.4 Bottleneck Elimination

Armed with a thorough knowledge of the system architecture, and knowing if an application is CPU-bound or graphics-bound, you can turn to analyzing the application code. This process includes analysis of hardware fast path utilization and identification of bottlenecks.

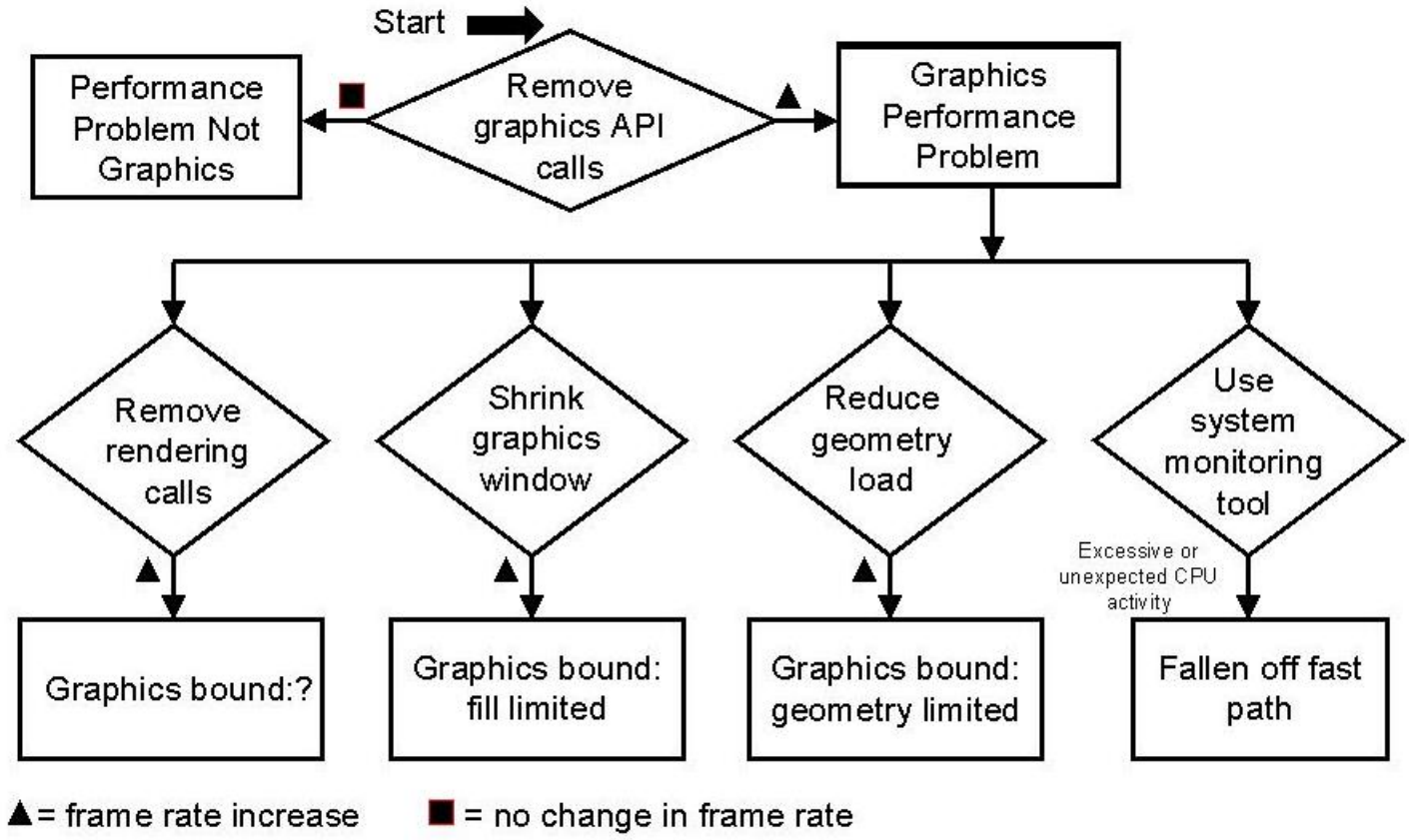


Figure 4.6: Graphics performance analysis procedure.

4.4.1 Falling Off the Fast Path

Most graphics hardware has a fast path that draws a subset of rendering operations much faster than others. These rendering operations are usually based on the primitives and modes directly supported by the underlying hardware. Using a primitive type or rendering mode that is not directly supported by the hardware causes the graphics rendering pipeline to fall-back to a software path, which can have a significant impact on rendering performance. Common examples include anti-aliased polygons, anti-aliased wide lines, and local lights sources. As described in Section 4.3.5, unexpected CPU activity is a sure sign that an application has fallen off the hardware fast path.

Know the hardware fast paths and stay on them whenever possible. One technique to determine hardware fast paths is by reading vendor supplied documentation. If your vendor does not supply fastpath documentation, or it is unclear, ask the vendor to supply this information. When targeting more than one platform, use a least-common denominator approach to stay on the intersection between the different hardware fast paths if possible. Another method for determining hardware fast paths is by using GLPerf or a similar test suite or test program as described in Section 2.2.3. However, be careful when using a test program not to introduce other overhead that may invalidate the results.

Armed with knowledge of the hardware fast paths, examine the graphics state and modes used by the application to determine if the graphics hardware is effectively utilized. Use the knowledge gained in the previous paragraphs or re-examine the source code again to determine if the hardware is effectively utilized. Later, tools will be described that may make this task easier. 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 hardware rendering features.

4.4.2 Identifying Bottlenecks

Understanding the potential and actual bottlenecks is crucial to effectively tuning an application. There will always be bottlenecks within an application which limit the performance of the system. The goal of tuning an application should be 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 isn't always a negative. Sometimes, you can take advantage of a bottleneck and use the time for the bottleneck to clear to perform other tasks. Sometimes this waiting can be used to actually 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. Listed below are a few common causes of bottlenecks within an application categorized by the subsystem in which they occur. All the bottlenecks listed here affect graphics rendering performance, regardless of the subsystem within which they occur.

Graphics

Bottlenecks are most common in the graphics subsystem. Where, when, and how severe a bottleneck is depends largely upon the combination of hardware and software which implements the graphics pipeline.

- **Non-native graphics formats.** Pixel and texture data that is not in a format native to the graphics hardware must be reprocessed by the graphics driver to repackage it into a native format before it can be rendered. An example of this might be the conversion of RGB data to RGBA. This increased

rendering overhead could create a bottleneck within the system. A list of native data formats can be found in the graphics hardware documentation.

- **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 as 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 actually 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). Avoid redundant state calls and cache important state information within the application. For example, one way to sort data may be:

  - By transform.
  - By lighting model(one vs. two side, local vs infinite, etc).
  - By texture.
  - By material.
  - By primitive(triangle strips, quads, lines, etc).
  - By color.

  However, exercise caution and don't 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 receive graphics data and attribute information from an application and render the resulting primitives according to the current state settings. Avoid querying the pipeline for state information as this will break the inherent pipelining and cause the graphics subsystem to stall. Cache important state information within the application.

- **Inefficiently packaged graphics primitives.** Render like primitives together, combining them into strips if possible, to reduce the rendering overhead of setup time in the graphics subsystem. One benefit is that strips are more likely to have good cache behavior. A second benefit is that the graphics driver and hardware can often pipeline rendering of strips.

