

SCALABLE GRAPHICS ARCHITECTURES:
INTERFACE & TEXTURE

A DISSERTATION
SUBMITTED TO THE COMPUTER SCIENCE DEPARTMENT
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Homan Igehy
May 2000

© Copyright by Homan Igehy 2000
All Rights Reserved

I certify that I have read this dissertation and that in my opinion
it is fully adequate, in scope and quality, as a dissertation for
the degree of Doctor of Philosophy.

Pat Hanrahan (Principal Advisor)

I certify that I have read this dissertation and that in my opinion
it is fully adequate, in scope and quality, as a dissertation for
the degree of Doctor of Philosophy.

Mark Horowitz

I certify that I have read this dissertation and that in my opinion
it is fully adequate, in scope and quality, as a dissertation for
the degree of Doctor of Philosophy.

Bill Dally

Approved for the University Committee on Graduate Studies:

Abstract

With today's technology, it is possible to place a significant amount graphics processing power on a single chip. While this allows computers to render very impressive imagery, many interactive graphics applications require several orders of magnitude more in processing power. Parallelism is one way of achieving increased power, and scalable solutions achieve parallelism through the replication of a basic unit. In this dissertation, we discuss scalable graphics architectures and present novel techniques for scaling two important aspects of graphics architectures that have not been addressed by the literature: interface and texture.

First, we analyze parallelism in the graphics pipeline. By looking at the virtual machine defined by the graphics API and analyzing its dependencies, we are able to examine the sources of parallelism. We define the metrics of scalability and analyze the extent to which existing graphics architectures are able to scale. Second, we present a novel parallel graphics interface that allows for scalable input rates. This interface allows multiple graphics contexts to simultaneously submit commands to the graphics system while explicitly ordering the drawing of graphics primitives. Even with scenes that require a total order, fully scalable submission and rendering are demonstrated. Finally, we present a scalable texture architecture based on a shared texture memory. In order to tolerate the high and variable latencies of a shared texture memory, a novel texture prefetching architecture is described. The effects of parallel texture caching are examined in detail, demonstrating the applicability of such an approach across a wide variety of rasterization architectures.

Acknowledgements

I would like to thank all the people who helped me through my years at Stanford. Because I did my undergraduate education here as well as my graduate education, I have spent 9 years out of my 27 years of life here: one-third of my life. You'll understand if I forget someone....

First and foremost, I would like to thank my advisor, Pat Hanrahan. Stylistically, he was very much a hands-off advisor that let me pursue my interests. Sometimes this would mean video games and guitar, and sometimes this would mean fruitful research. The research in this dissertation, as well as research beyond the scope of this dissertation, was guided by a person who had the knowledge to push me in the right direction and the wisdom to let me find my way from there. I would also like to thank Bill Dally and Mark Horowitz, the other members of my reading committee. Mark's comments over the years have always been to the point and have helped me gain a greater technical understanding of the work in this thesis. Bill's viewpoints called into question many of my assumptions and have helped me gain a better understanding of graphics architectures from a more general perspective.

My colleagues in the Stanford graphics lab have been a source of fun and learning. The course project of implementing a graphics pipeline in a class by Marc Levoy sparked my interest in graphics architecture, and I still recall trying to keep up with the torrent pace of lecturing by Leo Guibas in a couple of classes. I also owe a great debt to the other students who helped me with my research over the years, particularly Matthew Eldridge, Gordon Stoll, Kekoa Proudfoot, John Owens, Matt Pharr, Milton Chen, Ian Buck,

James Davis, Bill Mark, Maneesh Agrawala, Timothy Purcell, Lucas Pereira, Greg Humphreys, Afra Zomorodian, and Phil Lacroute.

I would like to thank my parents, Jeff and Golie, for their love and support over the years. I would like to thank my brother, Alex, and his fiancée, Dana, for looking out for me over the years. I thank my friends, who have made me laugh through good times and bad times, and I thank Sepi, who puts a smile on my face every day.

Finally, I would like to thank DARPA, under contract DABT63-95-C-0085-P00006, and Intel Corporation for the financial support of my research.

Contents

Abstract	v
Acknowledgements	vii
Chapter 1 Introduction	1
1.1 Trends in Graphics Architecture	2
1.2 The Rendering Problem	5
1.2.1 Representations and Algorithms	5
1.2.2 Interactive Rendering	7
1.3 The Graphics Pipeline	8
1.4 Summary of Original Contributions	14
Chapter 2 Analysis of Parallel Graphics	17
2.1 Sources of Parallelism	18
2.1.1 Instruction Set	19
2.1.2 Data Dependencies	21
2.1.2.1 Dependencies and Parallelism	21
2.1.2.2 Data Dependencies in Graphics Architectures	23
2.1.3 Control Dependencies	26
2.1.4 Discussion	27
2.2 Scaling the Graphics Pipeline	29

2.2.1 Scalability Metrics	29
2.2.2 Analysis of Scalable Architectures	32
2.2.2.1 Sort-First Architectures	35
2.2.2.2 Sort-Middle Architectures	38
2.2.2.3 Fragment-sorting Architectures	42
2.2.2.4 Image-Composition Architectures	43
2.2.2.5 Pomegranate Architecture	45
2.3 Conclusion	48
Chapter 3 Scalable Graphics Interface	49
3.1 Introduction	50
3.2 Motivation	51
3.3 Related Work	53
3.4 The Parallel API Extensions	55
3.4.1 Existing Constructs	56
3.4.2 The Wait Construct	57
3.4.3 Synchronization Constructs	58
3.5 Using the Parallel Graphics API	59
3.5.1 Simple Interactive Loop	60
3.5.2 Marching Cubes	61
3.6 Implementations	63
3.6.1 Argus: A Software Implementation	63
3.6.1.1 Architecture	63
3.6.1.2 Performance	67
3.6.2 Pomegranate: A Hardware Implementation	73
3.6.2.1 Architecture	73
3.6.2.2 Performance	76
3.6.3 WireGL: A Transparent Implementation	77
3.7 Implementation Alternatives	79

3.7.1 Consistency and Synchronization	79
3.7.2 Architectural Requirements	80
3.8 Conclusion	82
Chapter 4 Scalable Texture Mapping	85
4.1 Prefetching in a Texture Cache	86
4.1.1 Introduction	87
4.1.2 Mip Mapping	88
4.1.3 Caching and Prefetching	89
4.1.3.1 Traditional Prefetching	90
4.1.3.2 A Texture Prefetching Architecture	91
4.1.4 Robust Scene Analysis	94
4.1.4.1 Texture Locality	94
4.1.4.2 The Benchmark Scenes	96
4.1.5 Memory Organization	98
4.1.5.1 Cache Efficiency	100
4.1.5.2 Bandwidth Requirements	101
4.1.5.3 Memory Models	105
4.1.6 Performance Analysis	106
4.1.6.1 Intra-Frame Variability	108
4.1.6.2 Buffer Sizes	109
4.2 Parallel Texture Caching	112
4.2.1 Previous Work	113
4.2.2 Parallel Texture Caching Architectures	114
4.2.3 Methodology	117
4.2.3.1 Parallel Rasterization Algorithms	117
4.2.3.2 Scenes	119
4.2.3.3 Simulation Environment	119
4.2.3.3.1 Data Organization	120

4.2.3.3.2 Performance Model	121
4.2.4 Results	122
4.2.4.1 Locality.....	123
4.2.4.2 Working Sets.....	127
4.2.4.3 Load Imbalance.....	129
4.2.4.4 Performance	132
4.2.5 Texture Updates	136
4.3 Conclusion	137
Chapter 5 Conclusion	139
Bibliography	143

List of Tables

Table 4.1: The Benchmark Scenes	96
Table 4.2: Memory Models	105
Table 4.3: Buffer Sizes	110

List of Figures

Figure 1.1: The Graphics Pipeline	10
Figure 2.1: The Sorting Classification.....	33
Figure 2.2: Sort-First Architectures.....	35
Figure 2.3: Sort-Middle Architectures.....	39
Figure 2.4: Fragment-Sorting Architectures.....	42
Figure 2.5: Image-Composition Architectures	43
Figure 2.6: Pomegranate Architecture	46
Figure 3.1: The Parallel Graphics Interface Extensions	56
Figure 3.2: Parallelizing a Simple Interactive Loop.....	60
Figure 3.3: Parallel Marching Cubes Traversal	62
Figure 3.4: The Argus Pipeline.....	66
Figure 3.5: Argus Speedup Graphs.....	69
Figure 3.6: Argus Speedup Graphs without Rasterization	70
Figure 3.7: Effects Buffering and Granularity on Argus	72
Figure 3.8: Barriers in Pomegranate.....	75
Figure 3.9: Pomegranate Speedup Graph.....	77
Figure 4.1: Mip Mapping.....	88

Figure 4.2:	A Texture Prefetching Architecture	93
Figure 4.3:	Texture Data Organization	99
Figure 4.4:	Cache Efficiency	102
Figure 4.5:	Bandwidth Variation	104
Figure 4.6:	Prefetching Performance	107
Figure 4.7:	Time-Varying Execution Time Characterization	109
Figure 4.8:	The Effects of Varying Buffer Sizes	111
Figure 4.9:	A Base Graphics Pipeline.....	116
Figure 4.10:	Dedicated and Shared Texture Memories	117
Figure 4.11:	Shared Texture Data Organization.....	120
Figure 4.12:	Bandwidth Due to Compulsory Misses.....	126
Figure 4.13:	The Effects of Cache Size.....	128
Figure 4.14:	Bandwidth Requirements of a 16 KB Cache	129
Figure 4.15:	Texture Load Imbalance.....	131
Figure 4.16:	Breakdown of Serial Time	132
Figure 4.17:	Speedup Graphs for Dedicated Texture Memory	134
Figure 4.18:	Speedup Graphs for Shared Texture Memory.....	135

Chapter 1

Introduction

The roots of this dissertation can be traced back a decade to my first experience, during high school, with an interactive graphics architecture during high school. An exuberant engineer was speaking during career day about his experiences at Silicon Graphics, Inc. To convey this excitement to an otherwise unimpressed audience, he brought his graphics workstation to demonstrate a simple flight simulator. When I first saw this demo, I truly felt it was *magic*—here was an imaginary world in which I was immersed, controlled by my whim! I could fly a plane, pause the world, and look at it from any angle with a few flicks of the wrist. Little did I understand the underlying graphics architecture, let alone the concept of computation in silicon. Looking back now, the experience was far from realistic—the polygon counts were low, the surface detail was minimal, the lighting models were simplistic, aliasing was occurring on the edges, etc. Furthermore, years of study have transformed the mystery of computers into mundane machinery. However, even today when I load up the latest 3D game, I am in awe. The goal of this dissertation is to advance the mundane machinery of graphics architectures that creates the magic of interactive 3D. In particular, this thesis examines techniques and algorithms for providing scalability in two areas of interactive graphics architectures that have not previously been addressed by the research community: interface and texturing.

1.1 Trends in Graphics Architecture

Interactive graphics architectures have improved dramatically over the past few decades due to improvements in both algorithms and semiconductor technology. In addition to improving raw performance, successive architectures incorporate additional features aimed at improving visual quality. According to one classification of workstation graphics [Akeley 1993], the first-generation graphics architectures of the late 1970s and early 1980s were able to transform and clip geometry at rates that enabled wire frame renderings of objects. In the late 1980s, second-generation architectures were able to incorporate enough memory and compute power to enable depth buffering for hidden surface removal and Gouraud shading for smooth lighting. These capabilities opened the door for applications such as computer-aided design, animation, and scientific visualization. Then in the 1990s, third-generation architectures introduced texture mapping and anti-aliasing capabilities that greatly enhance realism in virtual reality and simulation applications. During this same period, due to advances in semiconductor technology, architects were able to integrate compelling 3D graphics on a single chip, paving the way for low-cost mass-market applications such as gaming and other forms of computer entertainment. As an extension to this classification [Hanrahan 1997], we are moving towards a fourth generation of interactive graphics architectures that include programmable shading and other advanced lighting models. Eventually, the paradigm of local shading models will be broken by fifth-generation architectures that are able to compute global illumination in real time.

Rendering research deals with computationally efficient methods for image synthesis, but the definition of computational efficiency is elusive due to exponential rates of increase in available computational performance over the past several decades. As the size of the devices that can be placed on a chip decreases, a chip architect can exploit the resulting improvement in speed and density to increase computational power. Under the current regime of semiconductor devices, computing power at a given price point has doubled approximately every couple of years. Consequently, “computational efficiency”

is a moving target: computations and architectures that were impractical or impossible a decade ago are now quite feasible. This rate of exponential growth in semiconductor processing power, dubbed Moore’s Law, is expected to continue for at least another fifteen years using known manufacturing methodologies. Furthermore, it is quite conceivable that improvements in semiconductors or another technology will allow this exponential growth for quite some time. Graphics architects have been able to increase performance at exponential rates by utilizing these semiconductor advances and will continue to do so in the future.

Although the technological forces greatly influence the way graphics architectures are designed, market forces have an equally important role in shaping graphics architectures. As three-dimensional graphics architectures have become increasingly mainstream, significant changes have occurred in their designs. In the past, computational power was relatively scarce, and graphics architects had to use tens to hundreds of chips to harness enough power for a compelling 3D experience. These systems, though parallel, were not very scalable: at best, one or two aspects of performance could be scaled by a factor of two or four. Because of the large number of components, the cost of these systems was very high, and consequently the volumes were limited to the few customers to whom interactive graphics was mission-critical. Furthermore, as the amount of processing power that can be placed on a single chip has increased, architects have been able to design compelling graphics architectures at lower price points. The early graphics architectures were aimed at a few high-end customers who were willing to pay millions of dollars for flight simulators; however, later architectures, composed of a few to several chips, were aimed at personal workstations, whose customers could afford price ranges in the tens of thousands of dollars. Now, we are at a point where an impressive amount of graphics power can be provided at consumer price points on a single chip.

This mass-market trend has had a profound impact on graphics architectures because the engineering costs of designing mass-market architectures can be amortized over millions rather than thousands of units. The first and most obvious result is the massive economies of scale that occurs. At low volumes, engineering costs make up the majority

of the cost of a graphics unit, and the cost of the silicon makes for a small fraction. At high volumes, engineering costs are a fraction of silicon costs. Thus, high-volume designs are able to provide processing power at a cheaper rate. Furthermore, the low per-unit cost of engineering in high-volume designs allows for an increased engineering budget. This means that mass-market graphics architectures tend to have highly optimized designs that make better use of the silicon and faster product cycles. As a result, graphics architectures aimed at the consumer market provide a much better price-performance ratio than graphics architectures aimed at the professional market.

Because interactive graphics is still several orders of magnitude away from realism because of limited performance, many high-end customers need performance beyond what the consumer graphics chip can offer. The obvious answer is to make increased use of parallelism: the amount of computational power that can be placed on multiple chips is proportionally higher than that placed on a single chip, but these chips must be able to communicate and coordinate their computations. The subdivision of computation among different processing units is one of the main decisions for a graphics architect, whether it is within a single chip, with area constraints, or across multiple chips, with cost and bandwidth constraints. Traditionally, high-end graphics systems were composed of heterogeneous chip sets—one type of chip is built for command processing, another for composition, etc. Furthermore, these architectures did not place a high premium on scalability—at most one or two aspects of performance may be increased by adding more chips to the system. Because of the large price-performance advantage offered by low-end, single-chip graphics architectures, it is no longer cost-effective to engineer point solutions for the high-end graphics system composed of heterogeneous chips. Instead, the optimal way to build these high-end graphics architectures is by combining self-sufficient, low-end graphics chips in a scalable fashion to realize scaled performance.

In this thesis, we examine the scalability of parallel graphics architectures. First, we present a novel analysis of the graphics pipeline and a novel framework for the classification of graphics architectures, examining how current architectures fit into these frameworks. Then, we examine two aspects of scalability that have been largely ignored by

3D world and are by far the most common representation format because of their amenability to animation. Volume-based representations describe characteristics of the 3D environment (e.g., opacity) as a function of a 3D position in space. Often, this function is parameterized by uniformly spaced samples in space called voxels. A generalization of volume-based representations that parameterizes the environment according to a 2D direction in addition to 3D position leads to a 5D plenoptic function [McMillan & Bishop 1995]. Image-based representations, on the other hand, forgo a direct representation of the 3D environment and describe it as a set of 2D images. Some examples of such representations are 2D panoramas [Chen 1995] and 4D light fields [Levoy & Hanrahan 1996, Gortler et al. 1996]. Given the plethora of representation formats, the “correct” choice for an application depends on the ease of modeling the environment, the phenomena used in the rendering, and the time constraints of rendering.

While the algorithms used for rendering are quite varied, often being closely tied to a particular representation, they may be classified as projection-based or ray-based. With projection-based algorithms, surface elements, volume elements, or image elements are transformed onto the image plane according to a projective map. Because of regularity in computation and access patterns, these types of algorithms tend to be fast. Ray-based algorithms, on the other hand, compose images by tracing rays through the environment and computing interactions between these rays and the intersected scene elements. Such algorithms allow the description of much richer phenomena, but the higher computational costs lead them to be much slower. Again, the “correct” choice for a rendering algorithm depends on the phenomena and time constraints. For example, an animator may use an interactive projection-based graphics system to set an animation’s key frames during the day, and then she may use a Monte Carlo ray tracer to render the final animation overnight. Because projection-based algorithms tend to be more efficient computationally, most interactive rendering systems use such an algorithm because of the performance level required by human interaction.

parallel graphics research: we look at how to scale the interface into graphics systems in a way that is semantically compatible with existing interfaces; then, we will look at how the texturing subsystem of graphics architectures may be scaled. In addition to presenting designs for scalable interface and texturing, we will quantify the effectiveness of our approaches.

1.2 The Rendering Problem

Rendering is the field of study that deals with the synthesis of images from computer models of the world; vision is the complementary problem that deals with the analysis of images to create computer models of the world. While the two problems are inverses of each other, the state of the two fields from the point of computation is quite different. With vision, the computational process of creating a model of the world from real-world imagery is extremely difficult because it is, in general, an artificial intelligence problem. Thus, human-like understanding and reasoning are required, fundamental processes whose computational algorithms are largely unknown, and much research focuses on understanding these processes. On the other hand, the fundamental process behind rendering, the physics of light transport, is well understood. This understanding is so detailed that multiple levels of approximation must be applied to make image synthesis tractable. Thus, rendering research mainly deals with computationally efficient algorithms and models for image synthesis. Rendering systems can be classified according to two axes: the representation used to describe the environment and the algorithm used to create the images. In general, many algorithms may be used for a given representation, and many representations may be used in conjunction with an algorithm.

1.2.1 Representations and Algorithms

The various representation formats of 3D environments may be loosely classified as surface-based, volume-based, or image-based. Surface-based representations (e.g., NURBS, subdivision surfaces, and triangle meshes) describe the environment as 2D manifolds in a

1.2.2 Interactive Rendering

Key frame animation, modeling, computer-aided design, simulation, gaming, and many other applications of rendering require the interaction of a human. Human interaction is confined to a relatively constrained period [Foley et al. 1990]. The productivity of a human using a computer interactively for a task is dramatically dependent on the latency between the user’s input and the computer display’s output. In particular, if the latency exceeds beyond approximately 1 second, productivity drops dramatically: the computer’s responses to the human’s actions are too slow for the human to interact seamlessly. At the other end of the spectrum, the benefits of a low-latency system are limited by the physiological characteristics of the human brain that give the illusion of a real-time phenomenon. For example, the *motion fusion rate* is the frame rate required to fool the human eye into perceiving motion in discrete frames as a continuous motion. This is around 20 Hz for humans. The *flicker fusion rate* is the rate at which a set of images that are “flashed” onto a surface (akin to a strobe light) appears as a continuous source of illumination. This tends to be between 50 and 80 Hz for most humans, depending on lighting conditions. Virtual reality researchers have also found that for a human immersed in a virtual world, end-to-end latencies in the tens of milliseconds give the illusion of real-time, while latencies over about a hundred milliseconds give nausea. In fact, the minimal latency detectable by a human has been shown to be approximately ten milliseconds [Regan et al. 1999]. Thus, we define *interactive rendering* as the process of creating images from 3D environments at rates between approximately 1 and 100 Hz.

The latency requirements of interactive rendering place unique constraints on graphics architectures. First, because the utility of interactive computer graphics to most applications is related to a specific latency, performance increases are targeted at rendering scenes that are more complex rather than rendering scenes at a faster rate. For example, for a computer-aided design application, being able to render a machine part at several hertz gives the designer a good sense of the object. While increasing the frame rate by an order of magnitude makes the interaction noticeably smoother, in the end, the contribu-

tion of computer graphics to the design process does not increase significantly. Similarly, in a virtual reality application, increasing the frame rate from 100 Hz to 1000 Hz is not useful due to the perceptual characteristics of the human brain. Increasing scene complexity, however, results in an appreciable improvement. Another result of latency requirements on interactive graphics architectures is the focus on frame latency rather than frame throughput. It is not very useful to provide a frame rate of 10 Hz if the latency on each of those frames is 1000 milliseconds. Ideally, a frame rate of 10 Hz should be provided with a frame latency of 100 milliseconds. This has particular implications on how interactive graphics architectures should exploit parallelism—distributing each frame to a distinct graphics pipeline is not practical beyond a couple of pipelines (e.g., Silicon Graphics’s SkyWriter). Thus, most parallel architectures focus on intra-frame parallelism.

1.3 The Graphics Pipeline

The computer graphics community has settled on a general algorithm for performing interactive rendering called the *graphics pipeline*. It defines a projection-based rendering algorithm that mainly supports surface representation, although it can be extended to support volume-based and image-based representations. The de facto standard that currently specifies the graphics pipeline is OpenGL (Open Graphics Library) [Segal & Akeley 1992, Neider et al. 1993]. To programmers, OpenGL defines the application programmer’s interface (API) to the graphics system. To graphics architects, OpenGL specifies the OpenGL machine model—a virtual machine that in essence defines the OpenGL graphics pipeline. While many other graphics APIs and virtual graphics pipelines exist (e.g., Direct3D), we will focus our attention on OpenGL. However, much of this thesis is applicable to other APIs.

The most defining characteristic of the graphics pipeline is, as the name implies, that it is a *pipeline*. In a pipeline, data flows linearly from one stage to the next in a fixed order, and each stage processes some type of input data and produces some type of output

data for the next stage in the pipeline. Such a pipeline lends itself naturally to *pipeline parallelism*, where each stage in the pipeline is implemented by a separate processing unit optimized for that particular type of processing. A pipeline also provides for *strict ordering semantics*, the idea that data is processed and its effects are realized in a serial order as data moves along the pipeline. The graphics pipeline also has implications for data dependence: one piece of data is processed independently of other pieces of data. This characteristic is key for *data parallelism*, where data for any given stage of the pipeline can be distributed across a large number of processing units for parallel processing. We will examine data dependence issues in greater detail in Chapter 2. Generally, the lack of data dependencies carries with it many implications for graphics algorithms. For example, the graphics pipeline supports only a local shading model where the lighting calculations for one primitive must be computed independently of all other primitives. This precludes global illumination effects such as shadows, indirect illumination, and caustics.

The graphics pipeline is composed of several stages, as illustrated in Figure 1.1. The application enters data into the graphics pipeline, and the data flows through the various graphics stages. Each of these stages holds a certain amount of state that determines how data passing through is processed. For example, the position, direction, intensity, etc. of the light sources are held by the lighting stage; these parameters determine how the colors of the vertices passing through it are modified. This lighting state is set according to the application’s needs, and often the state changes are flowed through the pipeline alongside the graphics data. We now will briefly go over each of the stages, which are roughly grouped as geometry stages or rasterization stages.

- **Command Processing.** The command processor is responsible for processing the input stream of graphics commands. Some of these commands are state modifying commands that are simply propagated to the appropriate stage of the pipeline. While there is great variety in the number of types of state modifying commands, the frequency with which they occur is relatively low. The majority of commands that the command processor sees are vertex commands: commands that control the properties of vertices that are submitted to the rest of the graphics pipeline in the form of 3D object-space triangles. The amount of data associated with a vertex is approximately a couple hundred bytes—this includes properties such as position, normal, texture coordinates, color, and surface material. Because much of this data is not changing on a

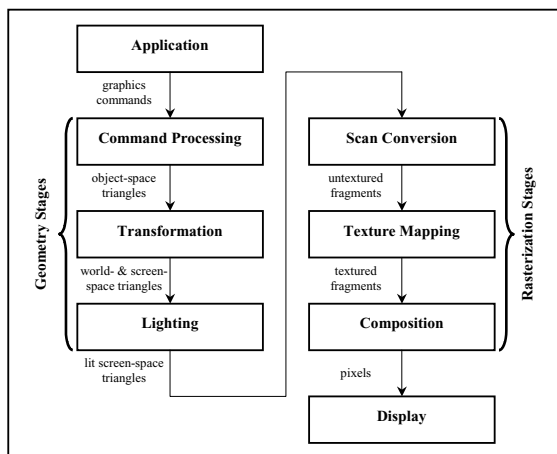


Figure 1.1: The Graphics Pipeline

per vertex granularity, the notion of a “current” vertex is used to minimize the bandwidth required for the interface. The current vertex holds all of the vertex properties except for position. As these various vertex properties are specified, the values of the properties are modified in the current vertex. Then, when a position is specified, the current vertex is emitted with the specified position. The command processor is responsible for determining how these vertices are connected to form 3D object-space triangles that are submitted to the rest of the pipeline. The command processor is also responsible for managing display lists, precompiled lists of commands that may be invoked with a single command later.

- **Transformation.** The incoming 3D object-space triangles are transformed to a world-space coordinate system according to a model-view matrix. This 4×4 matrix transforms homogenous coordinates in object-space to homogenous coordinates in a canonical world-space where the eye is located at the origin looking down the negative Z-axis. This matrix is able to express an arbitrary series of transformations that include rotation, translation, scaling, shearing, and projection. Surface normals are similarly transformed into world-space coordinates. The resulting 3D world-space triangles are then transformed into clipping coordinates according to another 4×4 projection matrix. In this space, clipping occurs against the edges of a canonical view frustum. These clip coordinates are then mapped to the screen by a perspective divide and a simple transformation. Additionally, the texture coordinates of the vertices are transformed according to another 4×4 matrix. These texture coordinates may either be specified by the application or generated according to the position of the vertices. Once all these transformations have occurred, we are left with world-space triangles for lighting and screen-space triangles for rasterization.
- **Lighting.** Using world-space positions and normals in conjunction with the material properties, the vertices of the triangles are lit from a number of light

sources according to a local lighting model: the lighting of any one vertex is independent of any other vertices. Light parameters typically include a position, various intensity coefficients, attenuation factors, and spotlight parameters.

