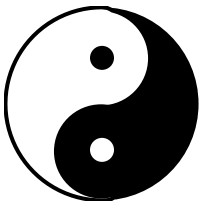
Developing Efficient Graphics Software:
The Yin and Yang of Graphics



A SIGGRAPH 99 Course

Course Organizer

Keith Cok

SGI

Course Speakers

Keith Cok

Alan Commike

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 the general optimization techniques for the C and C++ languages and overview current algorithms for reducing graphics and general system and algorithmic overhead. Finally, we provide a "laundry list" of graphics optimization techniques to improve software performance.

Preface

Course Schedule

1:30 PM	Introduction
1:35 PM	General Performance Overview
2:05 PM	Software and System Performance
2:40 PM	Break
2:55 PM	Profiling and Tuning Code
3:25 PM	Compiler and Language Considerations
3:55 PM	Graphics Techniques and Algorithms
4:40 PM	Tips and Tricks

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.

Alan Commike
MTS SGI
2011 N. Shoreline Blvd., Mountain View, CA 94039
commike@sgi.com

Alan Commike is a Member of Technical Staff at SGI. Alan works in optimizing large software applications for SGI platforms. His primary interests are in scientific visualization, volume rendering, and high-performance graphics. Lately, he has been teaching applications how to run in Reality Centers. Prior to joining SGI, he worked for Landmark Graphics developing Oil and Gas exploration software. Alan received a BS and MS in Computer Science from the State University of New York at Buffalo.

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, Matt Johnson, Brian Thatch, and Ann Johnson, for giving us the opportunity to develop this course and the course reviewers who gave us much needed feedback: Kurt Akeley, Roger Corron, Nigel Kerr, Peter Shafton, Robert Skinner, Kim Bayer, BJ Wishinsky, and Hansong Zhang. In particular, we thank Scott Nelson, Scott Senften, Steve Vance, and Mason Woo for their extensive course evaluation.

We gratefully acknowledge Pam Thuman-Commike for her work with the figures, proof reading, and LaTeXformatting.

Course Resources On the Web

The course notes and slides are available on the SGI web site:
http://www.sgi.com/events/siggraph99/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

This course, as presented at SIGGRAPH 99, is a half-day in length. Many details will necessarily be omitted from the presentation, and found only in the course notes. The presentation is intended to motivate thought about the issues surrounding high-performance graphics application development, whereas the notes provide details about the implementation and characterization necessary to achieve high-performance graphics applications.

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. Last, the course coalesces knowledge obtained from years of graphics application tuning into a “tips and tricks” section used as a quick reference guide to improving graphics application performance.

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

General Performance Overview

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. 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 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

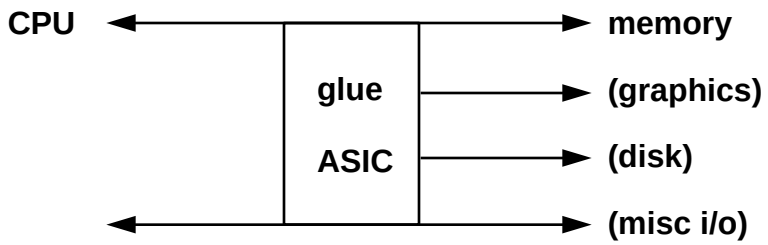


Figure 2.1: An abstract computer system fabric.

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.

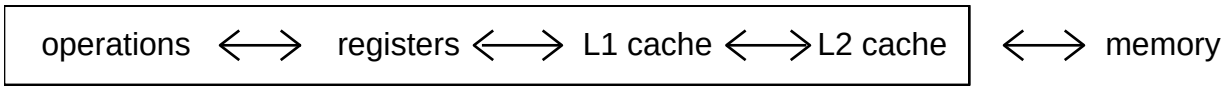


Figure 2.2: Abstract CPU.

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).

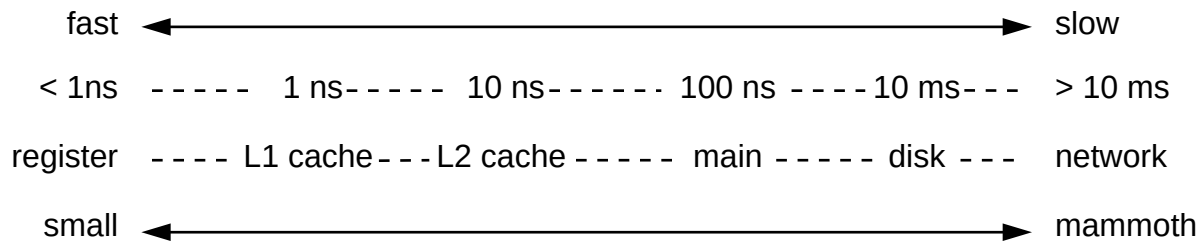


Figure 2.3: Approximate data latencies and capacities of typical system components.

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 on 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* [34] 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.

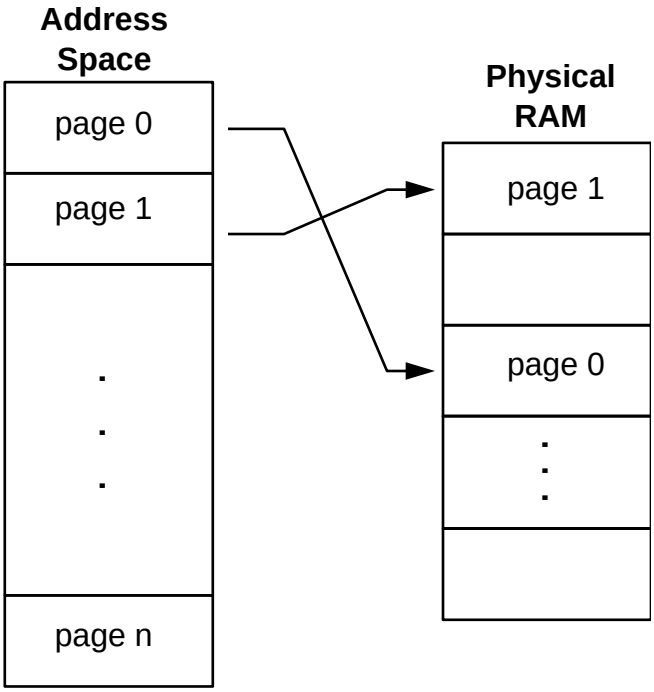


Figure 2.4: Virtual memory mapping active pages into RAM.

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.

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

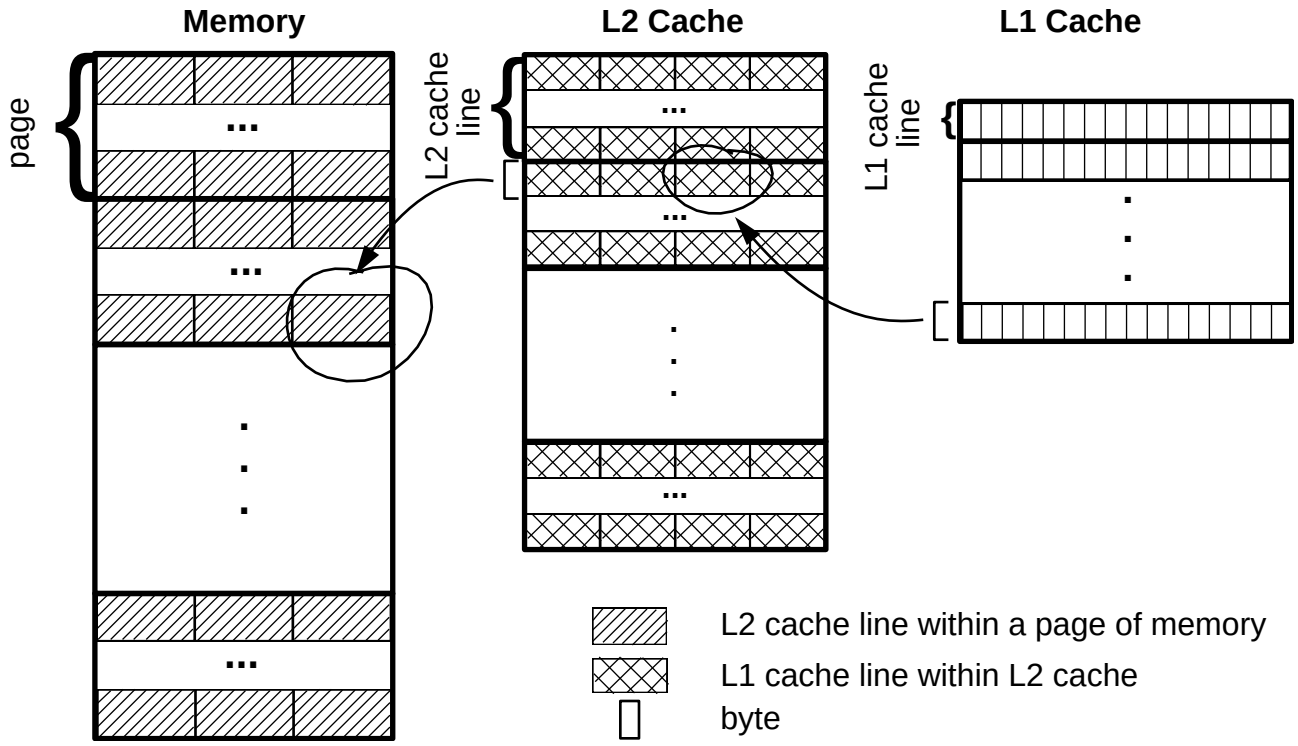


Figure 2.5: Cache line structure. Shown are pages of memory composed of multiple L2 cache lines; L2 cache composed of multiple L1 cache lines; and L1 cache composed of individual bytes.

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 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

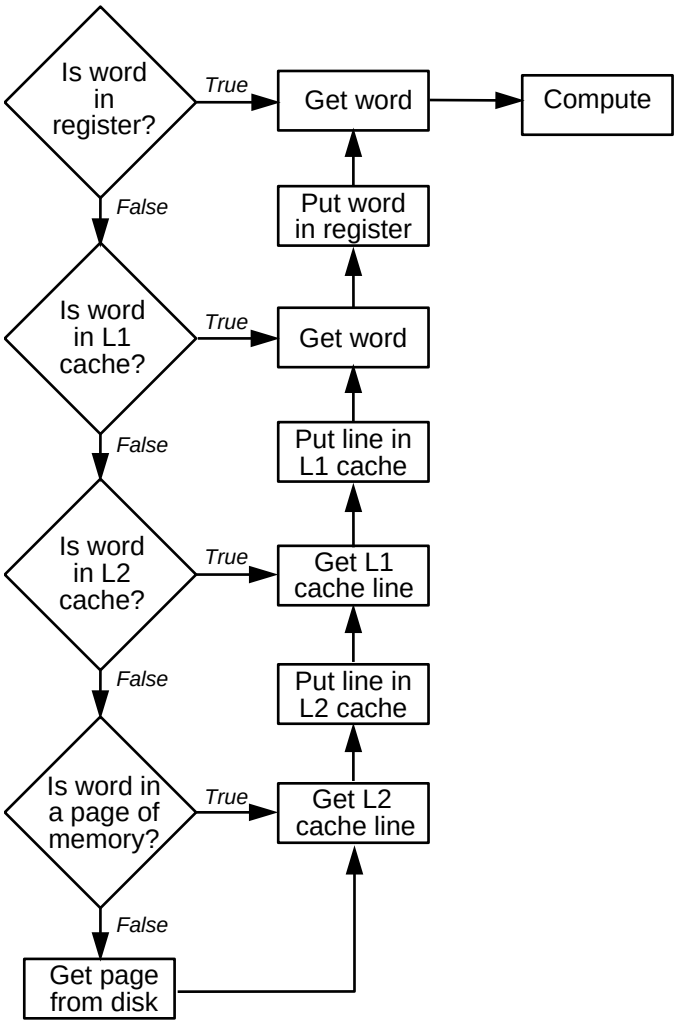
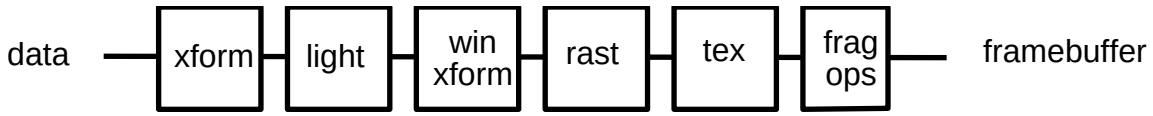


Figure 2.6: Register data request flowchart.

(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



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.

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[2].

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.

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 3.5.1.

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 is the subject of the next section.

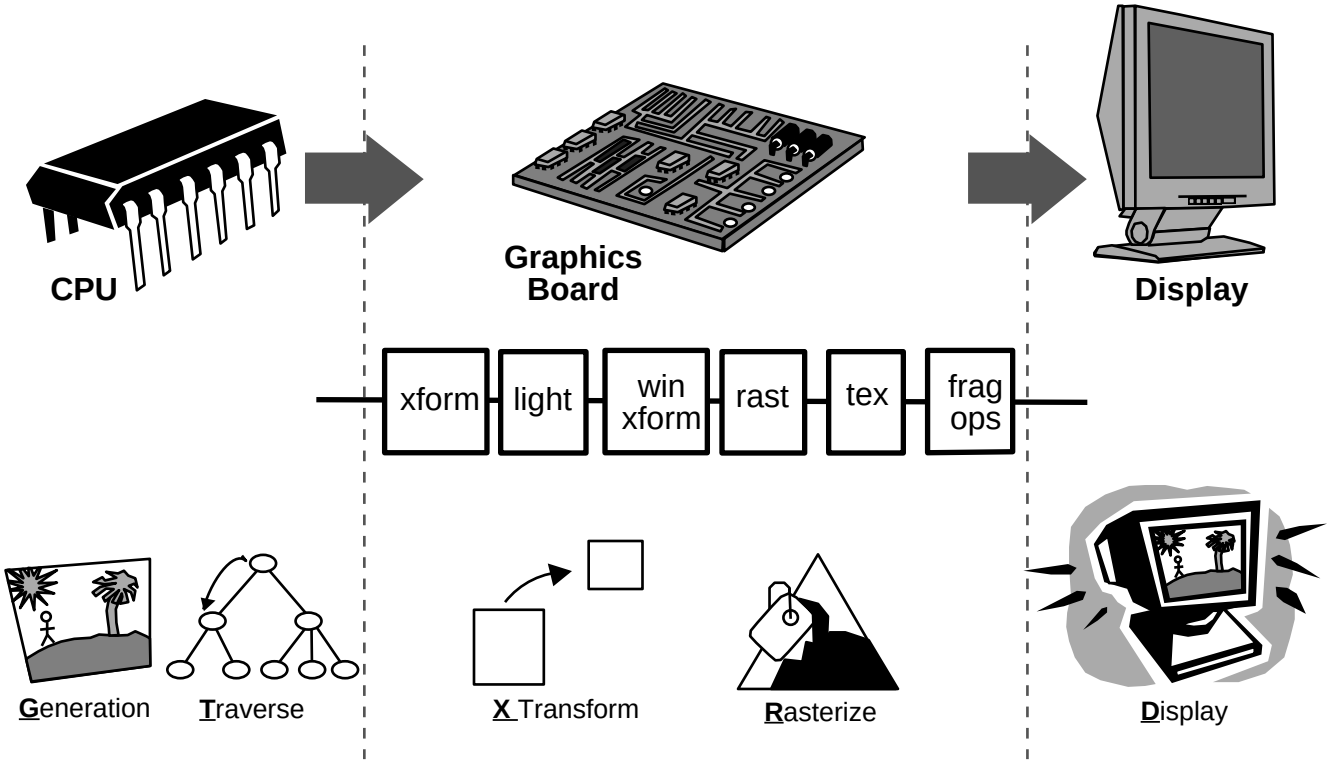


Figure 2.8: Graphics hardware pipeline and taxonomy.

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 between 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 256- or 512-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 3.4.1.

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

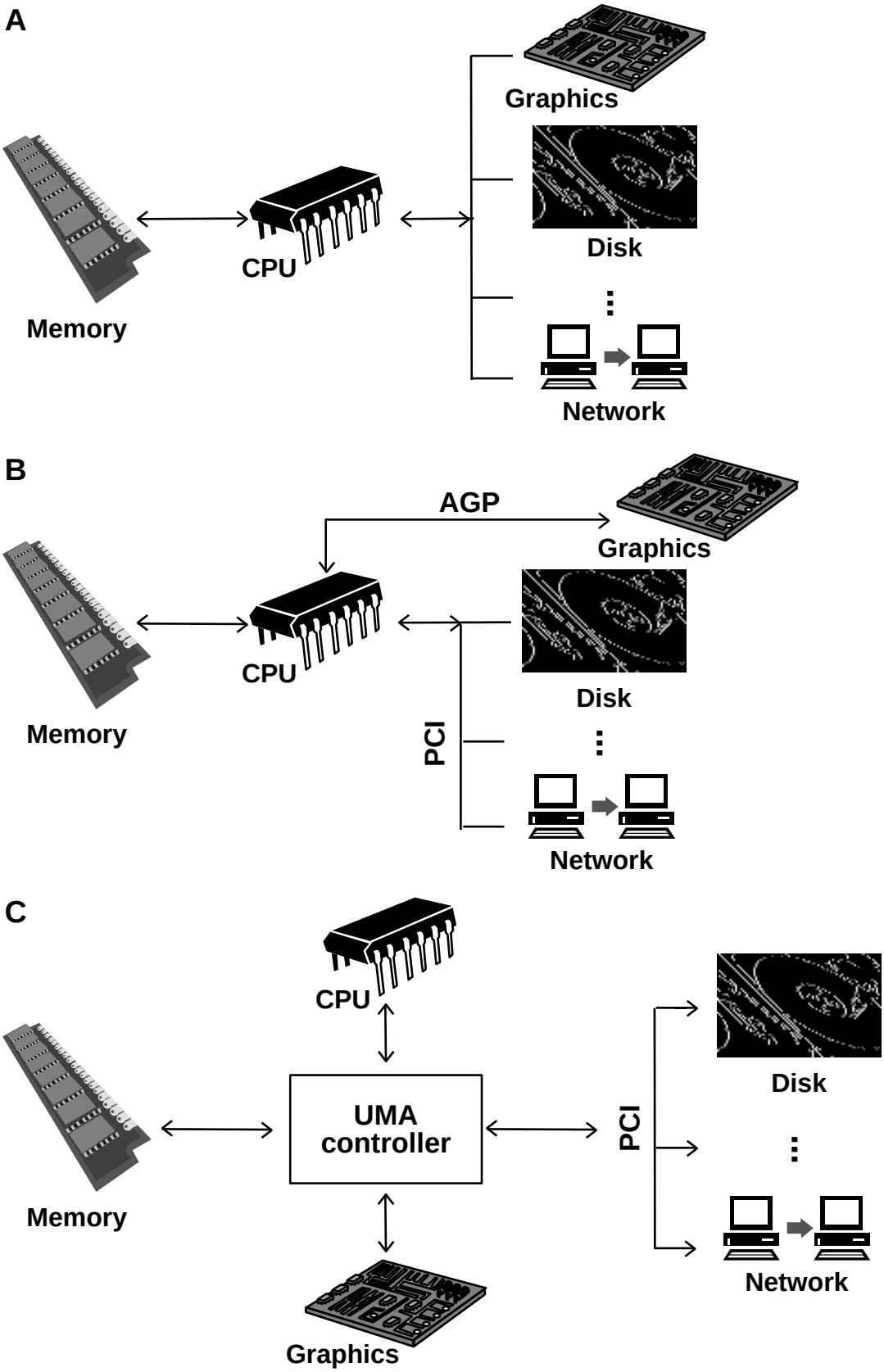


Figure 2.9: Schematic of system interconnection architectures: (A) PCI, (B) AGP, and (C) UMA.

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 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.

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[1]. 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 through a computing system.
  - Read the documentation provided by hardware vendors.

[1] http://www.spec.org/gpc

  - 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

Software and System Performance

3.1 Introduction

To achieve the highest performance from a computer graphics application is to maintain a delicate balance between the software requirements and the hardware capabilities. Out of balance, a system does not perform optimally. The goal of this section is to demonstrate the iterative process of tuning a computer graphics application to achieve equilibrium between the graphics requirements of the application and the capabilities of the various components that play a role in the overall system performance. Although targeted toward tuning existing applications, the ideas in this section apply to designing a new graphics application as well. Again, there is more to tuning a graphics application than tuning the actual graphics. All system components affect the overall performance of a graphics application.

The application tuning process consists roughly of four phases. The first phase quantifies performance to compare how the application performs with the ideal system performance. The second phase examines how system configuration impacts performance. The third phase considers system architecture to determine if an application is CPU or graphics-bound. The fourth phase examines the application code for potential bottlenecks that could impact performance.

The process described here is iterative and is never really complete. Once a bottleneck in the rendering pipeline has been identified and addressed, the tuning process should start anew in search of the next performance bottleneck. Code changes can cause bottlenecks to shift among the different stages of the rendering pipeline. Performance tuning is an ongoing process. A final sub-section describes how system tools can aid in the tuning process. This sub-section should be considered more of an appendix than an actual step in the overall process.

3.2 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. When this information is collected, you can compare it to the ideal performance of the graphics hardware to learn the relative performance of the application.

3.2.1 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.

Primitive Types

Determine the primitive types (triangles, quads, pixel rectangles, etc.) being used by the application and if there is a predominant primitive type. Record if the primitives are generally 2D or 3D and if the primitives 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. 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 running an application to gather primitive counts and other data, it is important to use the application and exercise code paths as a true user would. The performance that a bona fide user experiences, 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 3.1. 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 3.1A); rendering the triangles as a strip only requires 280 bytes of data (Figure 3.1A), which saves 224 bytes. In a real application, this 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

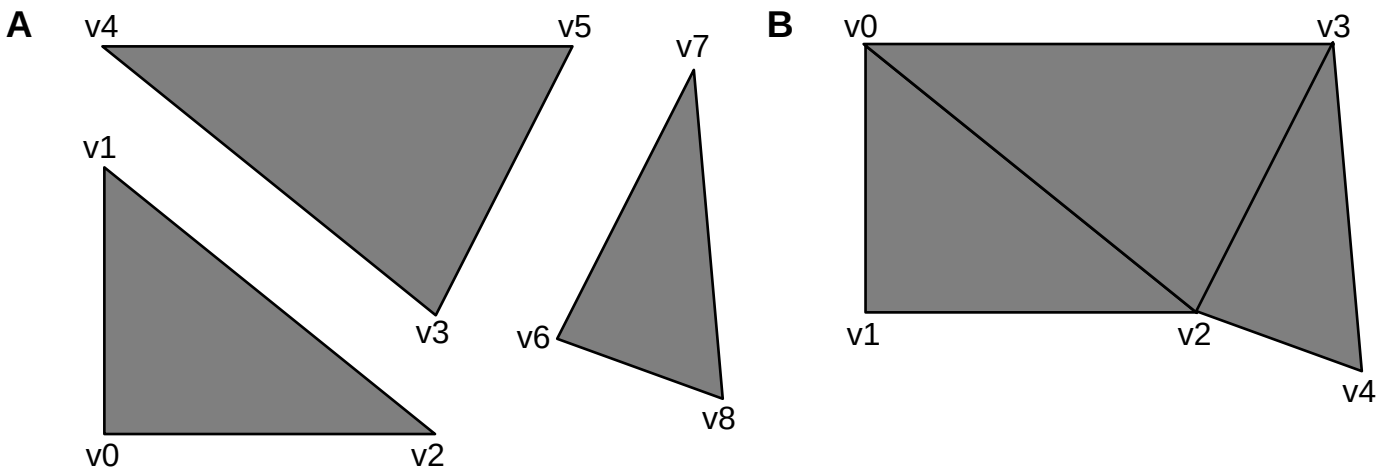


Figure 3.1: Worst case per-vertex data for triangles. (A) Shown are three triangles, each vertex containing position (XYZW), color (RGBA), normal (XYZ), and texture (STR). Rendering a single triangle requires 56 bytes of data per vertex, resulting in a total of 168 bytes of data. The set of triangles 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.

- 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 computation complexity and the number 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.

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. Some systems provide hooks (and tools, such as `osview`) into the hardware, which can measure framebuffer swaps for any application.

3.2.2 Compare Results

With the data collected above, you can make a full comparison between the current performance of the application and the ideal performance on a given system. Use this comparison to determine if the applica-

tion performance is characteristic given the capabilities of the available hardware. Methods for obtaining ideal performance data for a system are described in Section 2.2.3.

When making this comparison, consider 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 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, 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? This is highly unlikely. If so, then no amount of tuning in the graphics code will improve the user experience. (See Section 4 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.

3.3 Examine the System Configuration

Often the first system component considered regarding graphics performance is the graphics hardware. However, 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.

3.3.1 Memory

Insufficient memory in a system can cause excessive paging. Know 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 insufficient physical memory. Swapping memory pages to disk negatively impacts performance. Try to keep data in-cache as much as possible by creating and using small, tightly packed, and efficient data structures.

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 textures and other 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. The copy of data from system memory

to video memory is implemented as a DMA over a high-speed dedicated bus. One disadvantage of AGP texturing, for example, is that memory access to these 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 100-MHz, front-side bus between the CPU and main memory.

3.3.2 Display Configuration

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, subsequently, how an application uses the graphics hardware. It's important to 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.

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 force 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. Also, be wary of applications that choose default visuals and pixel formats because often the defaults are not the best performing or the most feature rich.

Once the display is configured properly by the window or operating system, 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. Know the fall-back strategy if an application can't get a requested 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.

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. For example most applications 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 size of the configured Z-buffer compares to the requirements of the application. Using a Z-buffer depth that does not match the requirements of the application can have a negative impact on performance. Using a visual or pixel format that does not support a hardware Z-buffer forces depth testing to be performed in software. The actual depth of the Z-buffer is important as well. Too many bits adds the overhead of manipulating additional unnecessary bits, while too few bits creates visual artifacts (Z-fighting) as too little precision available for conclusive depth testing.

Auxiliary Buffers Like the depth buffer, it's important to properly configure the color, stencil, and other auxiliary buffers to meet the demands of the application. The larger the number of color bits per pixel, the more bits that need to be manipulated during rendering and rasterization. A display configuration without a stencil buffer or other auxiliary buffer may force the rendering into a software path.

Buffer Swap Characteristics Determine if buffer swaps are tied to the vertical retrace of the graphics display. If so, as is traditionally the case, 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. 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.

3.3.3 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, and choose disks with the lowest latency and seek times. Once again, the disk requirements vary greatly by application, so use appropriate disk resources for the specific application.

3.3.4 Network

The network can also play a role in the performance of a graphical application program at several levels. Be careful about loading data and textures from a remote file system during rendering. Also, it is important to 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 rendered for display local, or are remote machines being used to offset some of the CPU requirements. Know the bandwidth requirements and capabilities of all systems in a remote-rendering scenario.

3.4 CPU-Bound or Graphics-Bound?

A computer graphics application is either CPU-bound or graphics-bound at any moment during the execution time. Like a pendulum, an application swings between varying degrees of these two states as rendering execution swings from CPU-based tasks to graphics hardware-based tasks. Tuning an application is an attempt 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 and special-purpose dedicated graphics hardware. However, before you can make the appropriate lifestyle adjustments to achieve this balance, you must be able to recognize the warning signs.

As application data is traversed by an application, it is passed through a graphics library that prepares it for passing over the interconnect fabric hardware (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 rate at which an application generates data and the rate that data is processed by the graphics subsystem.

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. 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.

On the other hand, if the graphics subsystem is processing data slower than the FIFO is being filled, eventually the FIFO issues 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.

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. The fill limit, usually specified in megapixels/sec is determined by the capabilities of the graphics accelerator card.

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. The geometry limit is determined by both the CPU and the graphics accelerator card depending on where the geometry calculations are performed.

3.4.1 Graphics Architecture

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.

GTXR-D As shown in Figure 3.2 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

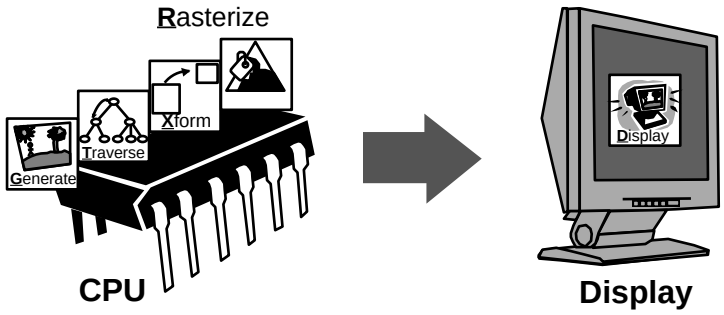


Figure 3.2: Schematic of the **GTXR-D** graphics subsystem.

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 4 and 5.

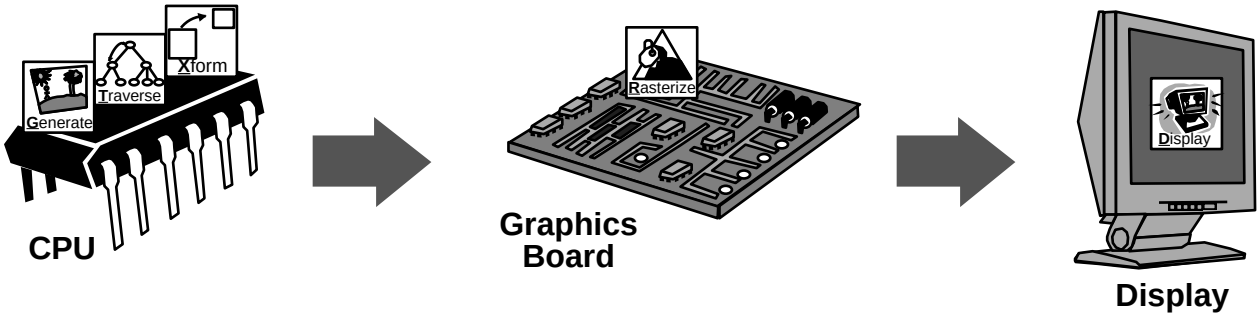


Figure 3.3: Schematic of the **GTX-RD** graphics subsystem.

GTX-RD As shown in Figure 3.3 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, the graphics FIFO fills causing the CPU to stall. When this happens, the application is graphics-bound. However, with the CPU still doing many of 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 [3], there is still a good chance that the application is CPU-bound.

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, make the graphics window smaller, which reduces the scan conversion area. Also, reduce depth complexity and the number of times a pixel is drawn by not rendering polygons that are occluded in the final image (see Section 6.2).

GT-XRD As shown in Figure 3.4 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. More often, the graphics FIFO fills up causing the CPU to stall while the graphics hardware performs all the transformation, lighting, and rasterization.

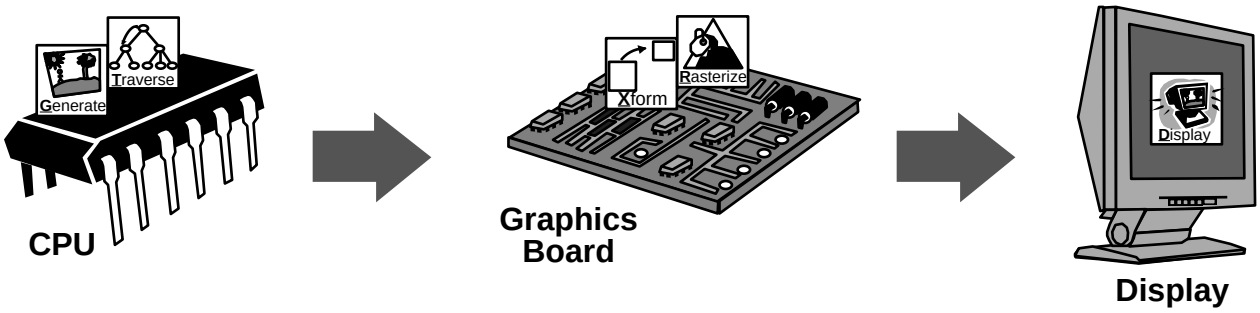


Figure 3.4: Schematic of the **GT-XRD** graphics subsystem.

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, use a lighting model implemented in the graphics hardware. Another good example is matrix operations. Graphics hardware in this category is specifically designed to efficiently perform matrix multiplication operations, pass these operations to 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 vertex arrays to limit setup time required by the graphics hardware. Also, try reducing lighting requirements to reduce the computational complexity of the lighting equations.

3.4.2 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. Does the frame rate improve? Then, the application is fill-limited as the overall performance is limited by the time required to update the graphics window.

- 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 in 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 to establish 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.

3.5 Code Considerations

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.

3.5.1 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 3.4.2, 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. Determine hardware fast paths by reading hardware vendor supplied documentation. Some vendors provide fast path documentation, others do not. If you are unsure about the fast paths on your target graphics adapter, 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.

Knowing 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.

3.5.2 Identifying Graphics Bottlenecks

Understanding the potential and actual bottlenecks is crucial to effectively tuning an application. There will always be a bottleneck, a part of the system which limits 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 smoothly as possible.

Independent of software or hardware implementation, rendering pipeline bottlenecks typically occur in these situations:

- Floating-point object coordinate to screen coordinate transformations and vertex color calculations
- Fixed or floating-point rasterization calculations for triangle slope and line slope

- Pixel fragment generation
- Scan conversion into frame buffer memory

Strangely enough, a bottleneck isn't always a negative. Sometimes, you can take advantage of a bottleneck and use the time to perform other tasks sometimes to the point of actually adding functionality to an application. For example, for a fill-limited application, you can add more geometry processing in the form of more sophisticated lighting and shading and/or a finer tessellation without impacting 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 common causes of bottlenecks within an application categorized by the subsystem in which they occur. All the bottlenecks listed here, independent of the subsystem within which they occur, affect graphics rendering performance.

Graphics

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 should be included in the graphics hardware documentation.

Excessive 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 changes can cause a bottleneck in the graphics subsystem when more time is actually spent validating the state than rendering. To avoid excessive state changes, organize data so that primitives with similar if not identical characteristics are rendered sequentially (without differing data in-between).

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. When primitives are packaged into strips, the graphics driver and rendering hardware can often pipeline rendering.

Large textures. Large textures may be paged-in and cached from main memory, depending on the graphics system architecture. 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 large textures, which do not fit in the local video memory, creates 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. Other programmatic solutions to this problem are to use mipmapping or clipmapping, as described in Section 6.3.3.

Large number of textures. For basically the same reason that large textures can create a bottleneck, using a large number of textures can create a bottleneck as well. Graphics subsystems typically

have a limited number of texture objects, which can remain resident depending upon the amount of graphics memory available. PCI-based graphics subsystems with limited local graphics memory must pre-fetch and load texture from system memory over the 132 MB/s PCI bus. This type of bottleneck is less of a problem under AGP, due to the dedicated graphics bus, and UMA, due to the lack of data transfer for textures resident in main-memory.

Unnecessary rendering modes. Using unnecessary rendering modes (two-sided lighting, normalization, dithering, etc.) causes increased rendering overhead that could potentially create an unnecessary bottleneck. Bottlenecks of this type can occur on 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.

Excessive geometry. Using large geometry can cause a bottleneck independent of where actual geometry calculations (that is, lighting, transformations, etc.) are performed. In every system, there is a point where the system becomes geometry-bound, where the system cannot transform and light the large amounts of geometry specified at satisfactory frame-rates. Avoid rendering excessive geometry — if you can't see it, don't draw it.

Depth complexity. Consider how many times the same pixels are filled. Avoid drawing small or occluded polygons by using techniques such as culling described in Section 6.2.

CPU

After graphics, the second most common place to uncover system bottlenecks is in the host CPU.

Excessive geometry. On systems that perform most if not all geometry processing (that is, lighting, transformations, etc.) on the host CPU, excessive geometry can easily cause a CPU bottleneck. Cull unseen geometry, reduce the resolution of surfaces, and reduce the detail of objects which are far from the viewer to solve this problem. See Section 6.2 for more information about these types of optimizations. Another solution is to use texture maps to simulate complex geometry.

Memory

The inefficient arrangement of graphics data within memory and inefficient memory management in general can cause a bottleneck in the memory system.

- Allocate all memory for graphics primitives before beginning the rendering loop.
- 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.
- Organize 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 can be transferred efficiently to the graphics hardware.

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).

Code and Language

Poor coding practices can be a source of application bottlenecks on the host CPU. General coding issues are addressed in Sections 4 and 5, but a couple of graphics-specific issues warrant discussion here.

API call overhead. A common cause of bottlenecks is function call overhead on the transfer rate between the host and graphics. While advanced 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. Inefficient use of the graphics API in both the sending of graphics data and in state and rendering mode changes can cause the CPU to do excessive work and create a bottleneck on the host, which leaves the graphics subsystem waiting for data. Avoid these types of bottlenecks by using the graphics API in the most efficient way possible. Use primitive strips, vertex arrays, display lists, and vector arguments rather than individual values to reduce overall graphics API overhead. Think of this differently as increasing the amount of work done within each function, to mitigate the costs of entering that function.

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, you must use additional CPU cycles to transform the data to the desired 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.

3.6 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.

3.6.1 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 3.5), APIMON (see Figure 3.6), ZapDB, and others provide these capabilities.

3.6.2 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 4.

3.6.3 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.

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.