- **Texture paging.** Textures which do not fit in texture cache on a graphics subsystem must be transferred to 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 which do not fit in the local texture cache can be a bottleneck. The AGP architecture attempts to solve this problem by providing a high-speed dedicated bus for the transfer of texture data from system memory to graphics memory. UMA systems also provide support for large textures by implementing the frame buffer directly in system memory. In the case of UMA, no copy of the texture data is required for rendering.

  Programmatic solutions to this problem are to use mipmapping or clipmapping, as described in Section 7.3.3. Another solution in OpenGL^{TM} is to use the texture LOD extension to reduce the

resolution and the texture size until a texture fits into texture memory. Use `glAreTexturesResident()` to verify that all textures are in memory before rendering.

In general, with textures, use texture objects and load texture images once for an application, not once per frame. Texture objects allow the graphics pipeline implementation to compile the texture into a form for optimal rendering and texture management. Using texture objects with ensure that textures are stored as efficiently as possible. If texture objects are not an option, consider encapsulating texture commands into a display list. Avoid unnecessary switching between textures by rendering primitives which use the same texture together.

Although texture objects optimize texture storage and management, there is a limit to the number of texture objects available. To stay within the limit of the number of texture objects available, combine multiple small textures into one large texture as a mosaic changing the texture coordinates accordingly to map into the larger super-texture. An alternative solution in OpenGL^{TM} may be to use `glTexSubImage*` routines to redefine part of an existing texture object.

- **Lighting model characteristics.** Using unnecessary lighting model characteristics such as two-sided lighting, and local viewer adds unnecessary complexity to the lighting model and the geometry processing required to render a scene. Bottlenecks of this type can occur in either the graphics subsystem or the CPU depending upon where the lighting model is implemented as described in Section 3.2.4.

  When a local viewer is specified, the calculation of the specular term in Equation 3.7 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^{TM}, 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^{TM} by setting `GL_LIGHT_MODEL_TWO_SIDE` to `GL_FALSE` in `glLightModel`. In order to use one-sided lighting effectively, all normals must be consistent with respect to the geometry. In otherwords, all normals must point either "out" or "in."

  Ensure that all lights, and the physical characteristics of those lights are required and add to the overall visual quality of the scene. For example, using directional or infinite light sources rather than local lights removes the per-vertex calculation of the attenuation factor in Equation 3.4. In OpenGL^{TM}, infinite or directional lights are specified with the fourth coordinate of `GL_POSITION` set to 0.0. Remove lights which don't add to the visual clarity of the scene. Each additional light requires evaluation of the lighting model equation at each primitive for flat shading and each vertex for Gouraud shading.

- **Normalization.** Normalization within the graphics rendering pipeline can create a bottleneck on the CPU or in the graphics hardware depending upon 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. In OpenGL^{TM} do this by disabling `GL_NORMALIZE`.

- **Rasterization and per-fragment operations.** Using rasterization operations which your application does not require increase rendering overhead and create a rasterization bottleneck. Rasterization

operations such as texture, fog, antialiasing, and other per-fragment operations (blending, depth, stencil, scissor, logic operations and dithering) as discussed in Section 3.2.10 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.

Examine application code to ensure that all rendering states enabled are required to achieve the resulting 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, buildings, etc. In another example, if low-quality texturing is acceptible, use only bilinear filtering instead of trilinear.

- **Geometry.** Processing large amounts of geometry (that is, lighting, transformations, etc.) can cause a bottleneck even with 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 specified at satisfactory frame-rates. Lessen geometric requirements — if you can't see it, don't draw it. See Section 7.2 to learn about various culling techniques.

  Consider using billboards to replace complex geometry as described in Section 7.2.3. Textures can also be used to implement approximate per-pixel lighting models for hardware which does not support per-pixel lighting. 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 see immediately how to apply it within an application. See [35] for further description and ideas.

- **Depth complexity.** Consider how many times the same pixels are filled. Avoid drawing small or occluded polygons by culling unseen or insignificant geometry as described in Section 7.2.

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 of graphics-specific issues warrant discussion here.

- **Function overhead.** A common cause of bottlenecks is function call overhead on the transfer rate between the host and graphics. 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 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, 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, leaving the graphics subsystem underutilized.

  Avoid these types of bottlenecks is fortunately 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 to reduce function-call overhead. Use vector arguements 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 state is set to the same value as is already current. Don't send state information which has not changed to the graphics subsystem.

- **Vertex formats.**

  When using vertex arrays, consider using interleaved and precompiled vertex arrays. Interleaved arrays allow for multiple arrays to be specified 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 using DMA.

  In the case of static geometry that is drawn many times, consider using a display list. A display list permits fast and simply traversal and may improve cache coherency. However, within a display list, for optimal performance, don't 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 over immediate mode rendering. One final caveat with display lists is to understand how nested display lists may create memory fragmentation and cacheing problems that will 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.

- **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 caused by multiple threads waiting to access a single graphics context, or multiple graphics contexts waiting for access to a single graphics device. Application programs that 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 guard against multiple threads 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 which performs the **G** and **T** stages of rendering prior to handing 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 4.7 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.

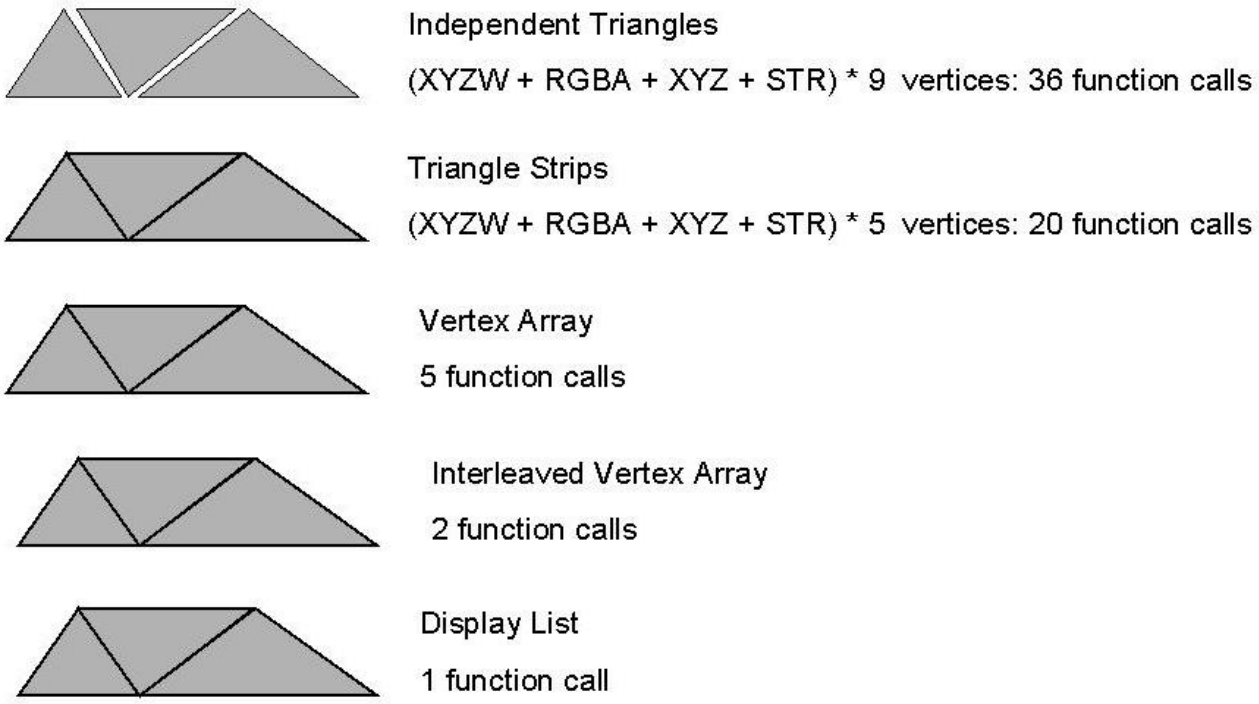


Figure 4.7: API Call Overhead.