- **Scan Conversion.** Scan conversion takes the three vertices of a screen-space triangle and generates samples from the interior of the triangle (*fragments*) everywhere there is a screen pixel. The values for these fragments are generated by linearly interpolating between the three vertices of a triangle. Typical interpolated values include color, depth, and texture coordinates. Because the projected triangles are being viewed under perspective, care must be taken to correctly interpolate the parameters, particularly texture coordinates. The resulting untextured fragments are then passed on for texture mapping.
- **Texture Mapping.** In its most basic form, the texture subsystem pastes an image onto a triangle that is viewed in perspective. This involves performing a lookup in the texturing unit's texture memory based on the incoming fragment's texture coordinates. Because the texture coordinate does not usually map exactly to a sample in the texture map (a *texel*), some sort of filtering must occur. Point sampling takes the nearest neighbor, while bilinear sampling takes a weighted average of the four closest texels. However, this can still result in aliasing artifacts because an arbitrary number of texels may fall within the region between adjacent pixels. Intuitively, one should use a weighted average of the texels within and around this region. A common approximation to this is mip mapping. For each texture, multiple pre-filtered copies of the texture are stored: one at full resolution, one at half the width and height, one at quarter the width and height, and so on, down to a single-texel texture. A typical pre-filtering technique utilized is a simple box filter—just take the average of the four corresponding pixels in the higher resolution image of the mip map. Then, when rendering occurs, not only are the texture coordinates for a fragment calculated, but also an approximation of the texel-

to-pixel ratio. This ratio is used to pick a level of the mip map from which bilinearly interpolated sample are taken. A ratio of one corresponds to full resolution, a ratio of two corresponds to half resolution, a ratio of four corresponds to quarter resolution, and so on. Because the texel-to-pixel ratio usually falls in between two mip map levels, a linearly interpolated average of bilinearly interpolated samples from the two adjacent mip map levels may be taken, resulting in trilinear mip mapping.

- **Composition.** Given a textured fragment with color, depth, and coverage, the composition unit's responsibility is to merge this fragment with the various buffers (color buffer, depth buffer, stencil buffer) that are collectively known as the framebuffer. Depth occlusion is usually resolved by comparing the fragment's depth with the depth stored for the pixel in the depth buffer and conditionally rejecting the fragment. A variety of blending modes and logic ops define how a fragment is combined with the color currently in the color buffer. The pixel coverage information is used in conjunction with the stencil buffer according to a stencil op to implement a variety of effects.
- **Display.** Many modern interactive display technologies are based on a raster scan. The display is in the state of constant refresh where the color on a certain portion of the display must be re-displayed every fraction of a second. In a monitor, for example, an electron gun starts out at the upper-left corner of the screen and scans out the first row of pixels. A horizontal retrace then moves the electron gun beam from the right side back to the left side, and another row is scanned out. This continues until the bottom row is reached, at which time a vertical retrace occurs and the whole process is repeated. The implication that this has on graphics architectures is that its color buffer must constantly be accessed to drive the display. Furthermore, if adjacent pixels of the color buffer are stored in logically separate locations, the pixels must be reassembled for scan-out.

These stages of the graphics pipeline form a basis for the work presented in this thesis. Our work relates to the scaling of the graphics pipeline—given the basic stages of the pipeline, we examine ways of distributing work amongst multiple copies of each stage in a way that gives scalable performance increases. This thesis focuses on providing scalability in two stages of the pipeline that have been previously ignored by the research community: the interface to the command processing unit and the texturing unit.

1.4 Summary of Original Contributions

The original contributions of this thesis fall in three areas. First, we present a novel analysis of parallelism in graphics architectures in Chapter 2:

- **Sources of Parallelism.** A lot of work has been done in microprocessor architecture that analyzes various aspects of computer systems. We present a novel analysis of the graphics pipeline that comparatively applies concepts found in computer architecture. We focus on examining how parallelism is exposed and constrained by architectural structures, semantics, and dependencies.
- **Analysis of Scalability.** We define the five parameters of a graphics architecture that benefit the end user (and hence, those that should be scaled) and the various semantics that a parallel architecture may support. Based on this, we examine the extent to which existing parallel graphics architectures scale with respect to these categories.

Then, we present and quantify algorithms for two aspects of scalability that have not been addressed by previous research in computer graphics. In Chapter 3, we present a novel scheme for parallelizing the interface to the graphics system:

- **Parallel Graphics Interface.** The rate of performance improvement in graphics architectures has outstripped the rate of performance improvement in microprocessor and computer architectures. As a result, the interface has be-

come a major bottleneck to improving the overall graphics system. To alleviate this bottleneck, we present a novel parallel interface for graphics systems. By introducing simple, well-understood synchronization constructs into the graphics interface, we are able to parallelize the submission of graphics command by the application, the bandwidth between the application and the graphics system, and the processing of graphics commands by the graphics system. These parallel interface extensions are compatible with the semantics applications have come to expect from a graphics interface, and we demonstrate that the interface can be used to attain near-linear speedup in graphics performance in three varieties of implementations. Much of this work was described in [Igehy et al. 1998b], and some was described in [Eldridge et al. 2000] and [Buck et al. 2000].

In Chapter 4, we examine and quantify the scaling of the texture subsystem. In particular, we focus on two aspects of making parallel texturing work:

- **Prefetching in a Texture Cache.** Texture subsystems have come to rely on caching to greatly reduce the bandwidth requirements of texture mapping. In a parallel texture caching architecture, particularly one that shares texture memory across many texturing units across a scalable network, memory latency can be both high and highly variable. We present a texture prefetching architecture that works in conjunction with texture caching. This architecture is able to eliminate virtually all the latency of a memory system. While this architecture is useful when a texture unit accesses a single texture memory with a small, fixed latency, it is critical for texturing units accessing a potentially shared memory across a potentially shared network. Much of this work was described in [Igehy et al. 1998a].
- **Parallel Texture Caching.** The concept of a parallelizing the texturing subsystem is simple: each texturing unit is responsible for accessing and applying texture to a fraction of the fragments generated by a scene. Additionally, the

texture data may be shared across the texture memories of the texture units instead of being replicated in a dedicated texture memory for each texturing unit. We present a detailed study of both shared and dedicated texture memory architectures, focusing in particular on quantifying how well texture caching parallelizes across a variety of parallel rasterization architectures. We find that texture caching works well with a variety of parallel rasterization schemes given the proper choice of parameters. Much of this work was described in [Igehy et al. 1999].

Finally, we conclude the thesis in Chapter 5.

Chapter 2

Analysis of Parallel Graphics

Interactive graphics architectures have evolved over the past few decades to a canonical graphics pipeline, which we briefly outlined in Section 1.3. OpenGL [Segal & Akeley 1992] formally specifies such a graphics pipeline and has gained widespread acceptance throughout the research, professional, and consumer communities as a de facto standard. This specification forms a formal starting point for our research, and its role in graphics architectures is quintessential. As the specification says, “To the programmer, OpenGL is a set of commands that allow the specification of geometric objects in two or three dimensions, together with commands that control how these objects are rendered into the framebuffer.” By using this set of commands, the programmer is able to write a graphics program that renders images on any computer that supports OpenGL, regardless of the underlying hardware. Conversely, a graphics architect is able to design a system that can perform the rendering of any program written using OpenGL, regardless of the original target system of the graphics programmer. In this view of OpenGL, the graphics API is a language for graphics architectures. In Section 2.1, we analyze the virtual graphics machine implicitly specified by this language, examining how this model exposes parallelism in some ways and constrains parallelism in other ways. Then, in Section 2.2, we present the relevant criteria for scaling in graphics architectures, and we evaluate how existing parallel graphics architectures scale with respect to these different criteria.

2.1 Sources of Parallelism

From the point of view of the graphics architect, an API such as OpenGL defines much more than just a language—it defines the virtual graphics machine that must be implemented. In this section, we analyze this virtual graphics machine. We take the novel approach of applying concepts found in traditional computer architecture (e.g., [Hennessy & Patterson 1996]) to graphics architectures. A formal understanding of this conceptual framework helps direct the graphics architect, whose job is to implement a specific instantiation of the virtual graphics machine under the constraints of cost, performance, time, and complexity. One of the main technical factors that has led to the proliferation of graphics architectures is the fact that the virtual graphics machine is *significantly* different from a traditional microprocessor—the specialized computation of graphics exposes parallelism that traditional architectures are not optimized to exploit.

Typically, microprocessor architectures are made up of four essential structures: processing units, instructions, registers, and memory. Processing units define the computation that can be performed; instructions define the control of the microprocessor; and registers and memory define the state of the system. While both registers and memory represent the state, the main distinction between the two is that registers are fixed and finite in number. Thus, registers can be addressed with much fewer bits, can be implemented with much faster logic, and are often given specialized functionality. An instruction stream forms a sequence of commands executed by the microprocessor. Each command instructs one or more of the processing units to modify the state of the system based on the contents of the registers and / or the memory. This modified state can be registers or memory. As with the microprocessor, the virtual graphics machine may be viewed in terms of the same four essential structures. The sequence of graphics commands from the API defines the instruction stream that the processing units execute. Memory is specialized into specific domains such as the framebuffer (color, depth, stencil), texture memory, display list memory, and the accumulation buffer. The graphics context, which determines how incoming primitives are combined with the various

memories (e.g., lighting state, blend mode, texturing mode), corresponds to the register set. Like the register set, the graphics context has an architecturally specified number of components.

2.1.1 Instruction Set

A basic premise of all microprocessor instruction sets is the idea of serial ordering semantics: even though many instructions may be processed simultaneously, the result of such processing must be identical to the result of processing the instructions serially. OpenGL and most other widely accepted graphics APIs also require this strict ordering semantics. Such semantics give the application programmer a clear, simple understanding of the underlying system. Individual instructions may expose parallelism by themselves, and adjacent instructions may expose parallelism in a way that does not violate strict ordering semantics.

Instruction sets are often classified with a certain design philosophy. In the early days of the integrated circuit, memory was the scarcest resource in a system. Consequently, microprocessors were designed with the CISC (complex instruction set computing) philosophy. Instructions were of variable size, with the most common instructions taking the fewest bytes; instructions treated registers non-orthogonally, often tying a specific register to a specific instruction, requiring additional bytes to use a different register; a single instruction could cause a large number of operations (e.g., cosine function, string copying instructions). Typically, each instruction took several cycles to compute. As compiler technology improved and pipelining became possible, instruction sets moved to the RISC (reduced instruction set computing) philosophy. Instructions were of fixed size; the number of registers increased and instructions could address them orthogonally; each instruction specified a single operation. Now, instruction fetch has become a critical bottleneck in microprocessor architecture—it is very difficult to keep all the execution units busy with useful work from the instruction stream. As a consequence, we are seeing a move towards VLIW (very long instruction word) instruction sets that allow a fixed

number of RISC instructions per instruction word and effectively move much of the dependency analysis to the compiler.

Within this framework, OpenGL is a CISC instruction set. In a certain sense, OpenGL instructions follow the CISC philosophy to a greater extent than microprocessor CISC instructions. The size of instructions can be anything from a few bytes to specify a change in the current color to dozens of bytes to specify a matrix to millions of bytes to specify a texture. The register set specified by the graphics context state is highly non-orthogonal. Each instruction accesses only specific registers in the graphics context, sometimes with special side effects. The amount of work specified by each instruction can vary by an enormous amount, anywhere from writing a single value into the current color to modifying every pixel in the framebuffer.

Graphics instructions can be broadly classified into four categories. By far, the most common type of command is the *context-modifying* instruction. Examples of such commands are: color, normal, material, and texture coordinate commands that specify the vertex attributes of the current vertex; commands that specify the lighting, commands that specify the blending modes, etc. The second type of command is the *buffer-modifying* instruction that reads some of the system state and modifies one or more of the buffers based on this state. The most common of these is the vertex command, which can specify the drawing of a triangle; however, clears, bitmap downloads, texture downloads, and display list downloads all fall into this category. *Readback* commands in the graphics interface allow the querying of state by the application program, and no analogue exists for this type of command in the microprocessor world. Finally, *special* instructions such as flush and finish commands fulfill a special need in the graphics interface. While the view exported to the application program by the graphics interface is a machine that executes commands serially, one at a time, with the modifications of all previous commands visible to the current command, the view exported to the display device is completely different. In order to synchronize the display to the state expressed by the command stream, one of two commands may be used. A flush command may return immediately but guarantees that all previous commands will execute in a finite amount of time.

the product of two other numbers. However, the two individual operations of multiplication and addition may not be executed simultaneously because of *intra-instruction data dependencies*: the multiplication must occur before the addition. Even with these dependencies, significant *pipeline parallelism* is exposed. If the system has enough functional units for a single addition and a single multiplication per instruction, the multiply-accumulate operations may be pipelined to allow a single multiple-accumulate instruction per cycle. On the first cycle, the multiplication for the first instruction occurs using the multiplication unit. On the second cycle, the first instruction uses the addition unit, thereby completing its execution, and the second instruction uses the multiplication unit. On the third cycle, the second instruction moves to the addition unit, and a third instruction starts using the multiplication unit, and so on. This pipelined execution can occur so long as there are no *inter-instruction data dependencies*: if the second instruction uses the result of the first multiply-accumulate instruction as an operand for its initial multiplication, it cannot be pipelined with the first instruction.

Instruction streams also expose what are known as *false dependencies*. A false dependency occurs when a dependency exists between instructions because of the use of the same registers (resource conflict), but no true dependency exists in the underlying data flow. For example, imagine an add instruction that uses a register as one of its operands followed by an add instruction that uses the same register to store the result of its operation. From the point of view of the register set, those operations must be done sequentially. However, from the point of view of the underlying computation, the second instruction does not require the results of the first instruction. This is called a write-after-read hazard (*anti-dependency*). If an add instruction that writes its result to a register is followed by an add instruction sometime later that also writes its result into the same register, a write-after-write hazard has occurred (*output dependency*). Virtualization is one solution for removing these dependencies: microprocessors typically provide more physical registers than the number of architectural registers specified by the instruction set and perform register renaming to provide a virtualized mapping between the two to address the problem of false dependencies.

A finish command does not return control to the application program until all previous commands are executed.

2.1.2 Data Dependencies

The instruction set of a machine greatly influences how much parallelism an implementation may exploit. While the instruction set may allow multiple operations to occur simultaneously, it can also introduce dependencies that inhibit parallelism. Here, we review general concepts in data dependencies and their relationship to different types of parallelism.

2.1.2.1 Dependencies and Parallelism

A well-understood source of parallelism in microprocessor instruction sets is bit-level parallelism: the processing units operate on multiple bits of data at a time. For example, a 1-bit addition instruction takes in two 1-bit numbers along with a carry-in bit to produce a 1-bit result and a carry-out bit. An 8-bit addition may be computed by eight applications of the 1-bit add instruction. However, if the instruction set defines an 8-bit add instruction, the implementation may parallelize the computation in novel ways (e.g., a chain of 1-bit adders, a 16-bit table lookup). Similarly, the instruction set may define a SIMD (single-instruction multiple-data) operation that allows eight 8-bit adds of eight distinct sets of values with a single instruction. This instruction exposes *data parallelism* to the architect by allowing the simultaneous use of eight adders on eight sets of distinct operands. Even if the eight additions were specified by eight separate instructions, the same parallelism is available, albeit at the cost of a more complex instruction fetch, because no *data dependencies* exist between the eight operations. Data parallelism is possible whenever there is a lack of data dependencies between instructions requiring operations of similar type, whether it is a simple addition or the entire transformation and lighting stage of the graphics pipeline.

However, not all instructions expose parallelism in such a straightforward manner. For example, a multiply-accumulate instruction is defined as the sum of a number with

2.1.2.2 Data Dependencies in Graphics Architectures

By examining how the various instructions of the graphics API are dependent on other instructions, we can understand the sources of parallelism in graphics architectures. Here, we look at the dependencies exposed by the four types of graphics commands: buffer-modifying commands, context-modifying commands, readback commands, and special commands.

Buffer-modifying commands perform read-modify-write operations on one or more of the buffers based on the content of the graphics context. A triangle-drawing command requires that the triangle first be transformed, light, and setup for rasterization according to the graphics context. A large amount of parallelism may be exploited at this level. Beyond the abundant bit-level parallelism available in the floating-point operations, significant data parallelism exists in the fact that most operations occur on four-element vectors (e.g., position, normal, color, texture) in SIMD fashion. Furthermore, many of the operations for a single triangle have no intra-instruction data dependencies: the projection matrix may be applied independent of the lighting; the contribution of each light may be calculated independently; etc. Even operations that cannot take advantage of this data parallelism because of intra-instruction data dependencies may utilize pipeline parallelism to an extremely large degree: e.g., one matrix-multiply unit may be transforming a vertex with the projection matrix while another matrix-multiply unit is transforming the next vertex with the model-view matrix. Because absolutely *no* inter-instruction data dependencies exist between triangles except at the framebuffer (we will look at this in more depth later), data parallelism may be used across multiple triangle instructions in addition to pipeline parallelism. In fact, many implementations chose to exploit inter-instruction data parallelism by distributing triangles amongst many units because infrequent clipping operations can cause excessive stalling in pipelined implementations.

Subsequent to the rasterization setup stage, triangles are broken up into fragments that correspond to individual pixels on the screen. Each fragment is then put through a series of steps that access the various buffers: texturing, alpha testing, depth testing, stencil test-

ing, blending, etc., some of which may terminate further processing on the fragment. As with transform, lighting, and setup, large amounts of SIMD parallelism exists for each fragment in addition to basic bit-level parallelism: texturing computes coordinates based on vectors, and all color operations compute on four components at a time. Additionally, many of the operations of a single fragment may be done in parallel: e.g., the texture lookup, the depth test, the alpha test, and the blending operation may all be performed simultaneously. The only dependencies exposed are the actual writes to the depth, stencil, and color buffers—everything else may be performed speculatively.

Across the different fragments of a single triangle, absolutely no dependencies exist. Reads to the texture memory may occur in any order because they are non-destructive, and modifications to the framebuffer may occur in any order because by definition each fragment from a triangle accesses a different location in memory. Similarly, the texture accesses of fragments from *different* triangles may be done in any order. Therefore, the only dependencies that exist between the fragments can be summarized as follows: the buffer modifications of fragments falling on the same pixel be done in-order with respect to the commands that generated the triangles. Other commands that modify the framebuffer (clears, bitmap downloads, etc.) also have similar characteristics.

Dependencies also exist between these framebuffer-modifying commands and commands that modify the texture data. When texturing is enabled, a true data dependency exists between the texture download and subsequent fragment-generating commands that use the texture data during the fragment pipeline. Additionally, a false data dependency exists between the texture download and previous fragment-generating commands that used the texture object; a renaming scheme (akin to register renaming) could be used to eliminate this false dependency. Of course, as with individual pixels in buffer modifications, dependencies exist at the level of individual texels rather than the whole texture. Therefore, it is possible to allow the texture access of a subsequent triangle to occur in a downloaded sub-region of a texture even before the entire texture is downloaded.

The interaction between buffer-modifying commands and context-modifying commands is quite interesting. A true data dependency exists between a buffer-modifying

command and previous context-modifying commands that modify parts of the context used by the buffer-modifying command. For example, a vertex command (which causes a triangle to be drawn) is dependent on a previous color command (which sets the color of the vertex). In the other direction, however, a false dependency exists. Imagine a sequence of commands that first sets the color, then draws a triangle, then sets the color, and then draws a second triangle. Although there is a true dependency between the first two commands as well as true dependency between the last two commands, there is a write-after-read hazard (anti-dependency) between the middle two commands. The first triangle command needs to read the “current color” register before the second color command writes it, but from a data-flow perspective, there is no dependency, and a renaming scheme may be used to eliminate this false dependency. Of course, no dependency exists between context-modifying commands and buffer-modifying commands that do not access that particular part of the graphics context.

The interaction between context-modifying commands and other context-modifying commands can be similarly characterized. Context-modifying commands nearly always modify independent portions of the graphics context in a write-only fashion, and thus no dependencies exist between distinct context-modifying commands. Between context-modifying commands of the same flavor, false dependencies exist. Again, imagine a sequence of commands in the following order: color, triangle, color, triangle. The second color command cannot set the “current color” register until after the first color command has done so. However, an architect can provide a virtualized view of the “current color” register to remove the output dependency (i.e., write-after-write hazard). This can be done with nearly all context-modifying commands, and the major exception to this rule has to do with the matrix state. In particular, matrix-modifying commands are truly dependent on previous commands that modified the current matrix because matrix transformations are cascaded with multiplies to the current matrix (matrix loads break this dependency chain). Similarly, many commands modify the graphics context in a way that depends on the current model-view matrix (e.g., light positions, clip plane coordinates,

etc.). As with buffer-modifying commands, true dependencies exist in one direction and anti-dependencies exist in the other direction.

Readback commands expose obvious data dependencies. Readback of a part of a context is truly dependent on a previous command that set the corresponding state. Readback of one of the buffers is truly dependent on the execution of buffer-modifying commands that write the corresponding buffer. Again, this dependency exists at the individual pixel level. From a data-flow point of view, the reverse direction (readback of the current color followed by a write) exposes a write-after-read anti-dependency, but such a situation cannot actually occur because readback commands are synchronous: the readback does not return (and hence, no other commands may be submitted) until the queried value is given to the program. The finish command is directly dependent on the execution of all buffer-modifying commands, but the flush command introduces no data dependencies.

2.1.3 Control Dependencies

One cornerstone of microprocessors and general purpose computing is the dependence of the instruction stream on memory and registers. First, because instructions typically reside in memory, the ability to modify memory gives the ability to modify actual instructions. This practice is uncommon, if not prohibited, on modern microprocessors. Second, and much more commonly, the memory address of the current instruction resides in an auto-incremented register that can be modified by the execution of certain instructions. The ability to modify the instruction stream conditionally based on the current state of the system gives the programmer the ability to use the microprocessor for general-purpose computation. As a simple example, a first instruction subtracts two numbers and a second instruction conditionally branches the location from which instructions are fetched if the result of the subtraction is non-zero. The control logic of the instruction fetch is dependent on a computation and thus exposes a control dependency. Extracting parallelism in the face of these conditional branches is of great difficulty to microprocessor architects.

Control dependencies are minimal in a graphics system. First, and most importantly, the graphics instruction stream presented by the graphics interface to the graphics system does not contain any control dependencies: the system CPU gives a series of instructions that all need to be executed, and the location of the next instruction is *never* dependent on previous instructions. Second, a graphics command may specify conditionality, but this conditionality is always in the form of conditional execution. The most common example of this is depth buffering: a fragment is merged into the framebuffer only if its depth value is less than the depth of the current pixel in the framebuffer. Dealing with a conditional operation is much simpler than dealing with a conditional branch that diverges instruction fetch; in fact, most modern microprocessors have introduced conditional operations to their instruction sets so that compilers may use them instead of expressing the same computation with conditional branches.

2.1.4 Discussion

In comparing graphics architectures to microprocessor architectures, the most important thing to note is that graphics architectures are constrained much less by the issues of instruction fetch and dependencies.

In general, the amount of work specified by a graphics instruction is much greater than the amount of work specified by a microprocessor instruction. A typical microprocessor instruction specifies a single operation such as a multiplication. On the other hand, a typical graphics instruction such as drawing an entire triangle embodies hundreds of operations at the very least. The lighting and transformation of the three vertices of a triangle as well as the rasterization setup constitute a few hundred floating-point operations. Furthermore, each triangle can generate anywhere from a few to thousands to millions of fragments, and each fragment requires a few hundred fixed-point operation. Because of the large number of operations per graphics instruction, instruction fetch is relatively easy compared to microprocessors. Even so, as the number of execution units available on a graphics chip increases exponentially and the bandwidth into a graphics chip remains limited, modern graphics architectures are limited by a serial interface. In Chapter 3, we

introduce thread-level parallelism to graphics architectures in order to alleviate this instruction fetch bottleneck.

The relative lack of data dependencies in graphics instructions has a great impact in the design of graphics architectures. This allows for, among other things, large amounts of latency toleration. In Chapter 4, we will see how this plays a critical part in texture prefetching. The scale of parallelism available from a graphics interface is several orders of magnitude larger than the amount of parallelism available in a microprocessor instruction set. Not only do the individual instructions contain more operations in a graphics interface, but also the effects of minimal dependencies are enormous. While a modern microprocessor keeps an instruction window of a few dozen instructions that are partially processed, modern parallel graphics architectures allow tens of thousands of outstanding triangles. Each chip in such an architecture incorporates hundreds of execution units that may operate in parallel with little contention. This is in direct contrast with the few execution found in microprocessors that are organized around alleviating contention to a general-purpose, shared register set. Because the dataflow is unknown a priori in a general-purpose microprocessor, much hardware is devoted to detecting data dependencies and extracting instruction-level parallelism whenever possible. In graphics architectures, most data dependencies are known a priori, and most hardware resources are expended on exploiting the vast amounts of parallelism. Handling control dependencies is one of the most difficult aspects of microprocessors: it adds a great deal of complexity to the designs, and in the end, the greatly limits the amount of parallelism in the computation. With graphics architectures, on the other hand, parallelism is not limited in any way by control dependencies, nor do they add any complexity to the designs. The fact that graphics architects have devised so many distinct parallel graphics architectures is a direct consequence of the dearth of dependencies in graphics architectures.

2.2 Scaling the Graphics Pipeline

Graphics architects have devised numerous architectures that attempt to exploit parallelism. In order to build parallel systems out of components that may be manufactured using standard processes and leverage the economies of scale that result from using replicated parts, a graphics architect must determine the division of labor between the various components of a parallel system and provide the necessary communication infrastructure. In this section, we first define a set of metrics that may be used in comparing the scalability of different graphics architectures. Then, we survey current parallel graphics architectures and to what extent they scale in each category.

2.2.1 Scalability Metrics

We can examine the scalability of parallel graphics architectures in terms of both quantitative and qualitative measures. Qualitatively, a scalable architecture may imply certain semantic constraints on the graphics interface. Often, systems sacrifice semantic flexibility for increased performance or simplicity in a scalable design. Here are the major semantic metrics to which a system may or may not adhere:

- **Mode Semantics.** If a system supports *immediate-mode* semantics, then the application submits commands to a graphics system one at a time, and the system executes them more or less immediately. Like the C programming language, it is an imperative model of programming that tells the graphics system step-by-step instructions on how to draw the scene. If a system supports *retained-mode* semantics, then the application first gives a high-level description of the entire scene to the system, and then the application draws the scene. Just as a program dataflow graph gives more information than a CPU instruction sequence, such a high-level description gives a better understanding of the scene to the graphics system. Furthermore, like the Prolog programming language, retained-mode is analogous to a declarative model of programming where the programmer specifies what is wanted rather than how it should be

computed. A graphics system may support immediate-mode, retained-mode, or both. In general, it is easy to provide a wrapper library around an immediate-mode system to provide a retained-mode interface, but not the other way around. OpenGL requires immediate-mode semantics.

- **Ordering Semantics.** Ordering semantics define a graphics system's ability to maintain a drawing order specified by the application programmer. For example, imagine a flight simulator that places heads-up display (HUD) information on top of the rendered scene. It is important for the programmer to be able to specify that the HUD be drawn after the scene is drawn. In a *strictly ordered* graphics system, everything is drawn in an exact order as specified by the application. However, a graphics system may choose to relax this constraint. For example, if depth buffering is enabled and only opaque primitives are specified, then the drawing order of the primitives does not affect the final image on the screen. With *relaxed ordering*, a graphics system is allowed to merge fragments into the framebuffer in any order, reducing dependencies in the graphics system. A strictly ordered graphics system also supports relaxed ordering, by definition, and it is possible for a system to support both ordering semantics by switching between strict and relaxed modes. OpenGL requires strict ordering semantics.
- **Frame Semantics.** Frame semantics define whether a graphics system must adhere to drawing images frame by frame. In a system with frame semantics, frame boundaries must be explicitly defined, and once a frame is drawn, additional drawing may not occur to it. In a system without frame semantics, drawing may occur incrementally on the framebuffer without specifying when a frame begins or ends, which is critical to single buffered applications. Furthermore, frame semantics introduce one frame of latency to the drawing process: none of the current frame may be drawn until the entire frame is defined. OpenGL does not impose frame semantics.