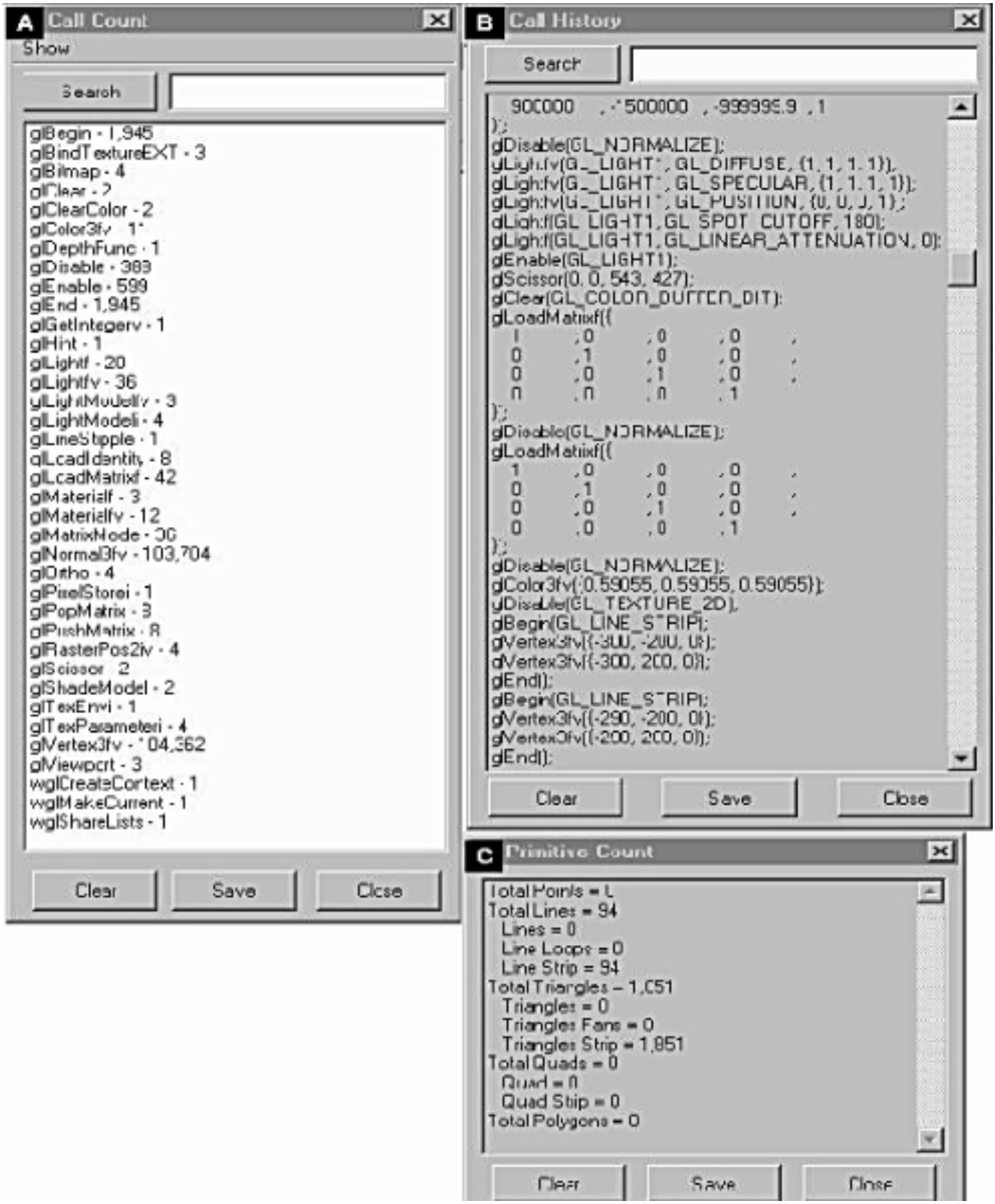


Figure 3.5: Sample output from ogldebug, an OpenGL tracing tool. (A) Call count output from a running OpenGL application. (B) A call history trace from the same OpenGL application. (C) Primitive count output from the same OpenGL application.

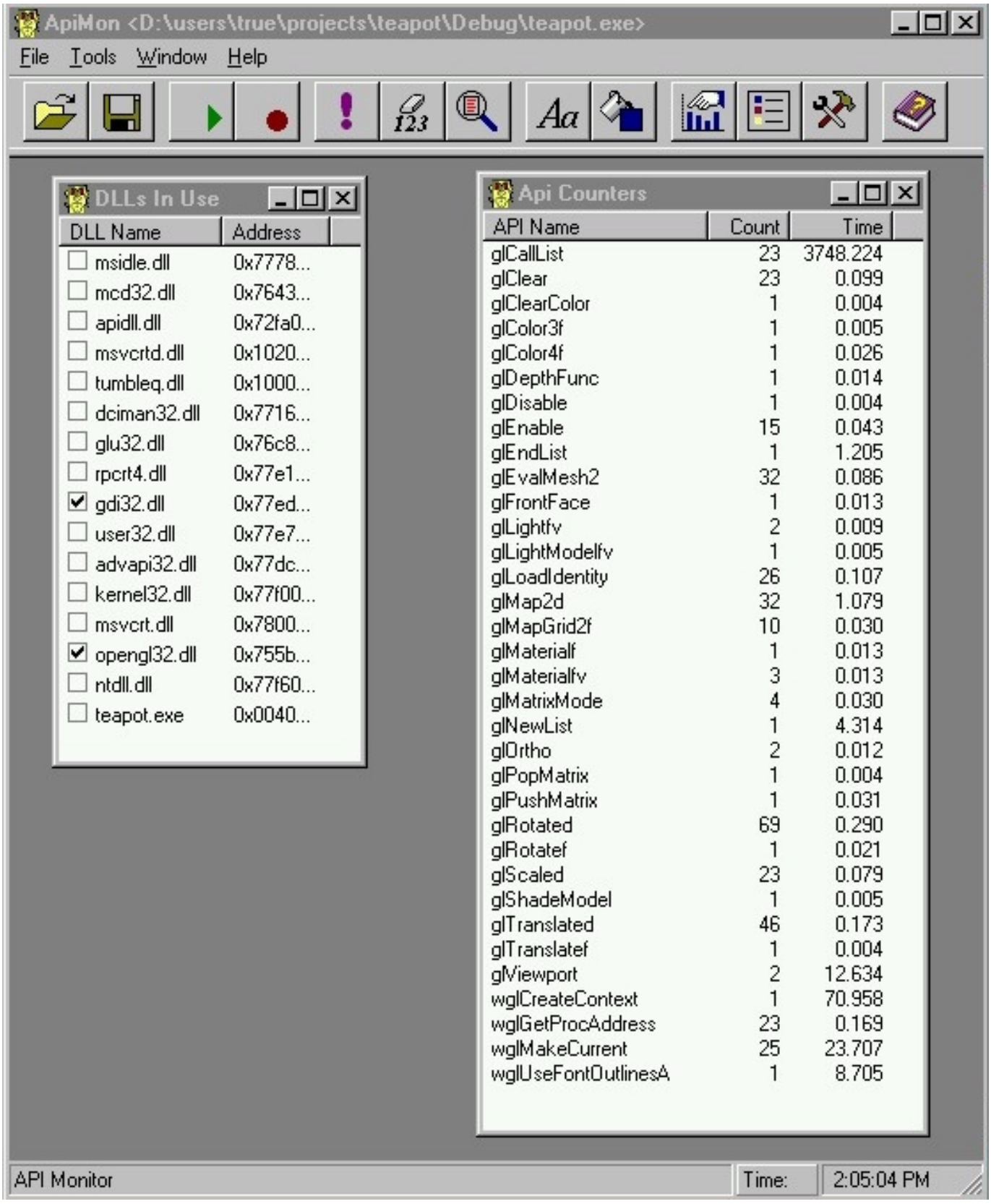


Figure 3.6: Using APIMON to trace graphics API usage.

3.7 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 4

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.

4.1 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.

4.2 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 3.4 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 3.2.1 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.

4.3 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.

4.3.1 Basic Block Profiling

It is fairly simple to profile code (Figure 4.1). 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 4.1: The steps performed during code profiling.

Consider the example `foo.exe` shown in Figure 4.2. 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 4.2A) is the original function prior to profiling, and the second function (Figure 4.2B), `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 4.2: 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 4.3 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 4.2A) is basically one large loop and nothing else. If you unroll the loop, and call the function `new_loop`, it now looks like Figure 4.2B. (More about loop unrolling in Section 5.4.4). After re-profiling the new executable, the analysis (Figure 4.3B) 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 4.3: 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 4.3C) 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 4.4) 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 4.4: Profile comparison of `new_loop` and `old_loop` using Visual C++ on an Intel CPU.

4.3.2 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 4.5 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 4.5A.

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 4.5 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 4.5B). However, under PC sampling analysis, a different behavior (Figure 4.5C) 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 4.5: 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 5

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++.

5.1 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 5.1 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 5.1: 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.

5.2 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.

5.3 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.

5.4 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 [10] also details a number of source code optimizations.

5.4.1 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 5.1A. 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 5.1B, 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.

5.4.2 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 5.1: 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 5.2 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.

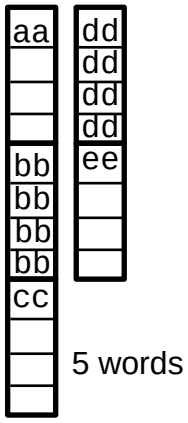
5.4.3 Source Code Organization

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

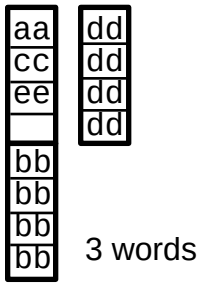


Figure 5.2: Example of how data structure packing affects memory size. (A) A non-packed data structure `foo`. (B) Memory space used by the `foo` data structure illustrating the wasted memory. Because the data structure is not packed, the 8-bit characters result in a waste of 16 bits, each resulting in 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.

5.4.4 Unrolling Loop Structures

Another common optimization technique is known as loop unrolling. Consider the function `old_loop` shown in Figure 5.3A 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 5.3B 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 5.3: Example of loop unrolling. (A) Original function `old_loop`. (B) Improved function `new_loop` with the loop unrolled.

In the example shown in Figure 4.3, 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.

5.4.5 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.

5.4.6 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.

5.4.7 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 5.4. 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 5.4: Example of optimization using temporary variables. (A) Original code. (B) Optimized version.

Figure 5.5 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 5.5: Example of optimization using temporary variables with a function. (A) Original code. (B) Optimized version.

5.4.8 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* [8]. 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 [8] (Figure 5.6A). 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 [36]. 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 5.6A would be

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

rewritten as shown in Figure 5.6B. Cook [8] 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.

5.5 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.

5.5.1 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.

5.5.2 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.

5.5.3 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.

5.5.4 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 6

Graphics Techniques and Algorithms

6.1 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 goal of this section is to take that step back and look at the holistic application. The following analogy 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 point of this analogy is to take a step back when trying to make an application run faster and begin to think about how the application works as a whole.

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.

6.2 Idioms

idiom: The syntactical, grammatical, or structural form peculiar to a language [49].

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) and reducing the complexity of the information that does get rendered (level of detail). 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.

6.2.1 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 [13, 51, 50, 5, 16] 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. Iones, *et al.*, have recently published a paper on the determination of the optimal bounding box orientation [25].

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

Closed opaque objects 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* [51]. Backface culling is done on a per-object, and sometimes per-primitive, basis.

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 ability of the user to “enter” an object needs to be handled specially 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.

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* [50], 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 detail 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.

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.

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 [47] 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 [52, 16]. 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 dawning 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 [50], which covers occlusion culling background material and an extensive algorithm for choosing the optimal occlusion set.

6.2.2 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 [5, 18]. 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 [30] or general adaptive algorithms if they allow the decimation factor to vary over the model in a view-dependent fashion [7, 21]. 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 [7] or calculated "on the fly" as a progressive refinement of the object [21].

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 [13, 38, 42]. 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 [13]. In many cases, the geometry can be totally replaced by a textured image [42], 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 [5] 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 [19] 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 [28, 24], hierarchical subdivision methods [46, 39, 9], feature methods [44], refinement methods [12, 20, 37, 14], decimation methods [29, 40], and optimal methods [1]. 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" [22]. 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.*[30] 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 [38] library's Active Surface Definition (ASD), Linstrom's [30] algorithm, and Hoppe's view-dependent progressive mesh [23] 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 [48, 41, 11, 21, 7, 22, 31] to solve this problem, each having advantages and disadvantages. Recently, these simplification algorithms have expanded the domain coverage to include real-time algorithms [26, 22, 31], and include view-dependent information [7, 22, 31], and attain higher compression rates for low-bandwidth transmission of data [4, 45, 17].

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 [13, 38]. 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.* [13] 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 [38, 32, 42, 43]. In this idiom, the geometry is pre-rendered into a texture and texture mapped onto a single polygon or simple polygon mesh during rendering. IRIS Performer [38] 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.* [42] 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.* [43] 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.

6.3 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.

6.3.1 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. 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 play a large role in benefiting 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 for 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 [38].

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. 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 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 6.2). 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 6.2.2). 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 state 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.

6.3.2 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.

6.3.3 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.

6.3.4 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. A simple center or radius for each object can go a long way. A full bounding box is also useful.

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 also include 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, 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.

Backing Store

Do not redraw the whole scene just because a menu item occluded parts of it. Save the scene and then restore the bits that have been occluded. More tips and tricks are discussed in Section 7.

Overlay Planes

Use the overlay planes for interactive editing or frequently changing simple geometry on top of a complex scene.

Section 7

Tips and Tricks

This section is a list of common techniques, tips, and hints used to optimize graphics code (and OpenGL code in particular.) Some have been mentioned earlier in this course, others were learned from experience or are public knowledge. Other sources of examples are found in [35], [15], [33], and [6]. Some items are trivial and others are difficult to implement, however, they are all included here as a checklist.

7.1 General Hints

- Identify the bottleneck
  - Fill-limited? Reduce window size to check. Make sure the application does not change behavior based on window size. Also, reduce the amount of effort done per pixel by disabling depth buffering, texturing, or alpha blending. If this test improves performance, the application is fill limited. Consider replacing larger polygons with smaller ones to balance the pipeline between fill rate and geometry transfer.
  - Geometry-limited? To check, substitute less expensive graphics calls with cheaper ones. This preserves the CPU and caching behavior while reducing the amount of work done by the graphics pipeline. For example, replace the `glVertex` and `glNormal` calls with `glColor` calls. This reduces the geometry drawing and lighting work while maintaining consistent CPU behavior. If these tests improve performance, the application is geometry-limited.

    Consider replacing smaller polygons with larger ones to balance the pipeline between geometry transfer and fill rate.
  - CPU-bound or memory-bound? Profile code to check.
- Does the code run efficiently on the test machine?
  - Profile the machine capabilities: use `glperf` or write test programs.
  - Query the machine capabilities and use code/extensions accordingly.
  - Use single-buffer mode when measuring performance. Although this often results in visual artifacts (such as “tearing”), it prevents the application from waiting for the next vertical retrace and provides repeatable frame rates that are not dependent on the monitor refresh rate.
- Remove redundant/unnecessary calls:

  - Avoid redundant state calls. Cache important state information. This can be more efficient than calling `glPushAttributes/glPopAttributes` for some graphics hardware.
  - Sort primitives in the following order to reduce unnecessary state changes [15]:
    * By transform
    * By lighting model (one vs. two side, local vs. infinite, and so on)
    * By texture
    * By material
    * By primitive mode (triangle strips, quads, lines, etc.)
    * By color
  - Avoid querying the graphics pipeline (for example, `glGet*`) for information as it may stall.
  - Send data to the pipeline in the most efficient format (float vs. double, RGBA-ABGR, vectors, vertex arrays) for that hardware.
  - Turn off unused features and attributes even when they have no visible effect.

- Debugging:
  - OpenGL errors can lower performance. Use the following macro to eliminate errors during the development phase [15]:

```
#if DEBUG
#define GLEND() glEnd();\
assert(glGetError()==GL_NO_ERROR);
#else
#define GLEND() glEnd()
#endif
```

7.2 Geometry Hints

- Place as much information as possible between `glBegin` and `glEnd`.
- Sort geometric data by geometry type (triangle, quad, etc). Avoid using `glBegin(GL_POLYGON)`.
- Use only concave polygons. Convex (including self-intersecting) polygons must be tesselated before being drawn.
- Use `GL_FASTEST` with all `glHint()` calls.
- Disable all per-fragment operations if possible: alpha blending, depth, stencil, scissor, logic ops, blending, dithering. Also turn off expensive rasterization operations: texture, fog, anti-aliasing. Enable these modes when necessary and try to sort primitives to reduce mode changes.
- Use display lists to draw static geometry multiple times. Keep display lists as flat as possible to reduce calls to `glCallList`. This allows a fast and simple traversal and may also improve cache coherency. Be reasonable about display list flattening: don't replace a single instantiation of an object with multiple copies. For example, don't replace many references to window geometry within a building with many copies of that geometry. Also, don't make the display list excessively small.

- Use `glPushMatrix/glPopMatrix` to cache matrix information instead of loading and storing matrices.
- Use `glTranslate`, `glRotate`, `glScale`, or `glLoadIdentity` instead of creating your own translate, rotation, or scaling matrix and calling `glLoadMatrix`.
- Texture simpler objects instead of using complex geometry. For example, textures can implement approximate lighting models to simulate smoother surfaces (for example, phong shading) without resorting to more tesselated geometry. Use one-dimensional textures to apply per-vertex color information more efficiently, using only one float per-vertex instead of three (to describe RGB color values).
- Use geometry strips to reduce the amount of vertex data to be sent to the graphics adapter. Performance can vary depending on tristrip length.
- Use smaller data types for colors and normals (such as bytes or shorts) to reduce the data sent per vertex. This helps eliminate data transfer as the bottleneck.
- Use backface culling when possible. Understand its implications — in some application domains backface culling can be faster when done on the CPU.
- Use vertex arrays. Use interleaved arrays if possible; precompiled vertex arrays are even better. Use `glDrawArrays` and `glDrawElements` to reduce function call overhead. In addition, for some implementations, `glInterleavedArrays` may be more efficient than specifying the arrays separately. The compiled array extension allows OpenGL to DMA the arrays to the pipeline. (Vertex arrays are available in OpenGL 1.1.)
- Use one normal per polygon when the polygon is flat-shaded (do not specify per-vertex normals if they are not used).
- Cull objects that will be out of view or too small to see.

7.3 Lighting

- Use flat shading if possible, which reduces the number of lighting calculations down to one per primitive from one per vertex.
- Use as few lights as possible. Different machines can efficiently handle different numbers of local or infinite lights. Adding another light may significantly reduce performance.
- Minimize calls to `glColorMaterial` and `glMaterial` (the expense of `glColorMaterial` vs. `glMaterial` is platform-dependent). Generally use `glMaterial` for infrequent changes and `glColorMaterial` when a single material property is changed often.
- Use directional (infinite) lighting. A quick way to do this is to find all references to `GL_POSITION` and set the fourth coordinate to 0.0.
- Use positional lights rather than spot lights.
- Disable `GL_NORMALIZE` if possible. Make sure all normals have a unit value.

- Avoid scaling operations in the model-view matrix. The normals will need to be renormalized if the model-view matrix has a scale operation.

- Make all normals consistent with respect to the geometry to allow use of one-sided lighting (all normals face "out" or "in"). Two-sided lighting may be expensive.

- [33] uses the following settings for peak performance lighting:

  - Single infinite light.
  - Nonlocal viewing. Set `GL_LIGHT_MODEL_LOCAL_VIEWER` to `GL_FALSE` in `glLightModel` (the default).
  - Single-sided lighting. Set `GL_LIGHT_MODEL_TWO_SIDE` to `GL_FALSE` in `glLightModel` (the default).
  - If two-sided lighting is used, use the same material properties for front and back by specifying `GL_FRONT_AND_BACK`.
  - Don't use per-vertex color.
  - Disable `GL_NORMALIZE`. Since it is usually only necessary to renormalize when the model-view matrix includes a scaling transformation, consider preprocessing the scene to eliminate scaling.

7.4 Visuals

- Pick the correct visual for the job. Many display and performance problems stem from poor visual selection. Different visuals may have different performance characteristics.

- Use hardware-accelerated visuals. Under Windows, check that the `PIXELFORMATDESCRIPTOR.dwFlags` structure for your visual has the `PFD_GENERIC_FORMAT` bit set to `FALSE`.

- Avoid context switches. Use direct contexts if possible. Use as few contexts as necessary.

  - Use one window/one context where possible. Use multiple viewports to switch views if necessary. Or try using one context across multiple windows [15].
  - Order of desirability:
    - One window, one context
    - Multiple windows, one context
    - Multiple windows, multiple contexts

- Avoid unnecessary round trips to the graphics hardware. If you need to flush the graphics pipe, use `glFlush` instead of `glFinish`.

- Put GUI elements in the overlay planes so as not to trigger unnecessary renders of the drawing area.

7.5 Buffers

- Use the depth buffer efficiently, turning it off when possible. For example, in large vis-sim applications, it's possible to draw the sky and ground with the depth buffer disabled (nothing drawn will be above the sky or below the ground) and anything flat on those polygons (runway, stars). Enable depth buffering before drawing for those objects on the ground (mountains, trees, buildings) or in the sky (airplanes, clouds).

- Use the OpenGL stencil buffer to perform interactive picking. A hierarchical scene graph could be created from the geometry that consists of nodes, each with 256 children (each uniquely identifying a stencil ID to be placed in the stencil buffer upon drawing). When picking, the stencil buffer at the pointer coordinates could be read back, and then only 1/256th of the scene would have to be re-rendered, dividing that subsection again in 256 separate stencil IDs. This could effectively accelerate picking by a factor of 100. [15]

- Edit one object in a scene of many objects without having to draw every object every frame. Try the following [15]:

  1. Render all objects except the edited object.
  2. Read color/depth buffers back into main memory.
  3. Calculate bounding rectangle (screen coordinates) for edited object.
  4. Draw edited object.
  5. Upon having to refresh scene, only copy color/depth for bounding rectangle from main memory.
  6. Draw edited object again in its altered form.

- Use hardware-accelerated offscreen buffers to do backing-store.

7.6 Textures

- Use texture objects (part of core OpenGL 1.1 and available through extensions in OpenGL 1.0) instead of display lists or calling `glTexImage` and `glTexParameter*` separately if possible. This is especially important if an application uses multiple textures. This technique allows the implementation to compile the object into a format for optimal rendering and texture management.

  To use texture objects, place calls to `glTexImage` and `glTexParameter*` into a texture object and bind it to the OpenGL context.

  If texture objects are not an option, consider encapsulating the texture commands into a display list.

- Load texture images once for an application, not once for every frame. Avoid frequent switching between textures.

- Combine multiple textures into one large texture as a mosaic, and change the texture coordinates accordingly to map into the larger super-texture. The application only then uses one texture as far as the graphics system is concerned.

- Use textures for mapping alternate data onto geometry. Think of textures generically as 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 is one of the interesting things to do with textures, but requires some thought to see immediately how to apply it to an application. See [27] for further description and ideas.
- Be explicit about specifying texture parameters. Don't rely on default parameters as a program will only get default performance.
- Try using `glTexSubImage` to replace part of an existing texture rather than creating a whole new texture.
- Avoid expensive texture filter modes such as trilinear filtering. Point sampling or bilinear filtering instead may be less expensive on some systems.
- Ensure that all of your textures are resident in texture memory before rendering.
  - Use `glAreTexturesResident` (available in OpenGL 1.1). Note that on those systems where the texturing is done on the host, this function call always returns `GL_TRUE`.
  - Reduce resolution or use the texture LOD extension to reduce texture size and make sure they fit. In general, smaller textures improve the performance of texture downloads.

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.

CPU: Central Processing Unit.

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.

Elide: To remove.

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.

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.

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.

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

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] 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).

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

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

[4] 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.

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

[6] Sharon Rose Clay. Optimization for real-time graphics applications. In *SIGGRAPH 98 Conference Course Notes*. ACM SIGGRAPH, August 1998.

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

[19] 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.

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

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

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

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

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

[29] 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.

[30] 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.

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

[32] 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.

[33] Tom McReynolds and David Blythe. Advanced graphics programming techniques using OpenGL. In *SIGGRAPH 98 Conference Course Notes*. ACM SIGGRAPH, August 1998.

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

[35] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley, first edition, 1993.

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

[37] 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.

[38] 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.

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

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

[41] 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.

[42] 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.

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

[44] 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.

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

[46] 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.

[47] 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.

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

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

[50] 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.

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

[52] 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.

Performance Implications of Pointer Aliasing

by Doug Cook
Silicon Graphics, Inc.
August, 1997

Pointer aliasing can have a severe impact on program performance. Understanding its implications is critical to writing high-performance code. This document provides a brief introduction to the problem, and suggests several approaches to solving it through source-code restructuring, compiler options, and C or C++ language extensions.

Aliasing

Here's a brief overview of aliasing. Consider the following function:

```
void
process_data(float *in, float *out, float gain, int nsamps)
{
        int i;

        for (i = 0; i < nsamps; i++) {
                out[i] = in[i] * gain;
        }
}
```

In C or C++, it is legal for the parameters *in* and *out* to point to overlapping regions in memory. When this happens, *in* and *out* are said to be *aliases*. When the compiler optimizes the function, it does not in general know whether *in* and *out* are aliases. It must therefore assume that any store through *out* can affect the memory pointed to by *in*, which severely limits its ability to reorder or parallelize the code (For some simple cases, the compiler could analyze the entire program to determine that two pointers cannot be aliases. But in general, it is impossible for the compiler to determine whether or not two pointers are aliases, so to be safe, it must assume that they are).

In the code above, the compiler must issue the instructions in roughly the following order. The code here is simplified a little bit:

```
top:    lwc1      $f1,0(v1)             # load in[i]           0
        mul.s     $f1,$f1,$f0           # multiply by gain     1
        addiu     v1,v1,4               # increment in         1 (
        swc1      $f1,0(v0)             # store the result     5
        addiu     v0,v0,4               # increment out        6
        bne       v0,a0,top             # see if we're done    7
```

In this code, the compiler decided that there was no benefit to unrolling the loop because of the limited options for reordering. This version theoretically requires 8 cycles per iteration.

For this example, we use the MIPS R5000 processor, an excellent processor to use for this type of benchmark: though it can be very fast, it is quite sensitive to instruction scheduling. Because the R10000 dynamically reorders instructions, it is considerably less sensitive to compiler scheduling, and thus less suitable for this type of benchmark. A few pertinent notes about the R5000:

1. The single-precision multiply or multiply/add (mul.s or madd.s) can be issued every cycle, but it has a 4-cycle latency. Therefore pipelining the floating-point instructions is critical to good performance.
2. A floating-point instruction can execute in parallel with an integer instruction.
3. lwc1 and swc1 (floating-point loads and stores) count as integer instructions.

The fundamental problem in this code example is that the compiled code has to wait for the result of a multiply and store it to memory before it can load the next value, because a store to *out[i]* might affect a subsequent load from *in[i+k]*. This completely prevents the compiler from pipelining the instructions, or from taking advantage of the parallelism in the processor.

Here is the performance benchmark for this version of the program (compiled with the MIPSPro 7.2 beta compilers, using **-n32 -mips4 -r5000 -O3**). The benchmark simply calls **process_data** a large number of times and looks at the elapsed clock time. All the data fits in the cache.

```
macondo 2% gainex 100000
22891752.000000 samps/sec (519.087341 voices @ 44.1kHz)
```

The numbers here assume that the data is audio, and measure the number of simultaneous **process_data** calls ("voices") that could be made in real-time at 44.1 kHz (44100 audio samples per second). But the techniques used here apply to any kind of data. The results will be more dramatic with floating-point data, where the independent floating-point unit provides more opportunities for instruction-level parallelism, and where the difference between FP

instruction latencies (sometimes several cycles) and FP instruction issue rates (usually 1 per cycle) provides more opportunities for software pipelining.

Improving Performance Through Code Restructuring

You can often improve code performance by manually unrolling the loops in your code. This is a conventional method for speeding up loop performance. An unrolled loop is often faster because it performs fewer loop-bound tests. But if done properly, loop unrolling can also minimize the effects of pointer aliasing.

The basic idea is to unroll a loop and defer the stores as long as possible in the unrolled loop. Then the compiler is free to reschedule all the instructions between the loads and the stores. Here is the previous example using temporary variables to defer the stores:

```
void
process_data(float * in, float * out, float gain, int nsamps)
{
        int i;
        float y,y1,y2,y3;

        for (i = 0; i < nsamps; i+= 4) {
                y = in[i] * gain;
                y1 = in[i+1] * gain;
                y2 = in[i+2] * gain;
                y3 = in[i+3] * gain;
                out[i] = y;
                out[i+1] = y1;
                out[i+2] = y2;
                out[i+3] = y3;
        }
}
```

Note that this version has the additional restriction of requiring that ***nsamps*** *be a multiple of 4. The common scheme for avoiding this restriction is to add a separate loop which does the leftover 0-3 iterations. Since that is irrelevant to the aliasing discussion, it is omitted here.*

```
macondo 1% /usr/tmp/gainex.new 100000
64134524.000000 samps/sec (1454.297607 voices @ 44.1kHz)
```

This method has some drawbacks. The compiler still cannot reorder operations past the stores, because it still does not know if the pointers are aliased. There are just more operations to reorder between the loads and stores. In other words, the store to **out[i]** could still affect subsequent values of **in[i+k]**, *except* **in[i]**, **in[i+1]**, **in[i+2]**, and **in[i+3]**, because those have already been loaded. So the compiler is only free to reorder the calculations based upon **in[i]**,

in[i+1], **in[i+2]**, and **in[i+3]**.

If we unroll the loop in larger blocks, we may get better performance. However, at some point, this hand-unrolling will backfire: the number of temporary variables will exceed the number of readily-available registers, or the loop's cache behavior will change, and its performance will degrade. The optimal block size can be difficult to determine, and changes from processor to processor, making it hard to write portable code.

Also, the hand-unrolled code becomes more difficult to read and maintain. What we'd really like is to keep the code simple and tell the compiler that pointers cannot be aliased, then let the compiler do the loop unrolling and scheduling as it sees fit. There are a couple of ways to do this.

Optimizer Options

Many compilers have options to override their default aliasing assumptions. The MIPSPro 7.X compilers support a set of **-OPT:alias** command-line switches for this purpose. For example, **-OPT:alias=restrict** implements the following semantics: Memory operations dereferencing different named pointers in the program are assumed not to alias with each other, nor with any named scalar in the program.

Example: if p and q are pointers, *p does not alias with *q, nor does *p alias with p, nor does *p alias with any named scalar variable.

There is also a switch, introduced in MIPSPro 7.2, called **-OPT:alias=disjoint**. This implements the following semantics: Memory operations dereferencing different named pointers in the program are assumed not to alias with each other, and in addition, different "dereferencing depths" of the same named pointer are assumed not to alias with each other.

Example: If p and q are pointers, *p does not alias with *q, nor does *p alias with **p, nor does *p alias with **q.

Both these switches make promises to the compiler about the behavior of the program; if either switch is enabled, programs violating the corresponding aliasing assumptions are liable to be compiled incorrectly.

Often these options are difficult to use; in a typical application, some pointers are aliases, and others are not. You may be able to move certain functions to separate source files and compile just

those files with these switches. In some cases, you cannot use these switches at all, because within a single function, some pointers are aliases and others are not.

The *restrict* keyword

Fundamentally, however, this problem is an issue with the C and C++ languages themselves, because there is no way to specify which pointers may be aliased, and which may not. The Numerical C Extensions Group / X3J11.1 proposed a **restrict** keyword for the C language to address the issue (see *Restricted Pointers in C, Numerical C Extensions Group / X3J11.1, Aliasing Subcommittee, June 1993*) The **restrict** keyword is a very clean, high-performance way of addressing the pointer aliasing problem, without many of the drawbacks of the other methods.

The **restrict** keyword tells the compiler to assume that dereferencing the qualified pointer is the only way the program can access the memory pointed to by that pointer. Hence loads and stores through such a pointer are assumed not to alias with any other loads and stores in the program, except other loads and stores through the same pointer variable. Here is the previous code example rewritten to indicate that *in* and *out* cannot overlap:

```
void
process_data(float *  restrict  in, float *  restrict  out, float g
{
        int i;

        for (i = 0; i < nsamps; i++) {
                out[i] = in[i] * gain;
        }
}
```

Another example may clarify things a bit. In this example, qualifying **a** with **restrict** is sufficient to indicate that **a** and **c** are not aliased. In other words, you need only qualify the pointers being stored through.

```
float x[ARRAY_SIZE];
float *c = x;

void f4_opt(int n, float * restrict a, float * restrict b)
{
  int i;
  /* No data dependence across iterations because of restrict */
  for (i = 0; i < n; i++)
    a[i] = b[i] + c[i];
}
```

The MIPSPro 7.2 compilers accept the restrict keyword with the **-LANG:restrict** option, and use it to perform aggressive

optimization. The 7.1 compilers do *not* support the restrict keyword, though they will not issue warnings if you use it. You can determine which compilers you have by using the command "cc -n32 -version". Additionally, the compilers will set the _COMPILER_VERSION preprocessor macro to indicate which version of the compiler is in use.

Here are the results of our original example using the **restrict** keyword. This code is over 3 times as fast as the original version, and somewhat faster than even the hand-unrolled version. For this simple example, performance would be identical using either the **-OPT:alias=restrict** option or the **restrict** keyword.

```
macondo 3% ./gainex 100000
70395424.000000 samps/sec (1596.268066 voices @ 44.1kHz)
```

In this case, the compiler was able to unroll the loop and freely reschedule instructions within the loop to maximize performance. Here is part of the unrolled loop which processes 4 samples. Note that all the multiplies are pipelined and overlapped with integer instructions as well. The result is that this version takes closer to 2 cycles per sample.

```
  [...]
                                        Cycle
  lwc1      $f0,24(a1)                  0
  mul.s     $f4,$f2,$f5                 1
  lwc1      $f1,20(a1)                  1
  mul.s     $f3,$f0,$f5                 2
  lwc1      $f0,16(a1)                  2
  mul.s     $f1,$f1,$f5                 3
  lwc1      $f2,44(a1)                  3
  beq       a1,a0,0x10000ed4            4
  mul.s     $f0,$f0,$f5                 5
  swc1      $f4,28(a2)                  5
  swc1      $f3,24(a2)                  6
  swc1      $f1,20(a2)                  7
  swc1      $f0,16(a2)                  8
  [...]
```

Since the **restrict** keyword is not recognized by many compilers, including older MIPS compilers, for portability you may want some construct like:

```
#if (defined(__sgi) && _COMPILER_VERSION >= 720)
#define RESTRICT restrict
#else
#define RESTRICT
#endif

process_data(float * RESTRICT in, float * RESTRICT out, float gain,
```

This construct will attempt to use **restrict** only in the presence of the MIPSPro 7.2 or later compilers.

The following table summarizes the results for this simple example using the four techniques discussed here: the original code; the hand-unrolled loop; with -OPT:alias=restrict; and with -LANG:restrict. We've included R10000 results for comparison. As mentioned earlier, the performance improvements for the R10000 are less dramatic; the code gets about 16% faster using **restrict** . The R5000 results used **-O3 -r5000 -mips4 -n32** ; the R10000 results used **-O3 -r10000 -mips4 -n32**.

	original	hand-unrolled	-OPT:alias=restrict	-LANG:rest
180MHz R5000	519	1454	1585	1596
195MHz R10000	1765	1891	2051	2051

From this, we can see that the **restrict** keyword is a clean, high-performance approach to writing number-crunching code.

Acknowledgments

Thanks much to **Raymond Lo** for his review, and to **Rohit Chandra** for precise descriptions of the compiler behavior and for some of the code examples.

Visibility Culling using Hierarchical Occlusion Maps

Hansong Zhang Dinesh Manocha Tom Hudson Kenneth E. Hoff III

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
{zhangh,dm,hudson,hoff}@cs.unc.edu
http://www.cs.unc.edu/~{zhangh,dm,hudson,hoff}

Abstract: We present hierarchical occlusion maps (HOM) for visibility culling on complex models with high depth complexity. The culling algorithm uses an object space bounding volume hierarchy and a hierarchy of image space occlusion maps. Occlusion maps represent the aggregate of projections of the occluders onto the image plane. For each frame, the algorithm selects a small set of objects from the model as occluders and renders them to form an initial occlusion map, from which a hierarchy of occlusion maps is built. The occlusion maps are used to cull away a portion of the model not visible from the current viewpoint. The algorithm is applicable to all models and makes no assumptions about the size, shape, or type of occluders. It supports approximate culling in which small holes in or among occluders can be ignored. The algorithm has been implemented on current graphics systems and has been applied to large models composed of hundreds of thousands of polygons. In practice, it achieves significant speedup in interactive walkthroughs of models with high depth complexity.
CR Categories and Subject Descriptors: I.3.5 **[Computer Graphics]:** Computational Geometry and Object Modeling
Key Words and Phrases: visibility culling, interactive display, image pyramid, occlusion culling, hierarchical data structures

1 Introduction

Interactive display and walkthrough of large geometric models currently pushes the limits of graphics technology. Environments composed of millions of primitives (e.g. polygons) are not uncommon in applications such as simulation-based design of large mechanical systems, architectural visualization, or walkthrough of outdoor scenes. Although throughput of graphics systems has increased considerably over the years, the size and complexity of these environments have been growing even faster. In order to display such models at interactive rates, the rendering algorithms need to use techniques based on visibility culling, levels-of-detail, texturing, etc. to limit the number of primitives rendered in each frame. In this paper, we focus on visibility culling algorithms, whose goal is to cull away large portions of the environment not visible from the current viewpoint.

Our criteria for an effective visibility culling algorithm are *generality*, *interactive performance*, and *significant culling*. Additionally, in order for it to be *practical*, it should be implementable on current graphics systems and work well on large real-world models.