Memory

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 causing rendering to stall until the system call returns and user mode state is restored. Allocate all memory for graphics primitives before beginning the rendering loop to prevent stalls of this type.

- **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, don't copy individual X, Y, and Z coordinates into a vector to make a graphics API call when the coordinates can be sent down 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 to allow use of vertex arrays. Vertex array code 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 using DMA.

- **Memory fragmentation.** Sparsely packed data causes memory fragmentation and as a result, poor cache behaviour. Avoid memory fragmentation by allocating memory for per-vertex data from a preallocated pool. This reduces expensive memory paging operations when traversing graphics data.

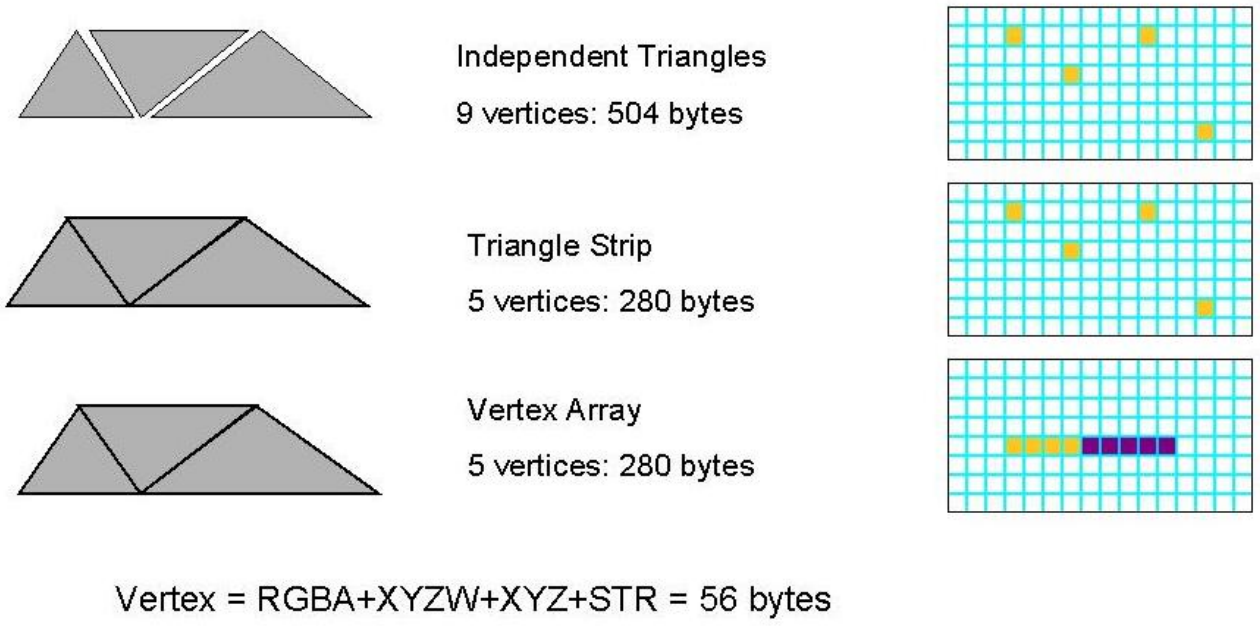


Figure 4.8: Memory Bandwidth and Fragmentation.

Figure 4.8 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 while using triangle strips, or a vertex array to render the same 3 triangles requires only 280 bytes of data. This reduces the required memory bandwidth by 45%. In the vertex array case, vertex data is contiquous in memory thereby reducing page faults and subsequent memory paging as the data is traversed.

CPU

Another common place to uncover system bottlenecks is the host CPU. This is especially true on systems which implement a large part of the graphics rendering pipeline in software. In this case, the most common bottleneck will be geometry processing as the CPU performs all transform and lighting calculations. To remedy this situation, follow the suggestions under **Geometry** in Section 4.4.2.

Disk

The inefficient storage 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 trying 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. Unfor-

tunately, 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. Do this by examining all the API calls between buffer swaps or screen updates. Be on the lookout for repeated calls to set graphics state and rendering modes between primitives. Tools such as OpenGL debug (see Figure 4.9), APIMON (see Figure 4.10), ZapDB, and others provide these capabilities.

4.5.2 Application Level

Profile the application program to determine where the most time and/or CPU cycles are being 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 caused by the application program or perhaps an external factor. This aids in the identification of 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.

- **Page Faults**

  A large number of page faults, indicating that a process is referring to a large number of virtual memory pages that are not currently in its working set in main memory, 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.