In addition to the above semantic metrics, parallel graphics architectures may also be evaluated according to several quantifiable performance metrics. While most previous work focuses on triangle rate and pixel rate, it is important for parallel systems to scale in all aspects. Here, we identify the critical metrics that are used in evaluating the performance of a graphics system from the point of view of the application:

- **Input Rate.** As a system scales, it is important to be able to increase the rate at which commands may be submitted to the graphics system. In an immediate-mode interface, this means that the CPU, interconnect, and command processor must all scale their abilities to input commands. In a retained-mode interface, the command processor and its bandwidth to the scene graph memory must scale. Scaling the interface is the subject of Chapter 3 and constitutes one of the main contributions of this thesis.
- **Triangle Rate.** The triangle rate is defined by the rate at which the graphics system may transform, light, and perform rasterization setup on primitives. Scaling the triangle rate allows graphics systems to model the environment in more detail, ideally to the point of using triangles that cover only a couple of pixels. Furthermore, a high triangle rate is critical to advanced techniques that use multi-pass shading algorithms.
- **Pixel Rate.** The pixel rate of a system determines how fast a graphics system can rasterize primitives into fragments and merge those fragments into the framebuffer. Scaling the pixel rate is important for high-resolution displays, high depth complexity, and multi-pass algorithms.
- **Texture.** As a system scales, it is important to scale the texture subsystem. First, in order to satisfy the requirements of an increased pixel rate, the bandwidth of the texture subsystem must be scaled. Furthermore, scaling the actual amount of texture memory allows applications to use more textures and / or textures with higher resolution. Higher resolution textures are particularly

important on high-resolution displays. Scaling the texture subsystem is the subject of Chapter 4 and constitutes a main contribution of this thesis.

- **Display.** Advances in technologies such as LCD's and plasma displays are driving display resolutions upwards. In addition, researchers are actively examining how interactive environments may be created by tiling large numbers of commodity displays. Thus, it is important to scale the display capabilities of graphics systems, allowing larger numbers of displays as well as displays of higher resolution. This entails scaling the amount of framebuffer memory, the bandwidth to that memory, and the number of output interfaces.

While a parallel architecture either does or does not inherently support a particular mode, ordering, or frame semantic, its scalability in each of the above performance metrics may be classified in three ways. If a system provides no scalability in a category, then no additional performance is realized by scaling the system. In a system with limited scalability, scaling the system realizes increased performance up to a certain amount of parallelism, beyond which no performance gains are seen. In a system with full scalability, there are no inherent limits how high the system may scale. In addition to raw scalability, it is also important to examine the parallel efficiency of systems as they are scaled: all else being equal, an architecture that scales up to 16 nodes with 90% parallel efficiency is more valuable than an architecture that scales up to 64 nodes with 20% parallel efficiency.

2.2.2 Analysis of Scalable Architectures

Many different interactive graphics architectures have been presented in the literature over the history of computer graphics. Exploiting parallelism is a key component of graphics architectures, and the flexibility that arises from the dearth of dependencies in the graphics API leads to a wide variety of parallel implementation choices. Ideally, an architecture should provide a division of work that is highly load balanced and has a minimal amount of redundant work in order to increase parallel efficiency. The commu-

texturing unit, a composition unit, and a display unit. Sort-first architectures sort 3D primitives; sort-middle architectures sort 2D primitives. There are two variations of sort-last architectures: fragment-sorting architectures distribute fragments generated by each primitive to the appropriate compositor while image-composition architectures distribute the post-rendering pixels of a framebuffer as part of the display process. Finally, the Pomegranate architecture, a multi-sort architecture, performs all-to-all communication at multiple stages of the graphics pipeline in order to increase parallel efficiency.

When we classify architectures according to its sorting methodology, it is important to understand how the framebuffer is divided amongst the various graphics nodes because this division determines how the sort must occur. First, the framebuffer partitioning can be either static or dynamic. While a static partitioning is typically simpler to implement, a dynamic partitioning can be desirable because the architecture is able to use scene-dependent knowledge to minimize redundant work and load imbalance. Second, the partitioning can be regular or irregular. A regular partitioning could consist of equally sized rectangular tiles while an irregular partitioning would use rectangular tiles of various sizes. On the extreme end of irregularity, a system could even use non-rectangular regions. Third, the granularity of partitioning could be fine-grained or coarse-grained. On one end of the scale, each region in the partitioning consists of only one or a few pixels. On the other end of the scale, each region in the partition contains so many pixels that each node is responsible for only one or a few regions. Fourth, the assignment of the regions could follow various algorithms. A graphics system could use a regular assignment algorithm (e.g., round-robin distribution), a random assignment algorithm, or even an assignment algorithm that distributes regions in an irregular pattern that results in one node being responsible for more pixels than other nodes. Finally, the assignment of pixels to nodes can be either shared (more than one node for a pixel) or disjoint (one node for each pixel). Some graphics architectures lend themselves more to certain framebuffer partitioning algorithms, leading to various parallel efficiency issues.

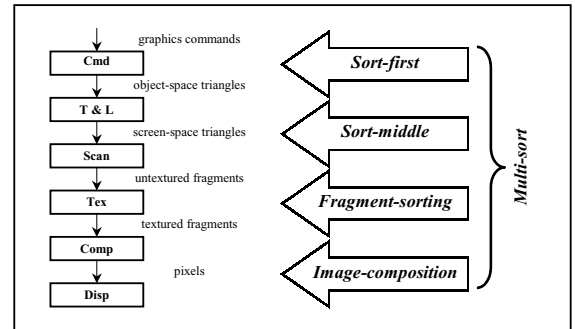


Figure 2.1: The Sorting Classification

A graphics architecture may be defined by the stage in the pipeline at which a sort, or all-to-all communication, occurs.

nication mechanism necessary for supporting such a division of labor should be minimal, and the system should not compromise any of the semantical requirements of the graphics interface (mode, ordering, frame). Finally, the complexity of the algorithm should be low. In this section, we examine how graphics architectures have made tradeoffs in these various requirements, resulting in various degrees of scalability according to the metrics set forth in Section 2.2.1.

One taxonomy [Molnar et al. 1994] classifies the parallel architectures by the place where a "sort" occurs. Primitives that begin on one node of a parallel machine must have their effects displayed on a particular location on the display, and an all-to-all communication based on image space location must occur at some point in the parallel graphics pipeline (i.e., a sort). In this section, we present a modified version of this taxonomy, illustrated in Figure 2.1. As a general point of reference, we will assume that the architectures are scaled by replicating several graphics "nodes", where each node is an individual graphics pipeline consisting of a command unit, a geometry unit, a rasterization unit, a

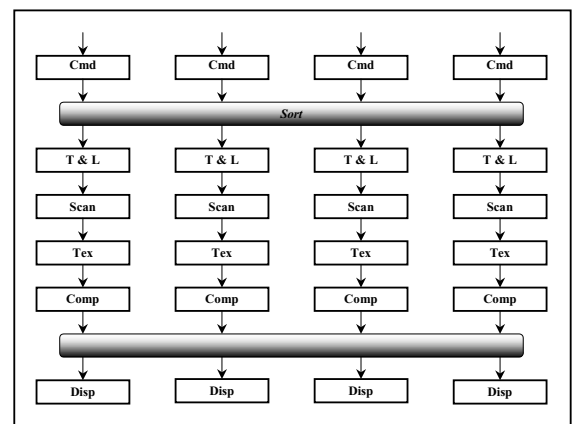


Figure 2.2: Sort-First Architectures

2.2.2.1 Sort-First Architectures

In a sort-first architecture, the command unit interprets the graphics commands and performs a small amount of computation to determine where the resulting primitive will fall in the framebuffer. Based on this, the primitive is transferred to the appropriate graphics node or graphics nodes where it goes through all of the rest of the stages of the graphics pipeline, including lighting, transformation, rasterization, texturing, and composition. These graphics pipelines unite again at the display stage: the disjoint regions of the framebuffer are combined to form a uniform display. One of the largest advantages of sort-first architectures is that it may be constructed by adding a simple communications infrastructure to a standard graphics pipelines with little additional modifications. In order to minimize the number of pipelines to which each primitive is sorted (and hence, redundant work), nearly all sort-first architectures utilize a coarse-grained partitioning of

the framebuffer. Additionally, in order to minimize the amount of computation necessary for classifying primitives, the cost of computing overlap is usually amortized over groups of several primitives (e.g., their bounding box). Many sort-first architectures have been described in the literature, and we will look at a few representative ones.

Mueller [Mueller 1995] makes the case for sort-first architectures and examines load balancing issues in such architectures. A sort-first architecture is outlined by this paper, although no actual architecture is implemented. While strict ordering is supported, only retained-mode interface semantics are supported, and frame semantics are imposed. By doing so, the system may take advantage of frame-to-frame coherence: an initial distribution of primitives is made according to the screen-space subdivision, and for each new frame, a small fraction of the primitives are redistributed according to the new scene parameters. This makes the input rate of such a system scale, albeit with the caveats of limited interface semantics and ungraceful degradation when the assumption of frame-to-frame coherence is violated. This can happen when a new object pops into the view frustum, or when the viewpoint rotation rate is too high to make use of frame-to-frame coherence (in interactive applications, this is a feed-forward loop that becomes worse and worse once it occurs). To avoid this, the screen needs to be subdivided into large regions. However, this has an adverse effect on load balancing to achieve scalable triangle and pixel rates. A static algorithm and an adaptive algorithm are compared to determine that a maximum-to-average triangle load balance ratio of 1.5 or less can be achieved with 9–25 static regions per processor or 1 adaptive region per processor. The adaptive algorithm made very good use of frame-to-frame coherence and required very little communication of primitives in its sort. Unfortunately, this work does not consider pixel load imbalance, nor does it consider how the display system of an adaptively subdivided screen may be built and load balanced, particularly in a scalable fashion. Though the paper mentions the ability of sort-first to handle large display resolutions, no scalable texture system is described to address the increased texture resolution necessitated by such a large display, and texture load imbalance is not explored.

to a sort-first architecture. Second, the architecture virtualizes the display system of each server by utilizing a fully scalable display system: Lightning2. Lightning2 takes in multiple Digital Visual Interface (DVI) streams that are output from each graphics card, reshuffle the data in these streams in a flexible way, and output many DVI streams that can subsequently be connected to a commodity display device. This efficient, display-speed pixel redistribution scheme allows for a high degree of scalability in triangle rate and pixel rate by utilizing a medium-grained, static partitioning of the screen, limited mainly by overlap. Such a medium-grained partitioning is particularly important in immediate-mode rendering systems because of temporal load imbalance effects—applications usually submit primitives in object-order, resulting in a pattern that draws many small triangles in a small region of the screen before moving on to another portion of the screen. Furthermore, as demonstrated in Chapter 4, such a partitioning load balances texture accesses well. Additionally, a client lazily sends texture data to a server only when required, allowing for some scalability in texture memory size. Unfortunately, as the tiling of the screen partitioning is made smaller for better rendering load balancing, lazy texture updates become less effective.

2.2.2.2 Sort-Middle Architectures

In sort-middle architectures, as with sort-first architectures, each node is responsible for a fraction of the framebuffer. When commands enter the pipeline, they are converted into 3D primitives that are subsequently light and transformed into 2D screen-space primitives. Each of these 2D primitives is then sorted to the appropriate rasterization nodes based on the screen-space subdivision. Here, texturing and composition occur. A display system then combines this disjoint framebuffer into a unified display. While most commercial systems have used a finely interleaved framebuffer [Akeley & Jermoluk 1988, Akeley 1993, Deering & Nelson 1993, Montrym et al. 1997] because of better pixel load balancing and a simple ordering mechanism, several research systems have examined the advantages of a coarse-grained subdivision scheme in both a static and an adaptive setting. Here, we look at the scaling properties of three sort-middle systems.

Samanta et al. [Samanta et al. 1999] examine the use of sort-first as an architecture for multi-projector systems. In this system, the focus is on providing an end-to-end rendering system that leverages commodity PC processors, commodity graphics cards, and a commodity network. A single client machine controls the rendering performed on eight server machines, each of which renders a single projector in a multi-projector display wall. By using a sort-first architecture that tiles the display on a per-graphics-card basis, the system is able to use unmodified graphics cards that were never intended for scalability. The interface for this system provides strict ordering, imposes frame semantics, and requires retained-mode semantics. The scene is described by a static scene graph hierarchy whose nodes contain approximately 100 polygons each. At each frame, the client machine computes potential visibility for nodes and sends the appropriate node ID tags to each server based on overlap. While the client machine is a point of serialization for this system, the amortization of a rendering instruction over 100 polygons provides significant leeway. However, the effects of a dynamic scene are unclear. In addition to a static algorithm, three adaptive load-balancing algorithms are examined—the system allows pixels rendered on one machine to be moved to another machine for final display. The scenes used for the study were all geometry-bound, so no conclusions are made regarding scalability in pixel rate, nor is texturing scalability addressed. The system is able to scale triangle rate with a parallel efficiency of 0.33 to 0.76 on eight PCs on a variety of scenes that tax the system by including frames that place the entire model on one projector. Of course, because each server machine adds a new display subsystem, the architecture’s display scales extremely well.

WireGL is a similar sort-first architecture based on PCs [Buck et al. 2000]. Again, this architecture leverages the use of unmodified graphics cards at commodity price points through standardized interfaces. This system is unique in two significant ways. First, rather than using a retained-mode interface based on a scene hierarchy, the architecture uses standard, unmodified OpenGL as its graphics interface. In order to overcome interface serialization, the parallel graphics interface described in Chapter 3 is implemented across many client machines, demonstrating the adaptability of such an interface

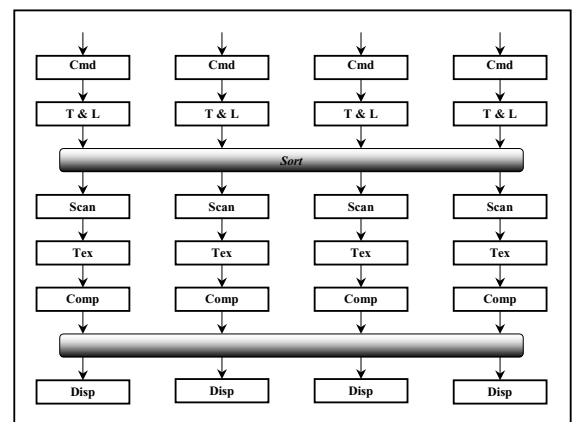


Figure 2.3: Sort-Middle Architectures

Pixel-Planes 5 is a scalable sort-middle architecture that utilizes a unique SIMD architecture for rasterization. The system consists of several heterogeneous units connected together by a network. A host interface creates and modifies the scene database through a retained-mode, frame-oriented API that provides strict ordering. Upon specification of the view parameters, several geometry nodes transform and light their fraction of the scene database. The screen is subdivided into coarse-grained virtualized tiles, the resulting 2D primitives are grouped according to the tiles they intersect, and a work queue of tiles is created. Then, each rasterization node takes a tile of work from the work queue, and performs scan conversion, texturing, and composition. Once the work of a tile is completed, the pixels of the resulting tile are transferred to a display node. In this architecture, the input rate is limited by a single host interface, but the retained-mode interface semantics keeps input requirements to a minimum. The triangle rate scales relatively

well because of three reasons. First, transform and lighting are well distributed across the geometry nodes. Second, the adaptive work queue algorithm load balances triangle work among the rasterizers. Third, the coarse-grained tiles have a low overlap factor, so the per-triangle work is not repeated across many tiles. At moderate levels of parallelism (10 or so rasterizers), the rasterizers have a parallel efficiency of approximately 0.7 when using 128 x 128 tiles. The pixel rate scales similarly to the triangle rate in this system, but the display was not built to be scalable, although it could be. The Pixel-Planes 5 architecture uses a non-scalable ring network that limits the maximum aggregate traffic that is available on the system, effectively placing a scalability limit on the triangle and pixel rates as well as the display.

The RealityEngine [Akeley 1993] is a sort-middle architecture that is representative of most commercial graphics systems. The interface unit receives commands from a host processor and distributes the resulting primitives among several geometry processors that perform transformation and lighting. The resulting 2D primitives are broadcast across a bus to the rasterization units in the order specified by the interface. Each rasterization unit is responsible for the scan conversion, texturing, and composition of a finely interleaved fraction of the framebuffer. The resulting pixels are then transferred to a display unit for final display. This architecture supports a strictly ordered, immediate-mode, single buffered interface. Because of the single interface unit, the input rate cannot scale. The triangle rate, on the other hand, can scale by a limited amount: as more geometry processors are added, more triangles may be light and transformed. However, because of the fine interleaving of the framebuffer across the rasterizers, every rasterizer must receive and process every primitive. This is problematic because of two reasons. First, broadcast communication does not scale well, whether it is implemented as a bus with electrical load limits or as a point-to-point network with quadratic growth characteristics. Second, because every rasterizer must receive and process every primitive, the overall system can never scale beyond a single rasterization unit's triangle rate. This fine interleaving does have a large advantage, however: the pixel rate scales linearly even in the face of temporal effects. This fine interleaving also has a negative effect on texture scal-

ability—in Chapter 4, we will see how a finely interleaved framebuffer causes excessive texture bandwidth requirements. Furthermore, in this architecture, texture memory is broadcast and replicated across the rasterizers, leading to no texture memory scalability. The display system of this architecture is also non-scalable.

Argus [Igehy et al. 1998b] is a sort-middle architecture that provides a large degree of scalability by subdividing the framebuffer into coarse-grained tiles while retaining an immediate-mode, strictly ordered interface with no frame semantics. Although Argus is a software system that runs on shared memory multiprocessors, many of its principles may be applied to hardware systems. Argus supports a scalable interface by implementing the parallel graphics API described in Chapter 3. Each of several interface units distributes blocks of 3D primitives to geometry units in a demand-driven fashion. These primitives are lit and transformed, and the resulting 2D primitives are placed in the appropriate tiles' reorder buffers. Each rasterizer performs scan conversion, texturing, and composition for each of its tiles, and a dynamic stealing algorithm load balances the triangle and pixel work both spatially and temporally. Display units then reassemble the tiles for final display. Unlike sort-middle schemes that use a fine framebuffer interleaving, the triangle rate scales well because each triangle is distributed to only the tiles it overlaps. Large triangles that overlap many tiles do not limit scalability because those triangles are limited by pixel work, a phenomenon described by Chen et al. [Chen et al. 1998]. The load balancing algorithm provides reasonable scaling in both pixel and triangle rates for moderate levels of parallelism. The texture bandwidth demands also scale well because of the large tile sizes, as shown in Chapter 4. Because the underlying communication mechanism is a scalable shared memory system [Laudon & Lenoski 1997], all the communication in the system scales well. Shared memory is particularly useful in the trivial implementation of a scalable texture memory and a scalable display memory: the process of accessing texture and display data on another node is provided through a fast, simple mechanism.

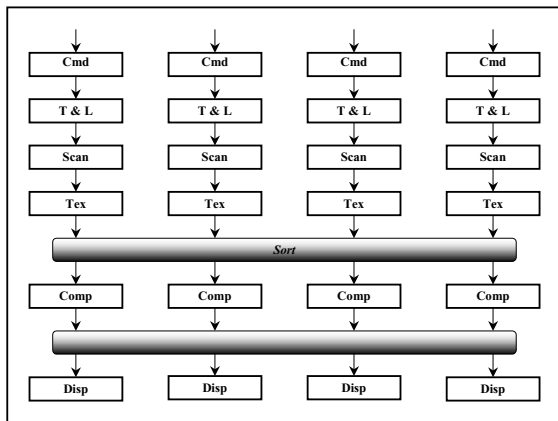


Figure 2.4: Fragment-Sorting Architectures

2.2.2.3 Fragment-sorting Architectures

In fragment-sorting architectures, each graphics pipeline is responsible for transforming, lighting, rasterizing, and texturing a fraction of the primitives in the scene. The textured fragments generated by each node are then sorted by an all-to-all network based on the image-space subdivision. These fragments are then composited into the framebuffer, and the display is generated by combining the pixels in each node's framebuffer. In order to ensure a highly balanced amount of pixel work, the framebuffer partitioning is usually very fine grained. As we will see in Chapter 4, unlike sort-middle, a fine-grained partitioning of the framebuffer does not increase texture bandwidth as the system scales because texturing occurs in object space independent of the framebuffer partitioning. Additionally, a parallel graphics interface may be utilized with a fragment-sorting architecture, allowing for scalable input rate. Unlike sort-first and sort-middle architectures, linearly

scaling the triangle rate in a fragment-sorting architecture is trivial. By distributing triangles among pipelines in a round-robin fashion, each triangle is processed only once, and each pipeline transforms, lights, and performs rasterization setup on an equal number of triangles. Scaling the pixel rate, however, is much more difficult. After transformation to image space, each triangle can generate anywhere from a single fragment to millions of fragments, and this information is unknown a priori. Thus, load imbalance leads to poor scalability in pixel rate. Adaptive load balancing schemes are difficult to implement given strict ordering requirements. Although a few fragment-sorting architectures have been implemented [Evans & Sutherland 1992, Kubota 1993], little technical detail is available on such systems. As with the interface and texture memory, although fragment-sorting can inherently support a scalable display system, no such systems have been built.

2.2.2.4 Image-Composition Architectures

Unlike the other architectures discussed thus far, image-composition architectures do not

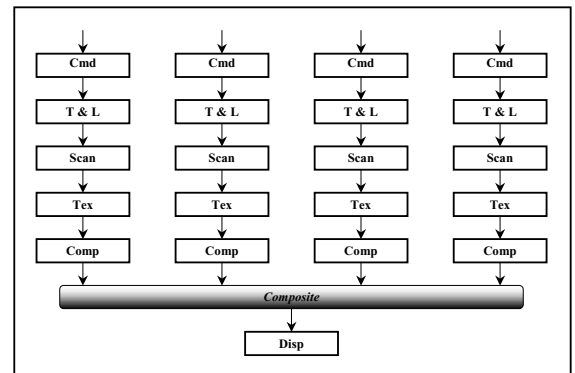


Figure 2.5: Image-Composition Architectures

force disjoint regions of the framebuffer to be allocated to only one node. Instead, each node draws into its own framebuffer (which may be as big as the display), and the framebuffers from each display are merged together through composition of the pixels. Typically, a depth comparison is done on pixels from different nodes to determine the visible pixel. As with sort-first architectures, a large advantage of image-composition architectures is the ability to use relatively unmodified graphics pipelines for scalability. Besides an interface network that allows distribution of primitives (which is usually provided by the host system), only a composition network needs to be built. Because pixel-to-pixel dependencies inherently cannot be satisfied by image-composition architectures, these systems break strict ordering semantics required by common graphics interfaces such as OpenGL. For many scenes, however, depth buffering alleviates the need for strict ordering.

PixelFlow [Molnar et al. 1992, Eyles et al. 1997] is an architecture that combines a rasterization architecture similar to Pixel-Planes 5 with deferred shading and an image-composition network to provide scalable rendering of scenes with advanced shading. In this system, each geometry transforms a portion of the scene and sorts them by coarse-grained tiles on the screen. Each rasterizer then scan converts, textures, and composites its resulting primitives for a small number of tiles, and then the pixels of this region are merged over a composition network into a shading module that performs lighting. Eventually, the regions are amassed into a framebuffer for display. The interface to this system can be immediate-mode or retained-mode, although frame semantics are required. The triangle rate scales linearly because primitives may be distributed round-robin and each primitive is handled only once. As with fragment-sorting architectures, pixel load imbalance can severely inhibit scalability in pixel rate for image-composition architectures if one node receives triangles that are much larger than other nodes. Because of the SIMD rasterization architecture used in PixelFlow, this imbalance is negligible. The rasterization node is able to rasterize a primitive covering an entire tile just as fast as a primitive covering a single pixel. Such an architecture would be unrealistic, however, in systems where framebuffer bandwidth is at a premium. The large-grained partitioning of

texture work balances texture bandwidth well, and a limited amount of scalability in texture memory can be achieved by only downloading texture data to shading nodes that require it, though no such algorithm is described in the literature. The display of this architecture is completely un-scalable due to the fact that doubling the display size doubles the amount of memory on each node's framebuffer as well as the rate at which it must be able to perform image-composition.

VC-1 [Nishimura & Kunii 1996] is an image-composition architecture that virtualizes the framebuffer. In this system, each node can hold only a fraction of the full framebuffer in a virtualized tile fashion. As rendering proceeds, a node eventually exhausts all the free tiles in its local framebuffer memory. At this point, some tiles are composited with the global framebuffer. This process continues until an entire frame is assembled for display. Additionally, because larger polygons take longer times to rasterize, it is important to load balance pixel work. To address this, polygons that are deemed to be too large by a geometry processor are broadcast to all the geometry processors. Although this algorithm breaks strict ordering constraints, it is irrelevant because VC-1 is an image-composition architecture that does not support strict ordering in any case. Although such a virtualized system reduces the amount of framebuffer memory required by each node in a scalable system, the display system does not scale because of the fixed bandwidth available on the image-composition network into the framebuffer.

2.2.2.5 Pomegranate Architecture

The Pomegranate architecture [Eldridge et al. 2000] is a novel architecture that maximizes parallel efficiency across all five scalability metrics while retaining a strictly ordered, immediate-mode API with no frame semantics. This is accomplished by treating the sort in graphics architectures not only as a means of transferring data to its final location, but also as a way of load balancing the work in a way that does not incur penalties for repeated work.