Main Contribution: In this paper, we present a new algorithm for visibility culling in complex environments with high depth complexity. At each frame, the algorithm carefully selects a small subset of the model as occluders and renders the occluders to build *hierarchical occlusion maps* (HOM). The hierarchy is an image pyramid and each map in the hierarchy is composed of pixels corresponding to rectangular blocks in the screen space. The pixel value records the *opacity* of the block. The algorithm decomposes the visibility test for an object into a two-dimensional overlap test, performed against the occlusion map hierarchy, and a conservative Z test to compare the depth. The overall approach combines an *object space* bounding volume hierarchy (also useful for view frustum culling) with the *image space* occlusion map hierarchy to cull away a portion of the model not visible from the current viewpoint. Some of the main features of the algorithm are:

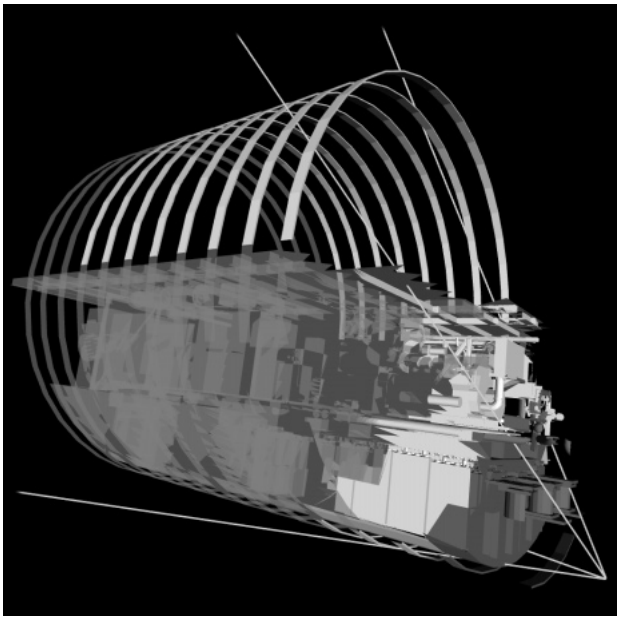


Figure 1: *Demonstration of our algorithm on the CAD model of a submarine's auxiliary machine room. The model has 632,252 polygons. The green lines outline the viewing frustum. Blue indicates objects selected as occluders, gray the objects not culled by our algorithm and transparent red the objects culled away. For this particular view, 82.7% of the model is culled.*

1. **Generality:** The algorithm requires no special structures in the model and places *no restriction* on the types of occluders. The occluders may be polygonal objects, curved surfaces, or even not be geometrically defined (e.g. a billboard).
2. **Occluder Fusion:** A key characteristic of the algorithm is the ability to *combine* a "forest" of small or disjoint occluders, rather than using only large occluders. In most cases,

the union of a set of occluders can occlude much more than what each of them can occlude taken separately. This is very useful for large mechanical CAD and outdoor models.

3. **Significant Culling:** On high depth complexity models, the algorithm is able to cull away a significant fraction (up to 95%) of the model from most viewpoints.
4. **Portability:** The algorithm can be implemented on most current graphics systems. Its main requirement is the ability to read back the frame-buffer. The construction of hierarchical occlusion maps can be accelerated by texture mapping hardware. It is not susceptible to degeneracies in the input and can be parallelized on multiprocessor systems.
5. **Efficiency:** The construction of occlusion maps takes a few milliseconds per frame on current medium- to high-end graphics systems. The culling algorithm achieves significant speedup in interactive walkthroughs of models with high depth complexity. The algorithm involves no significant preprocessing and is applicable to dynamic environments.
6. **Approximate Visibility Culling:** Our approach can also use the hierarchy of maps to perform *approximate* culling. By varying an *opacity threshold* parameter the algorithm is able to fill small transparent holes in the occlusion maps and to cull away portions of the model which are visible through small gaps in the occluders.

The resulting algorithm has been implemented on different platforms (SGI Max Impact and Infinite Reality) and applied to city models, CAD models, and dynamic environments. It obtains considerable speedup in overall frame rate. In Figure 1 we demonstrate its performance on a submarine's Auxiliary Machine Room.

Organization: The rest of the paper is organized as follows: . We briefly survey related work in Section 2 and give an overview of our approach in Section 3. Section 4 describes occlusion maps and techniques for fast implementation on current graphics systems. In Section 5 we describe the entire culling algorithm. We describe its implementation and performance in Section 6. Section 7 analyses our algorithm and compares it with other approaches. Finally, in Section 8, we briefly describe some future directions.

2 Related Work

Visibility computation and hidden surface removal are classic problems in computer graphics [FDHF90]. Some of the commonly used visibility algorithms are based on Z-buffer [Cat74] and view-frustum culling [Cla76, GBW90]. Others include Painter's Algorithm [FDHF90] and area-subdivision algorithms [War69, FDHF90].

There is significant literature on visible surface computation in computational geometry. Many asymptotically efficient algorithms have been proposed for hidden surface removal [Mul89, McK87] (see [Dor94] for a recent survey). However, the practical utility of these algorithms is unclear at the moment.

Efficient algorithms for calculating the visibility relationship among a static group of 3D polygons from arbitrary viewpoints have been proposed based on the binary space-partitioning (BSP) tree [FKN80]. The tree construction may involve considerable pre-processing in terms of time and space requirements for large models. In [Nay92], Naylor has given an output-sensitive visibility algorithm using BSPs. It uses a 2D BSP tree to represent images and presents an algorithm to project a 3D BSP tree, representing the model in object space, into a 2D BSP tree representing its image.

Many algorithms structure the model database into *cells* or regions, and use a combination of off-line and on-line algorithms for cell-to-cell visibility and the conservative computation of the potentially visible set (PVS) of primitives [ARB90, TS91, LG95]. Such approaches have been successfully used to visualize architectural models, where the division of a building into discrete rooms lends itself to a natural division of the database into cells. It is not apparent that cell-based approaches can be generalized to an arbitrary model.

Other algorithms for densely-occluded but somewhat less-structured models have been proposed by Yagel and Ray [YR96]. They used regular spatial subdivision to partition the model into cells and describe a 2D implementation. However, the resulting algorithm is very memory-intensive and does not scale well to large models.

Object space algorithms for occlusion culling in general polygonal models have been presented by Coorg and Teller [CT96b, CT96a] and Hudson et al. [Hud96]. These algorithms dynamically compute a subset of the objects as occluders and use them to cull away portions of the model. In particular, [CT96b, CT96a] compute an arrangement corresponding to a linearized portion of an aspect graph and track the viewpoint within it to check for occlusion. [Hud96] use shadow frusta and fast interference tests for occlusion culling. All of them are object-space algorithms and the choice of occluder is restricted to convex objects or simple combination of convex objects (e.g. two convex polytope sharing an edge). These algorithms are unable to combine a "forest" of small non-convex or disjoint occluders to cull away large portions of the model.

A hierarchical Z-buffer algorithm combining spatial and temporal coherence has been presented in [GKM93, GK94, Gre95]. It uses two hierarchical data structures: an octree and a Z-pyramid. The algorithm exploits coherence by performing visibility queries on the Z-pyramid and is very effective in culling large portions of high-depth complexity models. However, most current graphics systems do not support the Z-pyramid capability in hardware, and simulating it in software can be relatively expensive. In [GK94], Greene and Kass used a quadtree data structure to test visibility throughout image-space regions for anti-aliased rendering. [Geo95] describes an implementation of the Z-query operation on a parallel graphics architecture (PixelPlanes 5) for obscuration culling.

More recently, Greene [Gre96] has presented a hierarchical tiling algorithm using coverage masks. It uses an image hierarchy named a "coverage pyramid" for visibility culling. Traversing polygons from front to back, it can process densely occluded scenes efficiently and is well suited to anti-aliasing by oversampling and filtering.

For dynamic environments, Sudarsky and Gotsman [SG96] have presented an output-sensitive algorithm which minimizes the time required to update the hierarchical data structure for a dynamic object and minimize the number of dynamic objects for which the structure has to be updated.

A number of techniques for interactive walkthrough of large geometric databases have been proposed. Refer to [RB96] for a recent survey. A number of commercial systems like *Performer* [RH94], used for high performance graphics, and *Brush* [SBM+94], used for visualizing architectural and CAD models, are available. They use techniques based on view-frustum culling, levels-of-detail, etc., but have little support for occlusion culling on arbitrary models.

There is substantial literature on the visibility problem from the flight simulator community. An overview of flight simulator architectures is given in [Mue95]. Most notably, the Singer Company's Modular Digital Image Generator [Lat94] renders front to back using a hierarchy of mask buffers to skip over already cov-

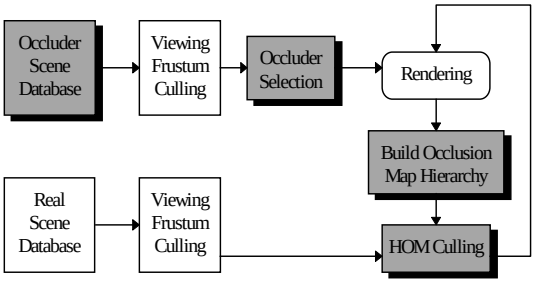


Figure 2: *Modified graphics pipeline showing our algorithm. The shaded blocks indicate components unique to culling with hierarchical occlusion map.*

ered spans, segments or rows in the image. General Electric's COMPU-SCENE PT2000 [Bun89] uses a similar algorithm but does not require the input polygons to be in front-to-back order and the mask buffer is not hierarchical. The Loral GT200 [LORA] first renders near objects and fills in a mask buffer, which is used to cull away far objects. Sogitec's APOGEE system uses the Meta-Z-buffer, which is similar to hierarchical Z buffer [Chu94].

The structure of hierarchical occlusion maps is similar to some of the hierarchies that have been proposed for images, such as image pyramids [TP75], MIP maps [Wil83], Z-pyramids [GKM93], coverage pyramids [Gre96], and two-dimensional wavelet transforms like the non-standard decomposition [GBR91].

3 Overview

In this paper we present a novel solution to the visibility problem. The heart of the algorithm is a hierarchy of occlusion maps, which records the aggregate projection of occluders onto the image plane at different resolutions. In other words, the maps capture the cumulative occluding effects of the occluders. We use occlusion maps because they can be built quickly and have several unique properties (described later in the paper). The use of occlusion maps reflects a decomposition of the visibility problem into two sub-problems: a two-dimensional overlap test and a depth test. The former decides whether the screen space projection of the potential occludee lies completely within the screen space projection of the union of all occluders. The latter determines whether or not the potential occludee is behind the occluders. We use occlusion maps for the overlap tests, and a *depth estimation buffer* for the conservative depth test. In the conventional Z-buffer algorithm (as well as in the hierarchical Z-buffer algorithm), the overlap test is implicitly performed as a side effect of the depth comparison by initializing the Z-buffer with large numbers.

The algorithm renders the occluders at each frame and builds a hierarchy (pyramid) of occlusion maps. In addition to the model database, the algorithm maintains a separate *occluder database*, which is derived from the model database as a preprocessing step. Both databases are represented as bounding volume hierarchies. The rendering pipeline with our algorithm incorporated is illustrated in Figure 2. The shaded blocks indicate new stages introduced due to our algorithm. For each frame, the pipeline executes in two major phases:

1. Construction of the Occlusion Map Hierarchy: The occluders are selected from the occluder database and rendered to build the occlusion map hierarchy. This involves:

- **View-frustum culling:** The algorithm traverses the bounding volume hierarchy of the occluder database to find occluders lying in the viewing frustum.
- **Occluder selection:** The algorithm selects a subset of the occluders lying in the viewing frustum. It utilizes temporal coherence between successive frames.
- **Occluder rendering and depth estimation:** The selected occluders are rendered to form an image in the framebuffer which is the highest resolution occlusion map. Objects are rendered in pure white with no lighting or texturing. The resulting image has only black and white pixels except for antialiased edges. A depth estimation buffer is built to record the depth of the occluders.
- **Building the Hierarchical Occlusion Maps:** After occluders are rendered, the algorithm recursively filters the rendered image down by averaging blocks of pixels. This process can be accelerated by texture mapping hardware on many current graphics systems.

2. Visibility Culling with Hierarchical Occlusion Maps: Given an occlusion map hierarchy, the algorithm traverses the bounding volume hierarchy of the model database to perform visibility culling. The main components of this stage are:

- **View-frustum Culling:** The algorithm applies standard view-frustum culling to the model database.
- **Depth Comparison:** For each potential occludee, the algorithm conservatively checks whether it is behind the occluders.
- **Overlap test with Occlusion Maps:** The algorithm traverses the occlusion map hierarchy to conservatively decide if each potential occludee's screen space projection falls completely within the opaque areas of the maps.

Only objects that fail one of the latter two tests (depth or overlap) are rendered.

4 Occlusion Maps

In this section, we present occlusion maps, algorithms using texture mapping hardware for fast construction of the hierarchy of occlusion maps, and state a number of properties of occlusion maps which are used by the visibility culling algorithm.

When an opaque object is projected to the screen, the area of its projection is made opaque. The *opacity* of a block on the screen is defined as the ratio of the sum of the opaque areas in the block to the total area of the block. An *occlusion map* is a two-dimensional array in which each pixel records the opacity of a rectangular block of screen space. Any rendered image can have an accompanying occlusion map which has the same resolution and stores the opacity for each pixel. In such a case, the occlusion map is essentially the α channel [FDHF90] of the rendered image (assuming α values for objects are set properly during rendering), though generally speaking a pixel in the occlusion map can correspond to a block of pixels in screen space.

4.1 Image Pyramid

Given the lowest level occlusion map, the algorithm constructs from it a hierarchy of occlusion maps (HOM) by recursively applying the average operator to rectangular blocks of pixels. This operation forms an *image pyramid* as shown in Figure 3. The resulting hierarchy represents the occlusion map at multiple resolutions. It greatly accelerates the overlap test and is used for approximate culling. In the rest of the paper, we follow the convention that the *highest* resolution occlusion map of a hierarchy is at *level* 0.

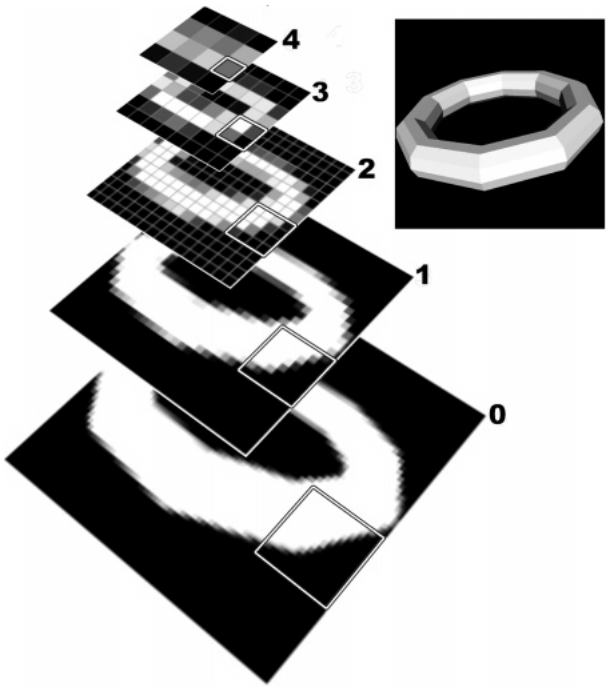


Figure 3: *The hierarchy of occlusion maps. This particular hierarchy is created by recursively averaging over 2 blocks of pixels. The outlined square marks the correspondence of one top-level pixel to pixels in the other levels. The image also shows the rendering of the torus to which the hierarchy corresponds.*

The algorithm first renders the occluders into an image, which forms the lowest-level and highest resolution occlusion map. This image represents an *image-space fusion* of all occluders in the object space. The occlusion map hierarchy is built by recursively filtering from the highest-resolution map down to some minimal resolution (e.g. 4×4). The highest resolution need not match that of the image of the model database. Using a lower image resolution for rendering occluders may lead to inaccuracy for occlusion culling near the edges of the objects, but it speeds up the time for constructing the hierarchy. Furthermore, if hardware multi-sampled anti-aliasing is available, the lowest-level occlusion map has more accuracy. This is due to the fact that the anti-aliased image in itself is already a filtered down version of a larger super-sampled image on which the occluders were rendered.

4.2 Fast Construction of the Hierarchy

When filtering is performed on 2×2 blocks of pixels, hierarchy construction can be accelerated by graphics hardware that supports bilinear interpolation of texture maps. The averaging operator for 2×2 blocks is actually a special case of *bilinear interpolation*. More precisely, the bilinear interpolation of four scalars or vectors $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ is:

$$(1-\alpha)(1-\beta)\mathbf{v}_0 + \alpha(1-\beta)\mathbf{v}_1 + \alpha\beta\mathbf{v}_2 + (1-\alpha)\beta\mathbf{v}_3,$$

where $0 \leq \alpha \leq 1, 0 \leq \beta \leq 1$ are the weights. In our case, we use $\alpha = \beta = 0.5$ and this formula produces the average of the four values. By carefully setting the texture coordinates, we can filter a $2N \times 2N$ occlusion map to $N \times N$ by drawing a two dimensional rectangle of size $N \times N$, texturing it with the $2N \times 2N$ occlusion map, and reading back the rendered image as the $N \times N$ occlusion map. Figure 4 illustrates this process.

The graphics hardware typically needs some setup time for the required operations. When the size of the map to be filtered is relatively small, setup time may dominate the computation. In such cases, the use of texture mapping hardware may slow down the computation of occlusion maps rather than accelerate it, and hierarchy building is faster on the host CPU. The break-even point between hardware and software hierarchy construction varies with different graphics systems.

[BM96] presents a technique for generating mipmaps by using a hardware accumulation buffer. We did not use this method because the accumulation buffer is less commonly supported in current graphics systems than texture mapping.

4.3 Properties of Occlusion Maps

The hierarchical occlusion maps for an occluder set have several desirable properties for accelerating visibility culling. The visibility culling algorithm presented in Section 5 utilizes these properties.

1. Occluder fusion: Occlusion maps represent the fusion of small and possibly disjoint occluders. No assumptions are made on the shape, size, or geometry of the occluders. Any object that is renderable can serve as an occluder.

2. Hierarchical overlap test: The hierarchy allows us to perform a fast overlap test in screen space for visibility culling. This test is described in more detail in Section 5.1.

3. High-level opacity estimation: The opacity values in a low-resolution occlusion map give an estimate of the opacity values in higher-resolution maps. For instance, if a pixel in a higher level map has a very low intensity value, it implies that almost all of its descendant pixels have low opacities, i.e. there is a low possibility of occlusion. This is due to the fact that occlusion maps are based on the average operator rather than the minimum or maximum operators. This property allows for a *conservative early termination* of the overlap test.

The opacity hierarchy also provides a natural method for *aggressive early termination*, or approximate occlusion culling. It may be used to cull away portions of the model visible only through small gaps in or among occluders. A high opacity value of a pixel in a low resolution map implies that most of its descendant pixels are opaque. The algorithm uses the *opacity threshold* parameter to control the degree of approximation. More details are given in Section 5.4.

5 Visibility Culling with Hierarchical Occlusion Maps

An overview of the visibility culling algorithm has been presented in Section 3. In this section, we present detailed algorithms for overlap tests with occlusion maps, depth comparison, and approximate culling.

5.1 Overlap Test with Occlusion Maps

The two-dimensional overlap test of a potential occludee against the union of occluders is performed by checking the opacity of the pixels it overlaps in the occlusion maps. An exact overlap test would require a scan-conversion of the potential occludee to find out which pixels it touches, which is relatively expensive to do in software. Rather, we present a simple, efficient, and *conservative* solution for the overlap test.

For each object in the viewing frustum, the algorithm conservatively approximates its projection with a screen-space bounding rectangle of its bounding box. This rectangle covers a superset of the pixels covered by the actual object. The extremal values of the bounding rectangle are computed by projecting the corners

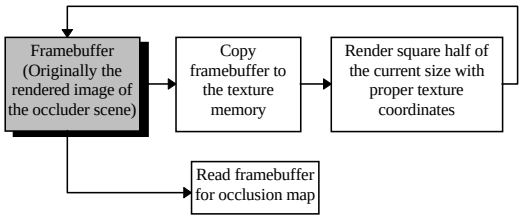


Figure 4: *Use of texture-mapping hardware to build occlusion maps*

of the bounding box. The main advantage of using the bounding rectangle is the reduced cost of finding the pixels covered by a rectangle compared to scan-converting general polygons.

The algorithm uses the occlusion map hierarchy to accelerate the overlap test. It begins the test at the level of the hierarchy where the size of a pixel in the occlusion map is approximately the same size as the bounding rectangle. The algorithm examines each pixel in this map that overlaps the bounding rectangle. If any of the overlapping pixels is not completely opaque [1], the algorithm recursively descends from that pixel to the next level of the hierarchy and checks all of its sub-pixels that are covered by the bounding rectangle. If all the pixels checked are completely opaque, the algorithm concludes that the occludee's projection is completely inside that of the occluders. If not, the algorithm conservatively concludes that the occludee may not be completely obscured by the occluders, and it is rendered.

The algorithm supports *conservative early termination* in overlap tests. If the opacity of a pixel in a low-resolution map is too low, there is small probability that we can find high opacity values even if we descend into the sub-pixels. So the overlap test stops and concludes that the object is not occluded. The *transparency thresholds* are used to define these lower bounds on opacity below which traversal of the hierarchy is terminated.

5.2 Depth Comparison

Occlusion maps do not contain depth information. They provide a necessary condition for occlusion in terms of overlap tests in the image plane, but do not detect whether an object is in front of or behind the occluders. The algorithm manages depth information separately to complete the visibility test. In this section, we propose two algorithms for depth comparison.

5.2.1 A Single Z Plane

One of the simplest ways to manage depth is to use a single Z plane. The Z plane is a plane parallel to and beyond the near plane. This plane separates the occluders from the potential occludees so that any object lying beyond the plane is farther away than any occluder. As a result, an object which is contained within the projection of the occluders and lies beyond the Z plane is completely occluded. This is an extremely simple and conservative method which gives a rather coarse bound for the depth values of all occluders.

5.2.2 Depth Estimation Buffer

The depth estimation buffer is a software buffer that provides a more general solution for conservatively estimating the depth of occluders. Rather than using a single plane to capture the depth of the entire set of occluders, the algorithm partitions the screen-space and uses a separate plane for each region of the partition. By using a separate depth for each region of the partition, the algorithm obtains a finer measure of the distances to the occluders. The depth estimation buffer is essentially a general-purpose software Z buffer that records the farthest distances instead of the nearest.

An alternative to using the depth estimation buffer might be to read the accurate depth values back from a hardware Z buffer after rendering the occluders. This approach was not taken mainly because it involves further assumptions of hardware features (i.e. there is a hardware Z-buffer, and we are able to read Z-values reasonably fast in a easily-usable format).

Construction of the depth estimation buffer: The depth estimation buffer is built at every frame, which requires determining the pixels to which the occluders project on the image plane. Scan-converting the occluders to do this would be unacceptably expensive. As we did in constructing occlusion maps, we conservatively estimate the projection and depth of an occluder by its screen-space bounding rectangle and the Z value of its bounding volume's farthest vertex. The algorithm checks each buffer entry covered by the rectangle for possible updates. If the rectangle's Z value is greater than the old entry, the entry is updated. This process is repeated for all occluders.

Conservative Depth Test: To perform the conservative depth test on a potential occludee, it is approximated by the screen space bounding rectangle of its bounding box (in the same manner as in overlap tests), which is assigned a depth value the same as that of the nearest vertex on the bounding box. Each entry of the depth estimation buffer covered by the rectangle is checked to see if any entry is greater than the rectangle's Z value. If this is the case then the object is conservatively regarded as being partly in front of the union of all occluders and thus must be rendered.

The cost of the conservative Z-buffer test and update, though far cheaper than accurate operations, can still be expensive as the resolution of the depth estimation buffer increases. Furthermore, since we are performing a conservative estimation of the objects' screen space extents, there is a point where increasing the resolution of the depth estimation buffer does not help increase the accuracy of depth information. Normally the algorithm uses only a coarse resolution (e.g. 64×64).

5.3 Occluder Selection

At each frame, the algorithm selects an occluder set. The *optimal* set of occluders is exactly the visible portion of the model. Finding this optimal set is the visible surface computation problem itself. Another possibility is to pre-compute global visibility information for computing the useful occluders at every viewpoint. The fastest known algorithm for computing the effects on global visibility due to a single polyhedron with m vertices can take $O(m^6 \log m)$ time in the worst case [GCS91].

We present algorithms to estimate a set of occluders that are used to cull a significant fraction of the model. We perform preprocessing to derive an occluder database from the model. At runtime the algorithm dynamically selects a set of occluders from that database.

5.3.1 Building the Occluder Database

The goal of the pre-processing step is to discard objects which do not serve as good occluders from most viewpoints. We use the following criteria to select good occluders from the model database:

- **Size:** Small objects will not serve as good occluders unless the viewer is very close to them.

[1] By definition, a pixel is completely opaque if its value is above or equal to the *opacity threshold*, which is defined in Section 5.4.

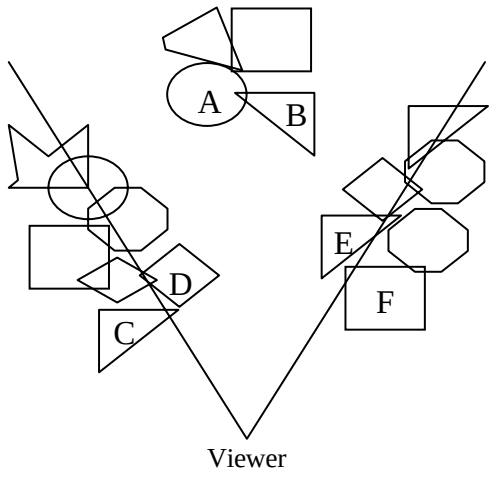


Figure 5: *Distance criterion for dynamic selection*

- **Redundancy:** Some objects, e.g. a clock on the wall, provide redundant occlusion and should be removed from the database.
- **Rendering Complexity:** Objects with a high polygon count or rendering complexity are not preferred, as scan-converting them may take considerable time and affect the overall frame rate.

5.3.2 Dynamic Selection

At runtime, the algorithm selects a set of objects from the occluder database. The algorithm uses a distance criterion, size, and temporal coherence to select occluders.

The single Z-plane method for depth comparison, presented in Section 5.2.1, is also an occluder selection method. All objects not completely beyond the Z-plane are occluders.

When the algorithm uses the depth estimation buffer, it dynamically selects occluders based on a distance criterion and a limit ($\mathcal{L}$) on the number of occluder polygons. These two variables may vary between frames as a function of the overall frame rate and percentage of model culled. Given $\mathcal{L}$, the algorithm tries to find a set of good occluders whose total polygon count is less than $\mathcal{L}$.

The algorithm considers each object in the occluder database lying in the viewing frustum. The distance between the viewer and the center of an object's bounding volume is used as an estimate of the distance from the viewer to the object. The algorithm sorts these distances, and selects the nearest objects as occluder until their combined polygon count exceeds $\mathcal{L}$. This works well for most situations, except when a good occluder is relatively far away. One such situation has been shown in Figure 5. The distance criterion will select C, D, E, F, etc. as occluders, but $\mathcal{L}$ will probably be exceeded *before* A and B are selected. Thus, we lose occlusion that would have been contributed by A and B. In other words, there is a hole in the occlusion map which decreases the culling rate.

Dynamic occluder selection can be assisted by visibility preprocessing of the occluder scene. The model space is subdivided by a uniform grid. Visibility is sampled at each grid point by surrounding the grid point with a cube and using an item buffer algorithm similar to the hemi-cube algorithm used in radiosity. Each grid point gets a lists of visible objects. At run-time, occluders can be chosen from visible object lists of grid points nearest to the viewing point.

5.4 Approximate Visibility Culling

A unique feature of our algorithm is to perform approximate visibility culling, which ignores objects only visible through small holes in or among the occluders. This ability is based on an inherent property of HOM: it naturally represents the combined occluder projections at different levels of detail.

In the process of filtering maps to build the hierarchy, a pixel in a low resolution map can obtain a high opacity value even if a small number of its descendant pixels have low opacity. Intuitively, a small group of low-opacity pixels (a "hole") in a high-resolution map can dissolve as the average operation (which involves high opacity values from neighboring pixels) is recursively applied to build lower-resolution maps.

The opacity value above which the pixel is considered completely opaque is called the *opacity threshold*, which is by default 1.0. The visibility culling algorithm varies the degree of approximation by changing the opacity threshold. As the threshold is lowered, the culling algorithm becomes more approximate. This effect of the opacity threshold is based on the fact that if a pixel is considered completely opaque, the culling algorithm does not go into the descendant pixels for further opacity checking. If the opacity of a pixel in a low-resolution map is not 1.0 (because some of the pixel's descendents have low opacities), but is still higher than the opacity threshold assigned to that map, the culling algorithm does not descend to the sub-pixels to find the low opacities. In effect, some small holes in higher-resolution maps are ignored. The opacity threshold specifies the size of the holes that can be ignored; the higher the threshold, the smaller the ignorable holes.

The opacity thresholds for each level of the hierarchy are computed by first deciding the maximum allowable size of a hole. For example, if the final image is 1024×1024 and a map is 64×64, then a pixel in the map corresponds to 16×16 pixels in the final image. If we consider 25 black pixels among 16×16 total pixels an ignorable hole, then the opacity threshold for the map is $1 - 25/(16 * 16) = 0.90$. Note that we are considering the worst case where the black pixels gather together to form the biggest hole, which is roughly a 5×5 black block. One level up the map hierarchy, where resolution is 32×32 and where a map pixel corresponds to 32×32 screen pixels, the threshold becomes $1 - 25/(32 * 32) = 0.98$.

Consider the k-th level of the hierarchy. Let n black pixels among m total pixels form an ignorable hole, then the opacity threshold is $O_k = 1 - \frac{n}{m}$. Since at the $k+1$-th level[2] each map pixel corresponds to four times as many pixels in the final image, the opacity threshold is

$$O_{k+1} = 1 - \frac{n}{4m} = 1 - \frac{1 - O_k}{4} = \frac{3 + O_k}{4}.$$

Let the opacity threshold in the highest resolution map be O_{min}. If a pixel in a lower resolution map has opacity lower than O_{min}, then it is not possible for all its descendant pixels have opacities greater than O_{min}. This means that if a high-level pixel is completely covered by a bounding rectangle and its opacity is lower than O_{min}, we can immediately conclude that the corresponding object is potentially visible. For pixels not completely covered by the rectangle (i.e. pixels intersecting the rectangle's edges), the algorithm always descends into sub-pixels.

To summarize the cases in the overlap test, a piece of pseudocode is provided in 5.4.

Approximate visibility is useful because in many cases we don't expect to see many meaningful parts of the model through small holes in or among the occluders. Culling such portions of the model usually does not create noticeable visual artifacts.

[2] Remember that highest resolution map is level 0. See Figure 3.

```
CheckPixel(HOM, Level, Pixel, BoundingRect)
{
    Op = HOM[Level](Pixel.x, Pixel.y);
    Omin = HOM[0].OpacityThreshold;

    if (Op > HOM[Level].OpacityThreshold)
        return TRUE;
    else if (Level = 0)
        return FALSE;
    else if (Op < Omin AND
             Pixel.CompletelyInRect = TRUE)
        return FALSE;
    else
    {
        Result = TRUE;
        for each sub-pixel, Sp, that
               overlaps BoundingRect
        {
            Result = Result AND CheckPixel(HOM,
                   Level-1, Sp, BoundingRect);
            if Result = FALSE
        return FALSE;
        }
    }
    return TRUE;
}

OverlapTest(HOM, Level, BoundingRect)
{
    for each pixel, P, in HOM[HOM.HighestLevel]
            that intersects BoundingRect
    {
        if (CheckPixel(HOM, HOM.HighestLevel, P)
                = FALSE)
            return FALSE;
    }
    return TRUE
}
```

Figure 8: *Pseudo-code for the overlap test between the occlusion map hierarchy and a bounding rectangle. This code assumes that necessary information is available as fields in the HOM and Pixel structures. The meaning of the fields are easily inferred from their names. The* `CheckPixel` *function check the opacity of a pixel, descending into sub-pixels as necessary. The* `OverlapTest` *function does the whole overlap test, which returns* `TRUE` *if bounding rectangle falls within completely opaque areas and* `FALSE` *otherwise.*

Omitting such holes can significantly increase the culling rate if many objects are potentially visible only through small holes. In Figure 6 and Figure 7, we illustrate approximate culling on an environment with trees as occluders.

It should be noted that in some situations approximate culling may result in noticeable artifacts, even if the opacity threshold is high. For example, if objects visible only through small holes are very bright (e.g. the sun beaming through holes among leaves of a tree), then strong popping can be observed as the viewer zooms closer. In such cases approximate culling should not be applied. Furthermore, approximate culling decreases accuracy of culling around the edges of occluders, which can also result in visual artifacts.

5.5 Dynamic Environments

The algorithm easily extends to dynamic environments. As no static bounding volume hierarchy may be available, the algorithm uses oriented bounding boxes around each object. The occluder selection algorithm involves no pre-processing, so the occluder database is exactly the model database. The oriented bounding boxes are used to construct the depth estimation buffer as well as to perform the overlap test with the occlusion map hierarchy.

6 Implementation and Performance

We have implemented the algorithm as part of a walkthrough system, which is based on OpenGL and currently runs on SGI platforms. Significant speed-ups in frame rates have been observed on different models. In this section, we discuss several implementation issues and discuss the system's performance on SGI Max Impacts and Infinite Reality platforms.

6.1 Implementation

As the first step in creating the occlusion map hierarchy, occluders are rendered in a 256×256 viewport in the back framebuffer, in full white color, with lighting and texture mapping turned off. Any one of the three color channels of the resulting image can serve as the highest-resolution occlusion map on which the hierarchy is based. An alternate method could be to render the occluders with the original color and shading parameters and use the α channel of the rendered image to construct the initial map. However, for constructing occlusion maps we do not need a "realistic" rendering of the occluders, which may be more expensive. In most cases the resolution of 256×256 is smaller than that of the final rendering of the model. As a result, it is possible to have artifacts in occlusion. In practice, if the final image is rendered at a resolution of 1024×1024, rendering occluders at 256×256 is a good trade-off between accuracy and time required to filter down the image in building the hierarchy.

To construct the occlusion map hierarchy, we recursively average 2×2 blocks of pixels using the texture mapping hardware as well as the host CPU. The resolution of the lowest-resolution map is typically 4×4. The break-even point between hardware and software hierarchy construction (as described in Section 4.2) varies with different graphics systems. For SGI Maximum Impacts, we observed the shortest construction time when the algorithm filters from 256×256 to 128×128 using texture-mapping hardware, and from 128×128 to 64×64 and finally down to 4×4 on the host CPU. For SGI InfiniteReality, which has faster pixel transfer rates, the best performance is obtained by filtering from 256×256 to 64×64 using the hardware and using the host CPU thereafter. Hierarchy construction time is about 9 milliseconds for the Max Impacts and 4 milliseconds for the Infinite Reality, with a small variance (around 0.5 milliseconds) between frames.

The implementation of the depth estimation buffer is optimized for block-oriented query and updates. The hierarchical overlap test is straight-forward to implement; It is relatively harder to optimize, as it is recursive in nature.

6.2 Performance

We demonstrate the performance of the algorithm on three models. These are:

- **City Model:** this is composed of models of London and has $312,524$ polygons. A bird's eye view of the model has been shown in Figure 11.
- **Dynamic Environment:** It is composed of dinosaurs and teapots, each undergoing independent random motion. The total polygon count is $986,800$. It is shown in Figure 12.
- **Submarine Auxiliary Machine Room (AMR):** It is a real-world CAD model obtained from industrial sources. The

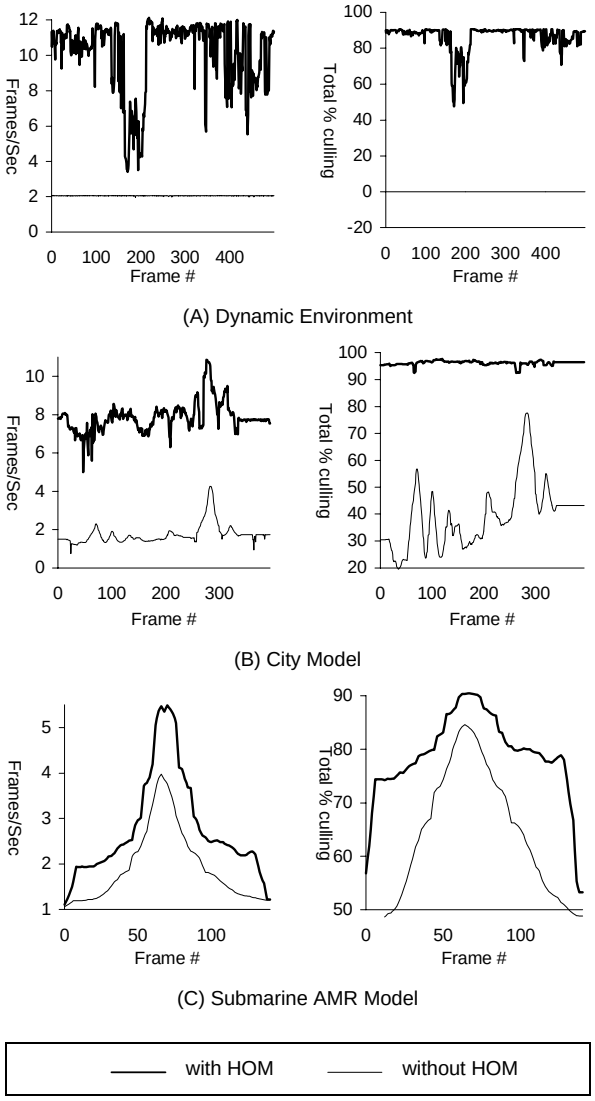


Figure 9: *The speed-up obtained due to HOM on different models. The left graphs show the improvement in frame rate and the right graphs show the percentage of model culled. The statistics were gathered over a path for each model.*

model has $632,252$ polygons. Different views of the model are shown in Figure 1 and Figure 13.

As mentioned earlier, our algorithm uses a bounding volume hierarchy (i.e. a scene graph) for both the original model database as well as the occluder database. Each model we used is originally a collection of polygons with no structure information. We construct an axis-aligned bounding box hierarchy for each database.