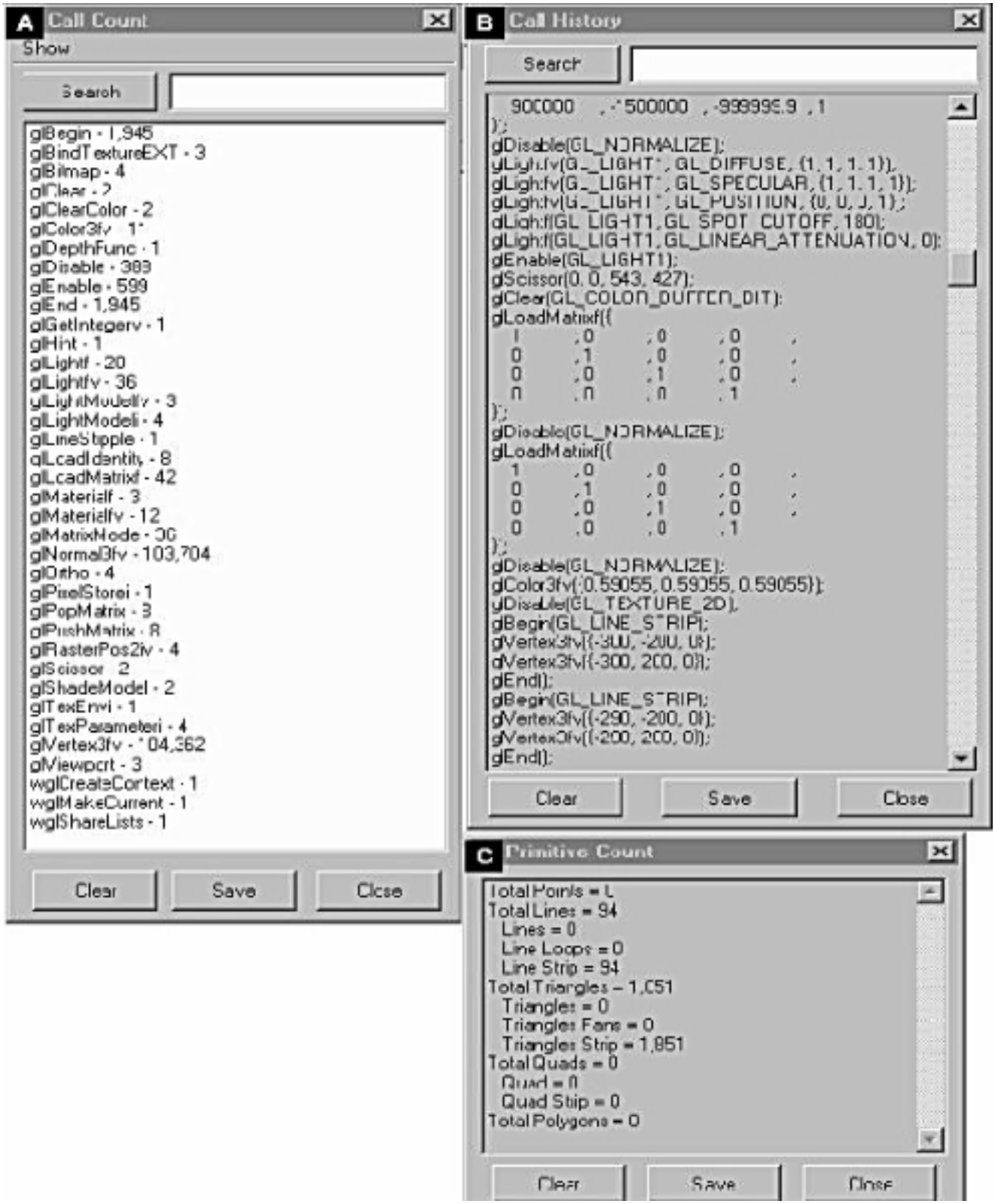


Figure 4.9: Sample output from ogldebug, an OpenGL tracing tool. (A) Call count output from a running OpenGL application. (B) A call history trace from the same OpenGL application. (C) Primitive count output from the same OpenGL application.

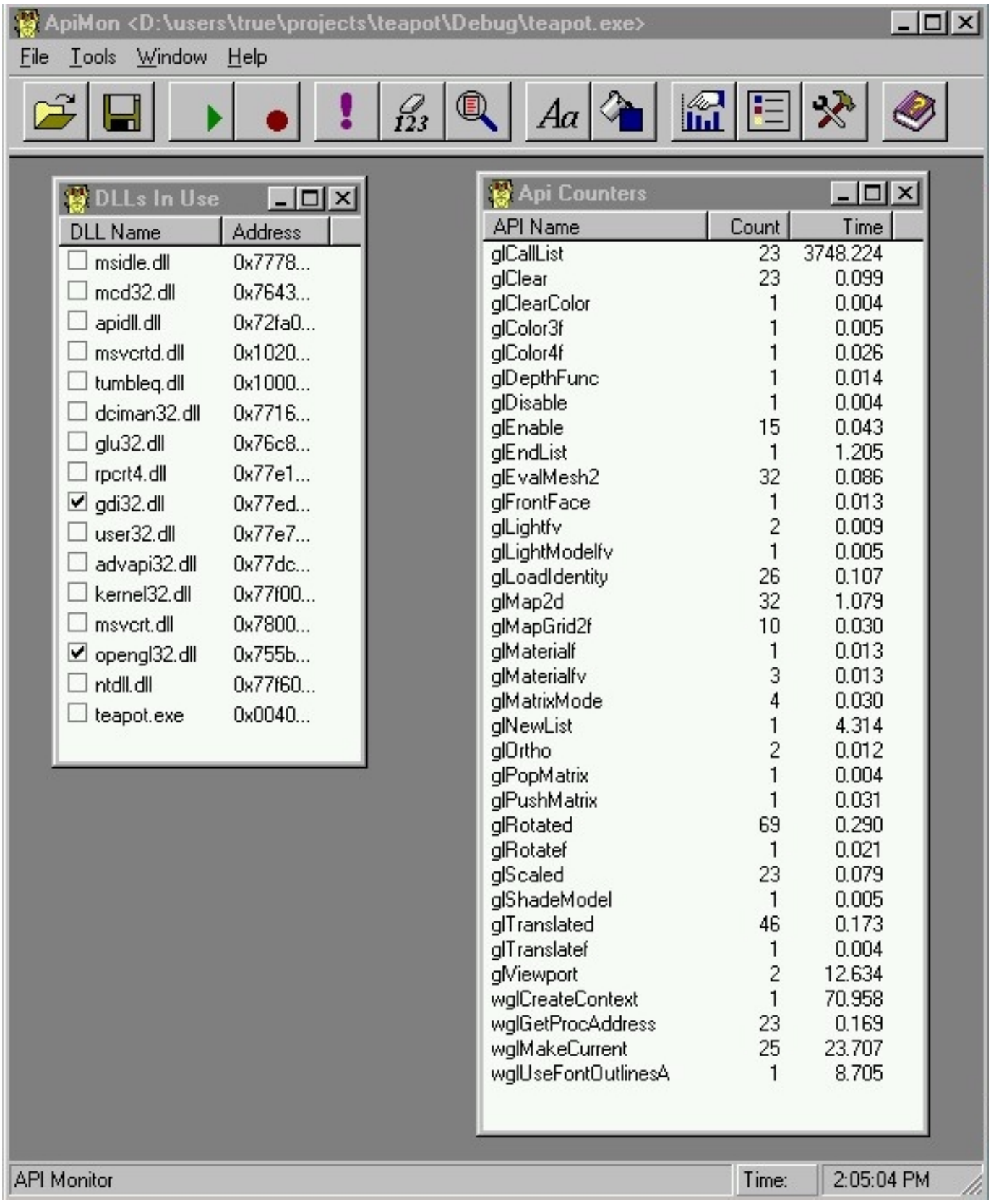


Figure 4.10: Using APIMON to trace graphics API usage.

Because tools differ by platform, it is impossible to adequately describe them here. More detail is presented in the next section but, in general, a developer should know the tools available on their 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. One key concept in graphics tuning is to try 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 performance of a graphics application. Applying the tuning procedures and tips described in this section to a graphics application will yield a more complete understanding of the graphics pipeline, application usage of that pipeline, and, after tuning, better utilization of that pipeline for faster application performance.

Section 5

Profiling and Tuning Code

By this point in the course, overall system graphics performance has been characterized, tuned, and can perform at an acceptable level. Now what? The next step is to profile the code, which simply means using system tools to identify the slow parts of application software. These tools insert extra counters in the executing assembly stream that track sections of code as they are executed. Analyzing the data output from these counters reveals where the code bottlenecks lie. The data generated from these counters can measure many aspects of application performance such as the number of times a line of code was executed or the number of CPU cycles taken to execute sections of code. You can use the results of this analysis to rewrite, or tune, slow sections of software, once the slow parts are identified.

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 others. Differing 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, one with 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 isn't difficult, but is absolutely necessary. Although it takes some time to become an expert at both generating and interpreting profiling data, the basics are simple to master. Profiling is essential to perform, because no amount of tuning the graphics code will improve performance if the work the application is doing between frames is the bottleneck. Profiling identifies slow sections of application code and helps direct tuning efforts. A well-tuned application is key to overall fast graphics.

5.2 System and Software Interaction

A necessary step before profiling software is to know 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? 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 high relative to the user-time, check your system calls.