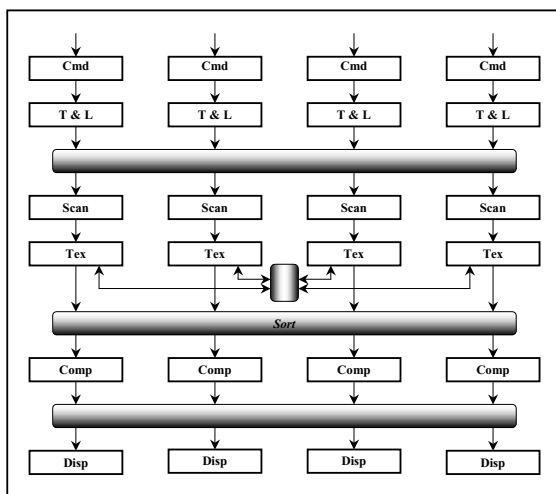


Figure 2.6: Pomegranate Architecture

Pomegranate is driven by the parallel graphics interface described in Chapter 3 to allow linear scalability of input bandwidth. Each interface unit is directly connected to a single geometry unit that transforms and lights all the primitives from an interface. Because the triangle rate of modern graphics systems are limited by the input rate of a single interface, there is no need to redistribute triangles from a single interface to multiple geometry units, although such an algorithm could be added. Each geometry unit then distributes groups of triangles in round-robin fashion among rasterization units that are responsible for scan conversion and texture mapping. Large triangles are tiled into smaller pieces by the geometry units and treated as separate triangles ensure that the pixel work in the round-robin distribution is load balanced. This all-to-all sort is not dependent on

any screen-space subdivision because, like fragment-sorting architectures, scan conversion is decoupled from composition into the framebuffer. Thus, each triangle is communicated and processed exactly once by a rasterizer, and because of the round-robin distribution, each rasterization unit receives a balanced number of triangles from each geometry unit that is fixed in sized regardless of the level of parallelism employed. This scales triangle rate. Each texturing unit may access texture memory attached to any other texturing unit, requiring an all-to-all sort. This scales the amount of texture memory linearly. By distributing texture memory across the units in a finely interleaved fashion, texture bandwidth from each texture memory is load balanced equally to each texturing unit. Because the distribution of primitives from geometry units to rasterization units is done in groups of triangles, object-space coherence keeps texture bandwidth requirements low (see Chapter 4). After texturing, the fragments are put through another all-to-all sort to place the them on the correct composition unit that merges fragments with the framebuffer. The fine-grained interleaving of the framebuffer as well as the load balancing of triangle sizes by the geometry unit distribution algorithm allows for linear scalability in pixel rate. Finally, each node's display processor fetches pixels for display refresh through an all-to-all sort with the composition units. Because each node linearly increases the amount of framebuffer memory available, and because a display's pixels are finely interleaved across the compositors in a way that balances refresh bandwidth requirements perfectly, a linearly scalable display system is provided. The several all-to-all sorts in Pomegranate are supported by a single scalable butterfly network. Because the architecture balances communications requests very well, even over short periods of time, and because the algorithms employed are latency-tolerant (e.g., the texture prefetching architecture of Section 4.1), such a network works extremely well. The system is thus able to scale performance linearly to large levels of parallelism, with a network that grows $O(n \log(n))$.

2.3 Conclusion

Large amounts of parallelism are available in graphics architectures. The relative lack of dependencies in the graphics API is a direct source of this parallelism. Even though the computation is very specialized, broad sets of implementation choices (classified by the sort taxonomy) are available to exploit this parallelism. In doing so, one must scale all parts of the graphics system in order to achieve full scalability. The rest of this dissertation focus on techniques for scaling two particular aspects of graphics architectures: interface and texture.

Chapter 3

Scalable Graphics Interface

It is increasingly difficult to drive a modern high-performance graphics system at full speed with a serial immediate-mode graphics interface. To resolve this problem, retained-mode constructs are integrated into graphics interfaces. While retained-mode constructs provide a good solution in many cases, at times they provide an undesirable interface model for the application programmer, and in some cases they do not solve the performance problem. In order to resolve these problems, this chapter presents a parallel graphics interface that may be used in conjunction with the existing API as a new paradigm for high-performance graphics applications. In a sense, we add thread-level parallelism on top of the instruction-level parallelism described in Section 2.1.

The parallel API extends existing ideas found in OpenGL and X11 that allow multiple graphics contexts to simultaneously draw into the same image. Through the introduction of synchronization primitives, the parallel API allows parallel traversal of an explicitly ordered scene. We give code examples that demonstrate how the API can be used to expose parallelism while retaining many of the desirable features of serial immediate-mode programming. The viability of the API is demonstrated by the performance of a software implementation that achieves scalable performance on a 24 processor system and a simulated hardware implementation that achieves scalable performance on a 64 processor system.

49

3.1 Introduction

Computer graphics hardware is rapidly increasing in performance. This has motivated immediate-mode graphics interfaces like OpenGL [Segal & Akeley 1992, Neider et al. 1993] to adopt constructs such as display lists and packed vertex arrays in order to alleviate system bottlenecks. However, these constructs may impose an undesired paradigm shift for the application programmer, and they may not be useful in resolving the particular performance bottleneck. Furthermore, with the increasing use of multiprocessor systems for graphics applications, a serial interface to the graphics system can be inelegant. A parallel graphics interface seeks to resolve these issues.

There are many challenges to designing a good parallel graphics interface; in formulating our design, we had several goals in mind. First and foremost were the ability to issue graphics primitives in parallel and the ability to explicitly constrain the ordering of these primitives. Ideally, the API should allow parallel issue of a set of primitives that need to be drawn in an exact order. The parallel API should be a minimal set of extensions to an immediate-mode interface such as OpenGL, and it should be compatible with existing features such as display lists. The design is constrained by the presence of state; this is required for a large feature set. A well designed parallel interface should be intuitive and useful in a wide variety of applications. And finally, the new API should extend the current framework of graphics architectures to provide a rich set of implementation choices. In the rest of this chapter, we present the motivations and issues involved in designing a parallel extension to a serial immediate-mode graphics interface with strict ordering and state. By adding synchronization commands (such as barriers and semaphores) into multiple graphics command streams, application threads can issue explicitly ordered primitives in parallel without blocking.

3.2 Motivation

Although graphics systems are on the same technology curve as microprocessors, graphics systems have reached a level of performance at which they can process graphics commands faster than microprocessors can produce them: a single CPU running an immediate-mode interface cannot keep up with modern graphics hardware. This is primarily due to an increasing use of parallelism within graphics hardware. Within a computer, there are three sources of bottlenecks in a graphics application. First, performance may be limited by the speed of the graphics system. In this case, the only solution is to use a faster graphics system. Second, performance may be limited by the rate of data generation. In this case, the programmer can use either a faster data generation algorithm or else, if the algorithm is parallelizable, multiple processors. Third, performance may be limited by the interface between the host system and the graphics system. Possible sources of this limitation are:

- 1) Overhead for encoding API commands.
- 2) Data bandwidth from the API host.
- 3) Data bandwidth into the graphics system.
- 4) Overhead for decoding API commands.

There are several possible ways to extend a serial immediate-mode API in order to address the interface bottlenecks:

- **Packed Primitive Arrays.** A packed primitive array is an array of primitives that reside in system memory. By using a single API call to issue the entire array of primitives instead of one API call per primitive, the cost of encoding API commands is amortized. Furthermore, because the arrays may be transferred by direct memory access (DMA), bandwidth limitations from the API processor may be bypassed. Nothing is done, however, about the bandwidth limitations into the graphics system. Furthermore, although the decoding may be somewhat simplified, all the primitives in the array still have to be decoded

on the graphics system. While packed primitive arrays are useful in a wide variety of applications, they may introduce an awkward programming model.

- **Display Lists.** A display list is a compiled set of graphics commands that resides on the graphics system. In a fashion similar to retained-mode interfaces, the user first specifies the list of commands to be stored in the display list and later invokes the commands within the display list. Because they are essentially command macros, display lists work well semantically with immediate-mode interfaces. In cases where the scene is small enough to fit in the graphics system and the frame-to-frame scene changes are modest, display lists trivially resolve the first three bottlenecks. If the scene is too large and therefore must reside in system memory, display lists are similar to packed primitive arrays and only the first two bottlenecks are resolved. Display lists provide an excellent solution for performance bottlenecks if the same objects are drawn from frame to frame. But on applications that re-compute the graphics data on every frame (e.g., [Hoppe 1997, Sederberg & Parry 1986]), display lists are not useful. Furthermore, the use of display lists burdens the programmer with the task of managing handles to the display lists.
- **Compression.** Whereas the idea of quantizing the data sent through the API has been used for quite some time, the idea of compressing the data has only recently been proposed. One system compresses the geometric data sent through the API [Deering 1995]; other systems compress the texture data [Beers et al. 1996, Torborg & Kajiya 1996]. All compression schemes increase the decoding costs, and systems that compress the data interactively increase the encoding costs. Systems that compress the data off-line, on the other hand, are useful only when the graphics data does not change.
- **Parallel Interface.** The motivation behind a parallel graphics interface is scalability: bottlenecks are overcome with increased parallelism. If the graphics system is too slow, it can be scaled by adding more graphics nodes. If the

processed one at a time by the X server. Though it is not explicitly encouraged, multiple clients may draw into the same drawable or even use the same graphics context.

While a 3D graphics interface was beyond the scope of the original design of X, OpenGL is a 3D interface that has been coupled with X. OpenGL is an immediate-mode interface whose state is kept within an X resource called the GLX context. In the interest of efficiency, both display lists and packed primitive arrays are supported. Furthermore, both texture data and display lists may be shared between contexts in order to allow the efficient sharing of hardware resources amongst related contexts [Kilgard 1996].

Strict ordering semantics are enforced in X and OpenGL: from the point of view of the API, every command appears to be executed once the API call returns. However, in the interest of efficiency, both interfaces allow implementations to indefinitely buffer commands. This introduces the need for two types of API calls. Upon return from the *flush* call (*XFlush*, *glFlush*), the system guarantees that all previous commands will execute in a finite amount of time from the point of view of the drawable. Upon return from a *finish* call (*XSync*, *glFinish*), the system guarantees that all previous commands have been executed from the point of view of the drawable.

Since OpenGL and X solve different problems, programs often use both. Because of buffering, however, a program must synchronize the operations of the two streams. Imagine a program that wants to draw a 3D scene with OpenGL and then place text on top of it with X. It is insufficient to simply make the drawing calls in the right order because commands do not execute immediately. Furthermore, a flush is insufficient because it only guarantees eventual execution. A finish, on the other hand, guarantees the right order by forcing the application to wait for the OpenGL commands to execute before issuing X commands. In a sense, however, the finish is too much: the application need not wait for the actual execution of the OpenGL commands; it only needs a guarantee that all prior OpenGL commands execute before any subsequent X commands. The call `glXWaitGL` provides this guarantee, and `glXWaitX` provides the complement.

data generation is too slow, more processors can be used to generate the data in parallel. Similarly, if the serial interface is too slow, then it should be parallelized. In a system with a single graphics port, a parallel API can be used to overcome the first two interface limitations. However, by building a scalable system with multiple graphics ports, all interface limitations can be overcome. This is the solution proposed in this chapter.

3.3 Related Work

In the field of parallel graphics interfaces, Crockett introduced the Parallel Graphics Library (PGL) for use in visualizing 3D graphics data produced by message-passing supercomputers [Crockett 1994]. Due to the characteristics of its target architecture and target applications, PGL was designed as a retained-mode interface. In parallel, each processor adds objects to a scene by passing pointers to graphics data residing in system memory. A separate command is used to render the objects into a framebuffer, and no ordering constraints are imposed by the interface. PixelFlow [Molnar et al. 1992, Eyles et al. 1997] is another system designed to support multiple simultaneous inputs from a parallel host machine, and PixelFlow OpenGL includes extensions for this purpose. However, due to the underlying image composition architecture, PixelFlow OpenGL also imposes frame semantics and does not support ordering. Because of these constraints, PGL and PixelFlow OpenGL do not meet the requirements of many graphics applications.

The X11 window system provides a parallel 2D graphics interface [Scheifler & Gettys 1986, Gettys & Karlton 1990]. A client with the proper permissions may open a connection to an X server and ask for X resources to be allocated. Among these resources are drawables (which are on- or off-screen framebuffers) and X contexts (which hold graphics state). Since resources are globally visible, any client may subsequently use the resource within X commands. Since X drawing calls always include references to a drawable and an X context, client requests are simply inserted into a global queue and

Hardware implementations of OpenGL typically provide support for a single context, and sharing of the hardware is done through a context switch. Though context switches are typically inexpensive enough to allow multiple windows, they are expensive enough to discourage fine-grained sharing of the graphics hardware between application threads. A few architectures actually provide hardware support for multiple simultaneous contexts drawing into the same framebuffer [Kirkland 1998, Voorhies et al. 1988], but all commands must go through a single graphics port. Furthermore, these architectures do not have a mechanism for maintaining the parallel issue of graphics commands when an exact ordering of primitives is desired.

3.4 The Parallel API Extensions

While OpenGL within X is not intended for multithreaded use due to the underlying implementations, the interface provides mechanisms for having multiple application threads work simultaneously on the same image. In this section, we first demonstrate how an interface like OpenGL may be used to attain parallel issue of graphics commands. Then we show how additional extensions can be used to increase the performance of parallel issue. The specification of the API extensions is given in Figure 3.1.

The API extensions are most easily motivated through the use of an example. Suppose that we want to draw a 3D scene composed of opaque and transparent objects. Though depth buffering alleviates the need to draw the opaque primitives in any particular order, blending arithmetic requires that the transparent objects be drawn in back-to-front order after all the opaque objects have been drawn. By utilizing the strict ordering semantics of the serial graphics API, a serial program simply issues the primitives in the desired order. With a parallel API, order must be explicitly constrained. We assume the existence of two arrays, one holding opaque primitives and the other holding transparent primitives ordered in back-to-front order. We also assume the existence of the following function:

```
DrawPrimitives(prims(first..last))
for p = first..last
  glColor(&prims[p].color)
  glPrimitive(&prims[p].coord)
glFlush()
```

3.4.1 Existing Constructs

As a first attempt at parallel issue, imagine two application threads using the same context to draw into the same framebuffer. In such a situation, a “set current color” command intended for a primitive from one application thread could be used for a primitive from the other application thread. In general, the sharing of contexts between application threads provides unusable semantics because of the extensive use of state. By using separate contexts, dependencies between the state-modifying graphics commands of the two streams are trivially resolved. Given two application threads using separate contexts

```
glBarrierCreate(GLuint barrier, GLuint numCtxs)
  barrier->reset = numCtxs;
  barrier->count = numCtxs;

glBarrierExec(GLuint barrier)
  barrier->count--;
  if (barrier->count == 0)
    barrier->count = barrier->reset;
    signal(all waiting contexts);
  else
    wait();

glBarrierDelete(GLuint barrier)

glSemaphoreCreate(GLuint sema, GLuint initial)
  sema->count = initial;

glSemaphoreP(GLuint sema)
  if (sema->count == 0)
    wait();
  sema->count--;

glSemaphoreV(GLuint sema)
  sema->count++;
  signal(one waiting context, if any);

glSemaphoreDelete(GLuint sema)

glWaitContext(GLXContext ctx)
  All subsequent commands from the issuing context execute
  after all prior commands from ctx have finished execution.
```

Figure 3.1: The Parallel Graphics Interface Extensions

on the same framebuffer, the following code could be used to attain parallel issue of the opaque primitives:

```
Thread1
DrawPrimitives(opaq(1..256))

Thread2
DrawPrimitives(opaq(257..512))
glFinish()
appBarrier(appBarrierVar)

appBarrier(appBarrierVar)
DrawPrimitives(tran(1..256))
glFinish()
appBarrier(appBarrierVar)
DrawPrimitives(tran(257..512))
```

Both application threads first issue their share of opaque primitives without regard for order. After synchronizing in lock-step at the application barrier, *Thread1* issues its half of the transparent primitives. These transparent primitives are guaranteed to be drawn in back-to-front order after *Thread1*'s share of opaque primitives through strict ordering semantics. They are also guaranteed to be drawn after *Thread2*'s share of opaque primitives through the combination of the finish and the barrier; the finish is used to guarantee the drawing of all previously issued commands. Through this same synchronization mechanism, *Thread2*'s share of transparent primitives are then drawn in back-to-front order after *Thread1*'s share of transparent primitives.

3.4.2 The Wait Construct

One inefficiency in the above code is the use of the finish command; in a sense, it is too much. Synchronization between the application threads does not require the actual execution of the graphics commands; it only requires a guarantee on the order of execution between the two graphics streams. In a fashion similar to that used in synchronizing X and OpenGL, we introduce the *wait context* call in order to make guarantees about the execution of commands between contexts. We refer the reader to Figure 1.1 for an exact specification. In synchronization situations, the wait call is more efficient than the finish call because it does not require any application thread to wait for the completion of graphics commands. The following code demonstrates how the example scene may be drawn using the wait command:

```
Thread1
DrawPrimitives(opaq(1..256))
appBarrier(appBarrierVar)
glWaitContext(Thread2Ctx)
DrawPrimitives(tran(1..256))
appBarrier(appBarrierVar)

Thread2
DrawPrimitives(opaq(257..512))
appBarrier(appBarrierVar)

appBarrier(appBarrierVar)
glWaitContext(Thread1Ctx)
DrawPrimitives(tran(257..512))
```

3.4.3 Synchronization Constructs

While the wait command provides an improvement, large problems remain in the above solution: the synchronization of the graphics streams is done by the application threads. Consequently, application threads are forced to wait for graphics streams. Why should the application thread wait when it could be doing something more useful? For example in the above code, the first thread must issue its entire half of the transparent primitives before the second thread can begin issuing its half. Every time an explicit ordering is needed between primitives from different threads, the interface degrades to a serial solution.

The answer to this problem is the key idea of our parallel API: synchronization that is intended to synchronize graphics streams should be done between graphics streams, not between application threads. To this end, we introduce a graphics barrier command into the graphics API. As with other API calls, the application thread merely issues the barrier command, and the command is later executed within the graphics subsystem. Thus, the blocking associated with the barrier is done on graphics contexts, not on the application threads. The code below achieves our primary objective, the parallel issue of explicitly ordered primitives: both application threads may execute this code without ever blocking.

```
Thread1
DrawPrimitives(opaq(1..256))
glBarrierExec(glBarrierVar)
DrawPrimitives(tran(1..256))
glBarrierExec(glBarrierVar)

Thread2
DrawPrimitives(opaq(257..512))
glBarrierExec(glBarrierVar)

glBarrierExec(glBarrierVar)
DrawPrimitives(tran(257..512))
```

We see the utility of the barrier primitive in the above code example, but what other synchronization primitives provide useful semantics within the realm of a parallel graphics interface? The barrier is an excellent mechanism for synchronizing a set of streams in lock-step fashion; however, it is not the best mechanism for doing point-to-point synchronization. Borrowing from the field of concurrent programming, semaphores provide an elegant solution for many problems [Dijkstra 1968]. Among them is a mechanism for signal-and-wait semantics between multiple streams. The specification of the barrier and semaphore commands can be found in Figure 3.1.

Barriers and semaphores have been found to be good synchronization primitives in the applications we have considered. If found to be useful, other synchronization primitives can also be added to the API. It is important to note that the requirements for synchronization primitives within a graphics API are somewhat constrained. Because the expression of arbitrary computation through a graphics API is not feasible, a synchronization primitive's utility cannot rely on computation outside of its own set of predefined operations (as do condition variables). Also, we intentionally do not specify anything regarding the allocation of synchronization primitives, except to note that they need to be global resources at the level of contexts and drawables.

3.5 Using the Parallel Graphics API

The most obvious way to use the interface is to call it directly from a parallel application. For existing serial applications, the parallel graphics interface provides a new paradigm for high-performance command issue. For existing parallel applications, it also provides a natural interface to the graphics system. We present two examples that make direct use of the parallel API.

<pre> Serial loop: glClear() get user input compute & draw glXSwapBuffers() glFinish() </pre>	<pre> Master loop: glClear() get user input appBarrier(appBarrierVar) compute & draw glBarrierExec(glBarrierVar) glXSwapBuffers() glFinish() </pre>	<pre> Slave loop: appBarrier(appBarrierVar) glWaitContext(masterCtx) compute & draw glBarrier(glBarrierVar) </pre>
---	--	--

Figure 3.2: Parallelizing a Simple Interactive Loop

Application computation and rendering are parallelized across slave threads, with a master thread coordinating per-frame operations.

3.5.1 Simple Interactive Loop

Figure 3.2a shows a simple interactive loop expressed in a strictly ordered serial interface. The goal in this example is to parallelize the compute and draw stage, yielding improved performance in the application, the issue of the graphics commands, and the execution of the graphics commands.

For parallel issue, a master thread (Figure 3.2b) creates a number of slave threads (Figure 3.2c) to help with the compute and draw stage. The master first issues a clear command and gets the user input. The application barrier ensures that the worker threads use the correct user input data for the rendering of each frame. This synchronizes the application threads, but not the graphics command streams. The slaves issue wait commands to ensure that the clear command issued by the master is executed first. The master is assured that the clear occurs first due to the strict ordering semantics of a single stream. After each thread issues its graphics commands, a graphics barrier is issued to restrict the swap operation to occur only after all the graphics streams have finished drawing their share of the frame. Finally, a finish operation is needed to ensure that the image is completed and displayed before getting user input for the next frame. The finish

itself is a context-local operation, only guaranteeing that all of the previous commands issued by the master are complete. However, in conjunction with the graphics barrier, the finish guarantees that the commands of the slaves are also completed.

3.5.2 Marching Cubes

As a more demanding example, consider the marching cubes algorithm [Lorensen & Cline 1987]. Marching cubes is used to extract a polygonal approximation of an isosurface of a function sampled on a 3D grid. In this example, we will discuss a simplification to 2D for brevity. In Figure 3.3a, the mechanics of surface extraction and rendering are abstracted as *ExtractAndRender*. *ExtractAndRender* operates on a single cell of the grid independently of any other. If any portion of the desired isosurface lies within the cell, polygons approximating it are calculated and issued to the graphics system immediately. Note that a cell may consist of many voxels. Due to the grid structure, it is fairly simple to perform the traversal in back-to-front order based on the current viewpoint, eliminating the need for depth buffering and allowing for alpha-based translucency. In our example, this corresponds to traversing the grid in raster order.

Due to the independence of the processing of different cells, marching cubes is easily parallelized. In Figure 3.3b, traversal is parallelized by interleaving the cells of the volume across processing elements. Unfortunately, this simple approach sacrifices back-to-front ordering. Figure 3.3d illustrates the dependence relationships between cells and their neighbors that must be obeyed in the ordered drawing of primitives. These dependencies can be expressed directly using semaphores injected into the graphics command streams. An implementation is shown in Figure 3.3c. Before processing a cell, the owner thread issues two *P* operations to constrain the rendering of a cell to occur after rendering of its two rear neighbor cells. After processing the cell, it issues two *V* operations to signal the rendering of its other neighbors. Note that the dependencies and traversal order given here are non-ideal; another approach is to keep the same dependencies and submit cells back-to-front in order of increasing $(i + j)$.

<pre> MarchSerialOrdered (M, N, grid) for (i=0; i<M; i++) for (j=0; j<N; j++) ExtractAndRender(grid[i, j]) </pre>	<pre> MarchParallel (M, N, grid) for (i=0; i<M; i++) for (j=(myProc+i)%numProcs; j<N; j+=numProcs) ExtractAndRender(grid[i, j]) </pre>
<pre> MarchParallelOrdered (M, N, grid, sema) for (i=0; i<M; i++) for (j=(myProc+i)%numProcs; j<N; j+=numProcs) if (i>0) glSemaphoreP(sema[i-1, j]) if (j>0) glSemaphoreP(sema[i, j-1]) ExtractAndRender(grid[i, j]) if (i<M-1) glSemaphoreV(sema[i, j]) if (j<N-1) glSemaphoreV(sema[i, j]) </pre>	<pre> (d) </pre>

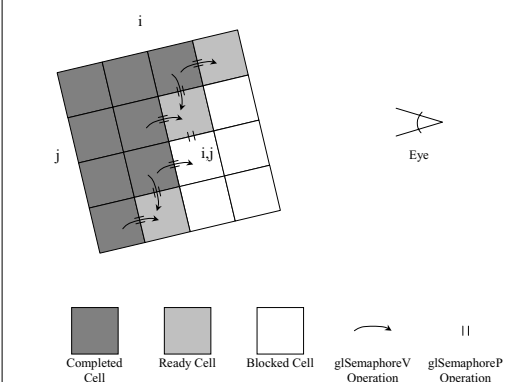


Figure 3.3: Parallel Marching Cubes Traversal

As rendering of cells completes, *glSemaphoreV* operations are performed by the graphics contexts to release dependent neighboring cells closer to the eye. Rendering of the white cells is still blocked on *glSemaphoreP* operations, waiting for rendering of their more distant neighbors.

3.6 Implementations

3.6.1 Argus: A Software Implementation

In order to test the viability of the parallel API extensions, we have implemented a software graphics library that is capable of handling multiple simultaneous graphics contexts. The name of this implementation is Argus, and the performance achieved with the parallel API using this system demonstrates the utility and feasibility of the ideas presented thus far.

3.6.1.1 Architecture

Argus is a shared memory multiprocessor graphics library that was designed to serve as a test-bed for various studies in graphics architecture. Argus implements a subset of OpenGL as well as the parallel API extensions. At the heart of Argus is a lightweight multiprocessor threads package. We implement a graphics architecture by allocating a thread for each processing node (e.g., geometry processor). A custom scheduler is used to schedule these threads onto system processors appropriately. Furthermore, if a system processor running application code is blocked for some reason due to the graphics (e.g., a buffer fills up or a *glFinish* is pending), the threads package will run graphics threads on the otherwise idle application processor.

There are three basic types of threads in the serial API version of Argus. An application thread runs application code and manages the graphics context. A geometry thread transforms and shades the primitives encoded in the graphics instruction stream. A rasterization thread is responsible for drawing these transformed primitives into the framebuffer. The version of Argus that implements the serial API is a sort-middle tiled parallel graphics system [Molnar et al. 1994]. Graphics commands from a single application thread fill a global command queue that is drained by many geometry threads. The number of geometry threads is scalable since the data in this global command queue can be read in parallel. Of course, the geometry threads must synchronize at a single point of

contention in order to distribute the work in the queue; however, because the contention is amortized over a large number of primitives, this cost is insignificant in our implementation. After the appropriate computation, the geometry threads distribute the transformed primitives among the appropriate tile rasterizers. Though the details are beyond the scope of this thesis, reorder buffers in front of each rasterizer are used to maintain the ordering found in the global command queue across the rasterizers. Since each tile rasterizer is responsible for a contiguous portion of the screen, no one rasterizer needs to see all of the primitives; thus, the rasterization architecture is scalable. Argus supports a variety of schemes for load balancing tile rasterization. For the results presented here, we used distributed task queues with stealing.

The version of Argus that implements the parallel API extends the serial API architecture to allow multiple simultaneous graphics streams. Each application thread is augmented by a local command queue and a synchronization thread. Instead of entering graphics commands onto the global command queue, each application thread fills its local command queue. The synchronization thread is then responsible for transferring commands from this local command queue onto the global command queue. Since the global command queue may be written in parallel, the architecture is scalable.

Figure 3.4 illustrates the pipeline in greater detail and explains how state management and synchronization commands are implemented within Argus. The pipeline contains several threads, shown as gray boxes, which communicate through a variety of queues. In this example, two application threads are drawing into the same framebuffer through two different contexts. The graphics data from the two contexts is shown with single- and double-underline type.

One key design issue that comes up in implementing the parallel API is the handling of the graphics state since most commands affect rendering through state changes. API commands are issued by the ‘App’ threads shown at the top of the diagram. Commands that modify state that is not necessary for the rendering of the current GL primitive (e.g., the bottom entries of the matrix stack) are tracked in the context state (e.g., CS). Commands that modify state that is necessary for rendering the current GL primitive (e.g., the

top entry of the matrix stack) are tracked in the current geometry state (e.g., GS₁), but old versions of the geometry state (e.g., GS₁) are kept until they are no longer needed by the rest of the pipeline. Commands that specify the current primitive (i.e., commands which are allowed within *glBegin* and *glEnd*, such as *glNormal* and *glVertex*) are grouped into fixed-size primitive blocks (denoted by P_i). A primitive block and its related geometry state contain all the information necessary for the rendering of the primitives, and multiple primitive blocks can share the same geometry state. For example, primitive blocks P₂ and P₃ both use the same geometry state GS₂. Every time a primitive block fills up or the geometry state changes, a pair of pointers (which are represented in the diagram by parentheses) is added to the local command queue (LCQ) by the ‘App’ thread; synchronization commands (Sema) are inserted into this queue directly.