For the dynamic environment and the city model, we use the model database itself as the occluder database, without any preprocessing for static occluder selection. For the AMR model, the pre-processing yields an occluder database of $217,636$ polygons. The algorithm removes many objects that have little potential of being a good occluder (like the bolts on the diesel engine, thin pipes etc.) from the original model. Further, most of these parts are densely tessellated, making them to expensive to be directly used as occluders. We use the simplified version of the parts which are produced by algorithms in [Cohen96]. Although many simplification algorithms give good error bounds on the simplified model, they do not guarantee that the projection of the simplified object lies within that of the original. Therefore, visibility artifacts may be introduced by the simplified occluders. We use very tight error bounds so that artifacts are rarely noticeable.

The performance of the algorithms has been highlighted in Figure 9. The graphs on the left show the frame rate improvement, while the graphs on the right highlight the percentage of the model culled at every frame. The performance of the city model was generated on a SGI Maximum Impact while the other two were rendered on an SGI Infinite Reality. The actual performance varies due to two reasons:

1. Different models have varying depth complexities. Furthermore, the percentage of occlusion varies with the viewpoint.
2. The ability of the occluder selection algorithm to select the "right" subset of occluders. The performance of the greedy algorithm, e.g. distance based criterion, varies with the model distribution and the viewpoint.

The occluder polygon count budget ($\mathcal{L}$) per frame is important for the performance of the overall algorithm. If too few occluders are rendered, most of the pixels in the occlusion map have low opacities and the algorithm is not able to cull much. On the other hand, if too many occluder polygons are rendered, they may take a significant percentage of the total frame time and slow down the rendering algorithm. The algorithm starts with an initial guess on the polygon count and adaptively modifies it based on the percentage of the model culled and frame rate. If the percentage of the model culled is low, it increases the count. If the percentage is high and the frame rate is low, it decreases the count.

Average time spent in the different stages of the algorithm (occluder selection and rendering, hierarchy generation, occlusion culling and final rendering) has been shown in Figure 10. The average time to render the model without occlusion culling is normalized to 100%. In these cases, the average time in occluder rendering varies between $10-25\%$.

7 Analysis and Comparison

In this section we analyze some of the main features of our algorithm and compare it with other approaches.

Our algorithm is generally applicable to all models and obtains significant culling when there is high depth complexity. This is mainly due to its use of occlusion maps to combine occluders in image space. The extensive use of screen space bounding rectangles as an approximation of the object's screen space projection makes the overlap tests and depth tests fast and cheap.

In terms of hardware assumptions, the algorithm requires only the ability to read back the framebuffer. Texture mapping with bilinear interpolation, when available, can be directly used to accelerate the construction of the occlusion map hierarchy.

In general, if the algorithm is spending a certain percentage of the total frame time in occluder rendering, HOM generation and culling (depth test and overlap test), it should at least cull away a similar percentage of the model so as to justify the overhead of occlusion culling. If a model under some the viewing conditions does *not* have sufficient occlusion, the overall frame rate may decrease due to the overhead, in which case occlusion culling should be turned off.

7.1 Comparison to Object Space Algorithms

Work on cells and portals[ARB90, TS91, LG95] addresses a special class of densely occluded environments where there are plenty

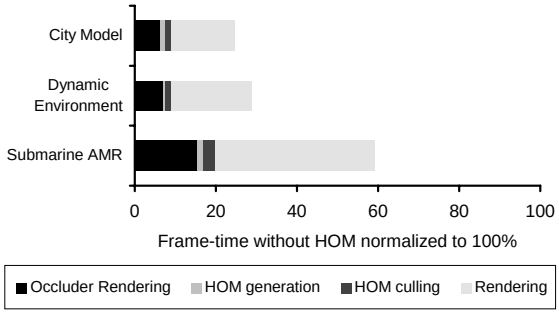


Figure 10: *Average speed-up obtained due to HOM culling on different models. The total time to render each model without HOM culling is normalized to 100%. Each bar shows the percentage of time spent in different stages of our algorithm.*

of cell and portal structures, as in an indoor architectural model. [ARB90, TS91] pre-processes the model to identify potentially visible set of primitives for each cell. [LG95] developed a dynamic version which eliminates the visibility pre-processing. These methods work very well for this particular type of environment, but are not applicable to models without cell/portal structures.

Our algorithm works without modification for environments with cells and portals, but occluder selection can be optimized for these environments. The cell boundaries can be used to form the occluder database. As an alternative, we can fill a viewport with white pixels and then render the portals in black to form the occlusion map. In general, however, we do not expect to outperform the specialized algorithms in cell/portal environments.

Two different object space solutions for more general models have been proposed by [CT96a, CT96b] and [Hud96]. They dynamically choose polygons and convex objects (or simple convex combination of polygons) as occluders and use them to cull away invisible portions of the model. However, many models do not have single big convex occluders. In such cases, merging small, irregular occluders is critical for significant culling, which is a difficult task in object space. Our algorithm lies between object space and image space and the occluder merging problem is solved in image space.

7.2 Comparison with Hierarchical Z-buffer Algorithm

In many ways, we present an alternative approach to hierarchical Z-buffer visibility [GKM93]. The main algorithm presented in [GKM93] performs updates of the Z-buffer hierarchy as geometry is rendered. It assumes special-purpose hardware for fast depth updating and querying to obtain interactive performance. It is potentially a very powerful and effective algorithm for visibility culling. However, we are not aware of any hardware implementation.

There is a possible variation of hierarchical Z-buffer algorithm which selects occluders, renders them, reads back the depth buffer once per frame, builds the Z-pyramid, and use the screen-space bounding rectangles for fast culling. The algorithm proposed in [GKM93] uses the exact projection of octree nodes, which requires software scan-conversion. In this case, the HOM approach and the hierarchical Z-buffer are comparable, each with some advantages over the other.

The HOM approach has the following advantages:

1. There is no need for a Z-buffer. Many low-end systems do not support a Z-buffer and some image generators for flight simulators do not have one. Tile-based architectures like PixelFlow[MEP92] and Talisman[TK96] do not have a full-screen Z-buffer, but instead have volatile Z-buffers the size of a single tile. This makes getting Z values for the entire screen rather difficult.
2. The construction of HOM has readily-available hardware support (in the form of texture mapping with bilinear interpolation) on many graphics systems. Further, if filtering is performed in software, the cost of the average operator is smaller than the minimum/maximum operator (due to the absence of branch instructions).
3. HOM supports conservative early termination in the hierarchical test by using a transparency threshold (Section 5.1) and approximate occlusion culling by using an opacity threshold (Section 5.4). These features result from using an average operator.

On the other hand, the Hierarchical Z-buffer has depth values, which the HOM algorithm has to manage separately in the depth estimation buffer. This results in the following advantages of Hierarchical Z-buffer:

1. Culling is less conservative.
2. It is easier to use temporal coherence for occluder selection because nearest Z values for objects are available in the Z-buffer. Updating the active occluder list is more difficult in our algorithm since we only have estimated farthest Z values.

7.3 Comparison with Hierarchical Tiling with Coverage Masks

Hierarchical polygon tiling [Gre96] tiles polygons in front-to-back order and uses a "coverage" pyramid for visibility culling. The coverage pyramid and hierarchical occlusion maps serve the same purpose in that they both record the aggregate projections of objects. (In this sense, our method has more resemblance to hierarchical tiling than to the hierarchical Z-buffer.) However, a pixel in a mask in the coverage pyramid has only three values (covered, vacant or active), while a pixel in an occlusion map has a continuous opacity value. This has lead to desirable features, as discussed above. Like HOM, the coverage masks do not contain depth information and the algorithm in [Gre96] uses a BSP-tree for depth-ordering of polygons. Our algorithm is not restricted to rendering the polygons front to back. Rather, it only needs a conservatively estimated boundary between the occluders and potential occludees, which is represented by the depth estimation buffer. Hierarchical tiling is tightly coupled with polygon scan-conversion and has to be significantly modified to deal with non-polygonal objects, such as curved surfaces or textured billboards. Our algorithm does not directly deal with low-level rendering but utilizes existing graphics systems. Thus it is readily applicable to different types of objects so long as the graphics system can render them. Hierarchical tiling requires special-purpose hardware for real-time performance.

8 Future Work and Conclusion

In this paper we have presented a visibility culling algorithm for general models that achieves significant speedups for interactive walkthroughs on current graphics systems. It is based on hierarchical occlusion maps, which represent an image space fusion of all the occluders. The overall algorithm is relatively simple, robust and easy to implement. We have demonstrated its performance on a number of large models.

There are still several areas to be explored in this research. We believe the most important of these to be occlusion preserving simplification algorithms, integration with levels-of-detail modeling, and parallelization.

Occlusion Preserving Simplification: Many models are densely tessellated. For fast generation of occlusion maps, we do not want to spend considerable time in rendering the occluders. As a result, we are interested in simplifying objects under the constraint of occlusion preservation. This implies that the screen space projection of the simplified object should be a subset of that of the original object. Current polygon simplification algorithms can reduce the polygon count while giving tight error bounds, but none of them guarantees an occlusion preserving simplification.

Integration with Level-of-Detail Modeling: To display large models at interactive frame rates, our visibility culling algorithm needs to be integrated with level-of-detail modeling. The latter involves polygon simplification, texture-based simplification and dynamic tessellation of higher order primitives.

Parallelization: Our algorithm can be easily parallelized on multi-processor machines. Different processors can be used for view frustum culling, overlap tests and depth tests.

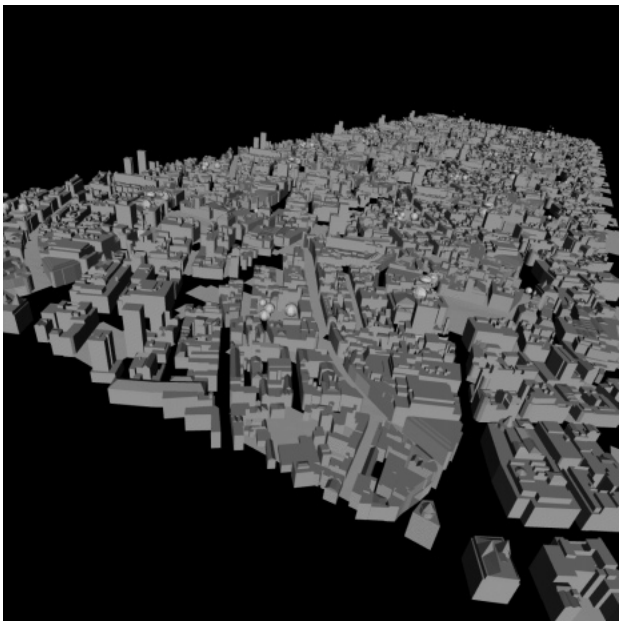


Figure 11: *City model with 312,524 polygons. Average speed-up obtained by our visibility culling algorithm is about five times.*

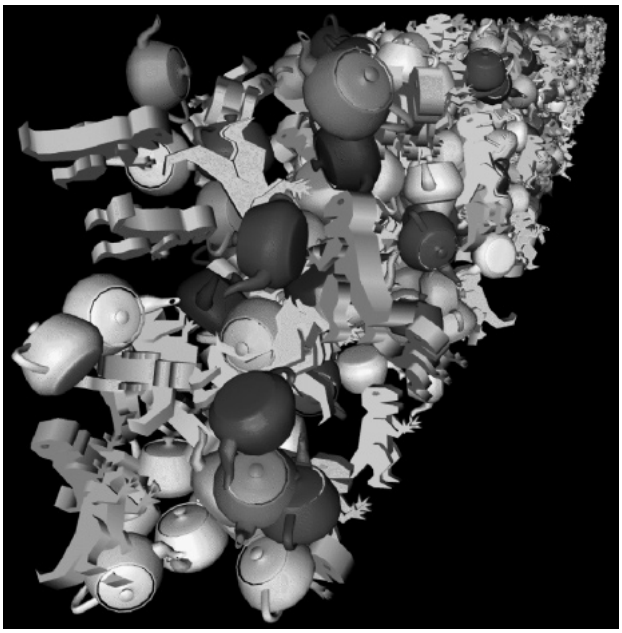


Figure 12: *Dynamic environment composed of dinosaurs and teapots. The total polygon count is 986,800. The HOM algorithm achieves about five times speed-up.*

9 Acknowledgements

We are grateful to the reviewers for their comments and to Fred Brooks, Gary Bishop, Jon Cohen, Nick England, Ned Greene, Anselmo Lastra, Ming Lin, Turner Whitted, and members of UNC Walkthrough project for productive discussions. The Auxiliary Machine Room model was provided to us by Greg Angelini, Jim Boudreaux and Ken Fast at Electric Boat, a subsidiary of General Dynamics. Thanks to Sarah Hoff for proofreading the paper.

This work was supported in part by an Alfred P. Sloan Foundation Fellowship, ARO Contract DAAH04-96-1-0257, DARPA Contract DABT63-93-C-0048, Intel Corp., NIH/National Center for Research Resources Award 2 P41RR02170-13 on Interactive Graphics for Molecular Studies and Microscopy, NSF grant CCR-9319957 and Career Award, an ONR Young Investigator Award, the NSF/ARPA Center for Computer Graphics and Scientific Visualization, and a UNC Board of Governors Fellowship.

References

[ARB90] J. Airey, J. Rohlf, and F. Brooks. Towards image realism with interactive update rates in complex virtual building environments. In *Symposium on Interactive 3D Graphics*, pages 41–50, 1990.

[BM96] D. Blythe and T. McReynolds. Programming with Opengl: Advanced course. *Siggraph'96 Course Notes*, 1996.

[Bun89] M. Bunker and R. Economy. Evolution of GE CIG Systems, SCSD document, General Electric Company, Daytona Beach, FL, 1989

[Car84] L. Carpenter. The A-buffer, an antialiased hidden surface method. *Proc. of ACM Siggraph*, pages 103–108, 1984.

[Cat74] E. Catmull. *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, University of Utah, 1974.

[Chu94] J. C. Chauvin (Sogitec). An advanced Z-buffer technology. IMAGE VII, pages 76–85, 1994.

[Cla76] J.H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[CT96a] S. Coorg and S. Teller. A spatially and temproally coherent object space visibility algorithm. Technical Report TM 546, Laboratory for Computer Science, Massachusetts Institute of Technology, 1996.

[CT96b] S. Coorg and S. Teller. Temporally coherent conservative visibility. In *Proc. of 12th ACM Symposium on Computational Geometry*, 1996.

[Dor94] S. E. Dorward. A survey of object-space hidden surface removal. *Internat. J. Comput. Geom. Appl.*, 4:325–362, 1994.

[FDHF90] J. Foley, A. Van Dam, J. Hughes, and S. Feiner. *Computer Graphics: Principles and Practice*. Addison Wesley, Reading, Mass., 1990.

[FKN80] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Proc. of ACM Siggraph*, 14(3):124–133, 1980.

[GBR91] R. Coifman G. Beylkin and V. Rokhlin. Fast wavelet transforms and numerical algorithms: I. *Communications of Pure and Applied Mathematics*, 44(2):141–183, 1991.

[GBW90] B. Garlick, D. Baum, and J. Winget. Interactive viewing of large geometric databases using multiprocessor graphics workstations. *Siggraph'90 course notes: Parallel Algorithms and Architectures for 3D Image Generation*, 1990.

[GCS91] Z. Gigus, J. Canny, and R. Seidel. Efficiently computing and representing aspect graphs of polyhedral objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):542–551, 1991.

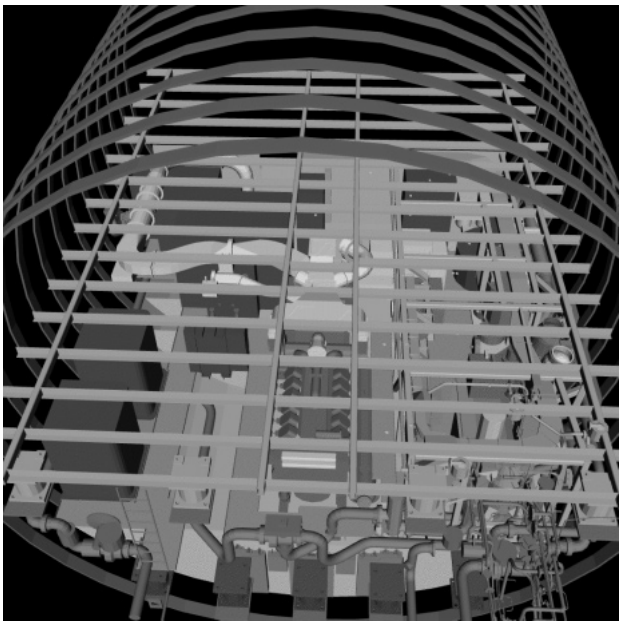


Figure 13: *A top view of the auxiliary machine room of a submarine composed of 632,252 polygons. Average speed-up is about two due to occlusion culling.*

[GK94] N. Greene and M. Kass. Error-bounded antialiased rendering of complex environments. In *Proc. of ACM Siggraph*, pages 59–66, 1994.

[GKM93] N. Greene, M. Kass, and G. Miller. Hierarchical Z-buffer visibility. In *Proc. of ACM Siggraph*, pages 231–238, 1993.

[Geo95] C. Georges. Obscuration culling on parallel graphics architectures. Technical Report TR95-017, Department of Computer Science, University of North Carolina, Chapel Hill, 1995.

[Gre95] N. Greene. *Hierarchical rendering of complex environments.* PhD thesis, University of California at Santa Cruz, 1995.

[Gre96] N. Greene. Hierarchical polygon tiling with coverage masks. In *Proc. of ACM Siggraph*, pages 65–74, 1996.

[Hud96] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff and H. Zhang. Accelerated occlusion culling using shadow frusta. Technical Report TR96-052, Department of Computer Science, University of North Carolina, 1996. To appear in Proc. of ACM Symposium on Computational Geometry, 1997.

[Lat94] R. Latham (CGSD). Advanced image generator architectures. Course reference material, 1994.

[LG95] D. Luebke and C. Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *ACM Interactive 3D Graphics Conference*, Monterey, CA, 1995.

[LORA] Loral ADS. GT200T Level II image generator product overview, Bellevue, WA.

[McK87] M. McKenna. Worst-case optimal hidden-surface removal. *ACM Trans. Graph.*, 6:19–28, 1987.

[MEP92] S. Molnar, J. Eyles and J. Poulton. PixelFlow: High speed rendering using image composition. *Proc. of ACM Siggraph*, pp. 231-248, 1992.

[Mue95] C. Mueller. Architectures of image generators for flight simulators. Technical Report TR95-015, Department of Computer Science, University of North Carolina, Chapel Hill, 1995.

[Mul89] K. Mulmuley. An efficient algorithm for hidden surface removal. *Computer Graphics*, 23(3):379–388, 1989.

[Nay92] B. Naylor. Partitioning tree imge representation and generation from 3d geometric models. In *Proc. of Graphics Interface*, pages 201–12, 1992.

[RB96] R. Brechner et al. Interactive walkthrough of large geometric databases. *Siggraph'96 course notes*, 1996.

[RH94] J. Rohlf and J. Helman. Iris performer: A high performance multiprocessor toolkit for realtime 3d graphics. In *Proc. of ACM Siggraph*, pages 381–394, 1994.

[SBM+94] B. Schneider, P. Borrel, J. Menon, J. Mittleman, and J. Rossignac. Brush as a walkthroughsystem for architectural models. In *Fifth Eurographics Workshop on Rendering*, pages 389–399, July 1994.

[SG96] O. Sudarsky and C. Gotsman. Output sensitive visibility algorithms for dynamic scenes with applications to virtual reality. *Computer Graphics Forum*, 15(3):249–58, 1996. Proc. of Eurographics'96.

[TK96] J. Torborg and J. Kajiya. Talisman: Commodity Realtime 3D Graphics for the PC. *Proc. of ACM Siggraph*, pp. 353-363, 1996.

[TP75] S. Tanimoto and T. Pavlidis. A hierarchical data structure for picture processing. *Computer Graphics and Image Processing*, 4(2):104–119, 1975.

[TS91] S. Teller and C.H. Sequin. Visibility preprocessing for interactive walkthroughs. In *Proc. of ACM Siggraph*, pages 61–69, 1991.

[War69] J. Warnock. A hidden-surface algorithm for computer generated halftone pictures. Technical Report TR 4-15, NTIS AD-753 671, Department of Computer Science, University of Utah, 1969.

[Wil83] L. Williams. Pyramidal parametrics. *ACM Computer Graphics*, pages 1–11, 1983.

[YR96] R. Yagel and W. Ray. Visibility computations for efficient walkthrough of complex environments. *Presence*, 5(1):1–16, 1996.

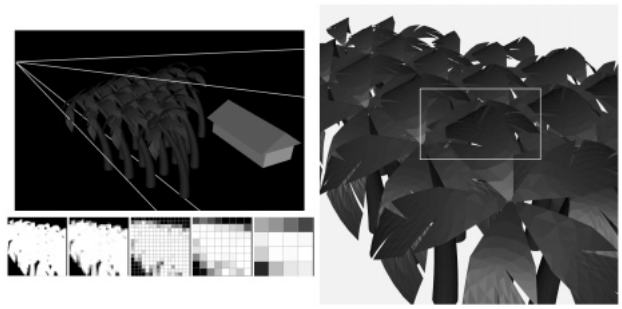


Figure 6: *These images show a view of an environment composed of trees and a house, with the trees as oclcuders. The green line in the left image indicates the view-frustum. The right image highlights the holes among the leaves with a yellow background. The screen space bounding rectangle of the house is shown in cyan. The occlusion map hierarchy is shown on the left.*

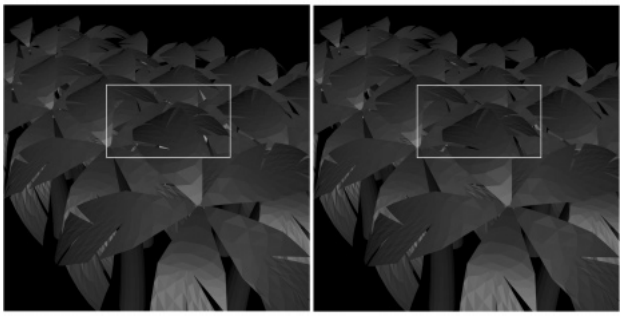


Figure 7: *Two images from the same view as in Figure 6. The left image is produced with no approximate culling. The right image uses opacity threshold values from* 0.7 *for the highest resolution map up to* 1.0 *for the lowest resolution map.*

Real-Time, Continuous Level of Detail Rendering of Height Fields

Peter Lindstrom* David Koller* William Ribarsky*
Larry F. Hodges* Nick Faust† Gregory A. Turner‡

*† Georgia Institute of Technology

‡ SAIC

Abstract

We present an algorithm for real-time level of detail reduction and display of high-complexity polygonal surface data. The algorithm uses a compact and efficient regular grid representation, and employs a variable screen-space threshold to bound the maximum error of the projected image. A coarse level of simplification is performed to select discrete levels of detail for blocks of the surface mesh, followed by further simplification through repolygonalization in which individual mesh vertices are considered for removal. These steps compute and generate the appropriate level of detail dynamically in real-time, minimizing the number of rendered polygons and allowing for smooth changes in resolution across areas of the surface. The algorithm has been implemented for approximating and rendering digital terrain models and other height fields, and consistently performs at interactive frame rates with high image quality.

1 INTRODUCTION

Modern graphics workstations allow the display of thousands of shaded or textured polygons at interactive rates. However, many applications contain graphical models with geometric complexity still greatly exceeding the capabilities of typical graphics hardware. This problem is particularly prevalent in applications dealing with large polygonal surface models, such as digital terrain modeling and visual simulation.

In order to accommodate complex surface models while still maintaining real-time display rates, methods for approximating the polygonal surfaces and using multiresolution models have been proposed [13]. Simplification algorithms can be used to generate multiple surface models at varying levels of detail, and techniques are employed by the display system to select and render the appropriate level of detail model.

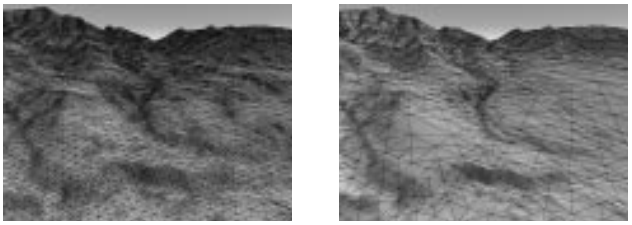


Figure 1: Terrain surface tessellations corresponding to projected geometric error thresholds of one (left) and four (right) pixels.

In this paper we present a new level of detail display algorithm that is applicable to surfaces that are represented as uniformly-gridded polygonal height fields. By extending the regular grid representation to allow polygons to be recursively combined where appropriate, a mesh with fewer polygons can be used to represent the height field (Figure 1). Such small, incremental changes to the mesh polygonalization provide for continuous levels of detail and a near optimal tessellation for any given viewpoint. The algorithm is characterized by the following set of features:

- **Large reduction in the number of polygons to be rendered.** Typically, the surface grid is decimated by several orders of magnitude with no or little loss in image quality, accommodating interactive frames rates for smooth animation.
- **Smooth, continuous changes between different surface levels of detail.** The number and distribution of rendered polygons change smoothly between successive frames, affording maintenance of consistent frame rates.
- **Dynamic generation of levels of detail in real-time.** The need for expensive generation of multiresolution models ahead of time is eliminated, allowing dynamic changes to the surface geometry to be made with little computational cost.
- **Support for a user-specified image quality metric.** The algorithm is easily controlled to meet an image accuracy level within a specified number of pixels. This parameterization allows for easy variation of the balance between rendering time and rendered image quality.

Related approaches to polygonal surface approximation and multiresolution rendering are discussed in the next section. The following sections of the paper describe the theory and procedures necessary for implementing the real-time continuous rendering algorithm. We conclude the paper by empirically evaluating the algorithm with results from its use in a typical application.

*Graphics, Visualization, & Usability Center, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332–0280. {lindstro, koller, ribarsky, hodges}@cc.gatech.edu.

†Center for GIS and Spatial Analysis Technologies, Georgia Tech Research Institute. nick.faust@gtri.gatech.edu.

‡Simulation Technology Division. gturner@std.saic.com.

2 RELATED WORK

A large number of researchers have developed algorithms for approximating terrains and other height fields using polygonal meshes. These algorithms attempt to represent surfaces with a given number of vertices, or within a given geometric error metric, or in a manner that preserves application specific critical features of the surface. Uniform grid methods or irregular triangulations are employed to represent the surfaces, and techniques including hierarchical subdivisions and decimations of the mesh are used for simplification and creation of multiresolution representations.

Much of the previous work on polygonalization of terrain-like surfaces has concentrated on triangulated irregular networks (TINs). A number of different approaches have been developed to create TINs from height fields using Delaunay and other triangulations [9, 10, 19], and hierarchical triangulation representations have been proposed that lend themselves to usage in level of detail algorithms [3, 4, 18]. TINs allow variable spacing between vertices of the triangular mesh, approximating a surface at any desired level of accuracy with fewer polygons than other representations. However, the algorithms required to create TIN models are generally computationally expensive, prohibiting use of dynamically created TINs at interactive rates.

Regular grid surface polygonalizations have also been implemented as terrain and general surface approximations [2, 7]. Such uniform polygonalizations generally produce many more polygons than TINs for a given level of approximation, but grid representations are typically more compact. Regular grid representations also have the advantage of allowing for easier construction of a multiple level of detail hierarchy. Simply subsampling grid elevation values produces a coarser level of detail model, whereas TIN models generally require complete retriangulation in order to generate multiple levels of detail.

Other surface approximation representations include hybrids of these techniques, and methods that meet application specific criteria. Fowler and Little [9] construct TINs characterized by certain "surface specific" points and critical lines, allowing the TIN representation to closely match important terrain features. Douglas [5] locates specific terrain features such as ridges and channels in a terrain model database, and represents the surface with line segments from these "information rich" features. This method generates only a single surface approximation, however, and is not easily adapted to produce multiresolution models. Gross et al. [12] use a wavelet transform to produce adaptive surface meshing from uniform grid data, allowing for local control of the surface level of detail. This technique, however, has not yet proven to yield interactive frame rates. The general problem of surface simplification has been addressed with methods for mesh decimation and optimization [14, 20], although these techniques are not suitable for on-the-fly generation of multiple levels of detail.

The issue of "continuous" level of detail representations for models has been addressed both for surfaces and more general modeling. Taylor and Barret [22] give an algorithm for surface polygonalization at multiple levels of detail, and use "TIN morphing" to provide for visually continuous change from one resolution to another. Many visual simulation systems handle transitions between multiple levels of detail by alpha blending two models during the transition period. Ferguson [8] claims that such blending techniques between levels of detail may be visually distracting, and discusses a method of Delaunay triangulation and triangle subdivision which smoothly matches edges across areas of different resolution.

3 MOTIVATION

The algorithm presented in this paper has been designed to meet a number of criteria desirable for a real-time level of detail (LOD) algorithm for height fields. These characteristics include:

(i) 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.

(ii) 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.

(iii) High frequency data such as localized convexities and concavities, and/or local changes to the geometry, should not have a widespread global effect on the complexity of the model.

(iv) Small changes to the view parameters (e.g. 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.

(v) 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.

Note that some applications do not require the satisfaction of all of these criteria. However, a polygon-based level of detail algorithm that supports all of these features is clearly of great importance in areas such as terrain rendering, which often requires both high frame rates and high visual fidelity, as well as fast and frequent queries of a possibly deformable terrain surface. Our algorithm successfully achieves all of the goals listed above.

Most contemporary approaches to level of detail management fail to meet at least one of these five criteria. TIN models, for example, do not in general meet the first two criteria. Generation of even modest size TINs requires extensive computational effort. Because TINs are non-uniform in nature, surface following (e.g. for animation of objects on the surface) and intersection (e.g. for collision detection, selection, and queries) are hard to handle efficiently due to the lack of a spatial organization of the mesh polygons. The importance of (ii) is relevant in many applications, such as games and military applications, where dynamic deformations of the mesh occur, e.g. in the form of explosions.

The most common drawback of regular grid representations is that the polygonalization is seldom optimal, or even near optimal. Large, flat surfaces may require the same polygon density as small, rough areas do. This is due to the sensitivity to localized, high frequency data within large, uniform resolution areas of lower complexity. (Most level of detail algorithms require that the mesh is subdivided into rectangular blocks of polygons to allow for fast view culling and coarse level of detail selection.) Hence, (iii) is violated as a small bump in the mesh may force higher resolution data than is needed to describe the remaining area of a block. This problem may be alleviated by reducing the overall complexity and applying temporal blending, or morphing, between different levels of detail to avoid "popping" in the mesh [16, 22].

Common to typical TIN and regular grid LOD algorithms is the discreteness of the levels of detail. Often, only a relatively small number of models for a given area are defined, and the difference in the number of polygons in successive levels of detail may be quite large. When switching between two levels of detail, the net change in the number of rendered polygons may amount to a substantial fraction of the given rendering capacity, and may cause rapid fluctuations in the frame rate.

Many LOD algorithms fail to recognize the need for an error bound in the rendered image. While many simplification methods are mathematically viable, the level of detail generation and

selection are often not directly coupled with the screen-space error resulting from the simplification. Rather, these algorithms characterize the data with a small set of parameters that are used in conjunction with viewpoint distance and view angle to select what could be considered "appropriate" levels of detail. Examples of such algorithms include TIN simplification [9], feature (e.g. peaks, ridges, and valleys) identification and preservation [5, 21], and frequency analysis/transforms such as wavelet simplification [6, 12]. These algorithms often do not provide enough information to derive a tight bound on the maximum error in the projected image. If image quality is important and "popping" effects need to be minimized in animations, the level of detail selection should be based on a user-specified error tolerance measured in screen-space, and should preferably be done on a per polygon/vertex basis.

The algorithm presented in this paper satisfies all of the above criteria. Some key features of the algorithm include: flexibility and efficiency afforded by a regular grid representation; localized polygon densities due to variable resolution within each block; screen-space error-driven LOD selection determined by a single threshold; and continuous level of detail, which will be discussed in the following section.

3.1 Continuous Level of Detail

Continuous level of detail has recently been used to describe a variety of properties [8, 18, 22], some of which are discussed below. As mentioned in (iii) and (iv) above, it is important that the complexity of the surface geometry changes smoothly between consecutive frames, and that the simplified geometry doesn't lead to gaps or popping in the mesh. In a more precise description of the term *continuity* in the context of multiresolution height fields, the continuous function, its domain, and its range must be clearly defined. This function may be one of the following:

(i) The elevation function $z(x,y,t)$, where $x,y,t \in \mathbf{R}$. The parameter t may denote time, distance, or some other scalar quantity. This function morphs (blends) the geometries of two discrete levels of detail defined on the same area, resulting in a virtually continuous change in level of detail over time, or over distance from the viewpoint to the mesh.

(ii) The elevation function $z(x,y)$ with domain $\mathbf{R}^2$. The function z is defined piecewise on a per block basis. When discrete levels of detail are used to represent the mesh, two adjacent blocks of different resolution may not align properly, and gaps along the boundaries of the blocks may be seen. The elevation z on these borders will not be continuous unless precautions are taken to ensure that such gaps are smoothed out.

(iii) The polygon distribution function $n(\mathbf{v},A)$. For any given area $A \subseteq \mathbf{R}^2$, the number of polygons used to describe the area is continuous with respect to the viewpoint $\mathbf{v}$.[1] Note that A does not necessarily have to be a connected set. Since the image of n is discrete, we define continuity in terms of the *modulus of continuity* $\omega(\delta,n)$. We say that n is continuous iff $\omega(\delta,n) \rightarrow \varepsilon$, for some $\varepsilon \leq 1$, as $\delta \rightarrow 0$. That is, for sufficiently small changes in the viewpoint, the change in the number of polygons over A is at most one. As a consequence of a continuous polygon distribution, the number of rendered polygons (after clipping), $n(\mathbf{v})$, is continuous with respect to the viewpoint.

Note that a continuous level of detail algorithm may possess one or more of these independent properties (e.g. (i) does not in general imply (iii), and vice versa). Depending on the constraints inherent in the tessellation method, criterion (iii) may or may not be satisfiable, but a small upper bound ε_{max} on ε may exist. Our algorithm, as presented here, primarily addresses definition (iii), but has been designed to be easily extensible to cover the other two definitions (the color plates included in this paper reflect an implementation satisfying (ii)).

[1] This vector may be generalized to describe other view dependent parameters, such as view direction and field of view.

4 SIMPLIFICATION CRITERIA

The surface simplification process presented here is best described as a sequence of two steps: a coarse-grained simplification of the height field mesh geometry that is done to determine which discrete level of detail models are needed, followed by a fine-grained retriangulation of each LOD model in which individual vertices are considered for removal. The algorithm ensures that no errors are introduced in the coarse simplification beyond those that would be introduced if the fine-grained simplification were applied to the entire mesh. Both steps are executed for each rendered frame, and all evaluations involved in the simplification are done dynamically in real-time, based on the location of the viewpoint and the geometry of the height field.

The height field is described by a rectilinear grid of points elevated above the *x-y* plane, with discrete sampling intervals of x_{res} and y_{res}. The surface corresponding to the height field (before simplification) is represented as a symmetric triangle mesh. The smallest mesh representable using this triangulation, the *primitive mesh*, has dimensions 3×3 vertices, and successively larger meshes are formed by grouping smaller meshes in a 2×2 array configuration (see Figure 2). For any level l in this recursive construction of the mesh, the vertex dimensions x_{dim} and y_{dim} are $2^l + 1$. For a certain level n, the resulting mesh is said to form a *block*, or a discrete level of detail model. A set of such blocks of fixed dimensions $2^n + 1$ vertices squared, describes the height field dataset, where the boundary rows and columns between adjacent blocks are shared. While the dimensions of all blocks are fixed, the spatial extent of the blocks may vary by multiples of powers of two of the height field sampling resolution, i.e. the area of a block is $2^{m+n}x_{res} \times 2^{m+n}y_{res}$ where m is some non-negative integer. Thus, lower resolution blocks can be obtained by discarding every other row and column of four higher resolution blocks. We term these decimated vertices the *lowest level vertices* of a block (see Figure 2c). A *quadtree* data structure [17] naturally lends itself to the block partitioning of the height field dataset described above.

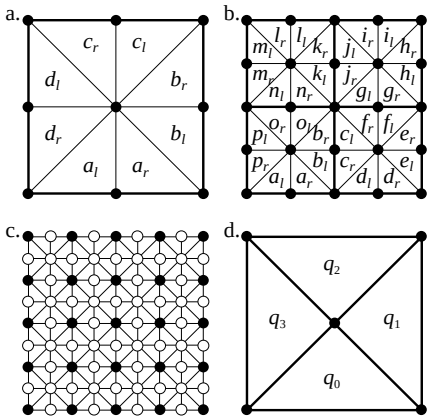


Figure 2: (a, b) Triangulation of uniform height fields of dimensions 3×3 and 5×5 vertices, respectively. (c) Lowest level vertices (unfilled). (d) Block quadrants.

In the following sections, we describe the different simplification steps. We begin by deriving a criterion for the fine-grained (vertex-based) simplification. The coarse-grained (block-based) level of detail selection is then described in terms of the former.

4.1 Vertex-Based Simplification

In the fine-grained simplification step, many smaller triangles are removed and replaced with fewer larger triangles. Conceptually, at the beginning of each rendered frame, the entire height field dataset at its highest resolution is considered. Wherever certain conditions are met, a *triangle/co-triangle pair* $(\triangle_{a_l}, \triangle_{a_r})$ is reduced to one single triangle $\triangle_{a_l} \oplus \triangle_{a_r}$, and the resulting triangle and its co-triangle (if one exists) are considered for further simplification in a recursive manner. In the *x-y* plane with $x_{res} = y_{res}$, a triangle/co-triangle pair is defined by the two congruent right triangles obtained by bisecting a larger isosceles right triangle. Recursive bisection of the resulting two triangles yields lower level triangle/co-triangle pairs. Triangle/co-triangle pairs within a block are descended from the four triangular quadrants of the block, defined by the block boundary and its diagonals (see Figure 2d). For arbitrary height field resolutions, the square mesh is simply stretched in either dimension while retaining the vertex connections. Figure 2a and 2b illustrate the lowest level pairs, where each pair has been assigned a unique letter.