In a well-balanced application, the system-time is a fraction of the user-time.[1] Not all system function calls are expensive, of course, but understanding the effects of a system call is important before using it. Similarly, libraries or utilities, which in turn execute system calls, for example, memory allocation functions, need to be handled carefully. Don't 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 themselves allocate memory; understand the work being done by the libraries in an application, 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 transfer of data (see Section 4.3 for more detail.) The queue can force a CPU stall if it becomes too full, thus stalling program execution. The state of the queue (stalled, full, or empty) during intense graphic activity can tell you if the host is flooding the graphics pipeline. Use tools described in Section 4.1.1 to gather data on the FIFO usage.

Additionally, some systems and CPUs indicate the amount of swapping or cache misses that are occurring while the program is executing. If swap activity is high, then more physical memory or better utilization of the existing memory is needed. If cache misses are high, then better packing of data in memory is needed. An analysis and re-write of the code is necessary to determine where to do this tuning. 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 will thrash even the largest cache. Use profiling to identify the offending code and re-write it.

5.3 Software Profiling

Once the code and system interaction is understood, the code is ready to be profiled. There are two basic methods of code profiling: *basic block counting* and *statistical sampling* using either the *program counter* (PC) or *callstack*.

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*) a section of code can achieve, regardless of how long an instruction might have taken to complete. Therefore, basic block profiling doesn't account for any memory latency issues. PC or callstack profiling uses statistical sampling to determine how many cycles or how much CPU time is actually spent executing a line of code. Statistical sampling indicates where the CPU spends its time and can be used to locate memory latency issues. Both basic block and statistical profiling reveal bottlenecks in code; the two methods tend to show slightly different results, however, and therefore it is important that both analysis be completed.

Be careful when profiling and debugging 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 which is in a state identical to that used to ship to customers) and not on debug code.

5.3.1 Basic Block Profiling

It is fairly simple to profile code (Figure 5.1). After the executable has been compiled and linked, an external tool instruments your code[2]. The next step is to run the instrumented code with a relevant data

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

[2] On SystemV UNIX, `prof` and `pixie`; on NT, Microsoft's `VisualStudio`

set and usage scenario. Choose the data set that best represents typical customer data. Run the software when profiling 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. Another consideration when profiling is that the execution of instrumented code can take significantly longer to complete. Running the instrumented executable produces a data file with timing results which 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.

Consider the example `foo.exe` shown in Figure 5.2. This example has two functions of interest, `old_loop` and `new_loop`, which add up and print the sum of all the values in array `x`. A third function, `setup_data`, is only used to set up the data, which we will ignore for now. The function `old_loop` (Figure 5.2A) is the original function prior to profiling, and the second function (Figure 5.2B), `new_loop`, is the improved function resulting 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 5.2: 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 5.3 provides the output for the test run. The function `old_loop` took 6168 cycles to complete. Now the fun begins — analysis of why the code is "slow" and how to make it better. How could this be rewritten to run faster? Notice that `old_loop` (Figure 5.2A) is basically one large loop and nothing else. If you unroll the loop, and call the function `new_loop`, it now looks like Figure 5.2B. (More about loop unrolling in Section 6.4.4). After re-profiling the new executable, the analysis (Figure 5.3B) shows that `new_loop` only takes 4625 cycles, a savings of 25%.

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 5.3: 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.

In addition to the amount of time each function takes, the analysis can tell you the lines of code that are repeated most often. The second part of the report (Figure 5.3C) 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 1024 times each. (This makes sense because the code was written that way.) Approximately 2 cycles per loop invocation were used by the loop overhead, and 4 cycles per loop invocation for the loop body. In the `new_loop` function, the loop body took 4615 cycles ($978 + 3 * 968$) to execute — a little more than with `old_loop` (4096). However, the loop overhead dropped from 2061 cycles (`old_loop`) to 733 (`new_loop`) because it is 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 5.4) show that `new_loop` beats `old_loop` by about 40 percent.

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

5.3.2 PC Sample Profiling

Basic block profiling counts the number of times a block of code was run, but does not record the amount of effort, or CPU cycles, needed to complete the block of code. PC sampling counts the number of cycles used, which is a measurement of the amount of effort required to execute a line of code. PC sampling therefore provides another useful analysis tool to determine where to tune application code.

Figure 5.5 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 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 5.5A.

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 5.5 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 5.5B). However, under PC sampling analysis, a different behavior (Figure 5.5C) is evident. The PC sampling analysis shows that the `loop_ijk` is much more efficient than `loop_kji` due to 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 taking the most CPU cycles, whereas basic block analysis points out the number of times 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 where additional care must be taken when building the data structures and code to compensate for memory latency.

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 5.5: Example PC 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 Issues

A good developer writes good code based on abilities honed throughout the educational process and work experience. However, a good software developer should also make effective use of the available tools such as compilers, linkers, and debuggers. While basic education is best left to the universities, this section shows how effective use of a compiler can greatly increase the overall performance of graphics software. This section concludes by addressing a variety of language considerations when writing software in either C or C++.

6.1 Compilers and Optimization

User guides, online documentation, and manual pages are essential tools in a developer's toolbox. However, surprisingly few people thoroughly read and use these resources. 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 could be enabled or disabled. Numerous optimizations for loop unrolling, processor architectures, error handling, etc., could 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 among these concerns 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 due to loss of needed precision. Compilers may gain a great deal of performance by interprocedural 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. 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 possible increase of code size. Some optimizations even require multipass compiles on the same source code. Are they worth it? Experiment on 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 consideration is that different compilers on different platforms come with different levels of quality and kinds of optimization. Study the compiler documentation carefully for insight into how certain optimizations perform and change the way code is generated. Often code compiled with debug information exhibits different bugs than code compiled with optimizations.

Discovering and working with optimizations can be well worth the effort. Consider the commonly known 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 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, 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]Interprocedural 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 is most often due to a problem in the code, and not in the compiler. Perhaps an inherently incorrect statement was used or one that doesn't 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 due to a change in arithmetic roundoff as a result of rearranged lines of code. The author has to make the final decision about each optimization, 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 OpenSource arena. Always check 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, allowing 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 potentially leading to better performance.

None of the advantages of 64-bit applications come without potential overhead. The memory space required by applications will increase due to the expanded data type sizes and additional alignment constraints. Additional performance may be elusive, and performance may actually degrade due to the additional data being pushed around the system. Note that OpenGL data is built on type *float* data, not type *double* data, thus leading to a conversion and performance loss when *double* data is specified to OpenGL

routines. Some applications require the extra precision of 64-bit parameters; this extra information may be lost when displaying data using OpenGL.

6.3 User Memory Management

Careful placement of objects in memory can lead to very efficient application operation. This result is due to the data access speed improvements associated with data accessed in L2 and L1 caches. These two caches typically access data an order of magnitude faster than data in main memory, so keeping data cache-resident is an obvious performance improvement. This section of the course discusses what a developer can do to increase the likelihood of data residing in cache.