Another key implementation design issue in any parallel API implementation is the merging of graphics streams and the resolution of synchronization commands. In Argus, each context has a ‘Sync’ thread which is responsible for moving data from its LCQ onto a global command queue (GCQ). ‘Sync’ threads execute the synchronization commands found in the LCQ (as illustrated by the dotted green line). When ‘Sync’ threads are not blocked due to synchronization, they copy the pointers from their LCQ onto the GCQ. This creates a sequence in the GCQ that is strictly ordered with respect to any one context and consistent with the constraints imposed by the synchronization commands. For example, the sequence found in the GCQ of the diagram keeps the order {P₁, P₂, P₃} and {P₁, P₂, P₃}. The sequence is also consistent with the semaphore pair (which requires an ordering that puts {P₁, P₂} in front of {P₂, P₃}).

Beyond the GCQ, the Argus pipeline is similar to a graphics pipeline that implements a serial API. The ‘Geom’ threads drain the GCQ and fill the triangle queue by converting the geometry state (GS_i) and the 3D data from primitive blocks (P_i) into rasterization state (RS_i) and 2D triangle blocks (T_i). Each ‘Rast’ thread is responsible for drawing into one tile of the framebuffer, and the ‘Geom’ threads insert pointers into the appropriate rasterization buffers based on the tiles that are overlapped by the triangles in the triangle block. These reorder buffers are used as a mechanism for maintaining ordering.

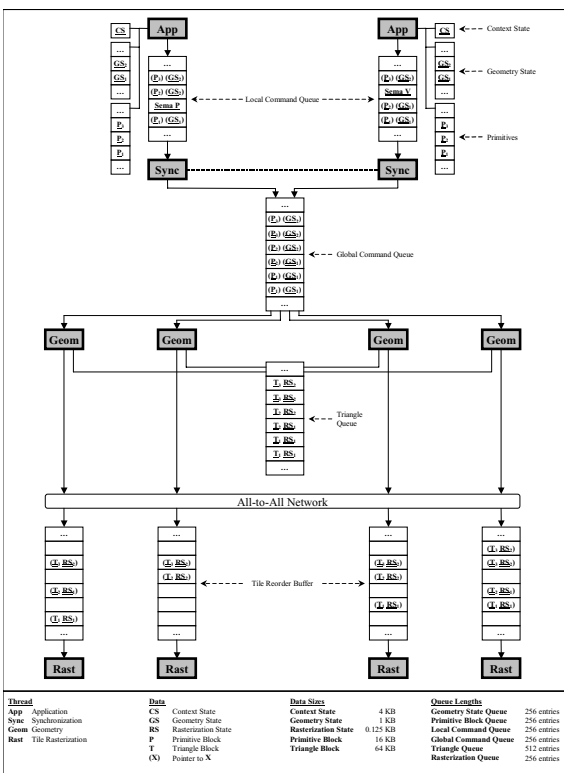


Figure 3.4: The Argus Pipeline

3.6.1.2 Performance

Because poor performance often hides architectural bottlenecks, Argus was designed with performance as one of its main criteria. Although Argus can run on many architectures, particular care was taken to optimize the library for the Silicon Graphics Origin system [Laudon & Lenoski 1997]. The Origin is composed of 195 MHz R10000 processors interconnected in a scalable NUMA architecture. Depending on the rendering parameters, the single processor version of Argus is able to render up to 200K triangles per second; this rendering rate scales up to 24 processors. In its original incarnation, Argus was designed for a serial interface and many serial applications were not able to keep up with the scalable performance of the graphics system. Remedying this situation led us to the development of the parallel API.

To study the performance of our parallel API implementation, we ran two applications: *Nurbs* and *March*. *Nurbs* is an immediate-mode patch tessellator parallelized by distributing the individual patches of a scene across processors in a round-robin manner. By tessellating patches on every frame, the application may vary the resolution of the patches interactively, and because depth buffering is enabled, no ordering constraints are imposed in the drawing of the patches—synchronization commands are utilized only on frame boundaries. Our second application, *March*, is a parallel implementation of the marching cubes algorithm [Lorenson & Cline 1987]. By extracting the isosurface on every frame, the application may choose the desired isosurfaces interactively. Rendering is performed in back-to-front order to allow transparency effects by issuing graphics semaphores that enforce the dependencies described in Section 3.5.2. One noteworthy difference between our implementation and the one outlined in Section 3.5.2 is that cells are distributed from a centralized task queue rather than in round-robin order because the amount of work in each cell can be highly unbalanced. The input characteristics and parameter settings used with each of these applications are shown below:

<i>Nurbs</i>	<i>March</i>
armadillo dataset	skull dataset
102 patches	256K voxels (64x64x64)
196 control points per patch	cell size at 16x16x16
117504 stripped triangles	53346 independent triangles
1200x1000 pixels	1200x1000 pixels

Figure 3.5a and Figure 3.5b show the processor speedup curves for *Nurbs* and *March*, respectively. The various lines in the graph represent different numbers of application threads. The serial application bottleneck can be seen in each case by the flattening of the "1 Context" curve: as more processors are utilized, no more performance is gained. Whereas the uniprocessor version of *Nurbs* attains 1.65 Hz, and the serial API version is limited to 8.8 Hz, the parallel API version is able to achieve 32.2 Hz by using four contexts. Similarly, the uniprocessor version of *March* gets 0.90 Hz, and the serial API version of *March* is limited to 6.3 Hz, but the parallel API version is able to attain 17.8 Hz by utilizing three contexts. These speedups show high processor utilization and highlight the implementation's ability to handle extra contexts gracefully.

One extension to Argus that we are considering is the use of commodity hardware for tile rasterization. Although this introduces many difficulties, it also increases rasterization rate significantly. In order to simulate the effects of faster rasterization on the viability of the parallel API, we stress the system by running Argus in a simulation mode that imitates infinite pixel fill rate. In this mode, the slope calculations for triangle setup do occur, as does the movement of the triangle data between the geometry processors and the tile rasterizers. Only the rasterization itself is skipped. The resulting system increases the throughput of Argus and stresses the parallel API: Figure 3.6a and Figure 3.6b show how a greater number of contexts are required to keep up with the faster rendering rate. The parallel API allows Argus to achieve peak frame rates of 50.5 Hz in *Nurbs* and 40.9 Hz in *March*. This corresponds to 5.9 million stripped triangles per second in *Nurbs* and 2.2 million independent triangles per second in *March*. These rates are approximately double the rate at which a single application thread can issue primitives into Argus even when *no* application computation is involved, thus demonstrating the importance of multiple input ports.

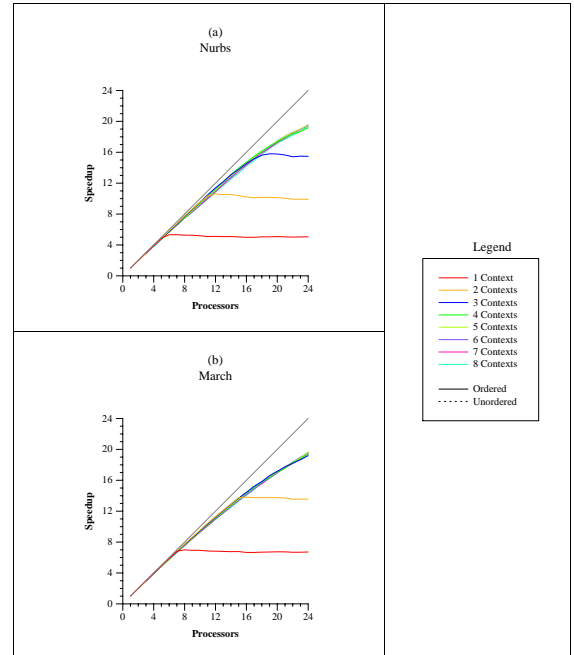


Figure 3.5: Argus Speedup Graphs
The speedup curves for two applications, *Nurbs* and *March*, are drawn in (a) and (b) for a varying numbers of contexts.

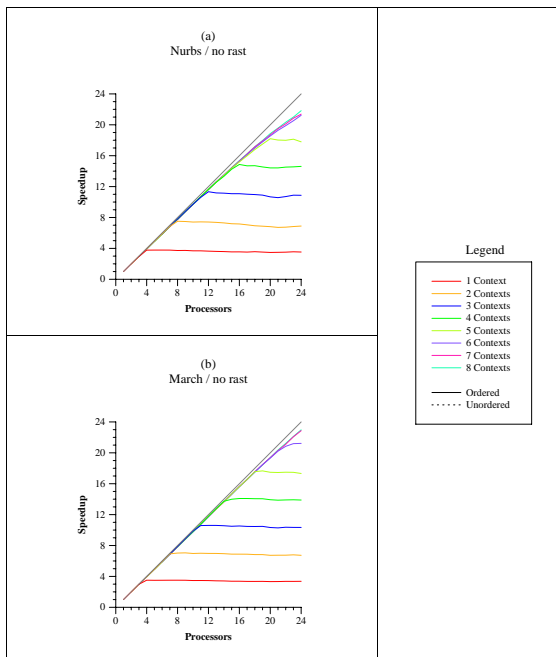


Figure 3.6: Argus Speedup Graphs without Rasterization
The speedup curves for *Nurbs* and *March* are drawn in (a) and (b) for a varying numbers of contexts assuming an infinite fill rate

One important aspect of any implementation of the parallel API is the amount of buffering required to make the API work. Without enough buffering, the parallel API serializes: in Argus, if a local command queue fills up before its synchronization commands are resolved, the application thread is forced to wait. Intuitively, we expect the amount of buffering required to be sensitive to the amount of synchronization between different threads. This is quantified in the speedup curves of Figure 3.7a for 24 processors. The number of entries in the local command queue (each can point to a 16 KB block of primitive commands or hold a single synchronization command) was varied from 1 to 256. The runs were performed on the *March* application with the semaphores both enabled (the solid "Ordered" lines) and disabled (the dotted "Unordered" lines). As one would expect, the ordered version requires significantly larger buffers.

Another key aspect of any parallel API implementation is its ability to minimize the cost of synchronization. If the granularity of the application is too fine, synchronization costs can dominate, and the application is forced to use a coarser subdivision of work. If the work is subdivided too coarsely, load imbalance can occur within the application. The effects of granularity on Argus were tested by varying the dimensions of the cells on both the ordered and unordered versions of *March*. The number of processors was held at 24 and timings were taken with varying numbers of contexts, as illustrated in Figure 3.7b. A granularity that is too fine deteriorates performance in both the application (as demonstrated by the unordered runs) as well as in the graphics system (as demonstrated by the extra performance hit taken by the ordered runs). For the *March* application, there is a wide range of granularities (well over an order of magnitude in the number of voxels) that work well since Argus was designed to keep the cost of synchronization low. When *March* is run without isosurface extraction and rendering (i.e., nothing but the synchronization primitives are issued), several hundred thousand semaphore operations are resolved per second.

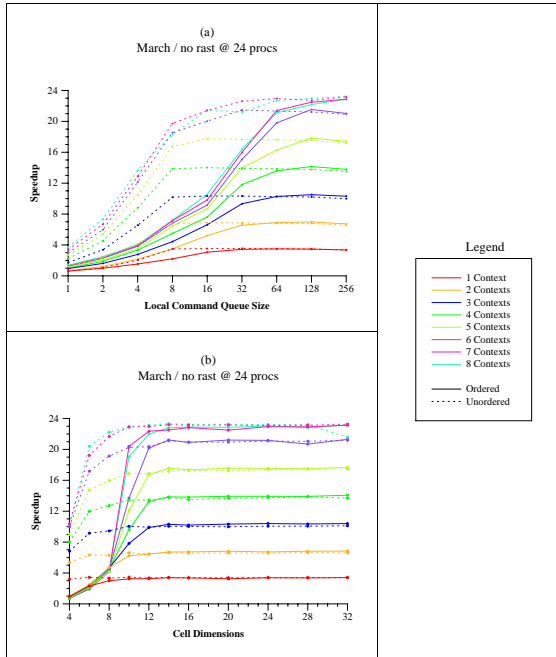


Figure 3.7: Effects Buffering and Granularity on Argus
The effects of varying buffer sizes for *March* are illustrated in (a), and the effects of synchronization granularity are illustrated in (b).

preemption—which has a much more coherent architecture compared to a graphics system—is generally viewed by computer architects as a great complication in high-performance microprocessors.

A second approach to the preemption problem, used by Pomegranate, is to resolve the API commands in software, utilizing the preemption resources of the microprocessor. With this approach, even though ordering constraints may be specified to the hardware, every piece of work specified is guaranteed by the software to eventually execute. Like Argus, each graphics context has an associated submit thread that is responsible for resolving the parallel API primitives. The application thread communicates with the submit thread via a FIFO, passing pointers to blocks of OpenGL commands and directly passing synchronization primitives. If the submit thread sees a pointer to a block of OpenGL commands, it passes this directly to the hardware. If the submit thread sees a parallel API command, it actually executes the command, possibly blocking until the synchronization is resolved. This allows the application thread to continue submitting OpenGL commands to the FIFO beyond a blocked parallel API command. In addition to executing the parallel API command, the submit thread passes the hardware a sequencing command that maintains the order resolved by the execution of the parallel API command. The important part of this hardware sequencing command is that even though an ordering is specified, the commands are guaranteed to be able to drain. Therefore, the hardware sequencing command for a *glSemaphoreP* will not be submitted until the hardware sequencing command for the corresponding *glSemaphoreV* is submitted. As with Argus, a blocked context is blocked entirely in software, and software context switching and resource reclamation may occur.

In order to keep hardware from constraining the total number of barriers and semaphores available to a programmer, Pomegranate’s internal hardware sequencing mechanism is based on sequence numbers. In Argus, a single global sequence number is used to order commands on a single global command queue. There are two problems with this approach. First, a global order is decided earlier than necessary in situations where no dependencies exist between graphics contexts. Delaying this decision until a later stage

3.6.2 Pomegranate: A Hardware Implementation

The Pomegranate architecture [Eldridge et al. 2000] is a hardware rendering system that achieves fully scalable performance in all the key metrics of Section 2.2.1 while retaining an ordered, immediate-mode interface. A key enabler of this scalability is the parallel API. By providing hardware support for multiple graphics contexts and the parallel API, the system is able to scale the input rate into the system.

3.6.2.1 Architecture

Hardware implementations of the parallel API pose a significant challenge. While support for multiple state and the actual implementation of barriers and semaphores are not very difficult, the parallel API requires that some or all of the graphics resources to be virtualized, and more importantly, subject to preemption and context switching. Imagine an application of $(n + 1)$ graphics contexts running on a system that supports only n simultaneous contexts in hardware. If a graphics barrier is executed by these $(n + 1)$ contexts, at least one of the n running contexts will need to be context switched out to allow the $(n + 1)$ th context to run. Furthermore, the parallel API introduces the possibility of deadlock. Imagine a poorly written graphics application that executes a *glSemaphoreP* on a semaphore that never receives a corresponding *glSemaphoreV*. At the very least, the system should be able to preempt the deadlocked graphics context and reclaim those resources. Performing this task in a software implementation is trivial due to the support for process preemption from the microprocessor and operating system.

Resolving the preemption problem while maintaining scalability in hardware performance was one of the most difficult challenges of the Pomegranate architecture. One solution to the preemption problem is the ability to read back *all* of the state of a hardware context and then restart the context at a later time. Although this may seem straightforward, it is a daunting task. Because a context may block at any time, the preempted state of the hardware is complicated by partially processed commands, large, partially-filled FIFOs, and in-flight network packets. As a point of comparison, microprocessor

Context A	Context B	Context C	Context D
Advance(A, seqA)	Advance(B, seqB)	Advance(C, seqC)	
			Await(A, seqA)
			Await(B, seqB)
			Await(C, seqC)
			Advance(D, seqD)
Await(D, seqD)	Await(D, seqD)	Await(D, seqD)	

Figure 3.8: Barriers in Pomegranate

This figure depicts the resolution of a parallel API barrier across four contexts once all the commands have reached the barrier. Three contexts generate sequence points and a wait command for the last context. The last context to arrive at the barrier submits wait commands for those three sequence points to ensure that every hardware context has reached the barrier. Additionally, context D emits a sequence point that signify its execution of the barrier, allowing the other contexts to move forward. Alternatively, each context could wait on all the other contexts, but that requires order $O(n^2)$ communication, while this solution is $O(n)$.

in the pipeline (e.g., ideally until fragment processing) could potentially lead to an improvement. Second, and more importantly, while the cost of a single point of synchronization for the global sequence number is small in a software system when it is amortized over several dozen primitives, such a global synchronization would have been a performance limit in Pomegranate, whose performance is two orders of magnitude greater. In order to avoid a global synchronization when only a point-to-point synchronization is necessary (i.e., semaphores), Pomegranate uses a single sequence number per hardware context. Upon executing a *glSemaphoreV* operation, the submit thread increments the hardware context’s sequence number by one to indicate a new ordering boundary, annotates the semaphore with a (ctx, seq) pair and issues an *Advance(ctx, seq)* command to the hardware. Upon completing the *glSemaphoreP* operation, the signaled submit thread removes the corresponding (ctx, seq) annotation from the semaphore and issues an *Await(ctx, seq)* command to the hardware. These commands are then executed just before the rasterization stage by the hardware. A similar mechanism is used for barriers, as illustrated in Figure 3.8. The sequence numbers are associated with a particular hardware context, not with a virtual graphics context, and when a context switch occurs, it is not reset. This allows the expression of dependencies for contexts that are switched out of the hardware, and thus the system can execute the $(n + 1)$ context barrier.

3.6.2.2 Performance

The performance of both *Nurbs* and *March* were measured on Pomegranate’s cycle-accurate simulator. The system can be scaled from 1 to 64 pipelines, and each pipeline is capable of receiving input at the rate of 1 GB per second. In order to tax the graphics system, CPU processing power is assumed to be infinite. The triangle rate is approximately 20 million triangles per second (depending on the number of lights, etc.), and the pixel rate is 400 million pixels per second per pipeline. In order to provide measurements at reasonable frame rates, the input scenes were scaled as follows:

<u>Nurbs</u>	<u>March</u>
armadillo dataset (textured)	orangutan dataset
1632 patches x 8 passes	64M voxels (400x400x400)
512 triangles per patch	cell size at 12x12x12
6.68M stripped triangles	1.53M independent triangles
2500x2000 pixels	2500x2000 pixels

In order to demonstrate the system’s ability to speed up a dataset with a total order, semaphores are used between patches to enforce a total order on *Nurbs*. The *Nurbs* dataset uses stripped textured triangles, allowing for over 22 million triangles per second from the interface of each pipeline, and the single pipeline system is limited by transformation and lighting to just over 17 million triangles per second. The *March* dataset uses untextured independent triangles, and a single pipeline system is limited by the input rate to around 10 million triangles per second.

Figure 3.9 shows Pomegranate’s speedup on *Nurbs* and *March*. As the number of pipelines and interfaces to the system are scaled from 1 to 64, near-linear speedup is achieved. Pomegranate’s novel sorting architecture (Section 2.2.2.5) is critical to providing linear speedup in triangle rate, and the parallel API scales the input rate. Even with a complete ordering specified, *Nurbs* is able to attain 99% efficiency at 64 pipelines. *March* attains a speedup of 58 at 64 pipelines. Although ordering is less constrained in *March* than in *Nurbs*, *March* requires many more synchronization primitives than *Nurbs* (3 semaphore pairs per 12^3 voxel containing an average of 38.8 triangles vs. 1 semaphore pair per patch containing 512 triangles).

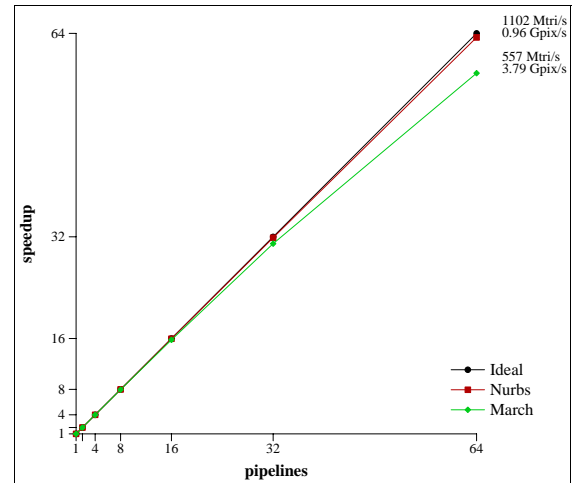


Figure 3.9: Pomegranate Speedup Graph

3.6.3 WireGL: A Transparent Implementation

WireGL is a sort-first architecture based on PCs [Buck et al. 2000]. As described in Section 2.2.2.1, this architecture achieves scalability by leveraging of unmodified graphics cards at commodity price points to render into a tiled display. Because a PC graphics card can render as fast as commands can be submitted, the parallel API is critical to achieving scalable performance. In this system, each application thread is a client hosted on a different PC, and graphics commands are sent across a network to the appropriate server PCs based on the screen-space subdivision where rendering occurs. Because each rendering node is an unmodified PC graphics card that does not support the parallel API,

the parallel API must be implemented in software in a transparent layer on top of the standard OpenGL implementation supported by the hardware. Parallel API commands are sent across the network, and the execution of the commands occurs in software on the servers by threads vying for the single graphics card.

State tracking plays a critical role in providing scalability in WireGL, both on the client side as well as on the server side. On the client side, blocks of primitive commands (i.e., those appearing between *glBegin* and *glEnd*) are packed into a compressed format and sent to only the appropriate rendering servers based on a bounding-box computation. For state-modifying commands, the client application thread allocates a dirty bit for each rendering server for each piece of state. For performance reasons, a hierarchical scheme is used to group each piece of state into one of 18 categories, and a dirty bit vector is kept for each category. When the application calls a state command, all the bits in the state bit vector are set to 1, and all the bits in the category vector are set to 1, indicating that the virtual context of the application thread is possibly out of sync with all the rendering servers. These bit changes accumulate until a primitive block is submitted to the application thread. At this point, the client computes the state difference between its virtual context and *only* the servers that the primitive block overlaps and sends state updates. The appropriate bits in the dirty bit vectors are then cleared. This lazy update scheme is important because broadcasting state would limit scalability.

On the server side, the same state tracking mechanism is used to support the parallel API. Each client needs a unique graphics context to support the parallel API, but PC graphics cards typically support a single hardware context. Because context switching on PC graphics cards is relatively slow (anywhere from a few dozen to several thousand context switches per second), context switching between client contexts would be prohibitively expensive because ordering constraints from the parallel API would force frequent switches. To remedy this, state tracking is used to perform a software context switch. A dirty bit vector is kept for each piece of state to indicate which clients contexts are out of sync with the hardware context. Each time a client modifies a piece of state, the bit vector is set. Then, when a client is context switched into the hardware because of

an ordering constraint, the appropriate bit of each vector is cleared if the client state is the same as the hardware state. As before, a hierarchical scheme is used to minimize the number of tests required.

3.7 Implementation Alternatives

Argus, Pomegranate, and WireGL are three implementations of the parallel API that perform well. Obviously, these architectures are not the only possible choices, and it is instructive to examine the design considerations of alternative implementations due to the special architectural requirements imposed by the extensions.

3.7.1 Consistency and Synchronization

Until now, we have not said much about how the operations of the parallel API can be interleaved. Supporting multiple contexts that share a framebuffer means that the system must provide a consistency model. We borrow the notion of sequential consistency from the field of computer architecture [Lamport 1979]. Imagine a system consisting of multiple processes simultaneously performing atomic operations. A sequentially consistent system computes a result that is realizable by some serial interleaving of these atomic operations. By making a single API command be the level of apparent atomicity, we define the notion of *command-sequential consistency*, the strongest form of consistency possible within the parallel API. At the other end of the spectrum is framebuffer-sequential consistency—only framebuffer accesses are atomic. A whole spectrum of consistency models can be enumerated in such a fashion. The OpenGL specification does not require an implementation to support any consistency model. In order to support the parallel API, however, a graphics system should provide at least *fragment-sequential consistency* in order to support features that depend on an atomic read-modify-write operation on the framebuffer (such as depth buffering).

The consistency model which an architecture supports is related to the location in the pipeline where synchronization constraints between graphics streams are resolved. The Argus pipeline described in Section 3.6.1 and the WireGL pipeline described in Section 3.6.3 synchronize and merge multiple graphics streams early in the pipeline (before geometry processing), thus supporting command-sequential consistency. One drawback with such architectures is that geometry processing cannot occur on primitives that are blocked due to synchronization constraints. Another problem is that ordering dependencies not required by the synchronization commands are introduced early in the pipeline. The Pomegranate pipeline described in Section 3.6.2, on the other hand, resolves final ordering dependencies just before the rasterization stage. This results in sequential consistency on the architecture's 2D screen-space primitives.

The choice of the point of synchronization has large implications for the overall architecture. For example, Argus originally merged graphics streams at the rasterizers. Because the system was in software, the entire pipeline up to and including the tile rasterization threads was replicated for each context. Every tile thread executed every synchronization command, and threads that share the same tile merge their streams by obtaining exclusive access to the tile. One disadvantage of this approach is the extra buffering requirements because the size of the graphics data expands as it gets farther down the pipeline. Another problem with this alternate approach is the high cost of synchronization since synchronization commands must be executed by every tile rasterizer—this proved prohibitively expensive in the framework of Argus.

3.7.2 Architectural Requirements

While a graphics system which implements the parallel API is in many respects similar to one which implements a serial API, an architecture should take special care in addressing three particular areas. First, the architecture must have a mechanism that efficiently handles multiple simultaneous input streams. Second, the state management capabilities of the architecture must be able to handle multiple simultaneous graphics states. And third,

the rasterization system must be able to handle texture data for multiple streams efficiently.

In designing current systems, graphics architects have gone to great lengths to allow the seamless sharing of the graphics hardware between multiple windows by significantly reducing the context switch time. Although this same mechanism can be used for the parallel API, the context switch time must be reduced even further in order to handle multiple input streams at a much finer granularity. Argus does this by making use of a thread library that can switch threads in less than a microsecond as well as allowing multiple input ports. A hardware system could allow multiple input ports by replicating command processors. Ideally, each of the command processors could handle either a single graphics stream at a high rate or multiple graphics streams at lower rates. This would result in peak performance on serial applications and good performance on highly parallel applications.

The parallel API imposes special requirements on the handling of state. In past architectures, state changes have been expensive due to pipeline flushing. Recent graphics architectures, however, have taken measures to allow large numbers of state changes [Montrym et al. 1997]. To a first order, the number of state changes for a given scene as issued by one application thread is the same as the number of state changes for the same scene as issued by multiple application threads since the number of inherent state changes in a scene is constant. However, the parallel API increases the amount of state that has to be accessible to the different portions of the graphics system: the various graphics processors must be able to switch between the states of different graphics streams without dramatically affecting performance. Hardware implementations that allow for multiple simultaneous contexts have already been demonstrated [Voorhies et al. 1988, Kirkland 1998]. In Argus, multiple simultaneous contexts are handled efficiently by taking advantage of state coherence in the state management algorithm using shared memory and processor caching. In Pomegranate, direct hardware support is provided. In WireGL, a novel state tracking system is used.