The conditions under which a triangle pair can be combined into a single triangle are primarily described by the amount of change in slope between the two triangles. For triangles $\triangle ABE$ and $\triangle BCE$, with *A*, *B*, and *C* in a plane perpendicular to the *x-y* plane, the slope change is measured by the vertical (*z* axis) distance $\delta_B = |B_z - \frac{A_z+C_z}{2}|$, i.e. the maximum vertical distance between $\triangle ACE = \triangle ABE \oplus \triangle BCE$ and the triangles $\triangle ABE$ and $\triangle BCE$ (see Figure 3). This distance is referred to as vertex *B*'s *delta value*. As the delta value increases, the chance of triangle fusion decreases. By projecting the *delta segment*, defined by *B* and the midpoint of $\overline{AC}$, onto the projection plane, one can determine the maximum perceived geometric (linear) error between the merged triangle and its corresponding sub-triangles. If this error is smaller than a given threshold, τ, the triangles may be fused. If the resulting triangle has a co-triangle with error smaller than the threshold, this pair is considered for further simplification. This process is applied recursively until no further simplification of the mesh can be made. Note that this scheme typically involves a reduction of an already simplified mesh, and the resulting errors (i.e. the projected delta segments) are not defined with respect to the highest resolution mesh, but rather relative to the result of the previous iteration in the simplification process. However, empirical data indicates that the effects of this approximation are negligible (see Section 7).

We now derive a formula for the length of the projected delta segment. Let $\mathbf{v}$ be the midpoint of the delta segment,[2] and define $\mathbf{v}^+ = \mathbf{v} + \begin{bmatrix} 0 & 0 & \frac{\delta}{2} \end{bmatrix}$, $\mathbf{v}^- = \mathbf{v} - \begin{bmatrix} 0 & 0 & \frac{\delta}{2} \end{bmatrix}$. Let $\mathbf{e}$ be the viewpoint and $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$, $\hat{\mathbf{z}}$ be the orthonormal eye coordinate axes expressed in world coordinates. Furthermore, let *d* be the distance from $\mathbf{e}$ to the projection plane, and define λ to be the number of pixels per world coordinate unit in the screen *x-y* coordinate system. (We assume that the pixel aspect ratio is 1:1.) The subscripts *eye* and *screen* are used to denote vectors represented in *eye coordinates* (after the view transformation) and *screen coordinates* (after the perspective projection), respectively. Using these definitions, the following approximations are made:

- When projecting the vectors $\mathbf{v}^+$ and $\mathbf{v}^-$, their midpoint $\mathbf{v}$ is always assumed to be in the center of view, i.e. along $-\hat{\mathbf{z}}$. This approximation is reasonable as long as the field of view is relatively small, and its effect is that the projected delta segments that represent the errors in the triangle simplification become relatively smaller at the periphery of the screen, where less detail is then used—an artifact that is often acceptable as human visual perception degrades toward the periphery.

- We assume $v^+_{eye_z} \simeq v^-_{eye_z} \simeq v_{eye_z}$ in the perspective division $\frac{1}{-v_{eye_z}}$. This is a fair assumption because, in general, $\delta \ll ||\mathbf{e}-\mathbf{v}|| = -v_{eye_z}$.

[2] One may safely substitute the vertex associated with the delta segment for its midpoint.

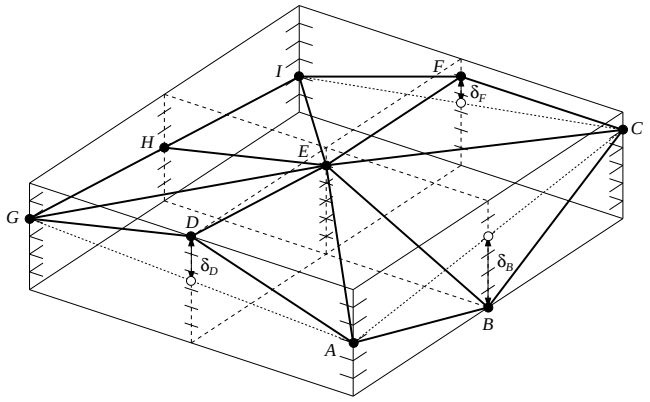


Figure 3: Geometric representation of delta values. $\delta_B = 4$, $\delta_D = 2.5$, $\delta_F = 1.5$, $\delta_H = 0$.

According to the first approximation, the viewing matrix is then:

$$\mathbf{M} = \begin{bmatrix} \hat{x}_x & \hat{y}_x & \frac{e_x - v_x}{||\mathbf{e}-\mathbf{v}||} & 0 \\ \hat{x}_y & \hat{y}_y & \frac{e_y - v_y}{||\mathbf{e}-\mathbf{v}||} & 0 \\ \hat{x}_z & \hat{y}_z & \frac{e_z - v_z}{||\mathbf{e}-\mathbf{v}||} & 0 \\ -\mathbf{e}\cdot\hat{\mathbf{x}} & -\mathbf{e}\cdot\hat{\mathbf{y}} & -\mathbf{e}\cdot\frac{\mathbf{e}-\mathbf{v}}{||\mathbf{e}-\mathbf{v}||} & 1 \end{bmatrix}$$

with $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ perpendicular to $\mathbf{e}-\mathbf{v}$ at all times. This definition of $\mathbf{M}$ leads to the following equalities:

$$\begin{aligned} \mathbf{v}^+_{eye} - \mathbf{v}^-_{eye} &= \mathbf{v}^+\mathbf{M} - \mathbf{v}^-\mathbf{M} \\ &= \delta \begin{bmatrix} \hat{x}_z & \hat{y}_z & \frac{e_z - v_z}{||\mathbf{e}-\mathbf{v}||} & 0 \end{bmatrix} \\ \hat{x}_z^2 + \hat{y}_z^2 &= 1 - \left(\frac{e_z - v_z}{||\mathbf{e}-\mathbf{v}||}\right)^2 \end{aligned}$$

The length of the projected delta segment is then described by the following set of equations:

$$\begin{aligned} \delta_{screen} &= ||\mathbf{v}^+_{screen} - \mathbf{v}^-_{screen}|| \\ &= \frac{d\lambda\sqrt{(v^+_{eye_x} - v^-_{eye_x})^2 + (v^+_{eye_y} - v^-_{eye_y})^2}}{-v_{eye_z}} \\ &= \frac{d\lambda\sqrt{(\delta\hat{x}_z)^2 + (\delta\hat{y}_z)^2}}{||\mathbf{e}-\mathbf{v}||} \\ &= \frac{d\lambda\delta\sqrt{1 - \left(\frac{e_z - v_z}{||\mathbf{e}-\mathbf{v}||}\right)^2}}{||\mathbf{e}-\mathbf{v}||} \\ &= \frac{d\lambda\delta\sqrt{(e_x - v_x)^2 + (e_y - v_y)^2}}{(e_x - v_x)^2 + (e_y - v_y)^2 + (e_z - v_z)^2} \end{aligned} \quad (1)$$

For performance reasons, δ^2_{screen} is compared to τ^2 so that the square root can be avoided:

$$\frac{d^2\lambda^2\delta^2\left((e_x-v_x)^2+(e_y-v_y)^2\right)}{\left((e_x-v_x)^2+(e_y-v_y)^2+(e_z-v_z)^2\right)^2} \leq \tau^2$$

An equivalent inequality that defines the simplification condition reduces to a few additions and multiplications:

$$\delta^2\left((e_x-v_x)^2+(e_y-v_y)^2\right) \leq \kappa^2\left((e_x-v_x)^2+(e_y-v_y)^2+(e_z-v_z)^2\right)^2 \quad (2)$$

where $\kappa = \frac{\tau}{d\lambda}$ is a constant. Whenever $e_x = v_x$ and $e_y = v_y$, i.e. when the viewpoint is directly above or below the delta segment, the projection is zero, and the triangles are coalesced. The probability of satisfying the inequality decreases as e_z approaches v_z, or when the delta segment is viewed from the side. This makes sense, intuitively, as less detail is required for a top-down view of the mesh (assuming a monoscopic view), while more detail is necessary to accurately retain contours and silhouettes in side views. The geometric interpretation of the complement of Equation 2 is a "bialy"—a solid circular torus with no center hole—centered at $\mathbf{v}$, with radius $r = \frac{d\lambda\delta}{2\tau}$ (see Figure 4). The triangles associated with $\mathbf{v}$ can be combined provided that the viewpoint is not contained in the bialy.

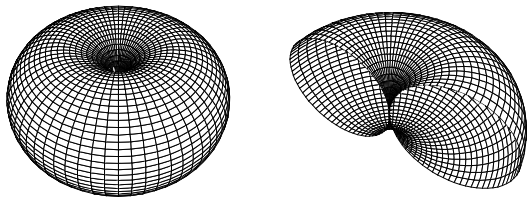


Figure 4: Geometric representation (and its cross-section) of the boundary of Equation 2.

4.2 Block-Based Simplification

Complex datasets may consist of millions of polygons, and it is clearly too computationally expensive to run the simplification process described in the previous section on all polygon vertices for each individual frame. By obtaining a conservative estimate of whether certain groups of vertices can be eliminated in a block, the mesh can often be decimated by several factors with little computational cost. If it is known that the maximum delta projection of all lowest level vertices in a block falls within τ, those vertices can immediately be discarded, and the block can be replaced with a lower resolution block, which in turn is considered for further simplification. Accordingly, a large fraction of the delta projections can be avoided.

The discrete level of detail selection is done by computing the maximum delta value, δ_{max}, of the lowest level vertices for each block. Given the axis-aligned bounding box of a block and δ_{max}, one can determine, for a given viewpoint, whether any of these vertices have delta values large enough to exceed the threshold τ. If none of them do, a lower resolution model may be used. We can expand on this idea to obtain a more efficient simplification algorithm. By using τ, the view parameters, and the constraints provided by the bounding box, one can compute the smallest delta value δ_l that, when projected, can exceed τ, as well as the largest delta value δ_h that may project smaller than τ. Delta values between these extremes fall in an *uncertainty interval*, which we denote by $I_u = (\delta_l, \delta_h]$, for which Equation 2 has to be evaluated. Vertices with delta values less than δ_l can readily be discarded without further evaluation, and conversely, vertices with delta values larger than δ_h cannot be removed. It would obviously be very costly to compute I_u by reversing the projection to get the delta value whose projection equals τ for every single vertex within the block, but one can approximate I_u by assuming that the vertices are dense in the bounding box of the block, and thus obtain a slightly larger superset of I_u. From this point on, we will use I_u to denote this superset.

To find the lower bound δ_l of I_u, the point in the bounding box that maximizes the delta projection must be found. From Equation 1, define $r = \sqrt{(e_x-v_x)^2+(e_y-v_y)^2}$ and $h = e_z - v_z$. We seek to maximize the function $f(r,h) = \frac{r}{r^2+h^2}$ subject to the constraints $r^2+h^2 \geq d^2$ and $\mathbf{v} \in B$, where d is the distance from the viewpoint to the projection plane and B is the set of points contained in the bounding box, described by the two vectors

$$\mathbf{b}_{min} = \begin{bmatrix} b_{min_x} & b_{min_y} & b_{min_z} \end{bmatrix}$$
$$\mathbf{b}_{max} = \begin{bmatrix} b_{max_x} & b_{max_y} & b_{max_z} \end{bmatrix}$$

We solve this optimization problem by constraining r, such that $d^2 - h^2 \leq r^2_{min} \leq r^2 \leq r^2_{max}$ (r and h are otherwise independent). Clearly, then, h^2 has to be minimized which is accomplished by setting $h = h_{min} = |e_z - \text{clamp}(b_{min_z}, e_z, b_{max_z})|$, where

$$\text{clamp}(x_{min}, x, x_{max}) = \begin{cases} x_{min} & \text{if } x < x_{min} \\ x_{max} & \text{if } x > x_{max} \\ x & \text{otherwise} \end{cases}$$

In the x-y plane, define r_{min} to be the smallest distance from $\begin{bmatrix} e_x & e_y \end{bmatrix}$ to the rectangular slice (including the interior) of the bounding box defined by $\begin{bmatrix} b_{min_x} & b_{min_y} \end{bmatrix}$ and $\begin{bmatrix} b_{max_x} & b_{max_y} \end{bmatrix}$, and define r_{max} to be the largest such distance. Via partial differentiation with respect to r, the maximum f_{max} of $f(r,h)$ is found at $r = h$. If no $\mathbf{v}$ exists under the given constraints that satisfies $r = h$, r is increased/decreased until $\mathbf{v} \in B$, i.e. $r = \text{clamp}(r_{min}, h, r_{max})$.

The upper bound, δ_h, is similarly found by minimizing $f(r,h)$. This is done by setting $h = h_{max} = \max\{|e_z - b_{min_z}|, |e_z - b_{max_z}|\}$. f_{min} is then found when either $r = r_{min}$ or $r = r_{max}$, whichever yields a smaller $f(r,h)$.

The bounds on I_u can now be found using the following equations:

$$\delta_l = \frac{\tau}{d\lambda f_{max}} \quad (3)$$

$$\delta_h = \begin{cases} 0 & \text{if } \tau = 0 \\ \frac{\tau}{d\lambda f_{min}} & \text{if } \tau > 0 \text{ and } f_{min} > 0 \\ \infty & \text{otherwise} \end{cases} \quad (4)$$

After computation of I_u, δ_{max} is compared to δ_l, and if smaller, a lower resolution level of detail block is substituted, and the process is repeated for this block. If $\delta_{max} > \delta_l$, it may be that a higher resolution block is needed. By maintaining $\delta^*_{max} = \max_i\{\delta_{max_i}\}$, the largest δ_{max} of all higher resolution blocks (or *block descendants*) for the given area, δ^*_{max} is compared to δ_l for the current block, and if greater, four higher resolution blocks replace the current block. As mentioned earlier, this implicit hierarchical organization of blocks is best represented by a quadtree, where each block corresponds to a quadnode.

4.3 Vertex Dependencies

As pointed out in Section 4.1, triangle fusion can occur only when the triangles in the triangle pair appear on the same level in the triangle subdivision. For example, in Figure 2b, $\triangle_{e_l} \oplus \triangle_{e_r}$ and

$\triangle_{f_l} \oplus \triangle_{f_r}$ cannot be coalesced unless the triangles in both pairs $(\triangle_{e_l}, \triangle_{e_r})$ and $(\triangle_{f_l}, \triangle_{f_r})$ have been fused. The triangles can be represented by nodes in a binary expression tree, where the smallest triangles correspond to terminal nodes, and coalesced triangles correspond to higher level, nonterminal nodes formed by recursive application of the $\oplus$ operator (hence the subscripts *l* and *r* for "left" and "right"). Conceptually, this tree spans the entire height field dataset, but can be limited to each block.

Another way of looking at triangle fusion is as vertex removal, i.e. when two triangles are fused, one vertex is removed. We call this vertex the *base vertex* of the triangle pair. Each triangle pair has a *co-pair* associated with it,[3] and the pair/co-pair share the same base vertex. The mapping of vertices to triangle pairs, or the nodes associated with the operators that act on the triangle pairs, results in a *vertex tree*, wherein each vertex occurs exactly twice; once for each triangle pair (Figures 5g and 5h). Hence, each vertex has two distinct parents (or dependents)—one in each of two binary subtrees T_0 and T_1—as well as four distinct children. If any of the descendants of a vertex *v* are included in the rendered mesh, so is *v*, and we say that *v* is *enabled*. If the projected delta segment associated with *v* exceeds the threshold τ, *v* is said to be *activated*, which also implies that *v* is *enabled*. Thus, the *enabled* attribute of *v* is determined by

$$\begin{aligned} &activated(v) \vee \\ &\quad enabled(left_{T_0}(v)) \vee \\ &\quad enabled(right_{T_0}(v)) \vee \\ &\quad enabled(left_{T_1}(v)) \vee \\ &\quad enabled(right_{T_1}(v)) \Rightarrow enabled(v) \end{aligned}$$

An additional vertex attribute, *locked*, allows the *enabled* flag to be hardwired to either **true** or **false**, overriding the relationship above. This may be necessary, for example, when eliminating gaps between adjacent blocks if compatible levels of detail do not exist, i.e. some vertices on the boundaries of the higher resolution block may have to be permanently disabled. Figures 5a–e show the dependency relations between vertices level by level. Figure 5f shows the influence of an enabled vertex over other vertices that directly or indirectly depend on it. Figures 5g and 5h depict the two possible vertex tree structures within a block, where intersections have been separated for clarity.

To satisfy continuity condition (ii) (see Section 3.1), the algorithm must consider dependencies that cross block boundaries. Since the vertices on block boundaries are shared between adjacent blocks, these vertices must be referenced uniquely, so that the dependencies may propagate across the boundaries. In most implementations, such shared vertices are simply duplicated, and these redundancies must be resolved before or during the simplification stage. One way of approaching this is to access each vertex via a pointer, and discard the redundant copies of the vertex before the block is first accessed. Another approach is to ensure that the attributes of all copies of a vertex are kept consistent when updates (e.g. *enabled* and *activated* transitions) occur. This can be achieved by maintaining a circular linked list of copies for each vertex.

5 ALGORITHM OUTLINE

The algorithm presented here describes the steps necessary to select which vertices should be included for rendering of the mesh. In Section 5.1, we describe how the mesh is rendered once the vertex selection is done. A discussion of appropriate data structures is presented in Section 6. Using the equations presented in previous sections, the algorithm is summarized by the pseudocode below. Unless qualified with superscripts, all variables are assumed to belong to the current frame and block.

[3] Triangle pairs with base vertices on the edges of the finite dataset are an exception.

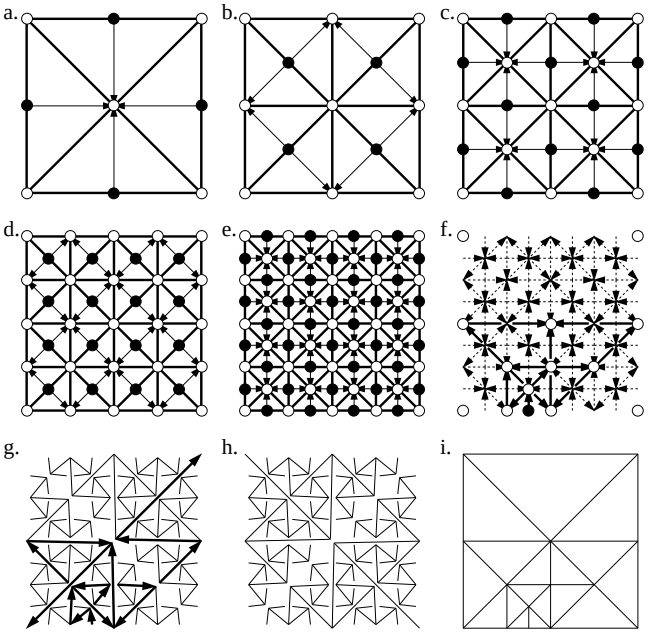


Figure 5: (a–e) Vertex dependencies by descending levels (left to right, top to bottom). An arc from *A* to *B* indicates that *B* depends on *A*. (f) Chain of dependencies originating from the solid vertex. (g, h) Symmetric binary vertex trees (the arcs in (g) correspond to (f)). (i) Triangulation corresponding to (f).

```
MAIN()
 1  for each frame n
 2      for each active block b
 3          compute I_u (Equations 3 and 4)
 4          if δ_max ≤ δ_l
 5              replace b with lower resolution block
 6          else if δ*_max > δ_l
 7              replace b with higher resolution blocks
 8      for each active block b
 9          determine if b intersects the view frustum
10      for each visible block b
11          I_0 ← (δ_l^{n-1}, δ_l^n]
12          I_1 ← (δ_h^n, δ_h^{n-1}]
13          for each vertex v with δ(v) ∈ I_0
14              activated(v) ← false
15              UPDATE-VERTEX(v)
16          for each vertex v with δ(v) ∈ I_1
17              activated(v) ← true
18              UPDATE-VERTEX(v)
19          for each vertex v with δ(v) ∈ I_u
20              EVALUATE-VERTEX(v)
21      for each visible block b
22          RENDER-BLOCK(b)

UPDATE-VERTEX(v)
 1  if ¬locked(v)
 2      if ¬dependency_i(v) ∀i
 3          if enabled(v) ≠ activated(v)
 4              enabled(v) ← ¬enabled(v)
 5              NOTIFY(parent_{T_0}(v), branch_{T_0}(v), enabled(v))
 6              NOTIFY(parent_{T_1}(v), branch_{T_1}(v), enabled(v))
```

```
EVALUATE-VERTEX(v)
 1  if ¬locked(v)
 2      if ¬dependency_i(v) ∀i
 3          activated(v) ← ¬Equation 2
 4          if enabled(v) ≠ activated(v)
 5              enabled(v) ← ¬enabled(v)
 6              NOTIFY(parent_T0(v), branch_T0(v), enabled(v))
 7              NOTIFY(parent_T1(v), branch_T1(v), enabled(v))

NOTIFY(v, child, state)
 1  if v is a valid vertex
 2      dependency_child(v) ← state
 3      if ¬locked(v)
 4          if ¬dependency_i(v) ∀i
 5              if ¬activated(v)
 6                  enabled(v) ← false
 7                  NOTIFY(parent_T0(v), branch_T0(v), false)
 8                  NOTIFY(parent_T1(v), branch_T1(v), false)
 9          else
10              if ¬enabled(v)
11                  enabled(v) ← true
12                  NOTIFY(parent_T0(v), branch_T0(v), true)
13                  NOTIFY(parent_T1(v), branch_T1(v), true)
```

The term *active block* refers to whether the block is currently the chosen level of detail for the area it covers. All blocks initially have I_u set to $[0,\infty)$, and so do blocks that previously were inactive. When deactivating vertices with delta values smaller than δ_l, the interval $I_0 \subseteq [0,\delta_l]$ is traversed. By inductive reasoning, vertices with deltas smaller than the lower bound of I_0 must have been deactivated in previous frames. Similarly, I_1 is used for vertex activation. In quadtree implementations, the condition on line 4 in MAIN may have to be supplemented; the condition $\delta_{max} \leq \delta_l$ should also hold for the three neighboring siblings of b before b can be replaced.

If a vertex's *enabled* attribute changes, all dependent vertices must be notified of this change so that their corresponding *dependency* flags are kept consistent with this change. The procedure UPDATE-VERTEX checks if $enabled(v)$ has changed, and if so, notifies v's dependents by calling NOTIFY. If the *enabled* flag of a dependent in turn is modified, NOTIFY is called recursively. Since line 2 in NOTIFY necessarily involves a change of a *dependency* bit, there may be a transition in $enabled(v)$ from **true** to **false** on line 6 provided $activated(v)$ is **false** as the vertex is no longer dependent. The evaluation of Equation 2 on line 3 in EVALUATE-VERTEX can be deferred if any of the vertex's *dependency* flags are set, which is of significant importance as this evaluation is one of the most computationally expensive parts of the algorithm. Note that there may be a one-frame delay before the *activated* attribute is corrected due to this deferral if the child vertices are evaluated after the dependent vertex (line 2 of EVALUATE-VERTEX and lines 4–5 of NOTIFY). The function $branch_T(v)$ refers to the field of the parent in tree T that reflects the *enabled* field of vertex v. Note that a check has to be made (line 1 in NOTIFY) whether a vertex is "valid" as some vertices have fewer than two dependents (e.g. boundary vertices).

5.1 Mesh Rendering

Once the vertex selection is made, a triangle mesh must be formed that connects the selected vertices. This mesh is defined by specifying the vertices encountered in a pre-order descent of the binary vertex trees. The recursive stopping condition is a **false** *enabled* attribute. To efficiently render the mesh, a triangle mesh graphics primitive, such as the one supported by IRIS GL and OpenGL [11, 15], may be used. For each specified vertex v, the previous two vertices and v form the next triangle in the mesh. At certain points, the previous two vertices must be swapped via a `swaptmesh()` call (IRIS GL), or a `glVertex()` call (OpenGL). A copy of the two-entry graphics vertex buffer, *my-buffer*, is maintained explicitly to allow the decision as to when to swap the entries to be made. The most recent vertex in this buffer is indexed by *ptr*.

The following pseudocode describes the mesh rendering algorithm. Each of the four triangular quadrants q_i are rendered in counterclockwise order, with the first vertex in each quadrant coincident with the last vertex in the previous quadrant (see Figure 2d). Hence, a single triangle mesh can be used to render the entire block. The indices q_{il}, q_{it}, and q_{ir} correspond to the left, top, and right vertex indices of quadrant q_i, respectively, with the "top" index being the center of the block. The block dimensions are 2^n+1 squared.

```
RENDER-BLOCK(b)
 1  enter triangle mesh mode
 2  render vertex v_q0l
 3  my-buffer_ptr ← q_0l
 4  previous-level ← 0
 5  for each quadrant q_i in block b
 6      if previous-level is even
 7          toggle ptr
 8      else
 9          swap vertices in graphics buffer
10      render vertex v_qil
11      my-buffer_ptr ← q_il
12      previous-level ← 2n+1
13      RENDER-QUADRANT(q_il, q_it, q_ir, 2n)
14  render vertex v_q0l
15  exit triangle mesh mode

RENDER-QUADRANT(i_l, i_t, i_r, level)
 1  if level > 0
 2      if enabled(v_it)
 3          RENDER-QUADRANT(i_l, (i_l+i_r)/2, i_t, level−1)
 4          if i_t ∉ my-buffer
 5              if level + previous-level is odd
 6                  toggle ptr
 7              else
 8                  swap vertices in graphics buffer
 9              render vertex v_it
10              my-buffer_ptr ← i_t
11              previous-level ← level
12          RENDER-QUADRANT(i_t, (i_l+i_r)/2, i_r, level−1)
```

The index $\frac{i_l+i_r}{2}$ corresponds to the (base) vertex that in the x-y plane is the midpoint of the edge $\overline{v_{i_l} v_{i_r}}$. Since *my-buffer* reflects what vertices are currently in the graphics buffer, line 9 in RENDER-BLOCK and line 8 in RENDER-QUADRANT could be implemented with a `glVertex()` call, passing the second most recent vertex in *my-buffer*.

6 DATA STRUCTURES

Many of the issues related to the data structures used with this algorithm have purposely been left open, as different needs may demand totally different approaches to their representations. In one implementation—the one presented here—as few as six bytes per vertex were used, and as many as 28 bytes per vertex were needed in another. In this section, we describe data structures that will be useful in many implementations.

For a compact representation, the vertex elevation is discretized and stored as a 16-bit integer. A minimum of six additional bits per vertex are required for the various flags, including the *enabled*, *activated*, and four *dependency* attributes. Optionally, the *locked* attribute can be added to these flags. The theoretical range of delta values becomes $[0, 2^{16}-1]$ in steps of $\frac{1}{2}$. We

elect to store each δ in "compressed" form as an 8-bit integer $\hat{\delta}$ in order to conserve space by encapsulating the vertex structure in a 32-bit aligned word. We define the decompression function as $\delta = \frac{1}{2}\lfloor(1+\hat{\delta})^{1+\hat{\delta}^2/(2^8-1)^2} - 1\rfloor$.[4] This exponential mapping preserves the accuracy needed for the more frequent small deltas, while allowing large delta values to be represented, albeit with less accuracy. The compression function is defined as the inverse of the decompression function. Both functions are implemented as lookup tables.

To accommodate tasks such as rendering and surface following, the vertices must be organized spatially for fast indexing. In Section 4.2, however, we implied that vertices within ranges of delta values could be immediately accessed. This is accomplished by creating an auxiliary array of indices, in which the entries are sorted on the corresponding vertices' delta values. Each entry uniquely references the corresponding vertex (i, j) via an index into the array of vertex structures. For each possible compressed delta value within a block, there is a pointer (index) $p_{\hat{\delta}}$ to a bin that contains the vertex indices corresponding to that delta value. The 2^8 bins are stored in ascending order in a contiguous, one-dimensional array. The entries in bin i are then indexed by $p_i, p_i+1, \ldots, p_{i+1}-1$ ($p_i = p_{i+1}$ implies that bin i is empty). For block dimensions up to 2^7+1, the indices can be represented with 16 bits to save space, which in addition to the 32-bit structure described above, results in a total of six bytes storage per vertex.

7 RESULTS

To show the effectiveness of the polygon reduction and display algorithm, we here present the results of a quantitative analysis of the number of polygons and delta projections, frame rates, computation and rendering time, and errors in the approximated geometry. A set of color plates show the resulting wireframe triangulations and textured terrain surfaces at different stages of the simplification and for different choices of τ. Two height field datasets were used in generating images and collecting data: a 64 km^2 area digital elevation model of the Hunter-Liggett military base in California, sampled at 2×2 meter resolution, and 1 meter height (z) resolution (Color Plates 1a–c and 2a–c); and a 1×1 meter resolution, 14 km^2 area of 29 Palms, California, with a z resolution of one tenth of a meter (Color Plates 3a–d). The vertical field of view is 60° in all images, which were generated on a two-processor, 150 MHz SGI Onyx RealityEngine^2 [1], and have dimensions 1024×768 pixels unless otherwise specified.

We first examine the amount of polygon reduction as a function of the threshold τ. A typical view of the Hunter-Liggett terrain was chosen for this purpose, which includes a variety of features such as ridges, valleys, bumps, and relatively flat areas. Figure 6 shows four curves drawn on a logarithmic scale (vertical axis). The top horizontal line, $n_0(\tau) = 13 \cdot 10^6$, shows the total number of polygons in the view frustum before any reduction method is applied. The curve second from the top, $n_1(\tau)$, represents the number of polygons remaining after the block-based level of detail selection is done. The number of polygons rendered, $n_2(\tau)$, i.e. the remaining polygons after the vertex-based simplification, is shown by the lowest solid curve. As expected, these two curves flatten out as τ is increased. The ratio $n_0(\tau)/n_2(\tau)$ ranges from about 2 ($\tau = 0$) to over 6,000 ($\tau = 8$). Of course, at $\tau = 0$, only coplanar triangles are fused. The ratio $n_1(\tau)/n_2(\tau)$ varies between 1.85 and 160 over the same interval, which clearly demonstrates the advantage of refining each uniform level of detail block.

We pay special attention to the data obtained at $\tau = 1$, as this threshold is small enough that virtually no popping can be seen in animated sequences, and the resulting surfaces, when textured, are seemingly identical to the ones obtained with no mesh simplification. Color Plates 1a–c illustrate the three stages of simplification at $\tau = 1$. In Color Plate 1c, note how many polygons are required for the high frequency data, while only a few, large polygons are used for the flatter areas. For this particular threshold, $n_0(1)/n_2(1)$ is slightly above 200, while $n_1(1)/n_2(1)$ is 18. The bottommost, dashed curve in Figure 6 represents the total number of delta values that fall in the uncertainty interval per frame (Section 4.2). Note that this quantity is generally an order of magnitude smaller than the number of rendered polygons. This is significant as the evaluations associated with these delta values constitute the bulk of the computation in terms of CPU time. This also shows the advantage of computing the uncertainty interval, as out of the eight million vertices contained in the view frustum, only 14,000 evaluations of Equation 2 need to be made when $\tau = 1$.

τ	displacement				
	mean	median	max	std. dev.	$> \tau$ (%)
0.000	0.00	0.00	0.00	0.00	0.00
0.125	0.03	0.00	0.52	0.05	6.41
0.250	0.06	0.00	0.85	0.09	4.52
0.500	0.11	0.04	1.56	0.15	3.14
1.000	0.21	0.07	2.88	0.29	2.61
2.000	0.42	0.13	5.37	0.59	2.84
4.000	0.88	0.23	10.41	1.24	3.27
8.000	1.38	0.19	16.69	2.08	1.38

Table 1: Screen-space error in simplified geometry.

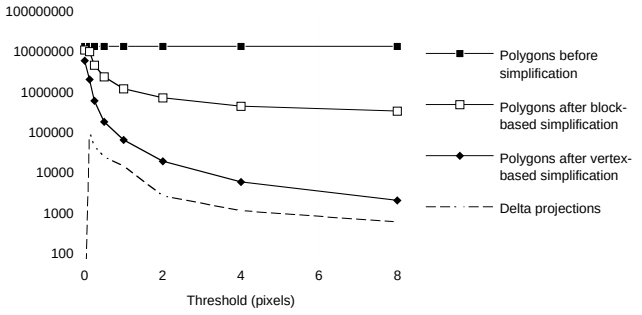


Figure 6: The number of polygons (n_0, n_1, n_2, from top to bottom) as a function of τ. The bottom curve shows the number of times Equation 2 was evaluated per frame.

In order to evaluate the errors due to the simplification, the points on the polygonal surface of the simplified mesh that have been displaced vertically, as well as the remaining triangle vertices, are perspective projected to screen-space and compared to the projections of the original set of vertices. Optimally, each such screen coordinate displacement should fall within the threshold distance τ. However, this constraint may in certain cases be violated due to the approximations discussed in Section 4.1. Table 1 was compiled for each mesh after vertex-based simplification was applied, and the surface points were correlated with the original eight million vertices shown in Color Plate 1a. The table summarizes the mean, median, maximum, and standard deviation of the displacements in number of pixels, as well as the fraction of displacements that exceed τ. In all cases, the average pixel error is well below τ. It can be seen that the approximations presented in Section 4.1 do not significantly impact the accuracy, as the fraction of displacements that exceed τ is typically less than five percent.

Color Plates 2a–c illustrate a checkerboard pattern draped over the polygonal meshes from Color Plates 1a–c. Qualitatively, these images suggest little or no perceivable loss in image quality for a

[4] This results in an upper bound $\frac{2^{16}-1}{2}$ for the delta values.

threshold of one pixel, even when the surface complexity is reduced by a factor of 200.

Figure 7 demonstrates the efficiency of the algorithm. The computation time associated with the delta projections (lines 10–20 in MAIN, Section 5) is typically only a small fraction of the rendering time. This data was gathered for the views shown in Color Plates 3a–d.

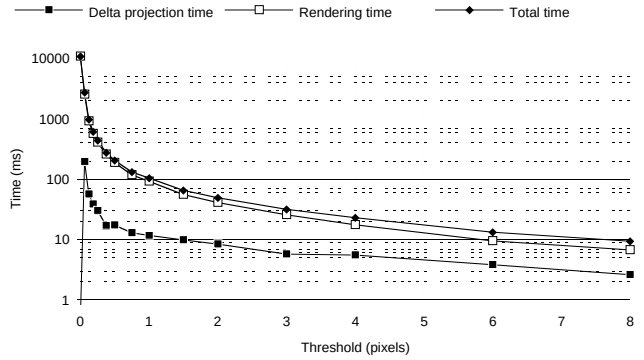


Figure 7: Rendering and evaluation times and their sum as functions of τ.

Figure 8 shows how the quantities in Figure 6, as well as the frame rate vary with time. The data collection for 3,230 frames was done over a time period of 120 seconds, with the viewpoint following a circular path of radius 1 km over the Hunter-Liggett dataset. The terrain was rendered as a wireframe mesh in a 640×480 window, with $\tau = 2$ pixels. It can be seen that the number of rendered polygons does not depend on the total number of polygons in the view frustum, but rather on the complexity of the terrain intersected by the view frustum. As evidenced by the graph, a frame rate of at least 20 frames per second was sustained throughout the two minutes of fly-through.

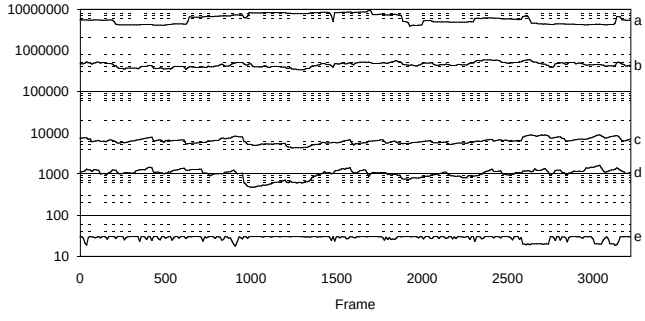


Figure 8: Time graph of (a) total number of polygons in view frustum, (b) number of polygons after block-based simplification, (c) number of polygons after vertex-based simplification, (d) number of delta projections, and (e) frames per second.

8 CONCLUSION

We have presented a height-field display algorithm based on real-time, per vertex level of detail evaluation, that achieves interactive and consistent frame rates exceeding twenty frames per second, with only a minor loss in image quality. Attractive features attributed to regular grid surface representations, such as fast geometric queries, compact representation, and fast mesh rendering are retained. The concept of continuous level of detail allows a polygon distribution that is near optimal for any given viewpoint and frame, and also yields smooth changes in the number of rendered polygons. A single parameter that can easily be changed interactively, with no incurred cost, determines the resulting image quality, and a direct relationship between this parameter and the number of rendered polygons exists, providing capabilities for maintaining consistent frame rates. The algorithm can easily be extended to handle the problem of gaps between blocks of different levels of detail, as well as temporal geometry morphing to further minimize popping effects.

Acknowledgement

This work was performed in part under contract DAKF11–91–D–0004–0034 from the U.S. Army Research Laboratory.

References

[1] AKELEY, K. RealityEngine Graphics. Proceedings of SIGGRAPH 93. In *Computer Graphics* Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH, pp. 109–116.

[2] COSMAN, M. A., MATHISEN, A. E., and ROBINSON, J. A. A New Visual System to Support Advanced Requirements. In *Proceedings, IMAGE V Conference*, June 1990, pp. 370–380.

[3] DE BERG, M. and DOBRINDT, K. T. G. On Levels of Detail in Terrains. In *11th ACM Symposium on Computational Geometry*, June 1995.

[4] DE FLORIANI, L. and PUPPO, E. Hierarchical Triangulation for Multiresolution Surface Description. *ACM Transactions on Graphics* 14(4), October 1995, pp. 363–411.

[5] DOUGLAS, D. H. Experiments to Locate Ridges and Channels to Create a New Type of Digital Elevation Model. *Cartographica* 23(4), 1986, pp. 29–61.