A quick survey of common data usage scenarios is the most effective means of determining what is necessary to allow data to reside in cache in those situations. One primary data structure 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 (unlike arrays that are a continuous segment of memory) data structures in 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 involve a developer managing 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 pre-creates a number of list elements close together in memory, then hands them to the application when a new one is required. This pre-allocated set of elements is known as a *pool* and is managed explicitly by a set of routines 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 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 things can be done in C++ with class-constructors and 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 greatly increases the likelihood that list elements will reside next to others in cache. Note that it does not ensure that elements will 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 allocated to doing the list management. One issue to consider is how much pre-allocation to do of list elements. If too many are allocated, overall memory requirements for the application may be increased, yet performance improved. If too few are allocated, then as the store of pre-allocated elements is exhausted, another segment will have to be allocated, taking time when the application expected a simple `create` was being issued and returning to the memory fragmentation case that the pool was trying to solve in

the first place. Again, it's 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's important to pursue changes that will most affect the application being tuned, so 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 which 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 spend effort tuning those.

6.4 C Language Considerations

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 while writing C code. Doud [16] also details a number of source code optimizations.

6.4.1 Data Structures

Data structures are essential to any application, including graphics applications. While writing and manipulating efficient data structures doesn't directly affect graphics *per se*, 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.

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`, will be 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 easily be rearranged, as shown in Figure 6.1B, so that when `next` or `previous` is referenced, `key` is likely to be cached in as well. Since `next`, `previous`, and `key` probably are only several bytes each, they should all fit in most cache lines. Thus, the reference to `key` is then much more efficient. This optimization improves cache effects only for a single record at a time since there is still the large `foo` data structure in between each of the `next` pointers. When traversing the list, it is likely that only a single `next` pointer will be brought into cache for each lookup. Allocating the `foo` data structure outside of the list and using a pointer to `foo` inside the list enables much more cache friendly searching and access to the data only a pointer reference away. Naturally, the size of cache lines changes the effectiveness of this optimization.

6.4.2 Data Packing and Memory Alignment

Understanding how your compiler arranges data structures in memory is an important aspect of writing efficient code. On some platforms, compilers may attempt the optimizations described in this section on behalf of an application developer. However, in the interest of achieving performance in a portable fashion, it's important to consider memory issues when developing data structures.

A computer uses one simple rule when organizing data in memory. This rule states that data larger or equal to a magic size must be placed on boundaries of that magic size. The magic size, referred to as

A
```
struct {
    str *next;
    str *prev;
    large_type foo; // lots of user
                    // data structures
    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 user
                    // data structures
} 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.

the alignment size, is typically the size of the largest basic-type (such as `float` or `double`). Units of data smaller than the alignment size can be placed on sub-alignment-size boundaries, and units of data 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 equal-to-or-larger-than-word-sized data 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 in together, resulting in better cache coherency. As access to data in cache is an order of magnitude faster than access to data in main memory, the program will therefore run faster. Customers also like this for obvious reasons.

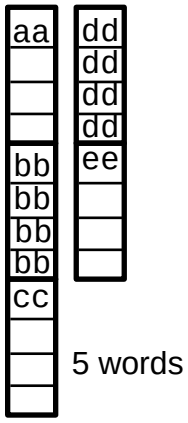
6.4.3 Source Code Organization

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 address space as the rest of the application data, and is therefore subject to the similar issues surrounding paging and caching (as described in Section 2.1.6).

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

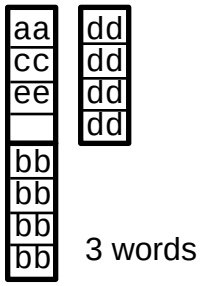


Figure 6.2: Example of how data structure packing affects memory size. (A) A non-packed data structure `foo`. (B) Memory space used by the `foo` data structure illustrating the wasted memory. Because the data structure is not packed, the 8-bit characters result in a waste of 16 bits, each resulting in a total space of 5 words. (C) Packed version of the data structure shown in A. (D) Memory space used by the `foo_packed` data structure. The packing allows all three characters to be placed in the same word resulting in only 8 bits of wasted memory and a total space of 3 words.

Tools are available on some platforms that rearrange procedures automatically. 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 used to generate these feedback files need to be chosen carefully because they heavily influence the overall effectiveness and relevance of these tools. Much as when profiling applications, choosing representative data is the most important factor to consider. 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 Unrolling Loop Structures

Another common optimization technique is known as loop unrolling. Consider the function `old_loop` shown in Figure 6.3A demonstrating a conventional loop consisting of the loop overhead (line 21) and the loop body (line 22). 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 checked against `NUM` for every loop iteration. The resulting modified loop, `new_loop`, is shown in Figure 6.3B consisting 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. Of course, since `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. 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 which is known to always be 1024 long. 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 5.3, the amount of work completed by the code segment took 6168 cycles. By reducing the loop overhead relative to the amount of work accomplished, the improved code took 4891 cycles, resulting 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 is small, the amount of savings is likely

to be negligible. Loops containing function calls should also be ignored as they are likely to be expensive. 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, as code is duplicated, loop unrolling can increase the code size. Last, the compiler may already have loop unrolling enabled as an optimization, and the compiler's work may obviate the effects of a manual unrolling.

6.4.5 Arrays

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 sub-arrays. The size of these sub-arrays can be designed to reside within either L1 or L2 cache. Another example is to consider a loop that walks down columns in an array. If each row is aligned such 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 accessed much more quickly. Try swapping row and column access for large array manipulations to see if performance improves. Note that different compilers place arrays differently in memory, so verify how the specific compiler being used allocates array memory. As Section 2.1.5 points out, data access to array elements in cache is far faster than those from main memory.

6.4.6 Inlining and Macros

Another issue in writing efficient software is that of small functions of one or several lines in length. Much like loops, the overhead in accessing a function must be offset by the work 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 using 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.

6.4.7 Temporary Variables

Another common optimization technique is to use 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 6.4. 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, it's best to 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 6.4: Example of optimization using temporary variables. (A) Original code. (B) Optimized version.

Figure 6.5 demonstrates how, within a function, a temporary variable, `tmp`, can replace several references to a global pointer, `new_pt`. References to global variables may induce caching penalties. The

substitution demonstrated results in better caching behavior and increased performance, which can result in up to a 50% faster loop with some compilers.

A

```
void tr_point1(float *oldPnt,
               float *m,
               flot *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 6.5: Example of optimization using temporary variables with a function. (A) Original code. (B) Optimized version.

6.4.8 Pointer Aliasing

In C and C++, pointers are used to 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* [14]. 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 use of those pointers by reordering or parallelizing the code. However, if it is known that the two pointers never overlap (be aliased), significant optimization can be accomplished.

Consider the code example from Cook [14] (Figure 6.6A). 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 referenced by `p2`. Therefore, any store through `p1` can potentially affect memory pointed to by `p2`. This prevents the compiler from taking advantage of instruction pipelining or parallelism inherent in the CPU. Loop unrolling may help here, but there is an even simpler solution in this case.

Optimally, the compiler would generally 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 not. However, the Numerical Extensions Group/X3J11.1 proposed a new keyword, `restrict`, for the C language to solve this problem [43]. The `restrict` keyword is used to indicate which pointers are aliased and which are not. The `const` keyword in C++ provides a similar capability by telling the compiler that certain variables will not be modified. Using `restrict`, the code in Figure 6.6A would

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

be rewritten as shown in Figure 6.6B. Cook [14] states a 300% performance improvement using this technique in his example over the original code. In addition, adding this to the code and recompiling is a much simpler and faster change than unrolling the loop.