One type of state that requires special attention is texture. Unlike the rest of the state associated with a context (with the exception of display lists), texture state can be shared amongst multiple contexts, thus exposing the need for efficient download of and access to shared texture data. The semantics of texture download are the same as all other graphics commands: it is susceptible to buffering, and synchronization must occur to guarantee its effects from the point of view of other contexts. Efficient implementations of synchronized texture download can be realized by extending the idea of the "texture download barrier" found in the SGI InfiniteReality [Montrym et al. 1997]. The access of texture memory may also require special care. Since hardware systems have a limited amount of local texture memory, applications issue primitives in an order that exploits texture locality. The parallel API can reduce this locality since the rasterizers can interleave the rendering of several command streams. In architectures that use implicit caching [Hakura & Gupta 1997, Cox et al. 1998], the effectiveness of the cache can possibly be reduced. In architectures that utilize local texture memory as an explicit cache, texture management is complicated. In Argus, shared texture download is facilitated by shared memory, and locality of texture access is provided by the caching hardware. In Pomegranate, shared texture download is ordered like all other graphics commands, and cache locality is provided by dividing texturing work into an appropriately sized granularity.

3.8 Conclusion

In this chapter, we have shown how thread-level parallelism may be introduced to graphics architectures by using a parallel API, addressing the performance limitations of a serial API. This parallel API consists of the simple addition of parallel synchronization constructs, like barriers and semaphores, to a standard interface such as OpenGL. These synchronization commands may be used to explicitly order the drawing of primitives across different graphics contexts, and command submission can proceed in parallel, without any loss in performance, even if an exact ordering is necessary among the graphics contexts. Furthermore, because of the large amount of parallelism available in graph-

ics commands due to a dearth of dependencies (as described in Section 2.1), fully scalable rendering rates for large numbers of graphics nodes are possible even when an exact ordering is required by the application. Contrary to the notion put forth by many graphics architects (e.g., [Molnar et al. 1992]), there is no need to forgo ordering in graphics architectures in order to achieve scalability. At worst, ordering introduces a manageable amount of complexity to the design of a graphics system.

Chapter 4

Scalable Texture Mapping

In this chapter, we present an architecture for scalable texture mapping and quantify its effectiveness. There are two important considerations in scaling texture mapping capabilities. First, the texture subsystem must be able to scale the number of fragments it can texture per second. An obvious scheme for this is to replicate the texturing unit to scale the amount of computational power available for texturing. In parallel systems, two effects that limit scalability are load imbalance and redundant work. Because each fragment is textured exactly once, no redundant computation occurs. Furthermore, so long as the parallel rendering algorithm is able to present an equal number of fragments to each rasterizer, the computational load to each texturing unit will be balanced. This is only part of the story, however. Modern texture mapping hardware relies heavily on texture caching to reduce the amount of bandwidth required for the texture subsystem. Parallel rasterization techniques tend to reduce the amount of locality available in each texturing unit's fragment stream, resulting in redundant work in terms of memory bandwidth. Furthermore, the bandwidth requirements of each texturing unit's fragment stream may vary greatly, resulting in load imbalance. These effects are studied in depth later in this chapter.

A second component of scalable texture memory is scaling the amount of texture memory available to an application. If each texturing unit is given its own dedicated tex-

85

ture memory, then textures downloaded by the application must be replicated across the texturing units, and the total amount of texture memory does not scale. On the other hand, if each texturing unit is able to share its texture memory with other texturing units, the total amount of texture memory available to the application scales linearly. Such a system presents two challenges. First, the texture data must be spread across the texture memories in a fashion that minimizes load imbalance in the number of requests to each texture memory. Second, unlike a dedicated memory system that delivers texture data with a fixed latency, network contention and memory contention in a shared texture memory system can cause highly variable latencies.

In Section 4.1, we examine and measure a texture cache architecture that can tolerate arbitrarily high and highly variable latencies. Such a system is applicable to serial texturing units accessing a dedicated texture memory or a shared system memory as well as parallel texturing units accessing a dedicated or shared texture memory. This architecture builds a foundation upon which a parallel texture architecture may be implemented. In Section 4.2, we present a framework for scaling texture mapping and quantify its performance across a variety of rasterization schemes.

4.1 Prefetching in a Texture Cache

Texture mapping has become so ubiquitous in real-time graphics hardware that most systems are able to perform filtered texturing without any penalty in fill rate. The computation rates available in hardware are outpacing the memory access rates, and texture systems are becoming constrained by memory bandwidth and latency. Caching in conjunction with prefetching can be used to alleviate this problem.

In this section, we introduce a prefetching texture cache architecture designed to take advantage of the access characteristics of texture mapping. The structures needed are relatively simple and are amenable to high clock rates. To quantify the robustness of our architecture, we identify a set of six scenes whose texture locality varies over nearly two orders of magnitude and a set of four memory systems with varying bandwidths and la-

encies. Through the use of a cycle-accurate simulation, we demonstrate that even in the presence of a high-latency memory system, our architecture can attain at least 97% of the performance of a zero-latency memory system.

4.1.1 Introduction

Texture mapping is expensive both in computation and in memory accesses. Continual improvement in semiconductor technology has made the computation relatively affordable, but memory accesses have remained troublesome. Several researchers have proposed and demonstrated texture cache architectures that can reduce texture memory bandwidth. Hakura and Gupta examine different organizations for on-chip cache architectures which are useful for exploiting locality of reference in texture filtering, texture magnification, and to a limited extent, repeated textures [Hakura & Gupta 1997]. Cox et al. extend this work to multi-level caching [Cox et al. 1998]. They demonstrate that on-chip caches in conjunction with large off-chip caches can be used to exploit all of the aforementioned forms of texture locality as well as inter-frame texture locality. Thus, memory bandwidth requirements can be dramatically reduced for scenes in which the working set of a frame fits into the off-chip cache.

A second troublesome point about texture memory access (which is not addressed by Hakura or Cox) is the high latencies of modern memory systems. In order to address this problem, several systems are described that make use of large pipelines that prefetch the texel data [Kilgard 1996, Torborg & Kajija 1996, Anderson et al. 1997]. Two of the systems [Kilgard 1996, Anderson et al. 1997] do not use any explicit caching, although their memory systems are organized for the reference patterns of texture filtering, but one system [Torborg & Kajija 1996] does employ prefetching as well as two levels of caching, one of which holds compressed textures. However, the algorithm that combines the prefetching with the caching is not described. Several other consumer-level architectures exist which undoubtedly utilize some form of prefetching, possibly with caching. Unfortunately, none of these algorithms are described in the literature. In this section, we introduce a texture architecture that combines prefetching and caching.

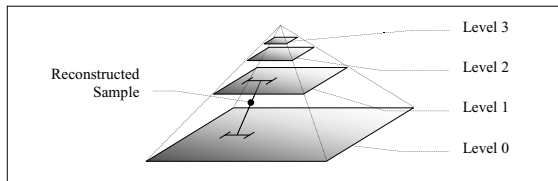


Figure 4.1: Mip Mapping

An image is filtered recursively into quarter-sized images. Trilinear interpolation reconstructs a sample by linearly interpolating between two adjacent levels of the mip map, each of which is sampled with bilinear filtering on the four closest texels in that level of the mip map.

4.1.2 Mip Mapping

Texture mapping, in its most basic form, is a process by which a 2D image is mapped onto a projected screen-space triangle under perspective. This operation amounts to a linear transformation in 2D homogeneous coordinates. The transformation is typically done as a backward mapping—for each pixel on the screen, the corresponding coordinate in the texture map is calculated. The backward mapped coordinate typically does not fall exactly onto a sample in the texture map, and the texture may be minified or magnified on the screen. Filtering is applied to minimize the effects of aliasing, and ideally, the filtering should be efficient and amenable to hardware acceleration.

Mip mapping [Williams 1983] is the filtering technique most commonly implemented in graphics hardware. In mip mapping, an image pyramid is constructed from the base image which serves as the bottom of the pyramid. Each successive level of the pyramid is constructed by resampling the previous level of the pyramid by half in each dimension, as illustrated in Figure 4.1. For each screen-space fragment, the rasterization process computes a texture coordinate and an approximate texel-to-pixel ratio (also known as the level-of-detail value). This ratio is used to compute the two closest corresponding mip map levels, and a bilinear interpolation is performed on the four nearest texels in each of

the two adjacent levels. These two values are then combined with linear interpolation based on the level-of-detail value, and the resulting trilinearly interpolated sample is passed to the rest of the graphics pipeline. If a fragment falls beyond either end of the mip map pyramid, the algorithm performs bilinear filtering on the one closest level of the mip map.

The popularity of mip mapping can be attributed to three characteristics. First, mip mapping reduces many aliasing artifacts. Although it is by no means an ideal filter, especially since it often blurs excessively, the results are quite acceptable for interactive applications. Second, the computational costs of mip mapping, though by no means cheap, are reasonable and fixed for each fragment. Finally, mip mapping is efficient with respect to memory. The additional space required for the pyramid representation is only one-third the space occupied by the original image. Furthermore, because the level-of-detail computation is designed to make one step in screen space correspond to approximately one step in the appropriate mip map level, the memory access pattern of mip mapping is very coherent.

4.1.3 Caching and Prefetching

For the past few decades, many aspects of silicon have been experiencing exponential growth. However, not all aspects have grown at the same rate. While memory density and logic density have seen tremendous growth, logic speed has experienced more moderate growth, and memory speed has experienced slight growth. These factors have made the cost of computation on a chip very cheap, but memory latency and bandwidth sometimes limit performance. Even with the advent of memory devices with high-speed interfaces [Crisp 1997], it is easy to build a texturing system that outpaces the memory it accesses. The problem of directly accessing DRAM in a texture system is aggravated by the fact that memory devices work best with transfers that do not match the access patterns of texture mapping: DRAM provides high bandwidth when moving large contiguous blocks of memory, but a fragment's texture accesses typically consist of several small non-contiguous memory references.

An obvious solution to this problem is caching. Many issues are resolved by integrating a small amount of high-speed, on-chip memory organized to match the access patterns of the texture system. According to our measurements (detailed in Section 4.1.5.1) as well as data found in other literature [Hakura & Gupta 1997, Cox et al. 1998], it is quite reasonable to expect miss rates on the order of 1.5% per access. Many texture systems are capable of providing the computation for a trilinearly mip mapped fragment on every clock cycle. Thus, because there are eight texture accesses per cycle, the per-fragment texel miss rate is 12%. Even if these misses could be serviced in a mere 8 cycles each, a calculation of the average memory access time shows that overall performance is cut in half. Clearly, this is not acceptable.

While caching can alleviate the memory bandwidth problem, it does not solve the memory latency problem. The latency problem with relation to texture caching is a special one. In current interactive graphics interfaces, texture accesses are read-only for large amounts of time, and address calculation for one texture access is never dependent on the result of another texture access. Thus, there are no inherent dependencies to limit the amount of latency that can be covered. This means that a prefetching architecture should be capable of handling arbitrary amounts of latency.

4.1.3.1 Traditional Prefetching

In the absence of caching, prefetching is very easy. When a fragment is ready to be textured, the memory requests for the eight texel accesses are sent to the memory system, and the fragment is queued onto a fragment FIFO. When the replies to the memory requests arrive, the fragment is taken off the FIFO, and the fragment is textured. The time a fragment spends in the FIFO is equal to the latency of the memory system, and if the FIFO is sized appropriately, fragments may be processed without ever stalling. For greater efficiency, part of the fragment FIFO can actually be a fragment processing pipeline [Kilgard 1996, Anderson et al. 1997]. Note that this non-caching prefetching architecture assumes that memory replies arrive in the same order that memory requests are

made, and that the memory system can provide the required bandwidth with small memory requests.

One straightforward way to combine caching with prefetching is to use the architecture found in traditional microprocessors that use explicit prefetch instructions. Such an architecture consists of a cache, a fully associative prefetch address buffer, and a memory request buffer. A fragment in such a system is processed as follows: first, the fragment's texel addresses are looked up in the cache tags, and the fragment is stored in the fragment FIFO. Misses are forwarded to a prefetch buffer that is made fully associative so that multiple misses to the same memory block can be combined. New misses are queued in the memory request buffer before being sent to the memory system. As data returns from the memory system, it is merged into the cache. When a fragment reaches the head of the fragment FIFO, the cache tags are checked again, and if all of the texels are found in the cache, the fragment can be filtered and textured. Otherwise, additional misses are generated, and the system stalls until the missing data returns from memory. Fortunately, the architecture works even in conjunction with an out-of-order memory system.

There are three problems with using the traditional microprocessor prefetch architecture for texture mapping. First, if the product of the memory request rate and the memory latency being covered is large compared to the size of the caches utilized, a prefetched block that is merged into the cache too early can cause conflict misses. Second, in order to support both reading and prefetching of texels at the full fragment rate, tag checks must be performed at twice the fragment rate, increasing the cost of the tag logic. Finally, as the product of the memory request rate and the memory latency increases, the size (and therefore the associativity) of the prefetch buffer must be increased proportionally.

4.1.3.2 A Texture Prefetching Architecture

While some of the problems with the traditional microprocessor prefetching architecture can be alleviated, we have designed a custom prefetching architecture that takes advantage of the special access characteristics of texture mapping. This architecture is illus-

trated in Figure 4.2. Three key features differentiate this architecture from the one described in Section 4.1.3.1. First, tag checks are separated in time from cache accesses, and tag checks are performed only once per texel access. Second, because the cache tags are only checked once and always describe the future contents of the cache, a fully associative prefetch buffer is not needed. And third, a reorder buffer is used to buffer memory requests that come back earlier than needed.

The architecture processes fragments as follows. As each fragment is generated, each of its texel addresses is looked up in the cache tags. If a tag check reveals a miss, the cache tags are updated with the fragment's texel address immediately and the address is forwarded to the memory request FIFO. The cache addresses associated with the fragment are forwarded to the fragment FIFO and are stored along with all the other data needed to process the fragment, including color, depth, and filtering information. As the request FIFO sends requests for missing cache blocks to the texture memory system, space is reserved in the reorder buffer to hold the returning memory blocks. This guarantee of space makes the architecture robust and deadlock-free in the presence of an out-of-order memory system. A FIFO can be used instead of the reorder buffer if responses from memory always return in the same order as requests sent to memory.

When a fragment reaches the head of the fragment FIFO, it can proceed only if all of its texels are present in the cache. Fragments that generated no misses can proceed immediately, but fragments that generated one or more misses must first wait for their corresponding cache blocks to return from memory into the reorder buffer. In order to guarantee that new cache blocks do not prematurely overwrite older cache blocks, new cache blocks are committed to the cache only when their corresponding fragment reaches the head of the fragment FIFO. Fragments that are removed from the head of the FIFO have their corresponding texels read from the cache and proceed onward to the rest of the texture pipeline.

Our simulated implementation can handle eight texel reads in parallel, consisting of two bilinear accesses to two adjacent mip map levels. To support these concurrent texel reads, we organize our cache tags and our cache memory as a pair of caches with four

banks each. Adjacent levels of a mip map are stored in alternating caches to allow both mip map levels to be accessed simultaneously. Data is interleaved so that the four accesses of a bilinear interpolation occur in parallel across the four banks of the respective cache. Cache tags are also interleaved across four banks in a fashion that allows the tag checks for a bilinear access to occur without conflict. The details of this layout can be found in Figure 4.3 of Section 4.1.5.

In order to make our architecture amenable to hardware implementation, we impose two limitations. First, the number of misses that can be added to the request FIFO is limited.

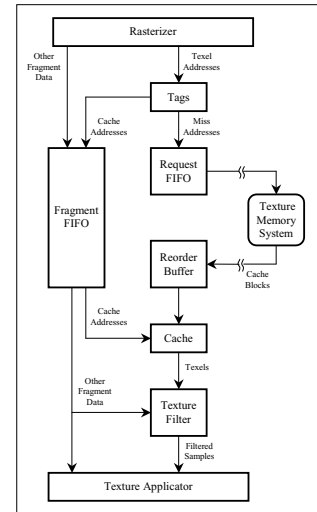


Figure 4.2: A Texture Prefetching Architecture

ited to one miss per cache per cycle. Second, the number of cache blocks that can be committed to the cache from the reorder buffer is similarly limited to one block per cache per cycle. These commits match up to the requests—groups of misses that are added to the request FIFO together are committed to the cache together. This means that each fragment may generate up to four groups of misses. Because our implementation can only commit one of these groups per cycle, a fragment that has more than one group of misses will cause the system to stall one cycle for every group of misses beyond the first.

4.1.4 Robust Scene Analysis

When validating an architecture, it is important to use benchmarks that properly characterize the expected workload. Furthermore, when validating interactive graphics architectures, an architect should look beyond averages due to various characteristics of the human perceptual system. For example, if a graphics system provides 60 Hz rendering for the majority of the frames, but every once in a while drops to 15 Hz for a frame, the discontinuity is distracting, if not nauseating. In designing a system, the graphics architect must evaluate whether or not sub-optimal performance is acceptable under bad-case conditions. Accordingly, a robust set of scenes that cover a broad range of workloads, from good-case to bad-case, should be utilized to validate a graphics architecture.

4.1.4.1 Texture Locality

The effectiveness of texture caching is strongly scene-dependent. For example, the size and distribution of primitives affect texture locality. Texture locality is also affected by what we call the scene's *unique texel to fragment ratio*. Every scene has a number of texels that are accessed at least once; these texels are called *unique texels*. Unless caches are big enough to exploit inter-frame locality (this requires several megabytes [Cox et al. 1998]), every unique texel must be fetched at least once by the cache, imposing a lower limit on the required memory bandwidth. If we divide this number by the number of fragments rendered for a scene, we can calculate the unique texel to fragment ratio. Note that this value is dependent on the screen resolution. A good-case scene will have a low

ratio, and a bad-case scene will have a high ratio. Ideally, the number of texels fetched by the caching architecture per fragment will be close to the scene's unique texel to fragment ratio.

Three factors affect the unique texel to fragment ratio of a scene. First, when a texture is viewed under magnification, each texel gets mapped to multiple screen pixels, and the ratio decreases. Second, when a texture is repeated across a surface, the ratio also decreases. This temporal coherence can be exploited by a cache large enough to hold the repeated texture. Third, when a mip map texture is viewed under minification, the ratio becomes dependent on the relationship between texel area and pixel area. This relationship is characterized by the level-of-detail value of the mip mapping computation that aims to keep the footprint of a backward-mapped pixel equal to the size of a texel in a mip map level. Although this value is normally calculated automatically, the application programmer may bias it in either direction, thus modifying the scene's unique texel to fragment ratio.

A more surprising effect that occurs even without biasing is characterized by the fractional portion of the level-of-detail value. The level-of-detail value determines the two levels of the mip map from which samples are taken; the fractional portion is proportional to the distance from the lower, more detailed level. Given a texture mapped polygon that is parallel to the screen, a fractional portion close to zero implies a texel area to pixel area ratio of nearly one in the lower mip map level and a quarter in the upper mip map level, yielding a texel to fragment ratio near 1.25. Likewise, a fractional portion close to one implies a texel area to pixel area ratio of four in the lower mip map level and one in the upper mip map level, yielding a texel to fragment ratio near 5. The ratios are lower for polygons that are not parallel to the screen. Normally, we expect a wide variation in the texel to fragment ratio due to the fractional portion of the level-of-detail value. However, most scenes exhibit worst-case behavior for short amounts of time, and a few scenes exhibit worst-case behavior for large amounts of time.

workload name	<i>quake</i>	<i>quake2x</i>	<i>flight</i>	<i>flight2x</i>	<i>qtv</i>	<i>qtv2x</i>
screen resolution	1280 x 1024	1280 x 1024	1280 x 1024	1280 x 1024	1280 x 1024	1280 x 1024
depth cmplxty.	3.29	3.29	1.06	1.06	1.00	1.00
percent trilinear	30%	47%	38%	87%	0%	100%
unique texels/frag	0.033	0.092	0.706	1.55	0.569	2.83

Table 4.1: The Benchmark Scenes

4.1.4.2 The Benchmark Scenes

In order to validate our texture caching architecture, we chose six real-world scenes that span a wide range of texture locality. These six scenes originated from three traces of OpenGL applications captured by *gltrace*, a tool implemented on top of the OpenGL Stream Codec. In the future, we expect to see more texture for a given screen resolution; this will increase the unique texel to fragment ratio. To simulate this effect, each of the traces was captured twice, once with the textures at original size, and once with the textures at double resolution. Table 4.1 summarizes our six scenes, and high resolution images can be found in the Color Plate. Our workloads span nearly two orders of magnitude in the unique texel to fragment ratio (0.033 to 2.83). This is in contrast to the ratios in the scenes used by Hakura (0.2 to 1.1) [Hakura & Gupta 1997] and the animations used by Cox (0.1 to 0.3) [Cox et al. 1998]. These workloads result from the fact that applications programmers choose the way they use texture according to the needs of the application and the constraints of the target systems. We now give a brief summary of each scene and highlight the points relevant to texture caching:

- ***quake***. This is a frame from the OpenGL port of the video game Quake. This application is essentially an architectural walkthrough with visibility culling. Color mapping is performed on all surfaces that are, for the most part, large polygons that make use of repeated texture. A second texturing pass blends low-resolution light maps with the base textures to provide realistic lighting effects. Approximately 40% of the base textures are magnified, and 100% of the light maps are magnified.
- ***quake2x***. In order to account for increasing texture resolutions needed for larger screen resolutions (Quake's content was geared towards smaller screens), the texture maps in *quake* were zoomed by a factor of two to create *quake2x*. This results in a scene that magnifies only the light maps.
- ***flight***. This scene from an SGI flight simulator demo shows a jet flying above a textured terrain map. The triangle size distribution centers around moderately sized triangles, and most textures are used only once. A significant portion of the texture (62%) is magnified.
- ***flight2x***. As texture systems become more capable of handling larger amounts of texture, applications will use larger textures to avoid the blurring artifact of filtered magnification. In *flight2x*, the textures of *flight* were zoomed by a factor of two. This results in a scene that only magnifies 13% of the texture.
- ***qtv***. This scene comes from an OpenGL-based QuickTime VR [Chen 1995] viewer looking at a panorama from Mars. This huge panorama, which measures 8K by 1K, is mapped onto a polygonal approximation of a cylinder made of tall, skinny triangles. Even though all of the texture is magnified, the lack of repeated texture keeps the number of unique texels per fragment high.
- ***qtv2x***. The texture of *qtv* was scaled up to 16K by 2K. This increases the number of unique texels accessed by the scene since all the texture is minified. Furthermore, the fractional portion of the level-of-detail value is always high

in *qtv2x* because the panorama is viewed more or less head-on at just the wrong zoom value. Note that these same effects would occur if *qtv* was run at quarter-sized screen resolution, and that *qtv2x* is by no means a hand-tailored pathological case. In fact, it was while gathering trace data on *qtv* that we first observed the texture locality effects of a level-of-detail fraction close to one. This scene is representative of a bad-case frame in a real-world application.

4.1.5 Memory Organization

In designing our prefetching cache architecture, we carefully chose the proper parameters for the cache and the memory system. To narrow our search space, we leveraged Hakura's findings on blocking [Hakura & Gupta 1997]. First, Hakura demonstrates the importance of placing texture into tiles according to cache block size. This addressing scheme is referred to as 4D blocking. Furthermore, tiles should be organized in a 2D blocked fashion according to the cache size in order to minimize conflict misses. This is called 6D blocking. In accordance with these guidelines, we employ 6D blocking for texture maps according to the cache block size and the cache size. The layout of texture data is illustrated in Figure 4.3. Figure 4.3 also illustrates how texture data is banked in both the cache tags as well as the cache memory in order to allow conflict-free access for bilinear interpolation. Additionally, rasterization also occurs in a blocked fashion rather than in scan line order, and we rasterize triangles in 8 pixel by 8 pixel blocks that are tiled in a 4 by 4 fashion. Note that for the purposes of this study, all texture data is stored as 32-bit RGBA values.

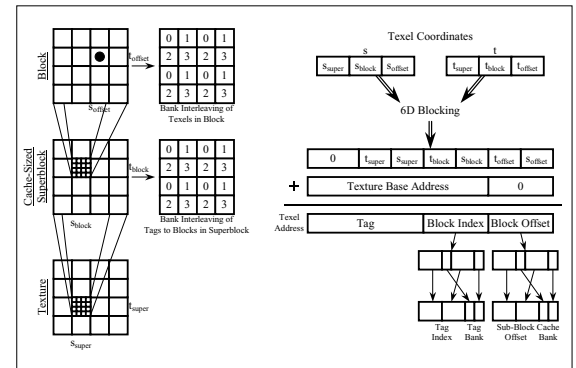


Figure 4.3: Texture Data Organization

In our architecture, textures are stored using a 6D blocking pattern. Each mip map level is divided into cache-sized superblocks, with each superblock further divided into blocks. Each block is a rectangular, linearly-addressable region of the original mip map level. Each of eight texel addresses is computed by adding an offset (formed by permuting the texel's coordinates) to a corresponding texture base address. The eight resulting texel addresses, four from each of two adjacent mip map levels, are then directed to two caches, each of which has four banks and services alternating levels of the mip map. Within each superblock, tags are interleaved on a block basis, causing all 2x2 texel accesses to fall onto one, two, or four adjacent blocks, with each block's tag stored in a separate bank of the tag memory. This interleaving is accomplished by permuting the bits of the block index, yielding a tag bank and a tag index for each texel address. Similarly, texels are interleaved within each block, causing all 2x2 texel accesses to fall into separate banks of the cache memory even if the texels of the 2x2 access do not all fall into the same block. A permutation of the block offset results in a cache bank and a sub-block offset for every texel address; used in conjunction with the block index, these values locate each texel in the cache memory. Note that both of these permutations extract the least significant bit of the corresponding *s* and *t* fields to determine the tag or cache bank.

4.1.5.1 Cache Efficiency

Since we have decided to provide a separate cache for each of the bilinear accesses that need to occur during every trilinear texture access, three cache parameters need to be chosen. The first choice is the cache block size. A small block size increases miss rates, but keeps bandwidth requirements low. A large block size can decrease miss rates, but bandwidth requirements and latency can skyrocket. An additional factor that needs consideration is that modern DRAM devices require large transfer sizes to sustain bandwidth. Hakura found that 16 texel tiles (64 bytes) work well, and most next-generation DRAM chips can achieve peak efficiency at such transfer sizes [Crisp 1997].

Given a 16 texel block size, we are left with choices for cache associativity and cache size. Figure 4.4 shows the miss rates for our six test scenes. We see that increasing associativity does not decrease the miss rate significantly. Intuitively, this makes sense since having a separate cache for alternate levels of a mip map minimizes conflict misses. Thus, a direct-mapped cache is quite acceptable if we use 6D blocking when alternate levels of the mip map are cached independently. According to Hakura, if a unified cache is used for trilinear accesses (and thus the bilinear accesses do not occur simultaneously), a 2-way set associative cache is appropriate. In the more general case of multi-texturing, m independent n -way set associative caches are well suited towards providing texture accesses at the rate of m bilinear accesses per cycle to $m \cdot n$ textures (in this scheme, trilinear accesses count as two accesses to two textures). Since we are limiting our study to a single trilinear access per cycle, two independent direct-mapped caches are appropriate.