[6] ECK, M., DEROSE, T., DUCHAMP, T., HOPPE, H., LOUNSBERY, M., and STUETZLE, W. Multiresolution Analysis of Arbitrary Meshes. Proceedings of SIGGRAPH 95. In *Computer Graphics* Proceedings, Annual Conference Series, 1995, ACM SIGGRAPH, pp. 173–182.

[7] FALBY, J. S., ZYDA, M. J., PRATT, D. R., and MACKEY, R. L. NPSNET: Hierarchical Data Structures for Real-Time Three-Dimensional Visual Simulation. *Computers & Graphics* 17(1), 1993, pp. 65–69.

[8] FERGUSON, R. L., ECONOMY, R., KELLY, W. A., and RAMOS, P. P. Continuous Terrain Level of Detail for Visual Simulation. In *Proceedings, IMAGE V Conference*, June 1990, pp. 144–151.

[9] FOWLER, R. J. and LITTLE, J. J. Automatic Extraction of Irregular Network Digital Terrain Models. Proceedings of SIGGRAPH 79. In *Computer Graphics* 13(2) (August 1979), pp. 199–207.

[10] GARLAND, M. and HECKBERT, P. S. Fast Polygonal Approximation of Terrains and Height Fields. Technical Report CMU-CS-95-181, CS Dept., Carnegie Mellon U., 1995.

[11] *Graphics Library Programming Guide*. Silicon Graphics Computer Systems, 1991.

[12] GROSS, M. H., GATTI, R., and STAADT, O. Fast Multiresolution Surface Meshing. In *Proceedings of Visualization '95*, October 1995, pp. 135–142.

[13] HECKBERT, P. S. and GARLAND, M. Multiresolution Modeling for Fast Rendering. In *Proceedings of Graphics Interface '94*, 1994, pp. 1–8.

[14] HOPPE, H., DEROSE, T., DUCHAMP, T., MCDONALD, J., and STUETZLE, W. Mesh Optimization. Proceedings of SIGGRAPH 93. In *Computer Graphics* Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH, pp. 19–26.

[15] NEIDER, J., DAVIS, T., and WOO, M. *OpenGL Programming Guide*. Addison-Wesley, 1993.

[16] ROHLF, J. and HELMAN, J. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. Proceedings of SIGGRAPH 94. In *Computer Graphics* Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH, pp. 381–394.

[17] SAMET, H. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys* 16(2), June 1984, pp. 187–260.

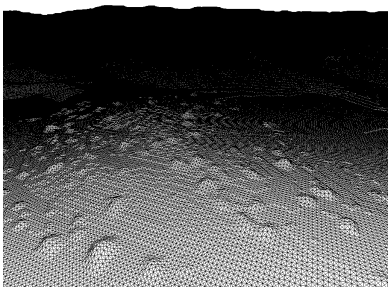
[18] SCARLATOS, L. L. A Refined Triangulation Hierarchy for Multiple Levels of Terrain Detail. In *Proceedings, IMAGE V Conference*, June 1990, pp. 114–122.

[19] SCHRODER, F. and ROSSBACH, P. Managing the Complexity of Digital Terrain Models. *Computers & Graphics* 18(6), 1994, pp. 775–783.

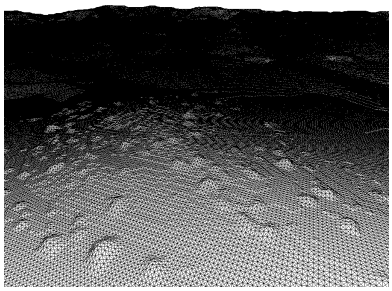
[20] SCHROEDER, W. J., ZARGE, J. A., and LORENSON, W. E. Decimation of Triangle Meshes. Proceedings of SIGGRAPH 92. In *Computer Graphics* 26(2) (July 1992), pp. 65–70.

[21] SOUTHARD, D. A. Piecewise Planar Surface Models from Sampled Data. *Scientific Visualization of Physical Phenomena*, June 1991, pp. 667–680.

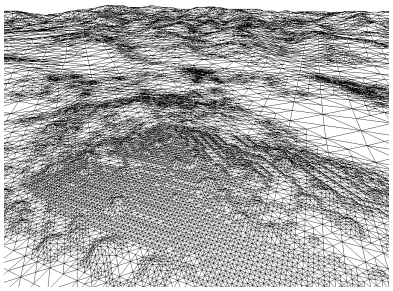
[22] TAYLOR, D. C. and BARRET, W. A. An Algorithm for Continuous Resolution Polygonalizations of a Discrete Surface. In *Proceedings of Graphics Interface '94*, 1994, pp. 33–42.



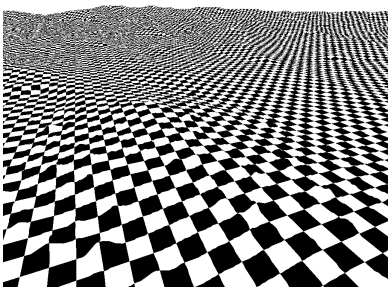
1a.



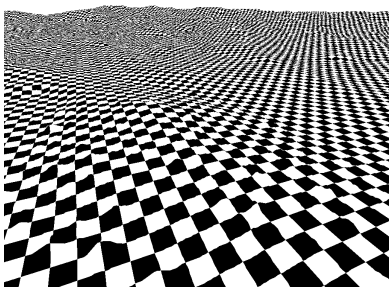
1b.



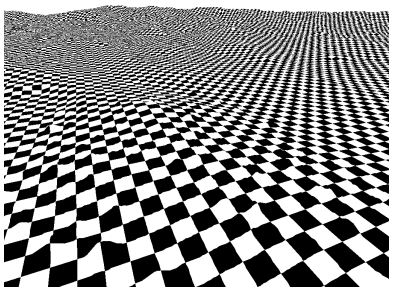
1c.



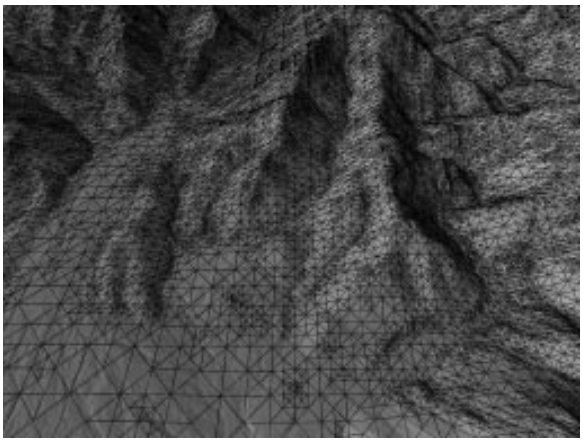
2a. Before simplification
13,304,214 polygons



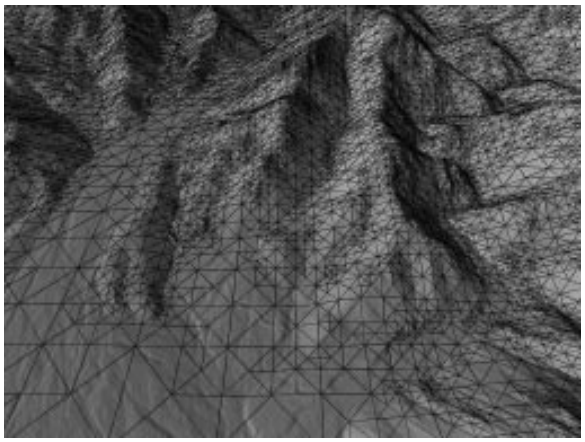
2b. After block−based LOD
1,179,690 polygons



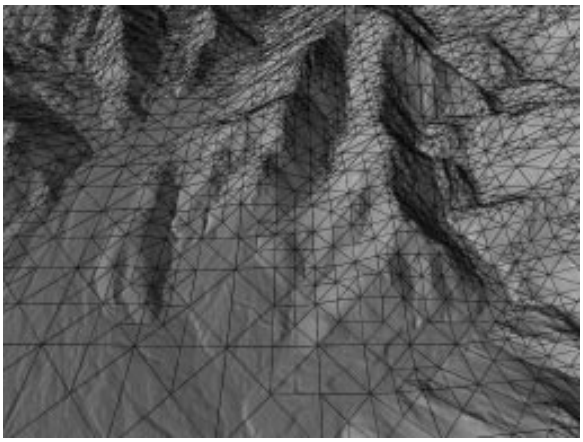
2c. After vertex−based LOD
64,065 polygons



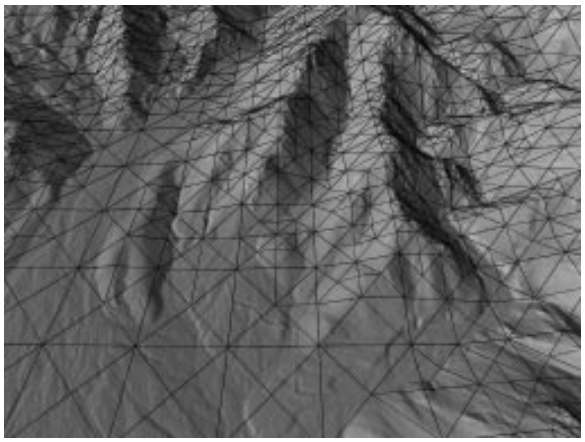
3a. $\tau = 0.5$, 62,497 polygons



3b. $\tau = 1.0$, 23,287 polygons



3c. $\tau = 2.0$, 8,612 polygons



3d. $\tau = 4.0$, 3,385 polygons

Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments

Thomas A. Funkhouser and Carlo H. Séquin
University of California at Berkeley[‡]

Abstract

We describe an adaptive display algorithm for interactive frame rates during visualization of very complex virtual environments. The algorithm relies upon a hierarchical model representation in which objects are described at multiple levels of detail and can be drawn with various rendering algorithms. The idea behind the algorithm is to adjust image quality adaptively to maintain a uniform, user-specified target frame rate. We perform a constrained optimization to choose a level of detail and rendering algorithm for each potentially visible object in order to generate the "best" image possible within the target frame time. Tests show that the algorithm generates more uniform frame rates than other previously described detail elision algorithms with little noticeable difference in image quality during visualization of complex models.

CR Categories and Subject Descriptors:
[Computer Graphics]: I.3.3 Picture/Image Generation – *viewing algorithms*; I.3.5 Computational Geometry and Object Modeling – *geometric algorithms, object hierarchies* ; I.3.7 Three-Dimensional Graphics and Realism – *virtual reality*.

1 Introduction

Interactive computer graphics systems for visualization of realistic-looking, three-dimensional models are useful for evaluation, design and training in virtual environments, such as those found in architectural and mechanical CAD, flight simulation, and virtual reality. These visualization systems display images of a three-dimensional model on the screen of a computer workstation as seen from a simulated observer's viewpoint under interactive control by a user. If images are rendered smoothly and quickly enough, an illusion of real-time exploration of a virtual environment can be achieved as the simulated observer moves through the model.

[‡]Computer Science Division, Berkeley, CA 94720

It is important for a visualization system to maintain an interactive frame rate (e.g., a constant ten frames per second). If frame rates are too slow, or too jerky, the interactive feel of the system is greatly diminished [3]. However, realistic-looking models may contain millions of polygons – far more than currently available workstations can render at interactive frame rates. Furthermore, the complexity of the portion of the model visible to the observer can be highly variable. Tens of thousands of polygons might be simultaneously visible from some observer viewpoints, whereas just a few can be seen from others. Programs that simply render all potentially visible polygons with some predetermined quality may generate frames at highly variable rates, with no guaranteed upper bound on any single frame time.

Using the UC Berkeley Building Walkthrough System [5] and a model of Soda Hall, the future Computer Science Building at UC Berkeley, as a test case, we have developed an adaptive algorithm for interactive visualization that guarantees a user-specified target frame rate. The idea behind the algorithm is to trade image quality for interactivity in situations where the environment is too complex to be rendered in full detail at the target frame rate. We perform a constrained optimization that selects a level of detail and a rendering algorithm with which to render each potentially visible object to produce the "best" image possible within a user-specified target frame time. In contrast to previous culling techniques, this algorithm guarantees a uniform, bounded frame rate, even during visualization of very large, complex models.

2 Previous Work

2.1 Visibility Determination

In previous work, visibility algorithms have been described that compute the portion of a model potentially visible from a given observer viewpoint [1, 11]. These algorithms cull away large portions of a model that are occluded from the observer's viewpoint, and thereby improve frame rates significantly. However, in very detailed models, often more polygons are visible from certain observer viewpoints than can be rendered in an interactive frame time. Certainly, there is no upper bound on the complexity of the scene visible from an observer's viewpoint. For instance, consider walking through a very detailed model of a fully stocked department store, or viewing an assembly of a complete airplane engine. In our model of Soda Hall, there are some viewpoints from which an observer can see more than eighty thousand polygons. Clearly, visibility processing alone is not sufficient to guarantee an interactive frame rate.

2.2 Detail Elision

To reduce the number of polygons rendered in each frame, an interactive visualization system can use *detail elision*. If a model can be described by a hierarchical structure of *objects*, each of which is represented at multiple *levels of detail* (LODs), as shown in Figure 1, simpler representations of an object can be used to improve frame rates and memory utilization during interactive visualization. This technique was first described by Clark [4], and has been used by numerous commercial visualization systems [9]. If different representations for the same object have similar appearances and are blended smoothly, using transparency blending or three-dimensional interpolation, transitions between levels of detail are barely noticeable during visualization.

241 Polygons 44 Polygons

Figure 1: Two levels of detail for a chair.

Previously described techniques for choosing a level of detail at which to render each visible object use static heuristics, most often based on a threshold regarding the size or distance of an object to the observer [2, 8, 9, 13], or the number of pixels covered by an average polygon [5]. These simple heuristics can be very effective at improving frame rates in cases where most visible objects are far away from the observer and map to very few pixels on the workstation screen. In these cases, simpler representations of some objects can be displayed, reducing the number of polygons rendered without noticeably reducing image quality.

Although static heuristics for visibility determination and LOD selection improve frame rates in many cases, they do not generally produce a *uniform* frame rate. Since LODs are computed independently for each object, the number of polygons rendered during each frame time depends on the size and complexity of the objects visible to the observer. The frame rate may vary dramatically from frame to frame as many complex objects become visible or invisible, and larger or smaller.

Furthermore, static heuristics for visibility determination and LOD selection do not even guarantee a *bounded* frame rate. The frame rate can become arbitrarily slow, as the scene visible to the observer can be arbitrarily complex. In many cases, the frame rate may become so slow that the system is no longer interactive. Instead, a LOD selection algorithm should adapt to overall scene complexity in order to produce uniform, bounded frame rates.

2.3 Adaptive Detail Elision

In an effort to maintain a specified *target frame rate*, some commercial flight simulators use an adaptive algorithm that adjusts the size threshold for LOD selection based on feedback regarding the time required to render previous frames [9]. If the previous frame took longer than the target frame time, the size threshold for LOD selection is increased so that future frames can be rendered more quickly.

This adaptive technique works reasonably well for flight simulators, in which there is a large amount of coherence in scene complexity from frame to frame. However, during visualization of more discontinuous virtual environments, scene complexity can vary radically between successive frames. For instance, in a building walkthrough, the observer may turn around a corner into a large atrium, or step from an open corridor into a small, enclosed office. In these situations, the number and complexity of the objects visible to the observer changes suddenly. Thus the size threshold chosen based on the time required to render previous frames is inappropriate, and can result in very poor performance until the system reacts. Overshoot and oscillation can occur as the feedback control system attempts to adjust the size threshold more quickly to achieve the target frame rate.

In order to *guarantee* a bounded frame rate during visualization of discontinuous virtual environments, an adaptive algorithm for LOD selection should be *predictive*, based on the complexity of the scene to be rendered in the current frame, rather than *reactive*, based only on the time required to render previous frames. A predictive algorithm might estimate the time required to render every object at every level of detail, and then compute the largest size threshold that allows the current frame to be rendered within the target frame time. Unfortunately, implementing a predictive algorithm is nontrivial, since no closed-form solution exists for the appropriate size threshold.

3 Overview of Approach

Our approach is a generalization of the predictive approach. Conceptually, every potentially visible object can be rendered at any level of detail, and with any rendering algorithm (e.g., flat-shaded, Gouraud-shaded, texture mapped, etc.). Every combination of objects rendered with certain levels of detail and rendering algorithms takes a certain amount of time, and produces a certain image. We aim to find the combination of levels of detail and rendering algorithms for all potentially visible objects that produces the "best" image possible within the target frame time.

More formally, we define an *object tuple*, (O, L, R), to be an instance of object O, rendered at level of detail L, with rendering algorithm R. We define two heuristics for object tuples: $Cost(O, L, R)$ and $Benefit(O, L, R)$. The *Cost* heuristic estimates the time required to render an object tuple; and the *Benefit* heuristic estimates the "contribution to model perception" of a rendered object tuple. We define S to be the set of object tuples rendered in each frame. Using these formalisms, our approach for choosing a level of detail and rendering algorithm for each potentially visible object can be stated:

$$\begin{aligned} &\text{Maximize :} && \textstyle\sum_S Benefit(O, L, R) \\ &\text{Subject to :} && \textstyle\sum_S Cost(O, L, R) \leq TargetFrameTime \end{aligned} \quad (1)$$

This formulation captures the essence of image generation with real-time constraints: "do as well as possible in a given amount of time." As such, it can be applied to a wide variety of problems that require images to be displayed in a fixed amount of time, including adaptive ray tracing (i.e., given a fixed number of rays, cast those that contribute most to the image), and adaptive radiosity (i.e., given

a fixed number of form-factor computations, compute those that contribute most to the solution). If levels of detail representing "no polygons at all" are allowed, this approach handles cases where the target frame time is not long enough to render all potentially visible objects even at the lowest level of detail. In such cases, only the most "valuable" objects are rendered so that the frame time constraint is not violated. Using this approach, it is possible to generate images in a short, fixed amount of time, rather than waiting much longer for images of the highest quality attainable.

For this approach to be successful, we need to find *Cost* and *Benefit* heuristics that can be computed quickly and accurately. Unfortunately, *Cost* and *Benefit* heuristics for a specific object tuple cannot be predicted with perfect accuracy, and may depend on other object tuples rendered in the same image. A perfect *Cost* heuristic may depend on the model and features of the graphics workstation, the state of the graphics system, the state of the operating system, and the state of other programs running on the machine. A perfect *Benefit* heuristic would consider occlusion and color of other object tuples, human perception, and human understanding. We cannot hope to quantify all of these complex factors in heuristics that can be computed efficiently. However, using several simplifying assumptions, we have developed approximate *Cost* and *Benefit* heuristics that are both efficient to compute and accurate enough to be useful.

4 Cost Heuristic

The $Cost(O, L, R)$ heuristic is an estimate of the time required to render object O with level of detail L and rendering algorithm R. Of course, the actual rendering time for a set of polygons depends on a number of complex factors, including the type and features of the graphics workstation. However, using a model of a generalized rendering system and several simplifying assumptions, it is possible to develop an efficient, approximate *Cost* heuristic that can be applied to a wide variety of workstations. Our model, which is derived from the *Graphics Library Programming Tools and Techniques* document from Silicon Graphics, Inc. [10], represents the rendering system as a pipeline with the two functional stages shown in Figure 2:

- *Per Primitive:* coordinate transformations, lighting calculations, clipping, etc.
- *Per Pixel:* rasterization, z-buffering, alpha blending, texture mapping, etc.

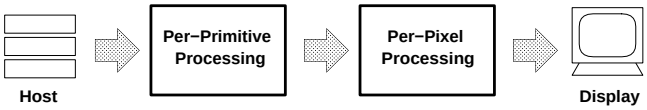


Figure 2: Two-stage model of the rendering pipeline.

Since separate stages of the pipeline run in parallel, and must wait only if a subsequent stage is "backed up," the throughput of the pipeline is determined by the speed of the slowest stage – i.e., the bottleneck. If we assume that the host is able to send primitives to the graphics subsystem faster than they can be rendered, and no other operations are executing that affect the speed of any stage of the graphics subsystem, we can model the time required to render an object tuple as the maximum of the times taken by any of the stages.

We model the time taken by the *Per Primitive* stage as a linear combination of the number of polygons and vertices in an object tuple, with coefficients that depend on the rendering algorithm and machine used. Likewise, we assume that the time taken by the *Per Pixel* stage is proportional to the number of pixels an object covers. Our model for the time required to render an object tuple is:

$$Cost(O, L, R) = max \left\{ \begin{array}{c} C_1 Poly(O, L) + C_2 Vert(O, L) \\ C_3 Pix(O) \end{array} \right\}$$

where O is the object, L is the level of detail, R is the rendering algorithm, and C_1, C_2 and C_3 are constant coefficients specific to a rendering algorithm and machine.

For a particular rendering algorithm and machine, useful values for these coefficients can be determined experimentally by rendering sample objects with a wide variety of sizes and LODs, and graphing measured rendering times versus the number of polygons, vertices and pixels drawn. Figure 3a shows measured times for rendering four different LODs of the chair shown in Figure 1 rendered with flat-shading. The slope of the best fitting line through the data points represents the time required *per polygon* during this test. Using this technique, we have derived cost model coefficients for our Silicon Graphics VGX 320 that are accurate within 10% at the 95% confidence level. A comparison of actual and predicted rendering times for a sample set of frames during an interactive building walkthrough is shown in Figure 3b.

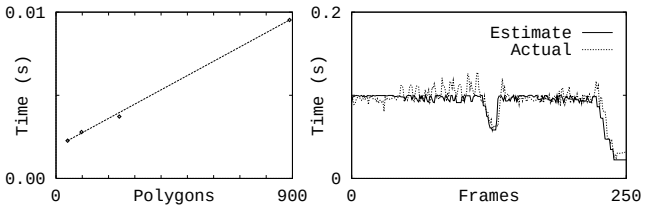


Figure 3: Cost model coefficients can be determined empirically. The plot in (a) shows actual flat-shaded rendering times for four LODs of a chair, and (b) shows a comparison of actual and estimated rendering times of frames during an interactive building walkthrough.

5 Benefit Heuristic

The $Benefit(O, L, R)$ heuristic is an estimate of the "contribution to model perception" of rendering object O with level of detail L and rendering algorithm R. Ideally, it predicts the amount and accuracy of information conveyed to a user due to rendering an object tuple. Of course, it is extremely difficult to accurately model human perception and understanding, so we have developed a simple, easy-to-compute heuristic based on intuitive principles.

Our *Benefit* heuristic depends primarily on the size of an object tuple in the final image. Intuitively, objects that appear larger to the observer "contribute" more to the image (see Figure 4). Therefore, the base value for our *Benefit* heuristic is simply an estimate of the number of pixels covered by the object.

Our *Benefit* heuristic also depends on the "accuracy" of an object tuple rendering. Intuitively, using a more detailed representation or a more realistic rendering algorithm for an object generates a higher quality image, and therefore conveys more accurate information to the user. Conceptually, we evaluate the "accuracy" of an object tuple rendering by comparison to an *ideal image* generated with an

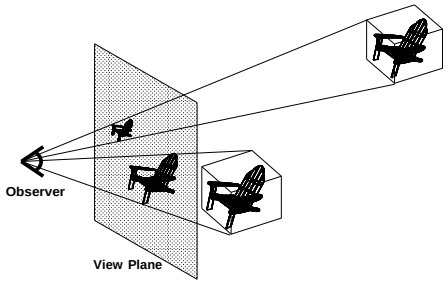
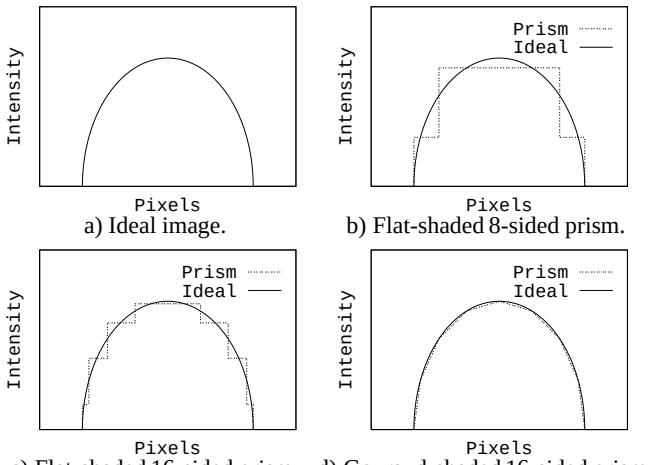


Figure 4: Objects that appear larger "contribute" more to the image.

ideal camera. For instance, consider generating a gray-level image of a scene containing only a cylinder with a diffusely reflecting Lambert surface illuminated by a single directional light source in orthonormal projection. Figure 5a shows an intensity plot of a sample scan-line of an ideal image generated for the cylinder.

First, consider approximating this ideal image with an image generated using a flat-shaded, polygonal representation for the cylinder. Since a single color is assigned to all pixels covered by the same polygon, a plot of pixel intensities across a scan-line of such an image is a stair-function. If an 8-sided prism is used to represent the cylinder, at most 4 distinct colors can appear in the image (one for each front-facing polygon), so the resulting image does not approximate the ideal image very well at all, as shown in Figure 5b. By comparison, if a 16-sided prism is used to represent the cylinder, as many as 8 distinct colors can appear in the image, generating a closer approximation to the ideal image, as shown in Figure 5c.



a) Ideal image. b) Flat-shaded 8-sided prism.

c) Flat-shaded 16-sided prism. d) Gouraud-shaded 16-sided prism.

Figure 5: Plots of pixel intensity across a sample scan-line of images generated using different representations and rendering algorithms for a simple cylinder.

Next, consider using Gouraud shading for a polygonal representation. In Gouraud shading, intensities are interpolated between vertices of polygons, so a plot of pixel intensities is a continuous, piecewise-linear function. Figure 5d shows a plot of pixel intensities across a scan line for a Gouraud shaded 16-sided prism. Compared to the plot for the flat-shaded image (Figure 5b), the Gouraud shaded image approximates the ideal image much more closely.

More complex representations (e.g., parametric or implicit surfaces) and rendering techniques (e.g., Phong shading, antialiasing or ray tracing) could be used to approximate the ideal image even more closely. Based on this intuition, we assume that the "error," i.e., the difference from the ideal image, decreases with the number of samples (e.g., rays/vertices/polygons) used to render an object tuple, and is dependent on the type of interpolation method used (e.g., Gouraud/flat). We capture these effects in the *Benefit* heuristic by multiplying by an "accuracy" factor:

$$Accuracy(O, L, R) = 1 - Error = 1 - \frac{BaseError}{Samples(L, R)^m}$$

where Samples(L, R) is #pixels for ray tracing, or #vertices for Gouraud shading, or #polygons for flat-shading (but never more than #pixels); and m is an exponent dependent on the interpolation method used (flat = 1, Gouraud = 2). The *BaseError* is arbitrarily set to 0.5 to give a strong error for a curved surface represented by a single flat polygon, but still account for a significantly higher benefit than not rendering the surface at all.

In addition to the size and accuracy of an object tuple rendering, our *Benefit* heuristic depends on on several other, more qualitative, factors, some of which apply to a static image, while others apply to sequences of images:

- **Semantics:** Some types of object may have inherent "importance." For instance, walls might be more important than pencils to the user of a building walkthrough; and enemy robots might be most important to the user of a video game. We adjust the *Benefit* of each object tuple by an amount proportional to the inherent importance of its object type.
- **Focus:** Objects that appear in the portion of the screen at which the user is looking might contribute more to the image than ones in the periphery of the user's view. Since we currently do not track the user's eye position, we simply assume that objects appearing near the middle of the screen are more important than ones near the side. We reduce the *Benefit* of each object tuple by an amount proportional to its distance from the middle of the screen.
- **Motion Blur:** Since objects that are moving quickly across the screen appear blurred or can be seen for only a short amount of time, the user may not be able to see them clearly. So we reduce the *Benefit* of each object tuple by an amount proportional to the ratio of the object's apparent speed to the size of an average polygon.
- **Hysteresis:** Rendering an object with different levels of detail in successive frames may be bothersome to the user and may reduce the quality of an image sequence. Therefore, we reduce the *Benefit* of each object tuple by an amount proportional to the difference in level of detail or rendering algorithm from the ones used for the same object in the previous frame.

Each of these qualitative factors is represented by a multiplier between 0.0 and 1.0 reflecting a possible reduction in object tuple benefit. The overall *Benefit* heuristic is a product of all the aforementioned factors:

$$Benefit(O, L, R) = Size(O) * Accuracy(O, L, R) * Importance(O) * Focus(O) * Motion(O) * Hysteresis(O, L, R)$$

This *Benefit* heuristic is a simple experimental estimate of an object tuple's "contribution to model perception." Greater *Benefit* is assigned to object tuples that are larger (i.e., cover more pixels in the image), more realistic-looking (i.e., rendered with higher levels of detail, or better rendering algorithms), more important (i.e.,

semantically, or closer to the middle of the screen), and more apt to blend with other images in a sequence (i.e., hysteresis). In our implementation, the user can manipulate the relative weighting of these factors interactively using sliders on a control panel, and observe their effects in a real-time walkthrough. Therefore, although our current *Benefit* heuristic is rather ad hoc, it is useful for experimentation until we are able to encode more accurate models for human visual perception and understanding.

6 Optimization Algorithm

We use the *Cost* and *Benefit* heuristics described in the previous sections to choose a set of object tuples to render each frame by solving equation 1 in Section 3.

Unfortunately, this constrained optimization problem is NP-complete. It is the Continuous Multiple Choice Knapsack Problem [6, 7], a version of the well-known Knapsack Problem in which elements are partitioned into candidate sets, and at most one element from each candidate set may be placed in the knapsack at once. In this case, the set *S* of object tuples rendered is the knapsack, the object tuples are the elements to be placed into the knapsack, the target frame time is the size of the knapsack, the sets of object tuples representing the same object are the candidate sets, and the *Cost* and *Benefit* functions specify the "size" and "profit" of each element, respectively. The problem is to select the object tuples that have maximum cumulative benefit, but whose cumulative cost fits in the target frame time, subject to the constraint that only one object tuple representing each object may be selected.

We have implemented a simple, greedy approximation algorithm for this problem that selects object tuples with the highest *Value* ($Benefit(O, L, R) / Cost(O, L, R)$). Logically, we add object tuples to *S* in descending order of *Value* until the maximum cost is competely claimed. However, if an object tuple is added to *S* which represents the same object as another object tuple already in *S*, only the object tuple with the maximum benefit of the two is retained. The merit of this approach can be explained intuitively by noting that each subsequent portion of the frame time is used to render the object tuple with the best available "bang for the buck." It is easy to show that a simple implementation of this greedy approach runs in $O(n \log n)$ time for n potentially visible objects, and produces a solution that is at least half as good as the optimal solution [6].

Rather than computing and sorting the *Benefit*, *Cost*, and *Value* for all possible object tuples during every frame, as would be required by a naive implementation, we have implemented an incremental optimization algorithm that takes advantage of the fact that there is typically a large amount of coherence between successive frames. The algorithm works as follows: At the start of the algorithm, an object tuple is added to *S* for each potentially visible object. Initially, each object is assigned the LOD and rendering algorithm chosen in the previous frame, or the lowest LOD and rendering algorithm if the object is newly visible. In each iteration of the optimization, the algorithm first increments the accuracy attribute (LOD or rendering algorithm) of the object that has the highest subsequent *Value*. It then decrements the accuracy attributes of the object tuples with the lowest current *Value* until the cumulative cost of all object tuples in *S* is less than the target frame time. The algorithm terminates when the same accuracy attribute of the same object tuple is both incremented and decremented in the same iteration.

This incremental implementation finds an approximate solution that is the same as found by the naive implementation if *Values* of object tuples decrease monotonically as tuples are rendered with greater accuracy (i.e., there are diminishing returns with more complex renderings). In any case, the worst-case running time for the algorithm is $O(n \log n)$. However, since the initial guess for the LOD and rendering algorithm for each object is generated from the previous frame, and there is often a large amount of coherence from frame to frame, the algorithm completes in just a few iterations on average. Moreover, computations are done in parallel with the display of the previous frame on a separate processor in a pipelined architecture; they do not increase the effective frame rate as long as the time required for computation is not greater than the time required for display.

7 Test Methods

To test whether this new cost/benefit optimization algorithm produces more uniform frame rates than previous LOD selection algorithms, we ran a set of tests with our building walkthrough application using four different LOD selection algorithms:

a) **No Detail Elision:** Each object is rendered at the highest LOD.

b) **Static:** Each object is rendered at the highest LOD for which an average polygon covers at least 1024 pixels on the screen.

c) **Feedback:** Similar to *Static* test, except the size threshold for LOD selection is updated in each frame by a feedback loop, based on the difference between the time required to render the previous frame and the target frame time of one-tenth of a second.

d) **Optimization:** Each object is rendered at the LOD chosen by the cost/benefit optimization algorithm described in Sections 3 and 6 in order to meet the target frame time of one-tenth of a second. For comparison sake, the *Benefit* heuristic is limited to consideration of *object size* in this test, i.e., all other *Benefit* factors are set to 1.0.

All tests were performed on a Silicon Graphics VGX 320 workstation with two 33MHz MIPS R3000 processors and 64MB of memory. We used an *eye-to-object* visibility algorithm described in [12] to determine a set of potentially visible objects to be rendered in each frame. The application was configured as a two-stage pipeline with one processor for visibility and LOD selection computations and another separate processor for rendering. Timing statistics were gathered using a $16\mu s$ timer.

In each test, we used the sample observer path shown in Figure 6 through a model of an auditorium on the third floor of Soda Hall. The model was chosen because it is complex enough to differentiate the characteristics of various LOD selection algorithms (87,565 polygons), yet small enough to reside entirely in main memory so as to eliminate the effects of memory management in our tests. The test path was chosen because it represents typical behavior of real users of a building walkthrough system, and highlights the differences between various LOD selection algorithms. For instance, at the observer viewpoint marked 'A', many complex objects are simultaneously visible, some of which are close and appear large to the observer; at the viewpoint marked 'B', there are very few objects visible to the observer, most of which appear small; and at the viewpoint marked 'C', numerous complex objects become visible suddenly as the observer spins around quickly. We refer to these marked observer viewpoints in the analysis, as they are the viewpoints at which the differences between the various LOD selection algorithms are most pronounced.

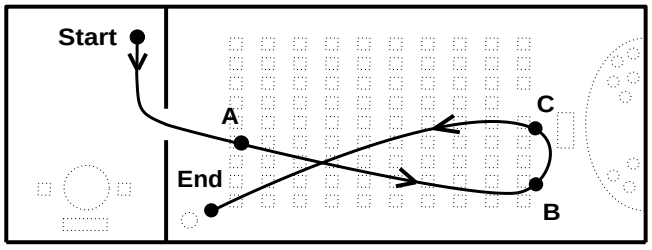


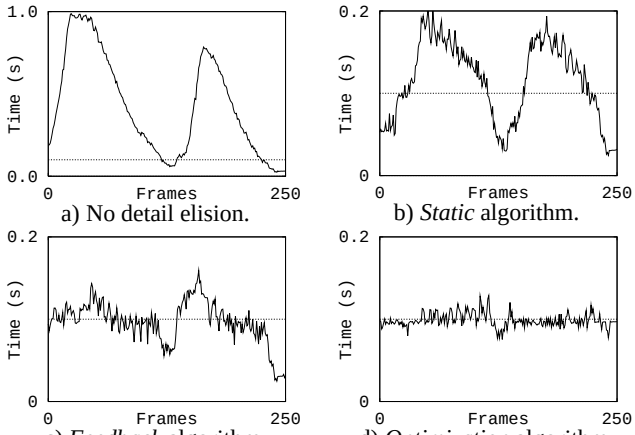
Figure 6: Test observer path through a model of an auditorium.

LOD Selection Algorithm	Compute Time		Frame Time		
	Mean	Max	Mean	Max	StdDev
None	0.00	0.00	0.43	0.99	0.305
Static	0.00	0.01	0.11	0.20	0.048
Feedback	0.00	0.01	0.10	0.16	0.026
Optimization	0.01	0.03	0.10	0.13	0.008

Table 1: Cumulative statistics for test observer path (in seconds).

8 Results and Discussion

Figure 7 shows plots of the frame time (seconds per frame) for each observer viewpoint along the test path for the four LOD selection algorithms tested. Table 1 shows cumulative compute time (i.e., time required for execution of the LOD selection algorithm) and frame time statistics for all observer viewpoints along the test path.

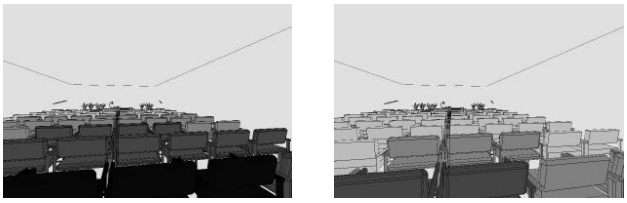


a) No detail elision. b) *Static* algorithm.
c) *Feedback* algorithm. d) *Optimization* algorithm.

Figure 7: Plots of frame time for every observer viewpoint along test observer path using a) no detail elision, b) static algorithm, c) feedback algorithm, and d) optimization algorithm. Note: the "Frame Time" axis in plot (a) is five-times larger than the others.

If no detail elision is used, and all potentially visible objects are rendered at the highest LOD, the time required for each frame is generally long and non-uniform, since it depends directly on the number and complexity of the objects visible to the observer (see Figure 7a). In our test model, far too many polygons are visible from most observer viewpoints to generate frames at interactive rates without detail elision. For instance, at the observer viewpoint marked 'A' in Figure 6, 72K polygons are simultaneously visible, and the frame time is 0.98 seconds. Overall, the mean frame time for all observer viewpoints on the test path is 0.43 seconds per frame.