6.5 C++ Language Considerations

This section describes a few performance issues to be aware of when designing and coding in C++. C++ provides many efficiencies in design, architecture, and re-use aspects of software development but also has associated performance implications to consider when implementing your designs.

6.5.1 General C++ Issues

There are only a few major issues to consider when writing C++ software, but there are many little issues 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 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, invoking a copy constructor, a potentially expensive operation. When passing arguments to functions, use references instead of passing arguments by value.

There are many other small 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 writes to read-only objects are detected at compile time. 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 when using 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.

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 only there. Virtual functions are implemented essentially as function tables stored within a class instance defining 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 using virtual functions.

Since the virtual function table is stored within a class instance in memory, there is associated memory overhead for this table. The increase in size of an instance means that an instance takes up more space in main memory and is similarly going to require more space when cached. Using more space when cached implies that less data overall can be in the cache, and, 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 2.1.6. 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 over a larger amount of work performed in that function. Using a method 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 has some important 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) in order 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 compiled with exceptions, but perhaps not even using them, to be slower. Compilers may

also not be able to optimize code as significantly with exceptions enabled. Catch exceptions that are not basic types by reference to reduce the number of copies made of exception objects. Use exceptions only to handle abnormal conditions — their overhead is too great for common error handling. Understand the implications of exception use for the operating systems and compilers used to build an application.

6.5.4 Templates

Templates are another language feature of C++ allowing high levels of code re-use. Templates preserve type-safety while allowing 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 only having to implement efficient code once. Templates can be difficult to debug, but are easily implemented originally as a concrete class, then templatized after they have been debugged. Another solution to efficient template usage is to use commercial libraries or the Standard Template Library (STL[1]), now part of the ANSI language specification. Extensive use of templates may cause code expansion due to techniques used by compilers to instantiate template code. Read compiler documentation to learn how templates are instantiated on a particular system.

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

Section 7

Graphics Techniques and Algorithms

7.1 Introduction

In this course, you've learned some 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 following metaphor for this section is this: "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 some additional techniques for improving application performance and show 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 differ sometimes 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. There are many more domains, each having their 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 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.

7.2 Idioms

idiom: The syntactical, grammatical, or structural form peculiar to a language [56].

The language of the computer is very specific — one misplaced symbol, and the computer no longer does what is expected. When that language is used for a graphics application, similar, although not as catastrophic, results can follow. For example, an application might not meet the needs of the users if it is not architected properly. There exist many idioms that help in architecting a graphics application, and

these idioms generally take the form of reducing the information that needs to be rendered. The basic premise of these idioms is that an application only needs 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 there exist precious few applications 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 does get 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 causes the amount of time available for non-rendering tasks to be reduced, which adds more geometry load to the graphics system due to less time to cull and calculate proper level of details, and so on, creating the feedback loop. Therefore, when using culling and multiple levels of detail, it is also necessary to have a frame-rate control mechanism that can balance the graphics and CPU load.

7.2.1 Caching

Caching is the well-known technique of locally storing data which 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 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 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, removing the need to retesselate. Both types of display list can contain meta-information such as bounding boxes to enable other optimizations. Since display list generation and editting take time, display lists are best for caching static geometry that will 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 will provide a useful performance boost at a reasonable cost.

Data caching - Paging, Tiling, and Bricking

Many applications roam through a large database stored on disk. The database may contain geometry, or in other cases, image or volume data formatted as texture. It can be worthwhile to organize the database spatially, and create a cache for data 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, since 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 frame buffer, 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 balanced 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 is best for environments where overlay planes are unavailable or where the frame buffer must be shared with other applications which do not use the system overlay planes.

7.2.2 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 [20, 58, 57, 12, 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 be tested. 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. Such objects include pens, trees, missiles, and railroad cars that 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 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. Ione*s, 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 GTX-RD 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), which are backfacing (not visible), and eliding those that are backfacing is called *backface culling* [58]. 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 (see section 3.2.6). This only eliminates rasterization, which is not helpful for applications bound

by transformation and lighting. In GTX-RD graphics systems, it can therefore 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 point regarding 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 not behind other objects or 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, and 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 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 & portal [54] culling algorithm to make a map of the visibility between cells. Only cells 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 [59, 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 feedback the depth buffer hit information outside of reading the full depth buffer. Check with your hardware vendor when implementing an occlusion culling algorithm to see if there are extensions that allow efficient occlusion culling algorithms.

More detail on occlusion culling can be found in Zhang [57], which covers occlusion culling background material and an extensive algorithm for choosing the optimal occlusion set.

Contribution Culling

Another area where you can use culling is 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* [57], 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 where 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, since objects with low pixel coverage are not good occluders.

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

7.2.3 Application specific heuristics and combinations of idioms

Caching, culling, and levels of detail are powerful techniques for accelerating the performance of a wide range of applications. Other techniques may be useful in particular circumstances. Some examples follow.

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 over 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(see section 3.2.9) 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 required for blending can add large amounts of data generation time to the GTXRD pipeline. The multiple renderings 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 frame buffer available, it can be worthwhile to implement antialiasing by *oversampling*. In this technique, the image is rendered once, in a large offscreen buffer, then filtered into the smaller visible window with an appropriate (and fast) kernel. When compared to blending, oversampling trades off frame buffer 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 frame buffer 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 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 a coarsely tesselated geometry for a finely tesselated 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. Since 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 requiring a large amount of frame buffer for 3D operations that conserve frame buffer 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.

7.2.4 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 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 [12, 26]. When an object is far from the viewer, fewer triangles need to be devoted to rendering the object to retain the same image fidelity.

There are many types of models 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 [38] or general adaptive algorithms if they allow the decimation factor to vary over the model in a view-dependent fashion [13, 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 [13] 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 framerate [20, 45, 49]. 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 [20]. In many cases, the geometry can be totally replaced by a textured image [49], thereby reducing the geometry load down to a single polygon.

Creating the LOD Models

Geometric models come from many sources and can vary by many orders of magnitude 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 from parametric surfaces generated from a modeling package. Clark [12] first proposed the use of simplified models as a means of increasing frame rate while rendering interactive applications. Since then, geometric surface simplification has been a strong research topic for many years. 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 only shared between 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 [53, 46, 15], feature methods [51], refinement methods [19, 28, 44, 21], decimation methods [37, 47], and optimal methods [7]. Many of these algorithms are very computational and therefore can only be used 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]. That is, 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 allowing 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.*[38] 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, leading 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 in order 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 incurred by the approximated geometry of the mesh. That is, there should exist a consistent and direct relationship 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 [45] library's Active Surface Definition (ASD), Linstrom's [38] 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 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, dependent on the viewer velocity and direction. This allows an infinite surface to be convincingly rendered. Furthermore, both ASD and Hoppe geomorph the vertices as the LOD level changes. This allows a very 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 [55, 48, 17, 29, 13, 30, 39] 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, 39], and include view-dependent information [13, 30, 39], and attain higher compression rates for low-bandwidth transmission of data [11, 52, 24].

Determining Which LOD to Use

Generating multiresolution models is only the first part of what is needed to effectively use LODs in an application. The second part of the problem is to decide when to use which LOD level [20, 45]. This is a very important problem 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 a 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 is dependent on the application domain. A visual simulation may need to run at 30 Hz or 60 Hz while 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 allowing an application to become arbitrarily slow.