Figure 4.4 also illustrates the effects of modifying the total cache size on the miss rates of the various scenes. We see that for scenes in which texture locality is not dependent on repeated textures (*flight*, *flight2x*, *qvr*, *qvr2x*), the miss rate curves flatten somewhere between a total cache size of 4 KB and 16 KB. This cache size represents the working set for filtering locality when rasterization is done in 8 by 8 blocks. On scenes that contain repeated texture (such as *quake* and *quake2x*) the miss rates are lower, but the miss rate curves flatten later (at 32 KB and 128 KB, respectively). These points cor-

respond to the working set sizes of the repeated textures in each scene. The miss rate realized once any of the curves flattens corresponds closely to the unique texel to fragment ratio of the respective scene.

We chose to use a 16 KB cache (composed of two direct-mapped 8 KB caches) for our study. According to our workloads, this size is large enough to exploit nearly all of the coherence found in scenes that demonstrate poor locality (such as *flight2x* and *qvr2x*), and even though a larger size could help in scenes with repeated textures (such as *quake* and *quake2x*), these scenes already perform very well. Though we stress that different choices can also be reasonable, we assume a cache architecture with two direct-mapped 8 KB caches (interleaved by mip map level) with 64-byte blocks for our study.

4.1.5.2 Bandwidth Requirements

In formulating bandwidth requirements, we can relate the number of texels of memory bandwidth required per fragment to the cache miss rate by the cache block size. These equivalent measures are shown as left- and right-axes in Figure 4.4. One key point of Table 4.1 and Figure 4.4 is that even though caching can work well sometimes, there are cases when the bandwidth requirements are extremely high. In the case of *qvr2x*, nearly 3 texels have to be fetched for each fragment no matter what size on-chip cache is utilized. This is quite high considering that eight texels are required to texture a trilinearly mip mapped fragment. However, this should not be seen as an argument for not having a cache: the cache still provides a way of matching the access patterns of mip mapping with the large block requests required for achieving high memory bandwidth. If a system wants to provide high performance robustly over a wide variety of scenes, it needs to provide high memory bandwidth even with the use of caching. If a system's target applications have high texture locality, or if cost is a primary concern, a memory system with lower memory bandwidth can be employed.

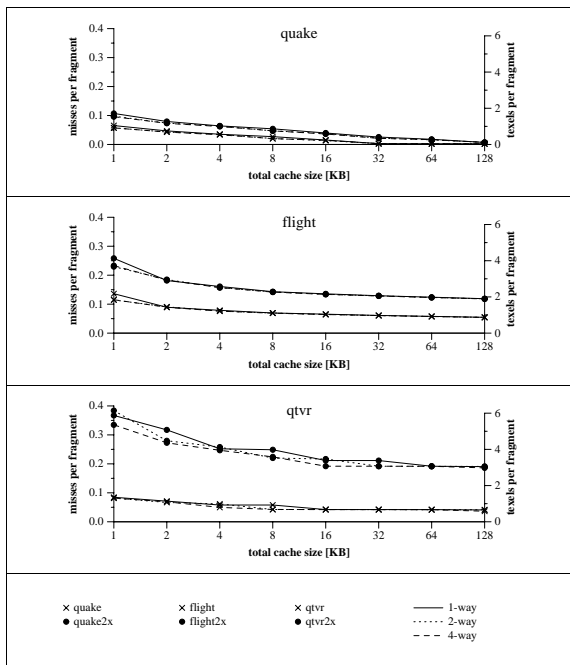


Figure 4.4: Cache Efficiency

Block size was set to 4 by 4 texels, and the six workloads were sent through the cache simulator with various cache sizes and cache associativities. Results are reported in terms of cache block misses per fragment rather than in terms of misses per access since most texturing architectures have clock cycles based on fragments. The cache block miss rate corresponds to a memory bandwidth requirement that can be expressed in terms of texels fetched per fragment.

Figure 4.4 can also be a bit misleading because the average bandwidth requirement does not tell the whole story. From the data, one could falsely infer that a memory system which provides enough bandwidth to supply 1 texel per fragment will perform perfectly on *quake2x* since, according to the graph, only 0.63 texels per fragment are required given the 16 KB cache size. Figure 4.5 illustrates why this is not the case. The average cache miss rate does not properly encapsulate temporal variations in bandwidth requirements. Even though the average bandwidth requirement is 0.63 texels per fragment over the whole frame, large amounts of time exist when the system needs double that bandwidth, and large amounts of time exist when the system does not need most of that bandwidth (i.e., when light maps are drawn). Because of the large separation in time between these two phases, a system cannot borrow from one to provide for the other, and thus the overall performance will decrease.

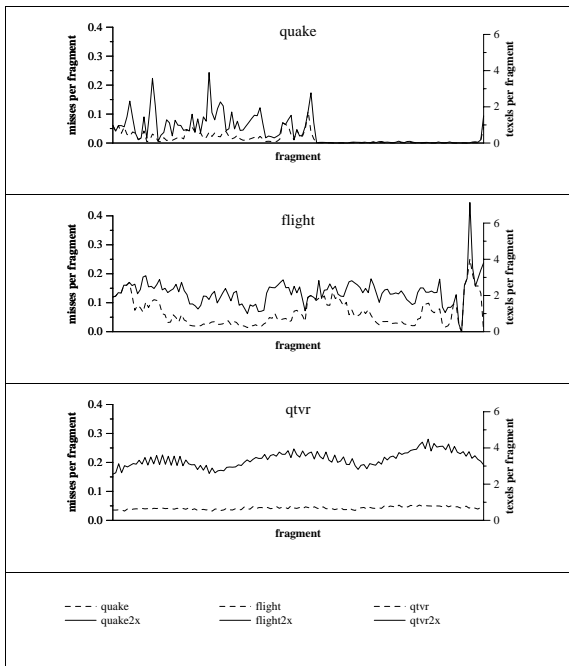


Figure 4.5: Bandwidth Variation

Even though the required memory bandwidth can be low on average, this value can vary widely over time. The graphs above show this variation with a pair of direct-mapped 8 KB caches. Each data point is a windowed average over 30,000 fragments in *quake* and 10,000 fragments in *flight* and *qtvr*. The variance in the required bandwidth is quite extreme in the cases of *quake* and *quake2x* as the application transitions from applying color maps to applying light maps.

- **rdram.** Direct RDRAM from Rambus [Crisp 1997] will serve as our baseline dedicated texture memory. These devices provide extremely high bandwidth (a sustainable 1.6 GB/sec) with reasonable latency (90 nsec) at high densities for commodity prices. We estimate that on-chip buffering logic adds 10 nsec of latency to this memory.
- **rdram2x.** In order to sustain the high and variable bandwidth requirements of scenes such of *flight2x* and *qtvr2x*, a texture architecture may choose to utilize 2 RDRAM parts for double the bandwidth of *rdram* at the same latency.
- **numa.** Although not based on any existing specification, we use the *numa* memory model to examine the feasibility of our prefetching architecture in novel and exotic texture memory architectures. The bandwidth of this memory model is the same as the bandwidth of *rdram2x*, but the latency of such a system is extremely high and highly variable. It can range anywhere between 250 nsec and 1.25 usec. This latency is in the range of what can be expected if texture is distributed across the shared memory of a NUMA multiprocessor [Laudon & Lenoski 1997] and is typical of the behavior that will be seen in shared texture memory systems analyzed in Section 4.2.

4.1.6 Performance Analysis

A cycle-accurate simulator was written to validate the robustness of the prefetching texture cache architecture proposed in this section. We analyze the architecture by running each scene with each memory model. First, the architecture is compared against an ideal architecture and an architecture with no prefetching. We then account for all of our execution time beyond the ideal execution time, demonstrating the architectures ability to hide nearly all the latency in the system.

Figure 4.6a presents the execution time for each of the scenes with each of the memory models on both our architecture and an architecture with no prefetching. Performance is normalized to the ideal execution time of 1 cycle per fragment. In all cases, our

	<i>agp</i>	<i>rdram</i>	<i>rdram2x</i>	<i>numa</i>
period	16	8	4	4
latency				

Table 4.2: Memory Models

The values reported here are in terms of a fragment clock cycle of 200 MHz, which corresponds to 5 nsec. The memory period determines the rate at which 64 byte blocks of memory can be provided. Thus, bandwidths of 1, 2, 4, and 4 texels per fragment are provided on *agp*, *rdram*, *rdram2x*, and *numa*, respectively.

4.1.5.3 Memory Models

In order to validate our texture prefetching architecture more precisely, we now explore the bandwidths and latencies provided by memory systems. For our study, we examine an architecture that can sustain the texturing performance projected for the near future. At the time of this study (1998), high-end architectures such as the SGI InfiniteReality [Montrym et al. 1997] provided approximately 200 million trilinear fragments per second from a single board. Low-end professional-level architectures provided approximately 30 million trilinear fragments per second [Kilgard 1996], as do many consumer-level graphics accelerators. Given these rates, we decided to set our nominal fragment clock rate at 200 MHz, meaning that under optimal memory conditions, the architecture provides a trilinearly sampled fragment every 5 nanoseconds. Based on this fragment clock rate, we decided to simulate four memory models, summarized by bandwidth and latency histogram in Table 4.2.

- **agp.** This models a system in which the texture cache requests blocks from system memory over Intel's Advanced Graphics Port [Intel 1998]. The AGP 4X standard can provide a sustained bandwidth of 800 MB/sec. Because system memory is shared with the host computer, we estimate that the latency of *agp* varies between 250 nsec and 500 nsec.

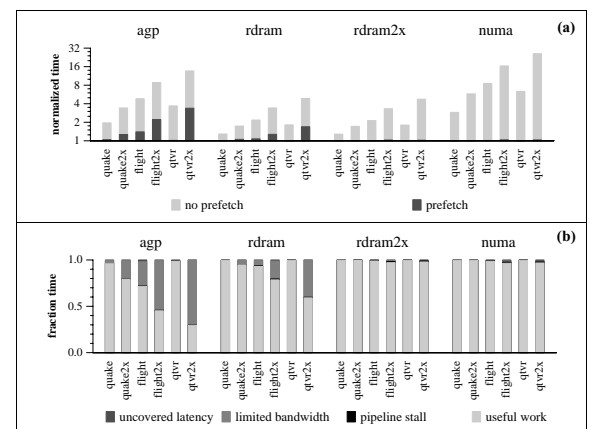


Figure 4.6: Prefetching Performance

In (a), we compare our prefetching architecture and one in which no prefetching takes place against an ideal architecture (one in which a fragment is generated on every clock cycle) on a logarithmic scale. On many configurations, the prefetching architecture is able to achieve near-ideal performance (as indicated by the near-total absence of a dark gray bar). Configurations that do not achieve near-ideal performance are bandwidth-limited, as illustrated in (b). This graph characterizes the architecture's execution time by useful work, pipeline stalls, limited memory bandwidth, and uncovered latency across the four memory models and the six scenes. For all of the cases in which near-ideal performance was not attained, memory bandwidth is by far the limiting factor. Thus, the architecture is able to hide nearly all of the latency of the memory system with little overhead.

architecture performs much better than an architecture lacking prefetching. However, we do not achieve an ideal 1 cycle per fragment across many of the scenes when running the *agp* and *rdram* memory models.

In order to account for lost cycles, we enumerate four components of our architecture's execution time:

- 1) A cycle is required to move each fragment through the texture pipeline.
- 2) If either cache has more than one miss for any fragment, the pipeline must stall.
- 3) The pipeline may stall due to insufficient texture memory bandwidth.
- 4) Cycles may be lost to uncovered latency in the prefetching architecture.

Each of these components can be calculated as follows. The number of cycles spent moving fragments through the pipeline is simply the number of fragments in the scene. The number of pipeline stalls attributed to multiple misses per fragment can be measured by counting the number of misses per cache per fragment beyond the first miss. Stalls occur infrequently, and our experiments show the performance lost to such pipeline stalls is typically less than 1%. Performance lost to insufficient memory bandwidth is determined by the execution time of the trace with the memory latency set to zero. Finally, when the scene is simulated with our memory latency model, any additional cycles not attributed to the first three categories are counted as uncovered latency in our architecture. Experimental results show that most of the latency of the memory system is indeed covered by our architecture, with at least 97% utilization of hardware resources using nominal sizes for the fragment FIFO, the memory request FIFO, and the reorder buffer. Most of the performance difference from an ideal system is caused by insufficient memory bandwidth. The breakdowns of the execution times for our configurations are presented in Figure 4.6b.

4.1.6.1 Intra-Frame Variability

A typical scene provides both an overall memory bandwidth demand over the course of the frame (several milliseconds) as well as localized memory bandwidth demands over several microseconds, as illustrated in Figure 4.5. Figure 4.7 shows how this translates into lost performance. The performance of the *quake2x* scene on the *agp* memory system is very different in the first and second half of the frame due to switching between color map textures and light maps. As predicted in Section 4.1.5.2, the fragment rate while drawing the color texture is limited by memory bandwidth while the pipeline runs at full

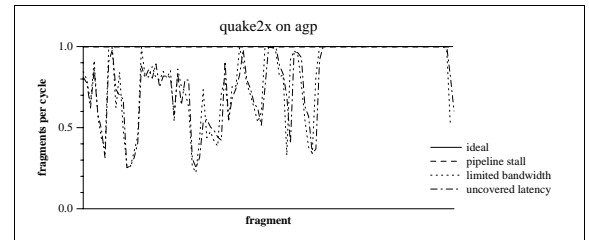


Figure 4.7: Time-Varying Execution Time Characterization

As predicted in Section 4.1.5.2, the performance of a workload can vary greatly over time if not enough memory bandwidth is provided. The graph above characterizes the execution time of the *quake2x* workload on the *agp* memory system. Even though the 1 texel per fragment bandwidth of *agp* by far exceeds *quake2x*'s average requirement of 0.63 texels per fragment, performance suffers due to the time-varying bandwidth requirements of *quake2x*.

speed while drawing the light maps. This does indeed cause an overall performance penalty even though the 1 texel per fragment bandwidth of *agp* far exceeds the average texel per fragment bandwidth requirement of *quake2x*. Figure 4.7 also illustrates that the performance of our architecture closely tracks the performance of a zero-latency memory system over time.

4.1.6.2 Buffer Sizes

The data in Figure 4.6 and Figure 4.7 was derived with a specific set of buffer sizes for each memory model. These sizes are presented in Table 4.3, and in all cases the buffers are reasonable in size when compared to the 16 KB of cache employed.

We determined the sizes of the three buffers—the fragment FIFO, the request FIFO, and the reorder buffer—by inspection and then validated them by experimentation. The fragment FIFO primarily masks the latency of the memory system. If the system is not to stall on a cache miss, it must be able to continually service new fragments while previous fragments are waiting for texture cache misses to be filled. Thus, the fragment FIFO

	Fragment FIFO Size	Request FIFO Size	Reorder Buffer Size
<i>agp</i>	128 slot 2.0 KB	8 slot 64 byte	8 slot 576 byte
<i>rdram</i>	64 slot 1.0 KB	8 slot 64 byte	8 slot 576 byte
<i>rdram2x</i>	64 slot 1.0 KB	16 slot 128 byte	16 slot 1.1 KB
<i>numa</i>	256 slot 4.0 KB	16 slot 128 byte	64 slot 4.5 KB

Table 4.3: Buffer Sizes

The numbers in each entry represent the sizes of the various buffers used in the various memory systems. Fragment FIFO entries are 16 bytes, memory request FIFO entries are 8 bytes, and reorder buffer entries are 72 bytes.

depth should at least match the latency of the memory system. The fragment FIFO also provides elasticity between the burstiness of texture misses and the constant rate at which the memory system can service misses, and therefore should be larger than just the memory system latency. The memory request FIFO also provides elasticity between the potentially bursty stream of miss addresses generated by the fragments and the fixed rate at which the memory system can consume them. The size of this buffer was determined primarily by experimentation. Finally, in order to provide a robust, deadlock-free solution that can handle out-of-order memory responses, our architecture requires that a reorder buffer slot be reserved when a memory request is made. Since a memory response will not be received and applied to the cache at least until after the memory latency has passed, the reorder buffer should be sized to be at least the ratio of the memory access time (latency) to the memory cycle time (period) entries deep.

The above guidelines were used to determine the approximate buffer sizes for each memory model, and then the choices were adjusted by measuring the performance of the system. We fine-tuned the buffer sizes by holding two of the buffer sizes constant and varying the third. If the buffer is sized appropriately, the performance of the overall system should decrease significantly when the buffer is made much smaller, and performance should increase very slowly if the buffer is made larger. The data for this process with the *flight2x* workload is shown in Figure 4.8. This process provided useful informa-

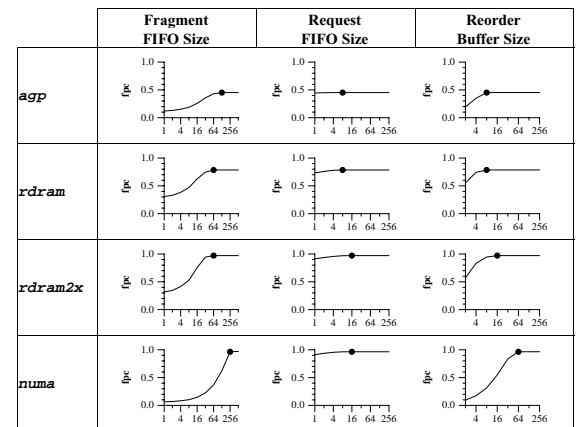


Figure 4.8: The Effects of Varying Buffer Sizes

The graphs above show the effects of varying buffer sizes on the *flight2x* workload across the different memory models. For each graph, one buffer size is varied while the other two are held fixed (at the values specified in Table 4.3). The results are reported in fragments per cycle (fpc), and the dot on each graph represents the final values used for the architecture on each memory model. The memory models whose fragments per cycle values do not approach 1.0 are bandwidth-limited.

tion in the cases of the *rdram* and *rdram2x* memory systems. The fragment FIFOs were originally sized to be 32 entries deep. However, simulation revealed that this did not provide enough elasticity, and increasing the FIFO depth to 64 entries improved performance by several percent. Similarly, simulation revealed that performance increased slightly when the reorder buffer size was increased to 8 slots and 16 slots for *rdram* and *rdram2x*, respectively.

One note should be made about the performance analysis of this section. In formulating a model for measuring the performance of our prefetching texture cache architecture, we assumed that the entire scene is rasterized by a renderer that is able to provide a fragment to the texture subsystem on every clock cycle. In a real system, this may not be the case. When triangles are smaller, caching does not work as well; but smaller triangles may also imply a lower fill rate (i.e., the scene is geometry limited), thus alleviating some of the penalty associated with the caching. A more detailed analysis of bandwidth requirements in a rasterization architecture can take this effect into account.

4.2 Parallel Texture Caching

The creation of high-quality images requires new functionality and higher performance in real-time graphics architectures. In terms of functionality, texture mapping has become an integral component of graphics systems, and in terms of performance, parallel techniques are used at all stages of the graphics pipeline. Two types of parallel texturing subsystems may be created: in a system with *dedicated texture memory*, each texturing unit has a dedicated texture memory that replicates all of the texture data in the system. As we demonstrated in Section 4.1, nearly all the latency of a texture cache may be hidden, thus making texture bandwidth the critical bottleneck in the texture memory subsystem. However, parallel rasterization divides work across multiple functional units, thus decreasing the locality of texture references and increasing the amount of texture bandwidth required in a parallel system. This can have an adverse effect on the scalability of the texturing performance, and in this section, we examine scalability related to this phenomenon. In a system with *shared texture memory*, each texturing unit may access shared texture memory from anywhere in the system, thus scaling the amount of texture memory in the system. However, this introduces two additional issues: variable latency for texture requests and load imbalance among the texture memory due to data distribution. We addressed latency in Section 4.1, and we will address data distribution in this section.

tion network. Because each of the independent pipelines has its own texturing unit and texture memory, the amount of texture memory available to an application could be scaled. However, the problem is not straightforward since there must be explicit software control to only send primitives to the particular pipeline holding its texture data. Such a scheme is greatly at odds with dynamically load balancing the amount of work in each pipeline, particularly given the irregularities of human interaction. If shading (and thus, texturing) is deferred until after the pixel merge is completed, the problem of dynamically load balancing shading work according to texture accesses is equally, if not more, challenging. There have been no published works to date that address these challenges. As with other parallel hardware architectures, the PixelFlow replicates texture memory [Molnar 1995]; furthermore, the locality of texture access is not exploited.

Vartanian et al. [Vartanian et al. 1998] have evaluated the performance of texture caching with both image-space parallel and object-space parallel rasterizers. They find that while object-space parallelism provides good speedup in a caching environment, image-space parallelism generates poor speedup. We believe that these results can be attributed to focused architectural choices and benchmark scenes that favor object-space parallelism both in terms of caching and rasterizer load balancing. In contrast, we find it more insightful to separate rasterizer load imbalance from texture load imbalance, and by exploring a more complete set of architectural choices, we find efficient design points for texture caching with both types of parallelism.

4.2.2 Parallel Texture Caching Architectures

A serial graphics pipeline is illustrated in Figure 4.9; performance can be increased by deploying multiple copies of some or all of the stages. Parallel rasterization distributes rasterization work amongst multiple copies of the rasterization stages. Looking at the texturing stage specifically, the role of the texture mapping units in a system is to take as input untextured fragments with texture coordinate information, access the appropriate data in the texture memory based on these coordinates and filtering modes, filter the data, and combine this filtered texture value with the value of the untextured fragment. In or-

Specifically, we quantify the effects of parallel rasterization on texture locality for a number of rasterization architectures, representing both current commercial products and proposed future architectures. A cycle-accurate simulation of the rasterization system demonstrates the parallel speedup obtained by these systems and quantifies inefficiencies due to redundant work, inherent parallel load imbalance, insufficient memory bandwidth, and resource contention. We find that parallel texture caching works well and is general enough to work with a wide variety of rasterization architectures.

4.2.1 Previous Work

Until recently, it had been difficult to provide the amount of computation required for texturing at high fragment rates within a single chip, so solutions were naturally parallel. Although texturing was used in the earlier dedicated flight simulators, one of the first real-time texture mapping workstations was the SGI RealityEngine [Akeley 1993]. This system parallelizes rasterization by interleaving vertical stripes of the framebuffer across 5, 10, or 20 fragment generator units. Each fragment generator is coupled with an independent texture memory. Because texture access patterns are independent of framebuffer interleaving patterns, any fragment generator needs to be able to access any texture data, so each fragment generator replicates the entire texture state. The successor to the RealityEngine, the InfiniteReality [Montrym et al. 1997], uses 1, 2, or 4 higher performance fragment generators, and thus replicates texture memory up to 4 times, instead of up to 20 times. The texture subsystems of these architectures made minimal use of texture locality. The stripe interleaving of rasterization used in aforementioned high-end machines has recently appeared as scan-line interleaving in consumer-level graphics accelerators such as the Voodoo2 SLI from 3Dfx. As with the SGI systems, texture memory is replicated across all the rasterizers.

One other class of scalable graphics architectures in which texture mapping has been implemented is the image composition architecture, as exemplified by PixelFlow [Molnar et al. 1992]. In such a system, multiple independent pipelines each generate a subset of the pixels for a scene, and these pixels are merged for display through an image composi-

der to scale the fragment rate (i.e., the rasterization performance), the number of texturing units must be increased to provide the necessary processing power. Additionally, the number of texture memories must also be scaled to provide the correspondingly increased bandwidth requirements.

Figure 4.10a shows a *dedicated* texture memory scheme for scaling the texture subsystem of a graphics pipeline. Each additional rasterization pipeline brings with it a dedicated texturing unit and texture memory. As the system scales, the total amount of texture memory increases, but due to replication, the *unique* texture memory remains constant. Figure 4.10b diagrams a *shared* texture memory scheme for scaling the graphics pipeline. In this architecture, an all-to-all texture-sorting network is introduced between the texturing units and the texture memories. This allows any texturing unit to access the data in any texture memory, allowing a single shared image of the texture data to be present across all of the texture memories. Many topologies exist for such networks [Duato et al. 1997], and highly scalable networks can be built if the system balances the data going in and out of the network. We will not focus on the network in this thesis, but the specifics of such a network can be found in other works [Eldridge et al. 2000].

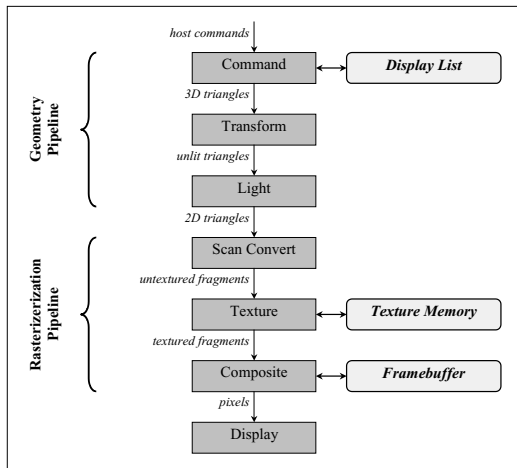


Figure 4.9: A Base Graphics Pipeline

The above diagram illustrates a typical graphics pipeline. A parallel rasterization architecture replicates the rasterization pipeline to achieve higher performance.

With the architectures of Figure 4.10a and Figure 4.10b, as with any parallel system, it is important to minimize the amount of redundant work introduced by parallelization and to balance the amount of work in each processing unit. Efficient parallel rasterization algorithms deal with presenting each texturing unit with a balanced number of untextured fragments that minimizes redundant work; this problem has been extensively studied, and we make use of a few such algorithms, as described in Section 4.2.3.1. The main focus of this section is to study the effects of parallel rasterization on texture locality. Assuming that the number of untextured fragments presented to each texturing unit is balanced, one requirement for good parallel performance is that the redundant fetching of

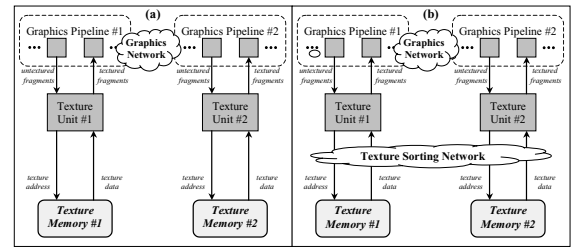


Figure 4.10: Dedicated and Shared Texture Memories

Multiple graphics pipelines simultaneously draw a scene by coordinating work over a graphics network. To apply texture, a fraction of the untextured fragments is distributed to each texture unit that holds a replicated version of the scene's texture data in a dedicated texture memory (a). In a shared texture memory system (b), each texturing unit can access the texture data of any texture memory, allowing for a single copy of the texture data system-wide. Texture is cached to reduce texture memory bandwidth.

the same texture data across texturing units be minimized. Furthermore, it is important to load balance the texture bandwidth required by each texturing unit, and in the case of a shared texture memory, the texture bandwidth required from each texture memory.

4.2.3 Methodology

While it is clear that the parallel architectures of Figure 4.10a and Figure 4.10b do potentially increase performance, the actual performance gains are still unclear in terms of texture bandwidth. In this section, we lay out a framework that will allow us to evaluate the performance of a parallel texture caching architecture.