If the *Static* LOD selection algorithm is used, objects whose average polygon is smaller than a size threshold fixed at 1024 pixels per polygon are rendered with lower LODs. Even though the frame rate is much faster than without detail elision, there is still a large amount of variability in the frame time, since it depends on the size and complexity of the objects visible from the observer's viewpoint (see Figure 7b). For instance, at the observer viewpoint marked 'A', the frame time is quite long (0.19 seconds) because many visible objects are complex and appear large to the observer. A high LOD is chosen for each of these objects independently, resulting in a long overall frame time. This result can seen clearly in Figure 8a which depicts the LOD selected for each object in the frame for observer viewpoint 'A' – higher LODs are represented by darker shades of gray. On the other hand, the frame time is very short in the frame at the observer viewpoint marked 'B' (0.03 seconds). Since all visible objects appear relatively small to the observer, they are rendered at a lower LOD even though more detail could have been rendered within the target frame time. In general, it is impossible to choose a single size threshold for LOD selection that generates uniform frame times for all observer viewpoints.

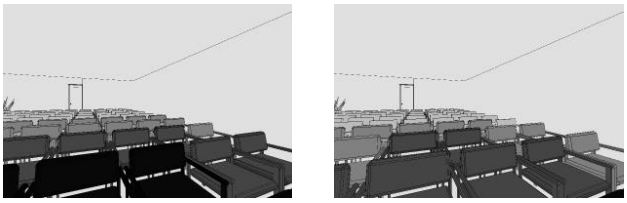


a) *Static* algorithm b) *Optimization* algorithm

Figure 8: Images depicting the LODs selected for each object at the observer viewpoints marked 'A' using the *Static* and *Optimization* algorithms. Darker shades of gray represent higher LODs.

The *Feedback* algorithm adjusts the size threshold for LOD selection adaptively based on the time taken to render previous frames in an effort to maintain a uniform frame rate. This algorithm generates a fairly uniform frame rate in situations of smoothly varying scene complexity, as evidenced by the relatively flat portions of the frame time curve shown in Figure 7c (frames 1–125). However, in situations where the complexity of the scene visible to the observer changes suddenly, peaks and valleys appear in the curve. Sometimes the frame time generated using the *Feedback* algorithm can be even longer than the one generated using the *Static* algorithm, as the *Feedback* algorithm is lured into a inappropriately low size threshold during times of low scene complexity. For instance, just before the viewpoint marked 'C', the observer is looking at a relatively simple scene containing just a few objects on the stage, so frame times are very short, and the size threshold for LOD selection is reduced to zero. However, at the viewpoint marked 'C', many chairs become visible suddenly as the observer spins around quickly. Since the adaptive size threshold is set very low, inappropriately high LODs are chosen for most objects (see Figure 9a), resulting in a frame time of 0.16 seconds. Although the size threshold can often adapt quickly after such discontinuities in scene complexity, some effects related to this feedback control (i.e., oscillation, overshoot, and a few very slow frames) can be quite disturbing to the user.

In contrast, the *Optimization* algorithm predicts the complexity of the model visible from the current observer viewpoint, and chooses an appropriate LOD and rendering algorithm for each object to meet the target frame time. As a result, the frame time generated using the *Optimization* algorithm is much more uniform than using any of



a) *Feedback* algorithm b) *Optimization* algorithm

Figure 9: Images depicting the LODs selected for each object at the observer viewpoints marked 'C' using the *Feedback* and *Optimization* algorithms. Darker shades of gray represent higher LODs.

the other LOD selection algorithms (see Figure 7d). For all observer viewpoints along the test path, the standard deviation in the frame time is 0.008 seconds, less than one third of any of the other three algorithms tested. The longest frame time is 0.13 seconds, and the shortest is 0.075 seconds.

As the *Optimization* algorithm adjusts image quality to maintain a uniform, interactive frame rate, it attempts to render the "best" image possible within the target frame time for each observer viewpoint. As a result, there is usually little noticeable difference between images generated using the *Optimization* algorithm and ones generated with no detail elision at all. A comparison of images for observer viewpoint 'A' generated using a) no detail elision, and b) using the *Optimization* algorithm to meet a target frame time of one tenth of a second are shown in Figure 10. Figure 10a has 72,570 polygons and took 0.98 seconds to render, whereas Figure 10b has 5,300 polygons and took 0.10 seconds. Even though there are less than a tenth as many polygons in Figure 10b, the difference in image quality is barely noticeable. For reference, the LOD chosen for each object in Figure 10b is shown in Figure 8b. Note that reduction in rendering time does not map to a linear reduction in polygon count since polygons representing lower levels of detail tend to be bigger on average.

The *Optimization* algorithm is more general than other detail elision algorithms in that it also adjusts the rendering algorithm (and possibly other attributes in the future) for each object independently. Examine Figure 11, which shows three images of a small library on the sixth floor of Soda Hall containing several textured surfaces. Figure $11a_1$, shows an image generated using no detail elision – it contains 19,821 polygons and took 0.60 seconds to render. Figures $11b_1$ and $11c_1$ show images generated for the same observer viewpoint using the *Optimization* algorithm with target frame times of b) 0.15 seconds (4,217 polygons), and c) 0.10 seconds (1,389 polygons). Although the *Optimization* algorithm uses lower levels of detail for many objects (see Figures $11b_1$ and $11c_1$), and generates images that are quite different than the one generated with no detail elision (see Figures $11b_2$ and $11c_2$), all three images look very similar. Notice the reduced tessellation of chairs further from the observer, and the omission of texture on the bookshelves in Figure $11b_1$. Similarly, notice the flat-shaded chairs, and the omission of books on bookshelves and texture on doors in Figure 11 c_1.

Having experimented with several LOD selection algorithms in an interactive visualization application, we are optimistic that variation in image quality is less disturbing to a user than variation in the frame times, as long as different representations for each object appear similar, and transitions between representations are not very noticeable. Further experimentation is required to determine which types of rendering attributes can be blended smoothly during interactive visualization.

9 Conclusion

We have described an adaptive display algorithm for fast, uniform frame rates during interactive visualization of large, complex virtual environments. The algorithm adjusts image quality dynamically in order to maintain a user-specified frame rate, selecting a level of detail and an algorithm with which to render each potentially visible object to produce the "best" image possible within the target frame time.

Our tests show that the *Optimization* algorithm generates more uniform frame rates than other previously described detail elision algorithms with little noticeable difference in image quality during visualization of complex models. Interesting topics for further study include algorithms for automatic generation of multiresolution models, and experiments to develop measures of *image quality* and *image differences*.

10 Acknowledgements

We are grateful to Thurman Brown, Delnaz Khorramabadi, Priscilla Shih and Maryann Simmons for their efforts constructing the building model. Silicon Graphics, Inc. has been very generous, allowing us to use equipment, and donating a VGX 320 workstation to this project as part of a grant from the Microelectronics Innovation and Computer Research Opportunities (MICRO 1991) program of the State of California. We appreciate the assistance of Greg Ward, Sharon Fischler, and Henry Moreton who helped generate the color prints for this paper. Finally, we thank Seth Teller for his spatial subdivisions, visibility algorithms, and other important contributions to this project.

References

[1] Airey, John M., Rohlf, John H., and Brooks, Jr., Frederick P. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. *ACM SIGGRAPH Special Issue on 1990 Symposium on Interactive 3D Graphics* , 24, 2 (1990), 41-50.

[2] Blake, Edwin H. A Metric for Computing Adaptive Detail in Animated Scenes using Object-Oriented Programming. *Eurographics '87*. G. Marechal (Ed.), Elsivier Science Publishers, B.V. (North-Holland), 1987.

[3] Brooks, Jr., Frederick P. Walkthrough - A Dynamic Graphics System for Simulating Virtual Buildings. *Proceedings of the 1986 Workshop on Interactive 3D Graphics* .

[4] Clark, James H. Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM*, 19, 10 (October 1976), 547-554.

[5] Funkhouser, Thomas A., Séquin, Carlo H., and Teller, Seth J. Management of Large Amounts of Data in Interactive Building Walkthroughs. *ACM SIGGRAPH Special Issue on 1992 Symposium on Interactive 3D Graphics* , 11-20.

[6] Garey, Michael R., and Johnson, David S. Computers and Intractibility: A Guide to the Theory of NP-Completeness. W.H. Freeman and Company, New York, 1979.

[7] Ibaraki, T., Hasegawa, T., Teranaka, K., and Iwase J. The Multiple Choice Knapsack Problem. *J. Oper. Res. Soc. Japan* 21, 1978, 59-94.

[8] Rossignac, Jarek, and Borrel, Paul. Multi-resolution 3D approximations for rendering complex scenes. *IFIP TC 5.WG 5.10 II Conference on Geometric Modeling in Computer Graphics*, Genova, Italy, 1993. Also available as IBM Research Report RC 17697, Yorktown Heights, NY 10598.

[9] Schachter, Bruce J. (Ed.). *Computer Image Generation*. John Wiley and Sons, New York, NY, 1983.

[10] *Graphics Library Programming Tools and Techniques* , Document #007-1489-01, Silicon Graphics, Inc., 1992.

[11] Teller, Seth J., and Séquin, Carlo H. Visibility Preprocessing for Interactive Walkthroughs. Proceedings of SIGGRAPH '91. In *Computer Graphics* 25, 4 (August 1991), 61-69.

[12] Teller, Seth J. *Visibility Computations in Densely Occluded Polyhedral Environments*. Ph.D. thesis, Computer Science Division (EECS), University of California, Berkeley, 1992. Also available as UC Berkeley technical report UCB/CSD-92-708.

[13] Zyda, Michael J. Course Notes, Book Number 10, Graphics Video Laboratory, Department of Computer Science, Naval Postgraduate School, Monterey, California, November 1991.

a) No detail elision

b) *Optimization* algorithm (0.10 seconds)

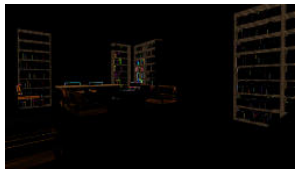
Figure 10: Images for observer viewpoint 'A' generated using a) no detail elision (72,570 polygons), and b) the *Optimization* algorithm with a 0.10 second target frame time (5,300 polygons).

a_1

a) No detail elision

b_1

b_2



b_3

b) *Optimization* algorithm (0.15 seconds)

c_1

c_2

c_3

c) *Optimization* algorithm (0.10 seconds)

Figure 11: Images of library generated using a) no detail elision (19,821 polygons), and the *Optimization* detail elision algorithm with target frame times of b) 0.15 seconds (4,217 polygons), and c) 0.10 seconds (1,389 polygons). LODs chosen for objects in b_1 and c_1 are shown in b_2 and c_c – darker shades of gray represent higher LODs. Pixel-by-pixel differences $abs(a_1 - b_1)$ and $abs(a_1 - c_1)$ are shown in b_3 and c_3 – brighter colors represent greater difference.

Progressive Meshes

Hugues Hoppe
Microsoft Research

ABSTRACT

Highly detailed geometric models are rapidly becoming commonplace in computer graphics. These models, often represented as complex triangle meshes, challenge rendering performance, transmission bandwidth, and storage capacities. This paper introduces the *progressive mesh* (PM) representation, a new scheme for storing and transmitting arbitrary triangle meshes. This efficient, lossless, continuous-resolution representation addresses several practical problems in graphics: smooth geomorphing of level-of-detail approximations, progressive transmission, mesh compression, and selective refinement.

In addition, we present a new mesh simplification procedure for constructing a PM representation from an arbitrary mesh. The goal of this optimization procedure is to preserve not just the geometry of the original mesh, but more importantly its overall appearance as defined by its discrete and scalar appearance attributes such as material identifiers, color values, normals, and texture coordinates. We demonstrate construction of the PM representation and its applications using several practical models.

CR Categories and Subject Descriptors: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - surfaces and object representations.

Additional Keywords: mesh simplification, level of detail, shape interpolation, progressive transmission, geometry compression.

1 INTRODUCTION

Highly detailed geometric models are necessary to satisfy a growing expectation for realism in computer graphics. Within traditional modeling systems, detailed models are created by applying versatile modeling operations (such as extrusion, constructive solid geometry, and freeform deformations) to a vast array of geometric primitives. For efficient display, these models must usually be tessellated into polygonal approximations—meshes. Detailed meshes are also obtained by scanning physical objects using range scanning systems [5]. In either case, the resulting complex meshes are expensive to store, transmit, and render, thus motivating a number of practical problems:

Email: hhoppe@microsoft.com
Web: http://www.research.microsoft.com/research/graphics/hoppe/

- *Mesh simplification*: The meshes created by modeling and scanning systems are seldom optimized for rendering efficiency, and can frequently be replaced by nearly indistinguishable approximations with far fewer faces. At present, this process often requires significant user intervention. Mesh simplification tools can hope to automate this painstaking task, and permit the porting of a single model to platforms of varying performance.
- *Level-of-detail (LOD) approximation*: To further improve rendering performance, it is common to define several versions of a model at various levels of detail [3, 8]. A detailed mesh is used when the object is close to the viewer, and coarser approximations are substituted as the object recedes. Since instantaneous switching between LOD meshes may lead to perceptible "popping", one would like to construct smooth visual transitions, *geomorphs*, between meshes at different resolutions.
- *Progressive transmission*: When a mesh is transmitted over a communication line, one would like to show progressively better approximations to the model as data is incrementally received. One approach is to transmit successive LOD approximations, but this requires additional transmission time.
- *Mesh compression*: The problem of minimizing the storage space for a model can be addressed in two orthogonal ways. One is to use mesh simplification to reduce the number of faces. The other is mesh compression: minimizing the space taken to store a particular mesh.
- *Selective refinement*: Each mesh in a LOD representation captures the model at a uniform (view-independent) level of detail. Sometimes it is desirable to adapt the level of refinement in selected regions. For instance, as a user flies over a terrain, the terrain mesh need be fully detailed only near the viewer, and only within the field of view.

In addressing these problems, this paper makes two major contributions. First, it introduces the *progressive mesh* (PM) representation. In PM form, an arbitrary mesh $\hat{M}$ is stored as a much coarser mesh M^0 together with a sequence of n detail records that indicate how to incrementally refine M^0 exactly back into the original mesh $\hat{M} = M^n$. Each of these records stores the information associated with a *vertex split*, an elementary mesh transformation that adds an additional vertex to the mesh. The PM representation of $\hat{M}$ thus defines a continuous sequence of meshes $M^0, M^1, \ldots, M^n$ of increasing accuracy, from which LOD approximations of any desired complexity can be efficiently retrieved. Moreover, geomorphs can be efficiently constructed between any two such meshes. In addition, we show that the PM representation naturally supports progressive transmission, offers a concise encoding of $\hat{M}$ itself, and permits selective refinement. In short, progressive meshes offer an efficient, lossless, continuous-resolution representation.

The other contribution of this paper is a new simplification procedure for constructing a PM representation from a given mesh $\hat{M}$. Unlike previous simplification methods, our procedure seeks to preserve not just the geometry of the mesh surface, but more importantly its overall appearance, as defined by the discrete and scalar attributes associated with its surface.

2 MESHES IN COMPUTER GRAPHICS

Models in computer graphics are often represented using triangle meshes.[1] Geometrically, a triangle mesh is a piecewise linear surface consisting of triangular faces pasted together along their edges. As described in [9], the mesh geometry can be denoted by a tuple (K, V), where K is a *simplicial complex* specifying the connectivity of the mesh simplices (the adjacency of the vertices, edges, and faces), and $V = \{\mathbf{v}_1, \ldots, \mathbf{v}_m\}$ is the set of vertex positions defining the shape of the mesh in $\mathbf{R}^3$. More precisely (cf. [9]), we construct a parametric domain $|K| \subset \mathbf{R}^m$ by identifying each vertex of K with a canonical basis vector of $\mathbf{R}^m$, and define the mesh as the image $\phi_V(|K|)$ where $\phi_V : \mathbf{R}^m \to \mathbf{R}^3$ is a linear map.

Often, surface appearance attributes other than geometry are also associated with the mesh. These attributes can be categorized into two types: *discrete* attributes and *scalar* attributes.

Discrete attributes are usually associated with faces of the mesh. A common discrete attribute, the *material identifier*, determines the shader function used in rendering a face of the mesh [18]. For instance, a trivial shader function may involve simple look-up within a specified texture map.

Many scalar attributes are often associated with a mesh, including diffuse color (r, g, b), normal (n_x, n_y, n_z), and texture coordinates (u, v). More generally, these attributes specify the local parameters of shader functions defined on the mesh faces. In simple cases, these scalar attributes are associated with vertices of the mesh. However, to represent discontinuities in the scalar fields, and because adjacent faces may have different shading functions, it is common to associate scalar attributes not with vertices, but with corners of the mesh [1]. A *corner* is defined as a (vertex,face) tuple. Scalar attributes at a corner (v, f) specify the shading parameters for face f at vertex v. For example, along a *crease* (a curve on the surface across which the normal field is not continuous), each vertex has two distinct normals, one associated with the corners on each side of the crease.

We express a mesh as a tuple $M = (K, V, D, S)$ where V specifies its geometry, D is the set of discrete attributes d_f associated with the faces $f = \{j, k, l\} \in K$, and S is the set of scalar attributes $s_{(v,f)}$ associated with the corners (v, f) of K.

The attributes D and S give rise to discontinuities in the visual appearance of the mesh. An edge $\{v_j, v_k\}$ of the mesh is said to be *sharp* if either (1) it is a boundary edge, or (2) its two adjacent faces f_l and f_r have different discrete attributes (i.e. $d_{f_l} \neq d_{f_r}$), or (3) its adjacent corners have different scalar attributes (i.e. $s_{(v_j,f_l)} \neq s_{(v_j,f_r)}$ or $s_{(v_k,f_l)} \neq s_{(v_k,f_r)}$). Together, the set of sharp edges define a set of *discontinuity curves* over the mesh (e.g. the yellow curves in Figure 12).

3 PROGRESSIVE MESH REPRESENTATION

3.1 Overview

Hoppe et al. [9] describe a method, *mesh optimization*, that can be used to approximate an initial mesh $\hat{M}$ by a much simpler one. Their optimization algorithm, reviewed in Section 4.1, traverses the space of possible meshes by successively applying a set of 3 mesh transformations: edge collapse, edge split, and edge swap.

We have discovered that in fact a single one of those transformations, *edge collapse*, is sufficient for effectively simplifying meshes. As shown in Figure 1, an edge collapse transformation $ecol(\{v_s, v_t\})$ unifies 2 adjacent vertices v_s and v_t into a single vertex v_s. The vertex v_t and the two adjacent faces $\{v_s, v_t, v_l\}$ and $\{v_t, v_s, v_r\}$ vanish in the process. A position $\mathbf{v}_s$ is specified for the new unified vertex.

[1] We assume in this paper that more general meshes, such as those containing n-sided faces and faces with holes, are first converted into triangle meshes by triangulation. The PM representation could be generalized to handle the more general meshes directly, at the expense of more complex data structures.

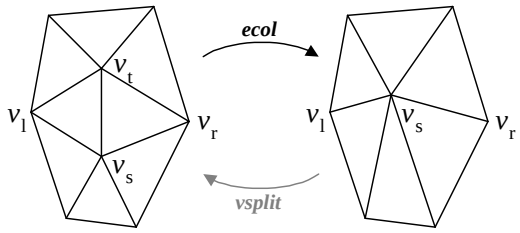


Figure 1: Illustration of the edge collapse transformation.

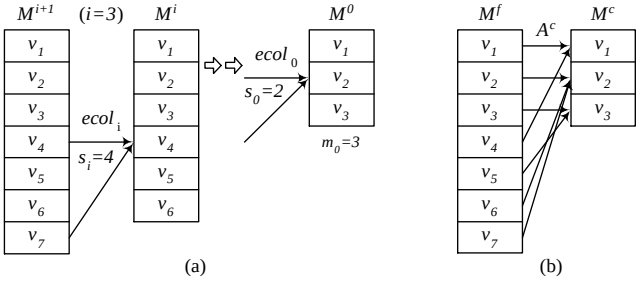


Figure 2: (a) Sequence of edge collapses; (b) Resulting vertex correspondence.

Thus, an initial mesh $\hat{M} = M^n$ can be simplified into a coarser mesh M^0 by applying a sequence of n successive edge collapse transformations:

$$(\hat{M} = M^n) \xrightarrow{ecol_{n-1}} \ldots \xrightarrow{ecol_1} M^1 \xrightarrow{ecol_0} M^0 .$$

The particular sequence of edge collapse transformations must be chosen carefully, since it determines the quality of the approximating meshes $M^i, i < n$. A scheme for choosing these edge collapses is presented in Section 4.

Let m_0 be the number of vertices in M^0, and let us label the vertices of mesh M^i as $V^i = \{v_1, \ldots, v_{m_0+i}\}$, so that edge $\{v_{s_i}, v_{m_0+i+1}\}$ is collapsed by $ecol_i$ as shown in Figure 2a. As vertices may have different positions in the different meshes, we denote the position of v_j in M^i as $\mathbf{v}_j^i$.

A key observation is that an edge collapse transformation is invertible. Let us call that inverse transformation a *vertex split*, shown as *vsplit* in Figure 1. A vertex split transformation $vsplit(s, l, r, t, A)$ adds near vertex v_s a new vertex v_t and two new faces $\{v_s, v_t, v_l\}$ and $\{v_t, v_s, v_r\}$. (If the edge $\{v_s, v_t\}$ is a boundary edge, we let $v_r = 0$ and only one face is added.) The transformation also updates the attributes of the mesh in the neighborhood of the transformation. This attribute information, denoted by A, includes the positions $\mathbf{v}_s$ and $\mathbf{v}_t$ of the two affected vertices, the discrete attributes $d_{\{v_s,v_t,v_l\}}$ and $d_{\{v_t,v_s,v_r\}}$ of the two new faces, and the scalar attributes of the affected corners ($s_{(v_s,\cdot)}$, $s_{(v_t,\cdot)}$, $s_{(v_l,\{v_s,v_t,v_l\})}$, and $s_{(v_r,\{v_t,v_s,v_r\})}$).

Because edge collapse transformations are invertible, we can therefore represent an arbitrary triangle mesh $\hat{M}$ as a simple mesh M^0 together with a sequence of n *vsplit* records:

$$M^0 \xrightarrow{vsplit_0} M^1 \xrightarrow{vsplit_1} \ldots \xrightarrow{vsplit_{n-1}} (M^n = \hat{M})$$

where each record is parametrized as $vsplit_i(s_i, l_i, r_i, A_i)$. We call $(M^0, \{vsplit_0, \ldots, vsplit_{n-1}\})$ a *progressive mesh* (PM) representation of $\hat{M}$.

As an example, the mesh $\hat{M}$ of Figure 5d (13,546 faces) was simplified down to the coarse mesh M^0 of Figure 5a (150 faces) using

6,698 edge collapse transformations. Thus its PM representation consists of M^0 together with a sequence of $n = 6698$ *vsplit* records. From this PM representation, one can extract approximating meshes with any desired number of faces (actually, within ± 1) by applying to M^0 a prefix of the *vsplit* sequence. For example, Figure 5 shows approximating meshes with 150, 500, and 1000 faces.

3.2 Geomorphs

A nice property of the vertex split transformation (and its inverse, edge collapse) is that a smooth visual transition (a *geomorph*) can be created between the two meshes M^i and M^{i+1} in $M^i \xrightarrow{vsplit_i} M^{i+1}$. For the moment let us assume that the meshes contain no attributes other than vertex positions. With this assumption the vertex split record is encoded as $vsplit_i(s_i, l_i, r_i, A_i = (\mathbf{v}_{s_i}^{i+1}, \mathbf{v}_{m_0+i+1}^{i+1}))$. We construct a geomorph $M^G(\alpha)$ with blend parameter $0 \le \alpha \le 1$ such that $M^G(0)$ looks like M^i and $M^G(1)$ looks like M^{i+1}—in fact $M^G(1) = M^{i+1}$—by defining a mesh

$$M^G(\alpha) = (K^{i+1}, V^G(\alpha))$$

whose connectivity is that of M^{i+1} and whose vertex positions linearly interpolate from $v_{s_i} \in M^i$ to the split vertices $v_{s_i}, v_{m_0+i+1} \in M^{i+1}$:

$$\mathbf{v}_j^G(\alpha) = \begin{cases} (\alpha)\mathbf{v}_j^{i+1} + (1-\alpha)\mathbf{v}_{s_i}^i & , \; j \in \{s_i, m_0+i+1\} \\ \mathbf{v}_j^{i+1} = \mathbf{v}_j^i & , \; j \notin \{s_i, m_0+i+1\} \end{cases}$$

Using such geomorphs, an application can smoothly transition from a mesh M^i to meshes M^{i+1} or M^{i-1} without any visible "snapping" of the meshes.

Moreover, since individual *ecol* transformations can be transitioned smoothly, so can the composition of any sequence of them. Geomorphs can therefore be constructed between *any* two meshes of a PM representation. Indeed, given a finer mesh M^f and a coarser mesh M^c, $0 \le c < f \le n$, there exists a natural correspondence between their vertices: each vertex of M^f is related to a unique ancestor vertex of M^c by a surjective map A^c obtained by composing a sequence of *ecol* transformations (Figure 2b). More precisely, each vertex v_j of M^f corresponds with the vertex $v_{A^c(j)}$ in M^c where

$$A^c(j) = \begin{cases} j & , \; j \le m_0 + c \\ A^c(s_{j-m_0-1}) & , \; j > m_0 + c \end{cases} .$$

(In practice, this ancestor information A^c is gathered in a forward fashion as the mesh is refined.) This correspondence allows us to define a geomorph $M^G(\alpha)$ such that $M^G(0)$ looks like M^c and $M^G(1)$ equals M^f. We simply define $M^G(\alpha) = (K^f, V^G(\alpha))$ to have the connectivity of M^f and the vertex positions

$$\mathbf{v}_j^G(\alpha) = (\alpha)\mathbf{v}_j^f + (1-\alpha)\mathbf{v}_{A^c(j)}^c .$$

So far we have outlined the construction of geomorphs between PM meshes containing only position attributes. We can in fact construct geomorphs for meshes containing both discrete and scalar attributes.

Discrete attributes by their nature cannot be smoothly interpolated. Fortunately, these discrete attributes are associated with faces of the mesh, and the "geometric" geomorphs described above smoothly introduce faces. In particular, observe that the faces of M^c are a proper subset of the faces of M^f, and that those faces of M^f missing from M^c are invisible in $M^G(0)$ because they have been collapsed to degenerate (zero area) triangles. Other geomorphing schemes [10, 11, 17] define well-behaved (invertible) parametrizations between meshes at different levels of detail, but these do not permit the construction of geomorphs between meshes with different discrete attributes.

Scalar attributes defined on corners can be smoothly interpolated much like the vertex positions. There is a slight complication in that a corner (v, f) in a mesh M is not naturally associated with any "ancestor corner" in a coarser mesh M^c if f is not a face of M^c. We can still attempt to infer what attribute value (v, f) would have in M^c as follows. We examine the mesh M^{i+1} in which f is first introduced, locate a neighboring corner (v, f') in M^{i+1} whose attribute value is the same, and recursively backtrack from it to a corner in M^c. If there is no neighboring corner in M^{i+1} with an identical attribute value, then the corner (v, f) has no equivalent in M^c and we therefore keep its attribute value constant through the geomorph.

The interpolating function on the scalar attributes need not be linear; for instance, normals are best interpolated over the unit sphere, and colors may be interpolated in a color space other than RGB.

Figure 6 demonstrates a geomorph between two meshes M^{175} (500 faces) and M^{425} (1000 faces) retrieved from the PM representation of the mesh in Figure 5d.

3.3 Progressive transmission

Progressive meshes are a natural representation for progressive transmission. The compact mesh M^0 is transmitted first (using a conventional uni-resolution format), followed by the stream of $vsplit_i$ records. The receiving process incrementally rebuilds $\hat{M}$ as the records arrive, and animates the changing mesh. The changes to the mesh can be geomorphed to avoid visual discontinuities. The original mesh $\hat{M}$ is recovered exactly after all n records are received, since PM is a lossless representation.

The computation of the receiving process should be balanced between the reconstruction of $\hat{M}$ and interactive display. With a slow communication line, a simple strategy is to display the current mesh whenever the input buffer is found to be empty. With a fast communication line, we find that a good strategy is to display meshes whose complexities increase exponentially. (Similar issues arise in the display of images transmitted using progressive JPEG.)

3.4 Mesh compression

Even though the PM representation encodes both $\hat{M}$ and a continuous family of approximations, it is surprisingly space-efficient, for two reasons. First, the locations of the vertex split transformations can be encoded concisely. Instead of storing all three vertex indices (s_i, l_i, r_i) of $vsplit_i$, one need only store s_i and approximately 5 bits to select the remaining two vertices among those adjacent to v_{s_i}.[2] Second, because a vertex split has local effect, one can expect significant coherence in mesh attributes through each transformation. For instance, when vertex $v_{s_i}^i$ is split into $v_{s_i}^{i+1}$ and $v_{m_0+i+1}^{i+1}$, we can predict the positions $\mathbf{v}_{s_i}^{i+1}$ and $\mathbf{v}_{m_0+i+1}^{i+1}$ from $\mathbf{v}_{s_i}^i$, and use delta-encoding to reduce storage. Scalar attributes of corners in M^{i+1} can similarly be predicted from those in M^i. Finally, the material identifiers $d_{\{v_s, v_t, v_l\}}$ and $d_{\{v_t, v_s, v_r\}}$ of the new faces in mesh M^{i+1} can often be predicted from those of adjacent faces in M^i using only a few control bits.

As a result, the size of a carefully designed PM representation should be competitive with that obtained from methods for compressing uni-resolution meshes. Our current prototype implementation was not designed with this goal in mind. However, we analyze the compression of the connectivity K, and report results on the compression of the geometry V. In the following analysis, we assume for simplicity that $m_0 = 0$ since typically $m_0 \ll n$.

A common representation for the mesh connectivity K is to list the three vertex indices for each face. Since the number of vertices is n and the number of faces approximately $2n$, such a list requires $6\lceil \log_2(n) \rceil n$ bits of storage. Using a buffer of 2 vertices, *generalized triangle strip* representations reduce this number to about

[2] On average, v_{s_i} has 6 neighbors, and the number of permutations $P_2^6 = 30$ can be encoded in $\lceil \log_2(P_2^6) \rceil = 5$ bits.

$(\lceil \log_2(n) \rceil + 2k)n$ bits, where vertices are back-referenced once on average and $k \simeq 2$ bits capture the vertex replacement codes [6]. By increasing the vertex buffer size to 16, Deering's *generalized triangle mesh* representation [6] further reduces storage to about $(\frac{1}{8}\lceil \log_2(n) \rceil + 8)n$ bits. Turan [16] shows that planar graphs (and hence the connectivity of closed genus 0 meshes) can be encoded in $12n$ bits. Recent work by Taubin and Rossignac [15] addresses more general meshes. With the PM representation, each $vsplit_i$ requires specification of s_i and its two neighbors, for a total storage of about $(\lceil \log_2(n) \rceil + 5)n$ bits. Although not as concise as [6, 15], this is comparable to generalized triangle strips.

A traditional representation of the mesh geometry V requires storage of $3n$ coordinates, or $96n$ bits with IEEE single-precision floating point. Like Deering [6], we assume that these coordinates can be quantized to 16-bit fixed precision values without significant loss of visual quality, thus reducing storage to $48n$ bits. Deering is able to further compress this storage by delta-encoding the quantized coordinates and Huffman compressing the variable-length deltas. For 16-bit quantization, he reports storage of $35.8n$ bits, which includes both the deltas and the Huffman codes. Using a similar approach with the PM representation, we encode V in $31n$ to $50n$ bits as shown in Table 1. To obtain these results, we exploit a property of our optimization algorithm (Section 4.3): when considering the collapse of an edge $\{v_s, v_t\}$, it considers three starting points for the resulting vertex position $\mathbf{v}_n$: $\{\mathbf{v}_s, \mathbf{v}_t, \frac{\mathbf{v}_s+\mathbf{v}_t}{2}\}$. Depending on the starting point chosen, we delta-encode either $\{\mathbf{v}_s - \mathbf{v}_n, \mathbf{v}_t - \mathbf{v}_n\}$ or $\{\frac{\mathbf{v}_s+\mathbf{v}_t}{2} - \mathbf{v}_n, \frac{\mathbf{v}_t-\mathbf{v}_s}{2}\}$, and use separate Huffman tables for all four quantities.

To further improve compression, we could alter the construction algorithm to forego optimization and let $\mathbf{v}_n \in \{\mathbf{v}_s, \mathbf{v}_t, \frac{\mathbf{v}_s+\mathbf{v}_t}{2}\}$. This would degrade the accuracy of the approximating meshes somewhat, but allows encoding of V in $30n$ to $37n$ bits in our examples. Arithmetic coding [19] of delta lengths does not improve results significantly, reflecting the fact that the Huffman trees are well balanced. Further compression improvements may be achievable by adapting both the quantization level and the delta length models as functions of the *vsplit* record index i, since the magnitude of successive changes tends to decrease.

3.5 Selective refinement

The PM representation also supports selective refinement, whereby detail is added to the model only in desired areas. Let the application supply a callback function REFINE(v) that returns a Boolean value indicating whether the neighborhood of the mesh about v should be further refined. An initial mesh M^c is selectively refined by iterating through the list $\{vsplit_c, \ldots, vsplit_{n-1}\}$ as before, but only performing $vsplit_i(s_i, l_i, r_i, A_i)$ if

(1) all three vertices $\{v_{s_i}, v_{l_i}, v_{r_i}\}$ are present in the mesh, and

(2) REFINE(v_{s_i}) evaluates to TRUE.

(A vertex v_j is absent from the mesh if the prior vertex split that would have introduced it, $vsplit_{j-m_0-1}$, was not performed due to the above conditions.)

As an example, to obtain selective refinement of the model within a view frustum, REFINE(v) is defined to be TRUE if either v or any of its neighbors lies within the frustum. As seen in Figure 7a, condition (1) described above is suboptimal. The problem is that a vertex v_{s_i} within the frustum may fail to be split because its expected neighbor v_{l_i} lies just outside the frustum and was not previously created. The problem is remedied by using a less stringent version of condition (1). Let us define the *closest living ancestor* of a vertex v_j to be the vertex with index

$$A'(j) = \begin{cases} j & , \text{ if } v_j \text{ exists in the mesh} \\ A'(s_{j-m_0-1}) & , \text{ otherwise} \end{cases}$$

The new condition becomes:

(1') v_{s_i} is present in the mesh (i.e. $A'(s_i) = s_i$) and the vertices $v_{A'(l_i)}$ and $v_{A'(r_i)}$ are both adjacent to v_{s_i}.

As when constructing the geomorphs, the ancestor information A' is carried efficiently as the *vsplit* records are parsed. If conditions (1') and (2) are satisfied, $vsplit(s_i, A'(l_i), A'(r_i), A_i)$ is applied to the mesh. A mesh selectively refined with this new strategy is shown in Figure 7b. This same strategy was also used for Figure 10. Note that it is still possible to create geomorphs between M^c and selectively refined meshes thus created.

An interesting application of selective refinement is the transmission of view-dependent models over low-bandwidth communication lines. As the receiver's view changes over time, the sending process need only transmit those *vsplit* records for which REFINE evaluates to TRUE, and of those only the ones not previously transmitted.

4 PROGRESSIVE MESH CONSTRUCTION

The PM representation of an arbitrary mesh $\hat{M}$ requires a sequence of edge collapses transforming $\hat{M} = M^n$ into a base mesh M^0. The quality of the intermediate approximations $M^i, i < n$ depends largely on the algorithm for selecting which edges to collapse and what attributes to assign to the affected neighborhoods, for instance the positions $\mathbf{v}^i_{s_i}$.

There are many possible PM construction algorithms with varying trade-offs of speed and accuracy. At one extreme, a crude and fast scheme for selecting edge collapses is to choose them completely at random. (Some local conditions must be satisfied for an edge collapse to be legal, i.e. manifold preserving [9].) More sophisticated schemes can use heuristics to improve the edge selection strategy, for example the "distance to plane" metric of Schroeder et al. [14]. At the other extreme, one can attempt to find approximating meshes that are optimal with respect to some appearance metric, for instance the E_{dist} geometric metric of Hoppe et al. [9].

Since PM construction is a preprocess that can be performed offline, we chose to design a simplification procedure that invests some time in the selection of edge collapses. Our procedure is similar to the mesh optimization method introduced by Hoppe et al. [9], which is outlined briefly in Section 4.1. Section 4.2 presents an overview of our procedure, and Sections 4.3–4.6 present the details of our optimization scheme for preserving both the shape of the mesh and the scalar and discrete attributes which define its appearance.

4.1 Background: mesh optimization

The goal of mesh optimization [9] is to find a mesh $M = (K, V)$ that both accurately fits a set X of points $\mathbf{x}_i \in \mathbf{R}^3$ and has a small number of vertices. This problem is cast as minimization of an energy function

$$E(M) = E_{dist}(M) + E_{rep}(M) + E_{spring}(M) \ .$$

The first two terms correspond to the two goals of accuracy and conciseness: the *distance energy* term

$$E_{dist}(M) = \sum_i d^2(\mathbf{x}_i, \phi_V(|K|))$$

measures the total squared distance of the points from the mesh, and the *representation energy* term $E_{rep}(M) = c_{rep}m$ penalizes the number m of vertices in M. The third term, the *spring energy* $E_{spring}(M)$ is introduced to regularize the optimization problem. It corresponds to placing on each edge of the mesh a spring of rest length zero and tension κ:

$$E_{spring}(M) = \sum_{\{j,k\} \in K} \kappa \|\mathbf{v}_j - \mathbf{v}_k\|^2 \ .$$

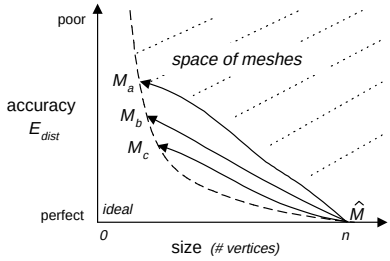


Figure 3: Illustration of the paths taken by mesh optimization using three different settings of c_{rep}.