Often, application developers that have not incorporated frame rate control into their applications rationalize the decision by saying they always want the fastest frame rate, hence they do not need set a target frame rate. This viewpoint is always countered by the fact that a frame-rate control mechanism, combined with LODs, allows 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, calculated on this difference, 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.* [20] 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 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 determine which modes and rendering types are fastest separately and in combination for a particular platform and 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 described above, you can control the frame rate with higher accuracy than with purely static or feedback methods. The accuracy of the predictions are highly dependent 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 is a feedback loop dependent 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 [45, 40, 49, 50]. In this idiom, the geometry is pre-rendered into a texture and texture mapped onto a single polygon or simple polygon mesh during rendering. This is an advance form of the texture for geometry substitution described in section 7.2.3. IRIS Performer [45] 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.* [49] creates a BSP tree of the scene and renders using 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 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, making the technique less effective.

Sillion *et al.* [50] have a similar approach, but instead of only using 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 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.

7.3 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.

7.3.1 Multithreading

Multithreading is used here as the general ability to have more than one thread of control sharing 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 will execute on seperate 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. Examples include the main thread 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 not have to wait for the main thread to stall or to context swap in order to get work done.

Multiple threads can be used for many of the computational tasks 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.

There are a few issues associated with using multiple threads. The primary concern becomes data exclusion and data synchronization. When multiple threads are acting on the same data, only one thread can be changing the data at a time. That change then needs to be propagated to all other threads so 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 based on multiple memory buffers is described in [45]. 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, it is useful to combine the two techniques for 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 using multiple hardware graphics pipelines. (see 2.1.10)The synchronization overhead is higher, but latency is reduced. A combination of the two approaches can have a pipelined renderer with asynchronous concurrent threads handling non-frame-critical aspects of the application such as I/O.The target system architecture will determine what is possible, while the application requirements will determine what is useful. The following 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 decides which geometric objects need to be drawn and which geometric objects can be safely elided from the scene (see section 7.2). 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

Use of multiple levels of detail (LOD) per object is one of the most effective ways of reducing geometric complexity (see 7.2.4). 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 allow 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 only load 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* where 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 in order 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 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 in-line as part of the rendering loop. This 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 they are responsible for is needed, the application needs to have a fall-back to replace the missing data. Database paging operations can first bring in small, low-resolution data that is quick to read to ensure there is some data 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.

7.3.2 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 repeating by drawing again to the back. Between issuing the buffer swap and the next graphics command the graphics system must wait for the current frame to finish scanning out 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 minimum latency of 13.89 milliseconds extra per frame.

Rendering Completes Before Frame

Completing rendering before the graphics system has finished scanning out to the output device usually results in the system blocking on the next graphics call. In a single threaded application, the time spent blocking can potentially be used to either spend more time rendering using more complex geometry and more resource-hungry rendering modes, or to use the extra time to run application-specific calculations. Another use for this time in a single threaded application is to synchronize the threads and update shared data. In general, the time between a buffer swap and the next iteration of the render loop should be used by an application to do additional work in anticipation of the subsequent frame.

Rendering Completes After Frame

Completion of rendering after the graphics system has finished scanning out a full frame means that the next possible buffer swap cannot occur for another full frame time. This effectively cuts the application frame rate in half. In contrast, a slight reduction in scene complexity can have the effect of doubling frame-rate.

7.3.3 Memory vs. Time vs. Quality Trade-Offs

There are many trade-offs between memory, time, and quality that need to be taken into account. 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 due to 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 very apparent and distracting. In either case, the use of LODs are 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 multiple power of two levels forming 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 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 working with large data sets, all of the data does not have to be loaded up-front. An application should be able to roam through an infinitely large scene if 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.

7.3.4 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. There are some small changes 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 time required and scene graph performance to use an off-the-shelf scene graph API.

Bounding Information

One of the easiest to use and most beneficial pieces of information 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 associated with them, for example, wheels of a car are always positioned relative to the center of the car, offset by some transform. 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 do similar concatenations for other states in the scene, namely rendering modes, colors, and even pre-calculating lighting in some situations.

State Changes

State changes are generally an expensive operation for most graphics systems. It is best to try to render all items with the same state in order to minimize the number of times state needs to be changed in a scene. Rendering a geometric checkerboard goes much faster by rendering 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 is using, then sorting the scene by state becomes possible and rendering 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 extremely useful when deciding how much can be drawn per frame. This timing information can be supplemented with information about the number and types of primitives 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, allow the user to modify the scene in some way, 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 also be of high quality, but primarily should be rendered with interactivity of a higher priority than higher quality.

An interactive scene needs to take advantage of 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 completely removing specific object representations by substituting them with bounding boxes.

Conclusion

Graphics software that truly runs efficiently on a computer system is built on three foundations. These foundations, of course, rely on knowledge of how software, graphics function calls, and the computer system interact with each other.

Since many graphics applications spend much of their time processing information not directly related to calling any graphics API, the first foundation is based on well written application software. This software is distinct from that used to call any graphics API, but is instead used to process data, take user input, or store data, etc. Delays in the execution of this part of the code will drag down over-all performance. Fortunately, a host of tools are available which 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 will not make up for a poor choice of graphics algorithms. Efficient algorithms, then, are the third foundation on which graphics performance rests. As is pointed out in the course, 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 which doesn't fit many situations. It's far easier - and cheaper in the long run - to look at how the software and system interact and modify the application software accordingly. This effort can be one of the most challenging and satisfying aspects of developing efficient graphics software.

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 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 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 defined by the smallest sphere that fits around the object.

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

BSP Tree: See Binary Space Partitioning.

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 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.

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 read from the memory. This is generally faster 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 designed to mitigate 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 able to 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's nearly ready to render.

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

Height Field: A mapping of a data value to a height 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 where successive levels have less geometric complexity.

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

Microcode: Instructions which 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.

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, and 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.

Polygon Rate: A measure of the speed polygons can be processed by the graphics pipeline. Polygon rates are reported as the number of triangles able to 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, 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 used to 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 once giving 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] Mesa3d website. *http://www.mesa3d.org*.

[4] Mpi website. *http://www.mpi-forum.org*.

[5] Opengl sample implementation open source project website. *http://oss.sgi.com/projects/ogl-sample*.

[6] Openmp website. *http://www.openmp.org*.

[7] 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).

[8] Kurt Akeley. The Silicon Graphics 4d/240gtx superworkstation. *IEEE Computer Graphics & Applications*, 9(4):71–83, 1989.

[9] Kurt Akeley and Thomas Jermoluk. High-performance polygon rendering. In *SIGGRAPH 88 Conference Proceedings*, Annual Conference Series, pages 239–246. ACM SIGGRAPH, 1988.

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

[11] 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.

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

[13] 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.

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

[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] James D. Foley, Andries van Dam, Seven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, second edition, 1990.

[19] 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.

[20] 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.

[21] 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.

[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] Paul Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. In *SIGGRAPH 90 Conference Proceedings*, Annual Conference Series, pages 309–318. ACM SIGGRAPH, 1990.

[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] 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.

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

[40] 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.

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

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

[43] Restricted pointers in C. *Numerical C Extensions Group / X3J11.1, Aliasing Subcommittee*, 1993.

[44] 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.

[45] 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.

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

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

[48] 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.

[49] 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.

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

[51] 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.

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

[53] 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.

[54] 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.

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

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

[57] 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.

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

[59] 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.