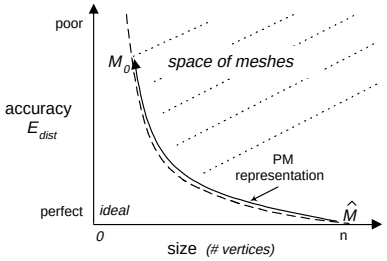


Figure 4: Illustration of the path taken by the new mesh simplification procedure in a graph plotting accuracy vs. mesh size.

The energy function $E(M)$ is minimized using a nested optimization method:

- *Outer loop*: The algorithm optimizes over K, the connectivity of the mesh, by randomly attempting a set of three possible mesh transformations: edge collapse, edge split, and edge swap. This set of transformations is complete, in the sense that any simplicial complex K of the same topological type as $\hat{K}$ can be reached through a sequence of these transformations. For each candidate mesh transformation, $K \rightarrow K'$, the continuous optimization described below computes $E_{K'}$, the minimum of E subject to the new connectivity K'. If $\Delta E = E_{K'} - E_K$ is found to be negative, the mesh transformation is applied (akin to a zero-temperature simulated annealing method).
- *Inner loop*: For each candidate mesh transformation, the algorithm computes $E_{K'} = \min_V E_{dist}(V) + E_{spring}(V)$ by optimizing over the vertex positions V. For the sake of efficiency, the algorithm in fact optimizes only one vertex position $\mathbf{v}_s$, and considers only the subset of points X that project onto the neighborhood affected by $K \rightarrow K'$. To avoid surface self-intersections, the edge collapse is disallowed if the maximum dihedral angle of edges in the resulting neighborhood exceeds some threshold.

Hoppe et al. [9] find that the regularizing spring energy term $E_{spring}(M)$ is most important in the early stages of the optimization, and achieve best results by repeatedly invoking the nested optimization method described above with a schedule of decreasing spring constants κ.

Mesh optimization is demonstrated to be an effective tool for mesh simplification. Given an initial mesh $\hat{M}$ to approximate, a dense set of points X is sampled both at the vertices of $\hat{M}$ and randomly over its faces. The optimization algorithm is then invoked with $\hat{M}$ as the starting mesh. Varying the setting of the representation constant c_{rep} results in optimized meshes with different trade-offs of accuracy and size. The paths taken by these optimizations are shown illustratively in Figure 3.

4.2 Overview of the simplification algorithm

As in mesh optimization [9], we also define an explicit energy metric $E(M)$ to measure the accuracy of simplified meshes $M = (K, V, D, S)$ with respect to the original $\hat{M}$, and we also modify the mesh M starting from $\hat{M}$ while minimizing $E(M)$.

Our energy metric has the following form:

$$E(M) = E_{dist}(M) + E_{spring}(M) + E_{scalar}(M) + E_{disc}(M) \ .$$

The first two terms, $E_{dist}(M)$ and $E_{spring}(M)$ are identical to those in [9]. The next two terms of $E(M)$ are added to preserve attributes associated with M: $E_{scalar}(M)$ measures the accuracy of its scalar attributes (Section 4.4), and $E_{disc}(M)$ measures the geometric accuracy of its discontinuity curves (Section 4.5). (To achieve scale invariance of the terms, the mesh is uniformly scaled to fit in a unit cube.)

Our scheme for optimizing over the connectivity K of the mesh is rather different from [9]. We have discovered that a mesh can be effectively simplified using edge collapse transformations alone. The edge swap and edge split transformations, useful in the context of surface reconstruction (which motivated [9]), are not essential for simplification. Although in principle our simplification algorithm can no longer traverse the entire space of meshes considered by mesh optimization, we find that the meshes generated by our algorithm are just as good. In fact, because of the priority queue approach described below, our meshes are usually better. Moreover, considering only edge collapses simplifies the implementation, improves performance, and most importantly, gives rise to the PM representation (Section 3).

Rather than randomly attempting mesh transformations as in [9], we place all (legal) candidate edge collapse transformations into a priority queue, where the priority of each transformation is its estimated energy cost ΔE. In each iteration, we perform the transformation at the front of the priority queue (with lowest ΔE), and recompute the priorities of edges in the neighborhood of this transformation. As a consequence, we eliminate the need for the awkward parameter c_{rep} as well as the energy term $E_{rep}(M)$. Instead, we can explicitly specify the number of faces desired in an optimized mesh. Also, a single run of the optimization can generate several such meshes. Indeed, it generates a continuous-resolution family of meshes, namely the PM representation of $\hat{M}$ (Figure 4).

For each edge collapse $K \rightarrow K'$, we compute its cost $\Delta E = E_{K'} - E_K$ by solving a continuous optimization

$$E_{K'} = \min_{V,S} E_{dist}(V) + E_{spring}(V) + E_{scalar}(V, S) + E_{disc}(V)$$

over both the vertex positions V and the scalar attributes S of the mesh with connectivity K'. This minimization is discussed in the next three sections.

4.3 Preserving surface geometry ($E_{dist} + E_{spring}$)

As in [9], we "record" the geometry of the original mesh $\hat{M}$ by sampling from it a set of points X. At a minimum, we sample a point at each vertex of $\hat{M}$. If requested by the user, additional points are sampled randomly over the surface of $\hat{M}$. The energy terms $E_{dist}(M)$ and $E_{spring}(M)$ are defined as in Section 4.1.

For a mesh of fixed connectivity, our method for optimizing the vertex positions to minimize $E_{dist}(V) + E_{spring}(V)$ closely follows that of [9]. Evaluating $E_{dist}(V)$ involves computing the distance of each point $\mathbf{x}_i$ to the mesh. Each of these distances is itself a minimization problem

$$d^2(\mathbf{x}_i, \phi_V(|K|)) = \min_{\mathbf{b}_i \in |K|} \|\mathbf{x}_i - \phi_V(\mathbf{b}_i)\|^2 \qquad (1)$$

where the unknown $\mathbf{b}_i$ is the parametrization of the projection of $\mathbf{x}_i$ on the mesh. The nonlinear minimization of $E_{dist}(V) + E_{spring}(V)$ is performed using an iterative procedure alternating between two steps:

1. For fixed vertex positions V, compute the optimal parametrizations $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_{|X|}\}$ by projecting the points X onto the mesh.
2. For fixed parametrizations B, compute the optimal vertex positions V by solving a sparse linear least-squares problem.

As in [9], when considering $ecol(\{v_s, v_t\})$, we optimize only one vertex position, $\mathbf{v}_s^i$. We perform three different optimizations with different starting points, $\mathbf{v}_s^i = (1-\alpha)\mathbf{v}_s^{i+1} + (\alpha)\mathbf{v}_t^{i+1}$ for $\alpha = \{0, \frac{1}{2}, 1\}$, and accept the best one.

Instead of defining a global spring constant κ for E_{spring} as in [9], we adapt κ each time an edge collapse transformation is considered. Intuitively, the spring energy is most important when few points project onto a neighborhood of faces, since in this case finding the vertex positions minimizing $E_{dist}(V)$ may be an under-constrained problem. Thus, for each edge collapse transformation considered, we set κ as a function of the ratio of the number of points to the number of faces in the neighborhood.[3] With this adaptive scheme, the influence of $E_{spring}(M)$ decreases gradually and adaptively as the mesh is simplified, and we no longer require the expensive schedule of decreasing spring constants.

4.4 Preserving scalar attributes (E_{scalar})

As described in Section 2, we represent piecewise continuous scalar fields by defining scalar attributes S at the mesh corners. We now present our scheme for preserving these scalar fields through the simplification process. For exposition, we find it easier to first present the case of continuous scalar fields, in which the corner attributes at a vertex are identical. The generalization to piecewise continuous fields is discussed shortly.

Optimizing scalar attributes at vertices Let the original mesh $\hat{M}$ have at each vertex v_j not only a position $\mathbf{v}_j \in \mathbf{R}^3$ but also a scalar attribute $\underline{\mathbf{v}}_j \in \mathbf{R}^d$. To capture scalar attributes, we sample at each point $\mathbf{x}_i \in X$ the attribute value $\underline{\mathbf{x}}_i \in \mathbf{R}^d$. We would then like to generalize the distance metric E_{dist} to also measure the deviation of the sampled attribute values $\underline{X}$ from those of M.

One natural way to achieve this is to redefine the distance metric to measure distance in $\mathbf{R}^{3+d}$:

$$d^2((\mathbf{x}_i \;\; \underline{\mathbf{x}}_i), M(K, V, \underline{V})) = \min_{\mathbf{b}_i \in |K|} \left\| (\mathbf{x}_i \;\; \underline{\mathbf{x}}_i) - (\phi_V(\mathbf{b}_i) \;\; \phi_{\underline{V}}(\mathbf{b}_i)) \right\|^2 .$$

This new distance functional could be minimized using the iterative approach of Section 4.3. However, it would be expensive since finding the optimal parametrization $\mathbf{b}_i$ of each point $\mathbf{x}_i$ would require projection in $\mathbf{R}^{3+d}$, and would be non-intuitive since these parametrizations would not be geometrically based.

Instead we opted to determine the parametrizations $\mathbf{b}_i$ using only geometry with equation (1), and to introduce a separate energy term E_{scalar} to measure attribute deviation based on these parametrizations:

$$E_{scalar}(\underline{V}) = (c_{scalar})^2 \sum_i \left\| \underline{\mathbf{x}}_i - \phi_{\underline{V}}(\mathbf{b}_i) \right\|^2$$

where the constant c_{scalar} assigns a relative weight between the scalar attribute errors (E_{scalar}) and the geometric errors (E_{dist}).

Thus, to minimize $E(V, \underline{V}) = E_{dist}(V) + E_{spring}(V) + E_{scalar}(\underline{V})$, our algorithm first finds the vertex position $\mathbf{v}_s$ minimizing $E_{dist}(V) + E_{spring}(V)$ by alternately projecting the points onto the mesh (obtaining the parametrizations $\mathbf{b}_i$) and solving a linear least-squares problem (Section 4.1). Then, using those same parametrizations $\mathbf{b}_i$, it finds the vertex attribute $\underline{\mathbf{v}}_s$ minimizing E_{scalar} by solving a single linear least-squares problem. Hence introducing E_{scalar} into the optimization causes negligible performance overhead.

Since ΔE_{scalar} contributes to the estimated cost ΔE of an edge collapse, we obtain simplified meshes whose faces naturally adapt to the attribute fields, as shown in Figures 8 and 11.

Optimizing scalar attributes at corners Our scheme for optimizing the scalar corner attributes S is a straightforward generalization of the scheme just described. Instead of solving for a single unknown attribute value $\underline{\mathbf{v}}_s$, the algorithm partitions the corners around v_s into continuous sets (based on equivalence of corner attributes) and for each continuous set solves independently for its optimal attribute value.

Range constraints Some scalar attributes have constrained ranges. For instance, the components (r, g, b) of color are typically constrained to lie between 0 and 1. Least-squares optimization may yield color values outside this range. In these cases we clip the optimized values to the given range. For least-squares minimization of a Euclidean norm at a single vertex, this is in fact optimal.

Normals Surface normals (n_x, n_y, n_z) are typically constrained to have unit length, and thus their domain is non-Cartesian. Optimizing over normals would therefore require minimization of a nonlinear functional with nonlinear constraints. We decided to instead simply carry the normals through the simplification process. Specifically, we compute the new normals at vertex $v_{s_i}^i$ by interpolating between the normals at vertices $v_{s_i}^{i+1}$ and $v_{m_0+i+1}^{i+1}$ using the α value that resulted in the best vertex position $\mathbf{v}_{s_i}^i$ in Section 4.3. Fortunately, the absolute directions of normals are less visually important than their discontinuities, and we have a scheme for preserving such discontinuities, as described in the next section.

4.5 Preserving discontinuity curves (E_{disc})

Appearance attributes give rise to a set of discontinuity curves on the mesh, both from differences between discrete face attributes (e.g. material boundaries), and from differences between scalar corner attributes (e.g. creases and shadow boundaries). As these discontinuity curves form noticeable features, we have found it useful to preserve them both topologically and geometrically.

We can detect when a candidate edge collapse would modify the topology of the discontinuity curves using some simple tests on the presence of sharp edges in its neighborhood. Let $sharp(v_j, v_k)$ denote that an edge $\{v_j, v_k\}$ is sharp, and let $\#sharp(v_j)$ be the number of sharp edges adjacent to a vertex v_j. Then, referring to Figure 1, $ecol(\{v_s, v_t\})$ modifies the topology of discontinuity curves if either:

- $sharp(v_s, v_l)$ and $sharp(v_t, v_l)$, or
- $sharp(v_s, v_r)$ and $sharp(v_t, v_r)$, or
- $\#sharp(v_s) \geq 1$ and $\#sharp(v_t) \geq 1$ and not $sharp(v_s, v_t)$, or
- $\#sharp(v_s) \geq 3$ and $\#sharp(v_t) \geq 3$ and $sharp(v_s, v_t)$, or
- $sharp(v_s, v_t)$ and $\#sharp(v_s) = 1$ and $\#sharp(v_t) \neq 2$, or
- $sharp(v_s, v_t)$ and $\#sharp(v_t) = 1$ and $\#sharp(v_s) \neq 2$.

If an edge collapse would modify the topology of discontinuity curves, we either disallow it, or penalize it as discussed in Section 4.6.

To preserve the geometry of the discontinuity curves, we sample an additional set of points X_{disc} from the sharp edges of $\hat{M}$, and define an additional energy term E_{disc} equal to the total squared distances of each of these points to the discontinuity curve from which it was sampled. Thus E_{disc} is defined just like E_{dist}, except that the points X_{disc} are constrained to project onto a set of sharp edges in the mesh. In effect, we are solving a curve fitting problem embedded within the surface fitting problem. Since all boundaries of the surface are defined to be discontinuity curves, our procedure preserves bound-

[3] The neighborhood of an edge collapse transformation is the set of faces shown in Figure 1. Using C notation, we set $\kappa = r < 4 \;?\; 10^{-2} : r < 8 \;?\; 10^{-4} : 10^{-8}$ where r is the ratio of the number of points to faces in the neighborhood.

ary geometry more accurately than [9]. Figure 9 demonstrates the importance of using the E_{disc} energy term in preserving the material boundaries of a mesh with discrete face attributes.

4.6 Permitting changes to topology of discontinuity curves

Some meshes contain numerous discontinuity curves, and these curves may delimit features that are too small to be visible when viewed from a distance. In such cases we have found that strictly preserving the topology of the discontinuity curves unnecessarily curtails simplification. We have therefore adopted a hybrid strategy, which is to permit changes to the topology of the discontinuity curves, but to penalize such changes. When a candidate edge collapse $ecol(\{v_s, v_t\})$ changes the topology of the discontinuity curves, we add to its cost ΔE the value $|X_{disc,\{v_s,v_t\}}| \cdot \|\mathbf{v}_s - \mathbf{v}_t\|^2$ where $|X_{disc,\{v_s,v_t\}}|$ is the number of points of X_{disc} projecting onto $\{v_s, v_t\}$. That simple strategy, although ad hoc, has proven very effective. For example, it allows the dark gray window frames of the "cessna" (visible in Figure 9) to vanish in the simplified meshes (Figures 5a–c).

Table 1: Parameter settings and quantitative results.

Object	Original $\hat{M}$		Base M^0		User param.		$\lvert X_{disc}\rvert$	V	Time
	$m_0 + n$	#faces	m_0	#faces	$\lvert X\rvert-(m_0+n)$	c_{color}		$\frac{\text{bits}}{n}$	mins
cessna	6,795	13,546	97	150	100,000	-	46,811	46	23
terrain	33,847	66,960	3	1	0	-	3,796	46	16
mandrill	40,000	79,202	3	1	0	0.1	4,776	31	19
radiosity	78,923	150,983	1,192	1,191	200,000	0.01	74,316	37	106
fandisk	6,475	12,946	27	50	10,000	-	5,924	50	19

5 RESULTS

Table 1 shows, for the meshes in Figures 5–12, the number of vertices and faces in both $\hat{M}$ and M^0. In general, we let the simplification proceed until no more legal edge collapse transformations are possible. For the "cessna", we stopped at 150 faces to obtain a visually aesthetic base mesh. As indicated, the only user-specified parameters are the number of additional points (besides the $m_0 + n$ vertices of $\hat{M}$) sampled to increase fidelity, and the c_{scalar} constants relating the scalar attribute accuracies to the geometric accuracy. The only scalar attribute we optimized is color, and its c_{scalar} constant is denoted as c_{color}. The number $|X_{disc}|$ of points sampled from sharp edges is set automatically so that the densities of X and X_{disc} are proportional.[4] Execution times were obtained on a 150MHz Indigo2 with 128MB of memory.

Construction of the PM representation proceeds in three steps. First, as the simplification algorithm applies a sequence $ecol_{n-1} \ldots ecol_0$ of transformations to the original mesh, it writes to a file the sequence $vsplit_{n-1} \ldots vsplit_0$ of corresponding inverse transformations. When finished, the algorithm also writes the resulting base mesh M^0. Next, we reverse the order of the *vsplit* records. Finally, we renumber the vertices and faces of $(M^0, vsplit_0 \ldots vsplit_{n-1})$ to match the indexing scheme of Section 3.1 in order to obtain a concise format.

Figure 6 shows a single geomorph between two meshes M^{175} and M^{425} of a PM representation. For interactive LOD, it is useful to select a sequence of meshes from the PM representation, and to construct successive geomorphs between them. We have obtained good results by selecting meshes whose complexities grow exponentially, as in Figure 5. During execution, an application can adjust the granularity of these geomorphs by sampling additional meshes from the PM representation, or freeing some up.

In Figure 10, we selectively refined a terrain (grid of 181×187 vertices) using a new REFINE(v) function that keeps more detail near silhouette edges and near the viewer. More precisely, for the faces F_v adjacent to v, we compute the signed projected screen areas $\{a_f : f \in F_v\}$. We let REFINE(v) return TRUE if

(1) any face $f \in F_v$ lies within the view frustum, and either

(2a) the signs of a_f are not all equal (i.e. v lies near a silhouette edge) or

(2b) $\sum_{f \in F_v} a_f > thresh$ for a screen area threshold $thresh = 0.16^2$ (where total screen area is 1).

6 RELATED WORK

Mesh simplification methods A number of schemes construct a discrete sequence of approximating meshes by repeated application of a simplification procedure. Turk [17] sprinkles a set of points on a mesh, with density weighted by estimates of local curvature, and then retriangulates based on those points. Both Schroeder et al. [14] and Cohen et al. [4] iteratively remove vertices from the mesh and retriangulate the resulting holes. Cohen et al. are able to bound the maximum error of the approximation by restricting it to lie between two offset surfaces. Hoppe et al. [9] find accurate approximations through a general mesh optimization process (Section 4.1). Rossignac and Borrel [12] merge vertices of a model using spatial binning. A unique aspect of their approach is that the topological type of the model may change in the process. Their method is extremely fast, but since it ignores geometric qualities like curvature, the resulting approximations can be far from optimal. Some of the above methods [12, 17] permit the construction of geomorphs between successive simplified meshes.

Multiresolution analysis (MRA) Lounsbery et al. [10, 11] generalize the concept of multiresolution analysis to surfaces of arbitrary topological type. Eck et al. [7] describe how MRA can be applied to the approximation of an arbitrary mesh. Certain et al. [2] extend MRA to capture color, and present a multiresolution Web viewer supporting progressive transmission. MRA has many similarities with the PM scheme, since both store a simple base mesh together with a stream of detail records, and both produce a continuous-resolution representation. It is therefore worthwhile to highlight their differences:

Advantages of PM over MRA:

- MRA requires that the detail terms (wavelets) lie on a domain with subdivision connectivity, and as a result an arbitrary initial mesh $\hat{M}$ can only be recovered to within an ϵ tolerance. In contrast, the PM representation is lossless since $M^n = \hat{M}$.
- Because the approximating meshes $M^i, i < n$ in a PM may have arbitrary connectivity, they can be much better approximations than their MRA counterparts (Figure 12).
- The MRA representation cannot deal effectively with surface creases, unless those creases lie parametrically along edges of the base mesh (Figure 12). PM's can introduce surface creases anywhere and at any level of detail.
- PM's capture continuous, piecewise-continuous, and discrete appearance attributes. MRA schemes can represent discontinuous functions using a piecewise-constant basis (such as the Haar basis as used in [2, 13]), but the resulting approximations have too many discontinuities since none of the basis functions meet continuously. Also, it is not clear how MRA could be extended to capture discrete attributes.

[4] We set $|X_{disc}|$ such that $|X_{disc}|/perim = c(|X|/area)^{\frac{1}{2}}$ where $perim$ is the total length of all sharp edges in $\hat{M}$, $area$ is total area of all faces, and the constant $c = 4.0$ is chosen empirically.

Advantages of MRA over PM:

- The MRA framework provides a parametrization between meshes at various levels of detail, thus making possible multiresolution surface editing. PM's also offer such a parametrization, but it is not smooth, and therefore multiresolution editing may be non-intuitive.
- Eck et al. [7] construct MRA approximations with guaranteed maximum error bounds to $\hat{M}$. Our PM construction algorithm does not provide such bounds, although one could envision using simplification envelopes [4] to achieve this.
- MRA allows geometry and color to be compressed independently [2].

Other related work There has been relatively little work in simplifying arbitrary surfaces with functions defined over them. One special instance is image compression, since an image can be thought of as a set of scalar color functions defined on a quadrilateral surface. Another instance is the framework of Schröder and Sweldens [13] for simplifying functions defined over the sphere. The PM representation, like the MRA representation, is a generalization in that it supports surfaces of arbitrary topological type.

7 SUMMARY AND FUTURE WORK

We have introduced the progressive mesh representation and shown that it naturally supports geomorphs, progressive transmission, compression, and selective refinement. In addition, as a PM construction method, we have presented a new mesh simplification procedure designed to preserve not just the geometry of the original mesh, but also its overall appearance.

There are a number of avenues for future work, including:

- Development of an explicit metric and optimization scheme for preserving surface normals.
- Experimentation with PM editing.
- Representation of articulated or animated models.
- Application of the work to progressive subdivision surfaces.
- Progressive representation of more general simplicial complexes (not just 2-d manifolds).
- Addition of spatial data structures to permit efficient selective refinement.

We envision many practical applications for the PM representation, including streaming of 3D geometry over the Web, efficient storage formats, and continuous LOD in computer graphics applications. The representation may also have applications in finite element methods, as it can be used to generate coarse meshes for multigrid analysis.

ACKNOWLEDGMENTS

I wish to thank Viewpoint Datalabs for providing the "cessna" mesh, Pratt & Whitney for the gas turbine engine component ("fandisk"), Softimage for the "terrain" mesh, and especially Steve Drucker for creating several radiosity models using Lightscape. Thanks also to Michael Cohen, Steven "Shlomo" Gortler, and Jim Kajiya for their enthusiastic support, and to Rick Szeliski for helpful comments on the paper. Mark Kenworthy first coined the term "geomorph" in '92 to distinguish them from image morphs.

REFERENCES

[1] APPLE COMPUTER, INC. *3D graphics programming with QuickDraw 3D.* Addison Wesley, 1995.

[2] CERTAIN, A., POPOVIC, J., DUCHAMP, T., SALESIN, D., STUETZLE, W., AND DEROSE, T. Interactive multiresolution surface viewing. *Computer Graphics (SIGGRAPH '96 Proceedings)* (1996).

[3] CLARK, J. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM 19,* 10 (Oct. 1976), 547–554.

[4] COHEN, J., VARSHNEY, A., MANOCHA, D., TURK, G., WEBER, H., AGARWAL, P., BROOKS, F., AND WRIGHT, W. Simplification envelopes. *Computer Graphics (SIGGRAPH '96 Proceedings)* (1996).

[5] CURLESS, B., AND LEVOY, M. A volumetric method for building complex models from range images. *Computer Graphics (SIGGRAPH '96 Proceedings)* (1996).

[6] DEERING, M. Geometry compression. *Computer Graphics (SIGGRAPH '95 Proceedings)* (1995), 13–20.

[7] ECK, M., DEROSE, T., DUCHAMP, T., HOPPE, H., LOUNSBERY, M., AND STUETZLE, W. Multiresolution analysis of arbitrary meshes. *Computer Graphics (SIGGRAPH '95 Proceedings)* (1995), 173–182.

[8] FUNKHOUSER, T., AND SÉQUIN, C. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics (SIGGRAPH '93 Proceedings)* (1995), 247–254.

[9] HOPPE, H., DEROSE, T., DUCHAMP, T., MCDONALD, J., AND STUETZLE, W. Mesh optimization. *Computer Graphics (SIGGRAPH '93 Proceedings)* (1993), 19–26.

[10] LOUNSBERY, J. M. *Multiresolution analysis for surfaces of arbitrary topological type.* PhD thesis, Dept. of Computer Science and Engineering, U. of Washington, 1994.

[11] LOUNSBERY, M., DEROSE, T., AND WARREN, J. Multiresolution analysis for surfaces of arbitrary topological type. Submitted for publication. (TR 93-10-05b, Dept. of Computer Science and Engineering, U. of Washington, January 1994.).

[12] ROSSIGNAC, J., AND BORREL, P. Multi-resolution 3D approximations for rendering complex scenes. In *Modeling in Computer Graphics*, B. Falcidieno and T. L. Kunii, Eds. Springer-Verlag, 1993, pp. 455–465.

[13] SCHRÖDER, P., AND SWELDENS, W. Spherical wavelets: Efficiently representing functions on the sphere. *Computer Graphics (SIGGRAPH '95 Proceedings)* (1995), 161–172.

[14] SCHROEDER, W., ZARGE, J., AND LORENSEN, W. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH '92 Proceedings) 26,* 2 (1992), 65–70.

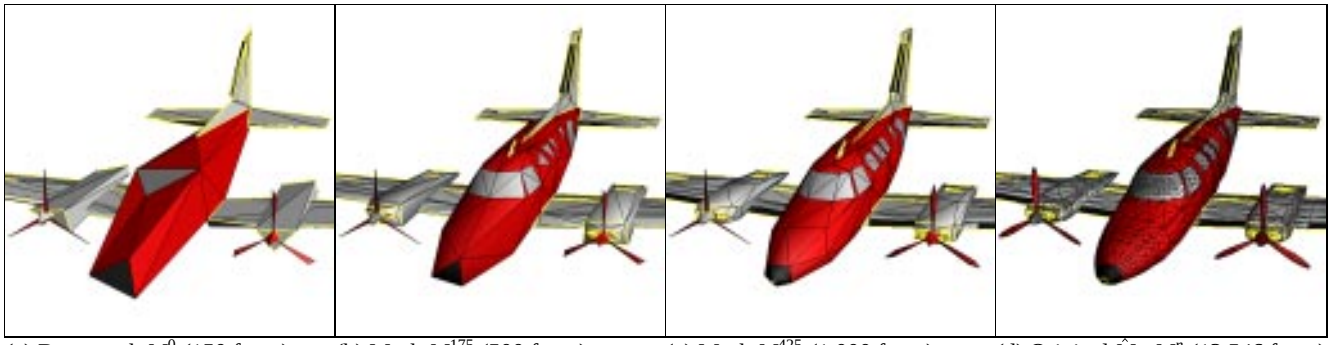
[15] TAUBIN, G., AND ROSSIGNAC, J. Geometry compression through topological surgery. Research Report RC-20340, IBM, January 1996.

[16] TURAN, G. Succinct representations of graphs. *Discrete Applied Mathematics 8* (1984), 289–294.

[17] TURK, G. Re-tiling polygonal surfaces. *Computer Graphics (SIGGRAPH '92 Proceedings) 26,* 2 (1992), 55–64.

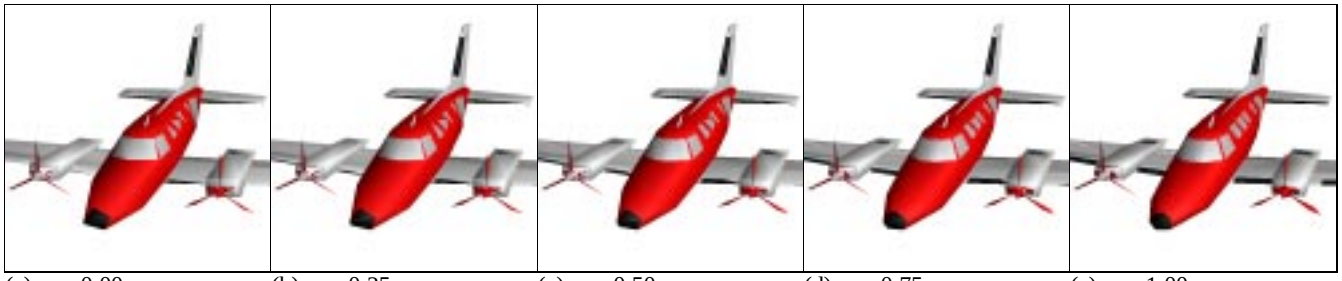
[18] UPSTILL, S. *The RenderMan Companion*. Addison-Wesley, 1990.

[19] WITTEN, I., NEAL, R., AND CLEARY, J. Arithmetic coding for data compression. *Communications of the ACM 30,* 6 (June 1987), 520–540.



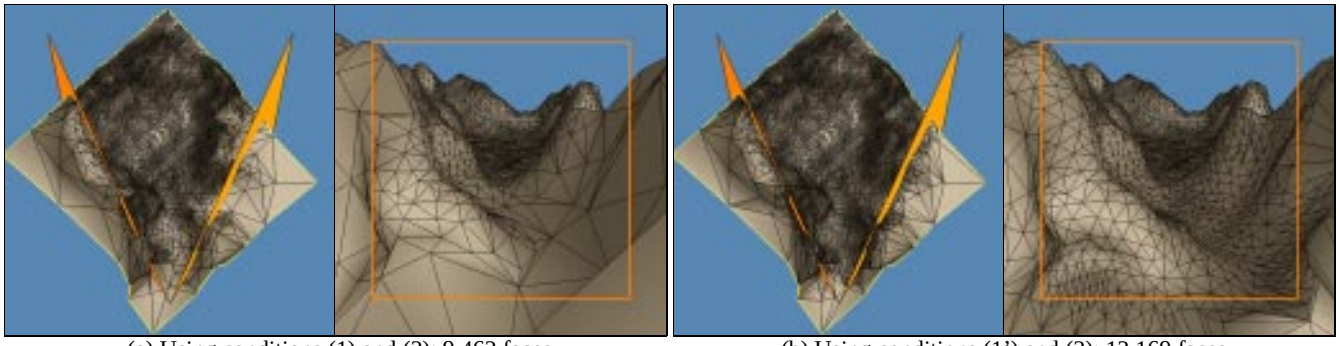
(a) Base mesh M^0 (150 faces) (b) Mesh M^{175} (500 faces) (c) Mesh M^{425} (1,000 faces) (d) Original $\hat{M} = M^n$ (13,546 faces)

Figure 5: The PM representation of an arbitrary mesh $\hat{M}$ captures a continuous-resolution family of approximating meshes $M^0 \ldots M^n = \hat{M}$.



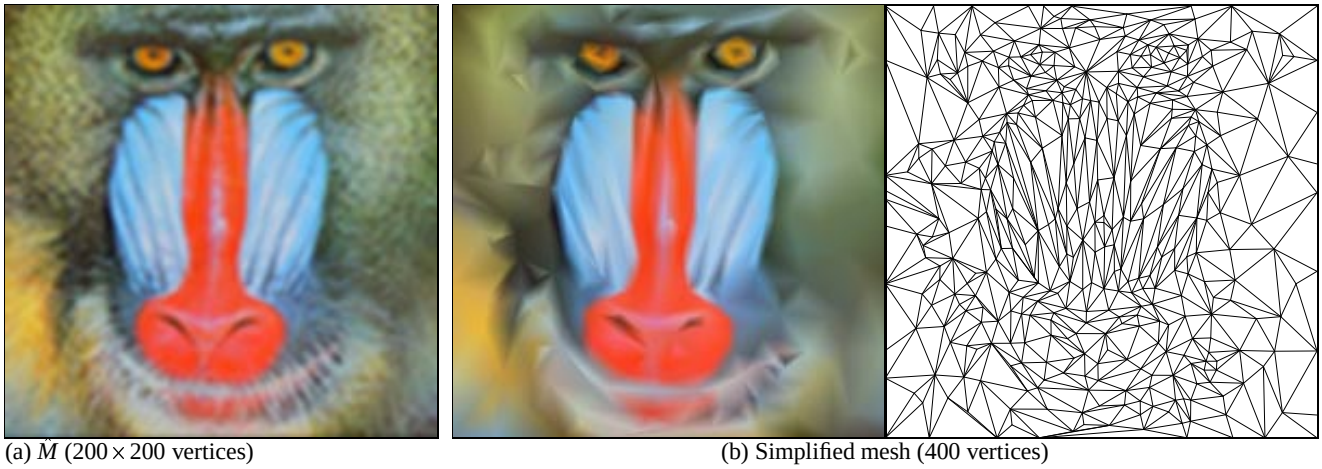
(a) $\alpha = 0.00$ (b) $\alpha = 0.25$ (c) $\alpha = 0.50$ (d) $\alpha = 0.75$ (e) $\alpha = 1.00$

Figure 6: Example of a geomorph $M^G(\alpha)$ defined between $M^G(0) \doteq M^{175}$ (with 500 faces) and $M^G(1) = M^{425}$ (with 1,000 faces).



(a) Using conditions (1) and (2); 9,462 faces (b) Using conditions (1') and (2); 12,169 faces

Figure 7: Example of selective refinement within the view frustum (indicated in orange).



(a) $\hat{M}$ (200×200 vertices) (b) Simplified mesh (400 vertices)

Figure 8: Demonstration of minimizing E_{scalar}: simplification of a mesh with trivial geometry (a square) but complex scalar attribute field. ($\hat{M}$ is a mesh with regular connectivity whose vertex colors correspond to the pixels of an image.)

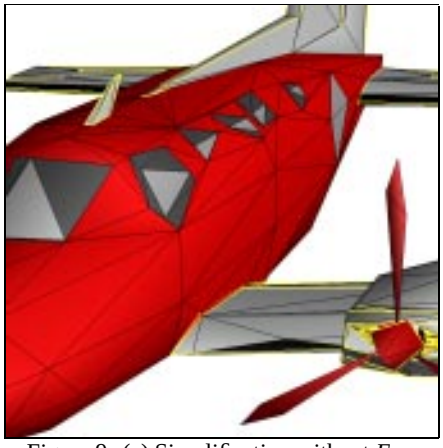


Figure 9: (a) Simplification without E_{disc}

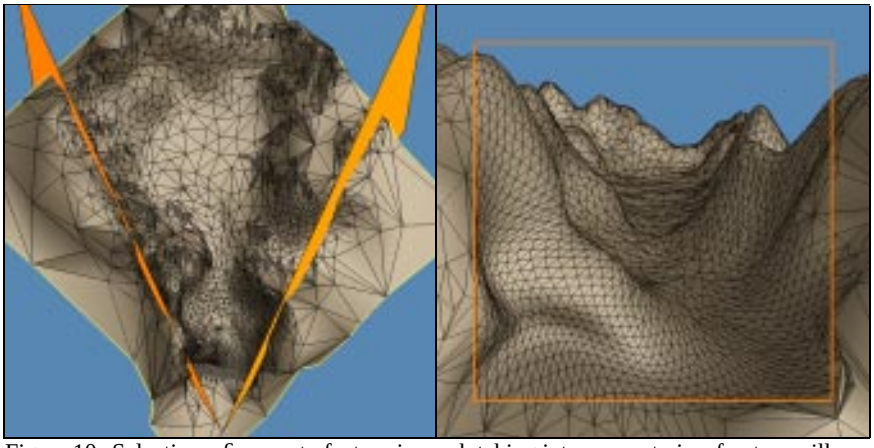
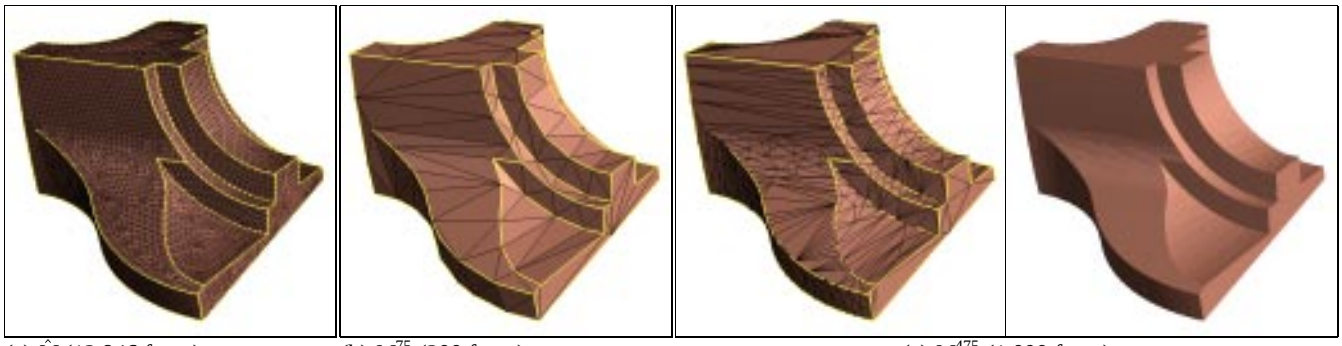


Figure 10: Selective refinement of a terrain mesh taking into account view frustum, silhouette regions, and projected screen size of faces (7,438 faces).

Figure 11: Simplification of a radiosity solution; left: original mesh (150,983 faces); right: simplified mesh (10,000 faces).



(a) $\hat{M}$ (12,946 faces) (b) M^{75} (200 faces) (c) M^{475} (1,000 faces)

(d) ϵ = 9.0 (192 faces) (e) ϵ = 2.75 (1,070 faces) (f) ϵ = 0.1 (15,842 faces)

Figure 12: Approximations of a mesh $\hat{M}$ using (b–c) the PM representation, and (d–f) the MRA scheme of Eck et al. [7]. As demonstrated, MRA cannot recover $\hat{M}$ exactly, cannot deal effectively with surface creases, and produces approximating meshes of inferior quality.