4.2.3.1 Parallel Rasterization Algorithms

The characteristics of parallel texture caching are highly dependent on the parallel rasterization algorithm because this algorithm determines which fragments are processed by which texturing units and in what order. There are a great number of different rasteri-

zation algorithms, and each algorithm has a number of parameters that can be varied. Because of the large number of variables, it is impractical to analyze every rasterization algorithm, and thus we choose a few representative algorithms.

Parallel rasterization algorithms can be characterized along three axes with regard to texturing. The first distinction to be made is whether work is partitioned according to image-space (each texturing unit is responsible for a subset of the pixels on the screen) or object-space (each texturing unit is responsible for a subset of the fragments generated by triangles). The second distinction is whether the texturing unit processes fragments immediately in the order primitives are submitted or buffers fragments and processes them in a different order. The third distinction is whether fragments destined for the same location in the framebuffer are processed in the order presented by the application. For this paper, all of the algorithms we present preserve application order.

- **tiled.** In a tiled architecture, the screen is subdivided uniformly into fixed-size square or near-square tiles and each texturing unit is responsible for a statically interleaved fraction of tiles. We have empirically found that 32 pixel tiles work well up to moderate levels of parallelism, and for this paper, we will assume that tile size. In *tiled-prim*, fragments are processed in primitive order. This means that if a triangle overlaps several tiles belonging to the same rasterizer, the fragments of that triangle are completely processed before moving on to the fragments of the next triangle. In *tiled-frame*, the fragments of a frame are processed in tile order. This means that a texturing unit processes all of the fragments for its first tile before moving on to any of the fragments that fall in its second tile.
- **osi.** Algorithms that subdivide work according to object-space usually distribute groups of primitives in a round-robin fashion amongst rasterizers, giving each rasterizer approximately the same amount of per-primitive work. Because the number of fragments generated by each primitive can vary greatly, it is important to also load balance fragment work either by dynam-

cally distributing primitives, by subdividing large primitives, or by combining the two techniques. In *object-space ideal (osi)*, we idealize the load balancing of fragments. First, we serially rasterize all the primitives to form a fragment stream, and then we round-robin groups of 1024 fragments amongst the texturing units.

- **striped.** Similar to both the RealityEngine and the InfiniteReality, fragments are subdivided according to an image-space subdivision of 2 pixel-wide vertical stripes. Each texturing unit is responsible for an interleaved fraction of the stripes, and processing is done in primitive order, as in *tiled-prim*.

4.2.3.2 Scenes

In order to quantify the effectiveness of parallel texture caching, we need to choose a set of representative scenes that cover a wide range of texture locality. A good measure of texture locality is the scene's unique texel to fragment ratio, and this ratio varies over nearly two orders of magnitude in our test scenes, which are identical to those presented in Section 4.1.4.2 and Table 4.1. All of these scenes make use of mip mapping for texture filtering. Mip mapping is crucial for providing locality in texture access patterns under minification, a characteristic that all texture caching rasterization architectures depend upon to run at full speed. Scenes that lack mip mapping will experience significant performance degradations under texture minification. The scenes we use in this section load balance fragment work relatively well with respect to the parallel rasterization algorithms of Section 4.2.3.1, as will be quantified in Section 4.2.4.3. Because these scenes load balance well under our parallel rasterization algorithm, texture bandwidth imbalance will not be hidden by fragment imbalance.

4.2.3.3 Simulation Environment

A cycle-accurate simulator of a parallel texturing subsystem was written in C++ to provide an environment for collecting the data presented in this paper. Our simulation infrastructure is based on a methodology for simulating hardware architectures [Mowry 1999].

The simulator takes as input texture data and untextured fragment data, and produces rendered images as well as trace data on the texture subsystem. The simulator is able to partition this fragment data among multiple instances of texturing units in accordance with the parallel rasterization algorithms of Section 4.2.3.1. Each texturing unit and each texture memory is made up of multiple functional units that run as threads in the C++ environment. The forward progress of each functional unit and the communication between functional units adhere to a simulation clock by making clock-awaits function calls within each thread at the appropriate points in the code. This allows us to simulate a graphics architecture with cycle-accuracy at varying levels of detail and collect data from the system in a non-intrusive fashion.

4.2.3.3.1 Data Organization

Given the high-level architecture of parallel texturing units that are connected to memories through a texture cache, we must decide how data is organized throughout the system. In accordance with Section 4.1, we group 2D texture data into 4D tiles so that each

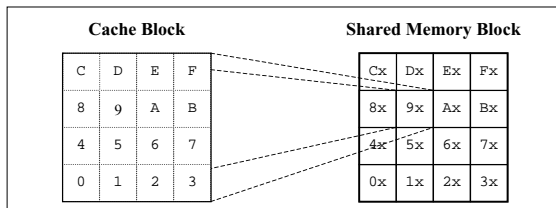


Figure 4.11: Shared Texture Data Organization

The above diagram correlates a location in the shared texture memory with an address, expressed in hexadecimal. An ‘x’ represents an arbitrary hexadecimal number. In this example, we are illustrating 16 texture memories with a block size of 16 texels. The left block shows the layout of a 4x4 cache block. A 4x4 grid of these 2D blocks gives rise to a 16x16 image laid out in the shared texture memory, illustrated in the center. The least significant hexadecimal digit of any texture address determines the pixel within the cache block, and the second least significant hexadecimal digit determines the texture memory holding that block.

cache block holds a square or near-square region of texture and use an additional level of tiling (6D tiling based on the number of cache sets) to reduce conflict misses. As before, we rasterize according to screen-aligned 8 by 8 tiles and another level of tiling to rasterization (every 32 by 32 pixels), resulting in 6D tiled rasterization. To give a consistent rasterization order across the studies in this paper, a serial rasterizer generates untextured fragments in this order and distributes them to the appropriate texturing unit according to the parallel rasterization algorithm.

For the shared texture memory architecture, we must decide on the distribution of texture data across the multiple texture memories. Texture data should be distributed in a finely interleaved fashion to prevent hot-spotting on any single memory for any significant period of time. In order to minimize the chance that nearby texture tiles fall onto the same texture memory, we distribute each cache block of texture data across the texture memories in a 4D tiled fashion. The exact parameters for this tiling are dependent on the cache block size and the number of texture memories used for a particular simulation. For example, with 16 texel cache blocks (organized in a 4 by 4 tile) and 16 texturing memories, each cache block in a 4 by 4 tile of cache blocks is given to a different texture memory. This is illustrated in Figure 4.11.

4.2.3.3.2 Performance Model

While caching characteristics may be analyzed statically without a performance model, such a model must be introduced in order to analyze resource contention and parallel speedup. Our simulated texturing unit is capable of texturing a single fragment every cycle, and we provide 2 texels per cycle of bandwidth to each texture memory, an amount large enough to cover most of the bandwidth demands of our scenes. This is a typical bandwidth in modern systems – see, for example, a calculation by Kirk [Kirk 1998]. The latency of each texture memory is set to 20 fragment clocks, and a 64 fragment FIFO is used to hide the latency of the memory, values we replicate from Section 4.1. Because arbitrary amounts of latency can be hidden using prefetching and a fragment FIFO, our results are not dependent on these values.

In a serial texturing unit, a fragment FIFO serves not only to hide the latency of the memory system, but also to smooth out variations in temporal bandwidth requirements. Even if a scene’s overall bandwidth requirement is low, temporal bandwidth requirements can get extremely high when several consecutive fragments miss in the texture cache. If this temporal imbalance is microscopic (e.g., over tens of fragments), then a fragment FIFO can smooth out the contention for the memory system. However, this imbalance is often macroscopic (e.g., over tens of thousands of fragments): a fragment FIFO is unable to resolve the fragment-to-fragment contention for the texture memory and performance suffers.

In a parallel texture caching system with shared texture memories, contention can also occur between texturing units for the texture memories, and thus, the network. In order to reduce the number of free variables in this paper, we choose to model the network as perfect (no latency and infinite bandwidth) and therefore focus on memory contention effects. Network contention is related to memory contention in a fully simulated system, and prefetching is able to successfully hide arbitrary amounts of network latency in texture caching (e.g., [Eldridge et al. 2000]).

4.2.4 Results

Parallel texture caching can be analyzed according to common parallel algorithm idioms. First, parallel texture caching incurs redundant work in the form of repeated texture data fetching. This reduction in locality is quantified in Section 4.2.4.1. The effect of multiple caches on working set size is described in Section 4.2.4.2. Second, it is essential that parallel texture caching be load balanced, and we quantify this in Section 4.2.4.3. Finally, in Section 4.2.4.4, we use a cycle-accurate simulation to demonstrate that good parallel speedup does in fact occur.

Contrary to traditional microprocessor cache studies, we present cache efficiency data in terms of bandwidth per fragment rather than miss rate per access. In microprocessor architecture, miss rate is of primary importance because only one or a few outstanding

misses can be tolerated before the processor stalls. Because of the lack of write hazards, texture caching can tolerate arbitrary numbers of outstanding reads (Section 4.1), and thus, performance is related more to its bandwidth demands.

4.2.4.1 Locality

As with most parallel algorithms, parallel texture caching induces inherent overhead beyond that found in a serial algorithm due to redundancies. For parallel texture caching, this is best characterized by the redundant fetching of the same texture data by multiple texturing units—a reduction of locality. In a serial graphics system, an ideal texture cache would fetch each texel used in the scene only once (assuming the cache is sized to exploit only intra-frame locality). The bandwidth required for such a cache can be computed by counting the number of compulsory misses (i.e., cold misses) taken by the cache that employs a block size of a single texel. As we make the block size larger, fewer misses are taken, but the amount of data read by each miss increases. Overall, we expect the total amount of data fetched due to compulsory misses to increase with the block size because of *edge effects*. Whenever a texture falls across the edge of a screen, the silhouette edge of an object, or the edge of a parallel work partitioning, larger block sizes force the cache to fetch larger portions of the texture data that are never used. By measuring the bandwidth attributable to compulsory cache misses, Figure 4.12 illustrates the reduction of locality caused by the various rasterization algorithms as the number of texturing units and block size are varied.

The lightest portions of the bars in Figure 4.12 indicate the average bandwidth required to satisfy the compulsory misses of a serial texture cache for the various rasterization algorithms. For a serial texturing unit, all of the algorithms perform equally because the number of compulsory misses is scene-dependent. We see that as block size is increased, the bandwidth requirement for a serial rasterizer increases slightly for the *flight* data set pair and negligibly for *quake* and *qivr* data set pairs. In *qivr*, the edge effects occur only near screen edges, which accounts for a negligible portion of the total work. In

quake, the texture used at the edge of polygons is repeated from the middle of the polygons, thus negating edge effects from polygons.

The bottom portion of each bar represents the optimal bandwidth requirements of a serial texture cache, and each successive portion of the bar represents the additional bandwidth required to satisfy additional texturing units. We simulate an *infinite* number of texturing units, the top-most portion of each bar, by assigning the smallest granularity of work for each rasterization algorithm to a unique texture unit. For a tiled architecture this quantum of work corresponds to a single tile, for object-space interleaving this corresponds to a single contiguous block of fragments. This defines the locality present in a rasterization algorithm's minimal unit of work. Also note that because we are counting compulsory misses, the order in which fragments are processed has no effect, and thus the results for *tiled-prim* and *tiled-frame* are identical.

As a detailed example, for a *tiled* architecture on the *flight2x* scene, we see that for a block size of 16 texels (arranged in a 4 by 4 tile), a single texturing unit requires approximately 1.67 texels per fragment. If work is distributed amongst two texturing units, then the bandwidth required increases to approximately 2.17 texels per fragment. This occurs because edge effects are introduced at the boundaries of tiles, reducing the tile-to-tile locality. For four texturing units, the bandwidth requirement slightly increases to 2.26 texels per fragment, but as work is distributed amongst additional texturing units, the bandwidth requirements do not increase significantly. The reason for this is that most of the texture in the scene is unique, and while the tiles of a two-way parallel system touch at their corners and thus share some of the texture data of an object (tile-to-tile locality), this adjacency goes away completely in four-way parallel and larger systems. We also see that as block size is increased from 1 to 16 to 64 texels, the bandwidth requirements increase significantly because the over-fetching of larger block sizes is multiplied by the large number of tile edges. These aforementioned behaviors are all mirrored in *flight*, *qtrvr*, and *qtrvr2x*.

Although the effects of larger block sizes are the same, the bandwidth requirements of *quake* and *quake2x* on the *tiled* architecture are quite different as the number of rasteriz-

ers is increased. The first thing to notice about these scenes is the low bandwidth requirements of the serial case due to the heavy use of repeated textures. Furthermore, as opposed to the other scenes, as the number of texturing units is increased, the bandwidth requirements always increase. The use of repeated textures causes this because the same texture data is used repeatedly across the image-space partitioning. However, even with an infinite number of texturing units, the total bandwidth requirement is still quite limited. In effect, although parallel rasterization diminishes texture locality due to repeated texture, locality due to filtering remains. This means that texturing subsystems that are designed to perform well only in conjunction with repeated textures do not parallelize well.

The behavior of *osi* largely mirrors the performance of *tiled*, with the exception that bandwidth requirements continue to increase as additional texturing units are utilized. This is explained by the fact that *osi* is fragment-interleaved, and the chance that a texturing unit's consecutive fragment groups utilize adjacent portions of a texture map decreases smoothly as the number of texturing units is increased. For both *tiled* and *osi*, we see that a block size of 16 texels provides reasonable locality given the granularity of access needed for efficient memory utilization and efficient network utilization. Thus, for the remainder of the section, we assume a block size of 16 texels for *tiled* and *osi*.

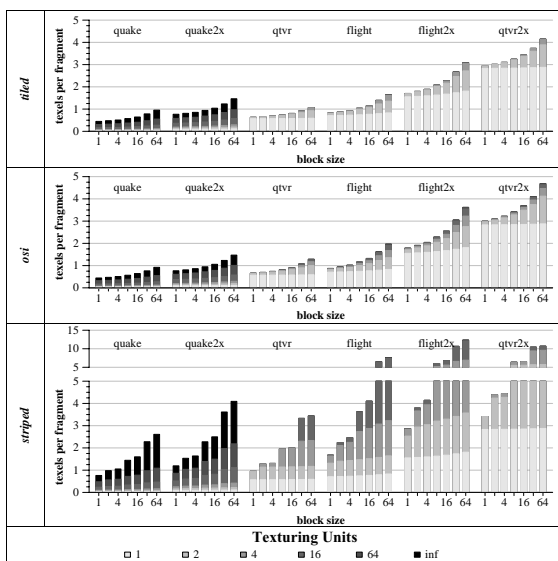


Figure 4.12: Bandwidth Due to Compulsory Misses

This study shows the bandwidth requirements (measured in average number of texels fetched per fragment) associated with compulsory misses as a function of rasterization algorithm, scene, block size, and number of texturing units. The top row represents the data for the *tiled* rasterization architecture, the middle row for the *osi* architecture, and the bottom row for the *striped* architecture. Scenes are sorted left to right by their unique texel to fragment ratio, which indicates the minimum bandwidth required. Each bar chart shows the bandwidth requirements for a different block size, and the shades of gray show the bandwidth requirements for differing numbers of texturing units. The shades of gray increase in darkness as the number of texturing units is increased, and the bandwidth required for greater numbers of texturing units increases monotonically. Finally, note that the bandwidth values for *striped* rasterization are shown with a split scale axis.

The behavior of the *striped* rasterization algorithm is markedly different from both *tiled* and *osi*. The most important thing to notice is that bandwidth requirements increase dramatically with increased block size. Because interleaving is done at every 2 pixels in the horizontal direction, edge effects occur very frequently. As block size is increased, a drastically larger number of texels that fall beyond a stripe's required set of texels are fetched. Thus, striped architectures reduce texture locality drastically. Even at a block size of 1 texel for *striped*, locality is much worse than at a block size of 16 for *tiled* or *osi*. We note that a single texel is a very small granularity of access for modern networks and memories, and that most modern devices perform highly sub-optimally at such granularities. Nonetheless, this is the only block size that preserves a modicum of locality for *striped*, and for the remainder of the section, we assume a block size of 1 texel for the *striped* architecture.

4.2.4.2 Working Sets

Now that we have an understanding of the effects of cache block size on locality under parallel rasterization, we move onto the effects of parallel rasterization on working set sizes by using limited-size caches that suffer misses beyond compulsory misses. As the number of texturing units is increased, the total amount of cache in the system increases, and thus we expect better performance. Figure 4.13 quantifies this notion by showing the bandwidth requirements of the various architectures with differing numbers of texturing units for the *flight2x* data set as the total cache size is varied. In general, there is a correlation between an algorithm's working set size and the point of diminishing returns in increasing cache size, illustrated as the "knee" in the curves of Figure 4.13. We see that as the number of texturing units increases, the working set size for each texturing unit decreases.

These same characteristics were found for all of the data sets. Because we want to pay attention to low levels of parallelism and systems that scale a serial texturing unit, we focus on a single cache size that works well for a serial algorithm. Choosing such a parameter outside of hardware implementation constraints is a bit of a black art, and thus

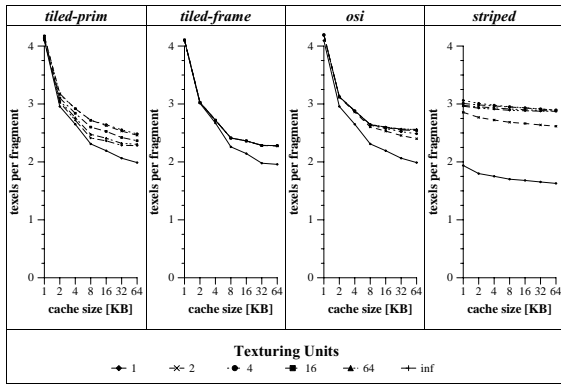


Figure 4.13: The Effects of Cache Size

The total bandwidth (measured in average number of texels per fragment) required to render *flight2x* is plotted as a function of the cache size. Each chart shows a different rasterization architecture, and each curve represents a different number of texturing units. Block size is set to 16 texels for all the graphs except *striped*, which has a block size of 1 texel.

we use parameters from Section 4.1.5.1 for consistency's sake and allocate a cache size of 16 KB (configured as two direct-mapped 8 KB caches) for the remainder of this paper. Figure 4.14 shows the bandwidth requirements of the various algorithms on the various scenes with a 16 KB cache. The first trend we notice is that while there is an initial jump in bandwidth demand when going from one texture unit to two texture units, the bandwidth demands are largely flat with increasing numbers of texture units. Moreover, for some traces, particularly *flight* and *flight2x*, the bandwidth demands actually decrease after the initial increase. This is a well-known phenomenon from parallel systems wherein the aggregate cache size increases more rapidly than the aggregate miss rate, resulting in improved cache behavior with increasing parallelism.

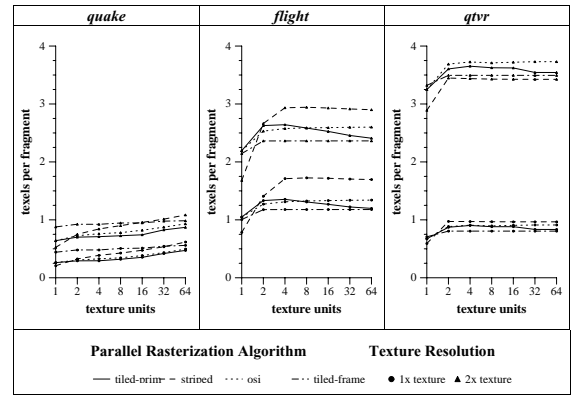


Figure 4.14: Bandwidth Requirements of a 16 KB Cache

In these graphs, bandwidth is displayed as a function of the number of rasterizers. Both the normal and the 2x resolution versions of each scene are shown on the same graph. Block sizes are the same as in Figure 4.13, and each curve shows the bandwidths for a different parallel rasterization algorithm.

One interesting result is that although *tiled-frame* performs better than *tiled-prim* for the *flight* and *qtvr* data set pairs, the opposite is true for the *quake* data set pair. In *flight*, and to a lesser extent *qtvr*, the disjoint drawing of triangles in image-space makes it advantageous to wait until all of the triangles of a tile are present before texturing due to increased temporal locality. In *quake*, however, it is more advantageous to texture large polygons that fall into multiple tiles immediately because the different regions of the polygon all use the same repeated texture data.

4.2.4.3 Load Imbalance

The performance of any parallel system is often limited by load imbalance: if one unit is given significantly more work than the other units, it will be the limiting factor, and per-

formance will suffer. In parallel texture caching, load imbalance can occur in one of three ways. First, the number of untextured fragments presented to each texturing unit can differ. Second, the bandwidth required for texturing the fragments of a texturing unit may vary. Third, the bandwidth required from each texturing memory can differ. In a dedicated texture memory system, the last two sources of imbalance are identical because each texturing unit is paired with a single texture memory.

Figure 4.15 shows the various types of load imbalance of the various scenes on the different architectures. The first trend to note is that all of the configurations load balance well in all respects when there are 16 or fewer texturing units (the worst imbalance is 9.7%). However, as the number of texturing units is increased to 32, and especially 64, there is a large imbalance in the bandwidth requirements of the texturing units. This imbalance is significantly larger than fragment imbalance, and the trend occurs in all of the rasterization algorithms except *striped* and on all of the data sets except the *qtvr* pair, which exhibits extreme regularity in texture access patterns. The *striped* algorithm is highly load balanced even at high numbers of texturing units because of its fine interleaving pattern. However, this positive result is moderated by the fact that the baseline for the *striped* data includes significantly more redundant bandwidth than the other rasterization algorithms.

The second important trend in Figure 4.15 is the effect of shared texture memory on texture memory load imbalance. In a dedicated texture memory system, the load imbalance between texture memories is equal to the load imbalance between texturing units. In a shared texture memory architecture, the load imbalance between texture memories is relatively small. Thus, we see that distributing blocks in a tiled fashion across the texture memories does in fact balance texture load well, usually to such an extent that shared texture memory imbalance is much lower than the dedicated texture memory imbalance.

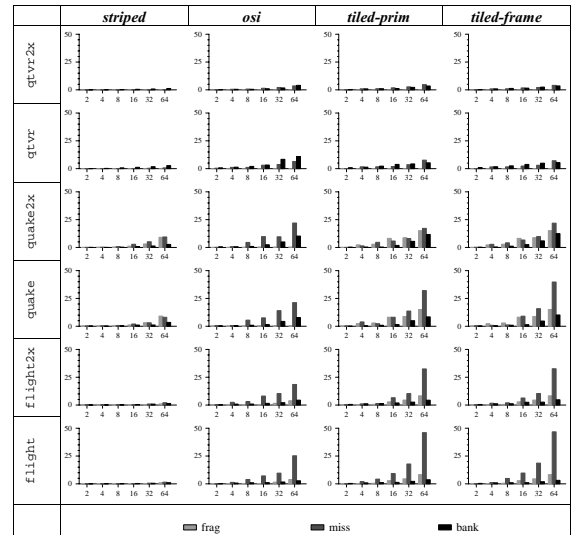


Figure 4.15: Texture Load Imbalance

Each graph shows load imbalance for different numbers of rasterizers. The y-axis of each graph shows the percent difference in the work performed by the busiest unit and the average unit. Each row shows a different scene, and each column shows a different parallel rasterization algorithm. The rows and columns have been sorted so that the scenes and rasterization algorithms that perform best are at the upper left and the ones that perform worst are at the lower right. Three types of load imbalance are shown. Fragment load imbalance is the maximum number of fragments given to a texturing unit divided by the average number of fragments per texturing unit. Miss load imbalance is the worst rasterizer's number of misses per fragment divided by the average number of misses per fragment. Bank load imbalance is the maximum number of access per texture memory in a shared memory architecture divided by the average number of accesses per memory. For these experiments, we use the same cache and block sizes as Figure 4.14.

4.2.4.4 Performance

While the experiments of the previous sections illuminate many characteristics of parallel texture caching, they say nothing about realized performance. In particular, the temporal effects of erratic intra-frame bandwidth demands are ignored. Even though a scene's memory bandwidth demands may be low when averaged over an entire frame, its memory bandwidth demands averaged over a few fragments can actually be quite high. Figure 4.16 divides the execution time of a serial texturing unit into three categories: time spent on fragment processing, time lost due to insufficient memory bandwidth, and time lost due to fragment-to-fragment contention for the memory. We see that only *flight2x*

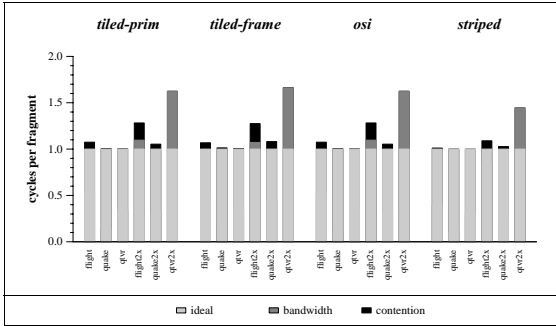


Figure 4.16: Breakdown of Serial Time

This graph breaks down the execution of a serial texturing unit across different scenes and rasterization algorithms. Execution time is normalized to cycles per fragment, and the light gray bar shows the ideal cost, assuming one fragment is processed per cycle. The dark gray bar shows the cost of insufficient memory bandwidth for a 2 texel per cycle memory system. If the ratio of a scene's average memory bandwidth requirement to the memory system's bandwidth supply is greater than one, then this cost is tallied. The black bar represents the cost of fragment-to-fragment memory contention incurred by the 64-entry fragment FIFO's inability to smooth out temporal bandwidth variations. These experiments use the same cache parameters as used in Figure 4.14.

and *qvr2x* have average memory bandwidth requirements beyond 2 texels per fragment, and thus even perfect smoothing of fragment-to-fragment contention could not achieve a performance of one cycle per fragment. We also see that contention time is nearly uniform across all rasterization architectures with the exception of *striped*, which performs better due to smaller block sizes. In general, uncovered contention occurs in scenes that have large variations in macroscopic bandwidth requirements. Whenever bandwidth requirements peak beyond 2 texels per fragment over large periods of time, temporal memory contention cannot be resolved by the 64-entry fragment FIFO. This occurs most in *flight2x*, and to a lesser extent, in *flight* and *quake2x*. In *qvr2x*, temporal bandwidth requirements are always over 2 texels per fragment, and in *quake* and *qvr*, bandwidth requirements are consistently under 2 texels per fragment, resulting in little temporal contention that is not covered by the 64-entry fragment FIFO.

The serial runs of Figure 4.16 serve as a baseline for computing speedup of parallel texture caching runs. In Figure 4.17, we graph the speedup of dedicated texture memory architectures. Across all of the runs, excellent speedup is achieved through 16 texturing units. For the scenes whose bandwidth requirements are usually met by the 2 texel per cycle memory system – *quake*, *quake2x*, *flight* (except in *striped*), and *qvr* – the speedup is near-linear. For the scenes that exceed this bandwidth – *flight2x* and *qvr2x* – the speedup efficiency is dictated by the inefficiency of the cache with respect to a serial cache, as graphed in Figure 4.16. Beyond 16 texturing units, some of the speedup curves exhibit lower speedup efficiency. Referring back to the load imbalance graphs in Figure 4.15, we see that this occurs in the configurations that exhibited significant load imbalance in the amount of bandwidth requested by each texturing unit. As expected intuitively, the fragment-to-fragment memory contention plays an insignificant role in speedup efficiency.

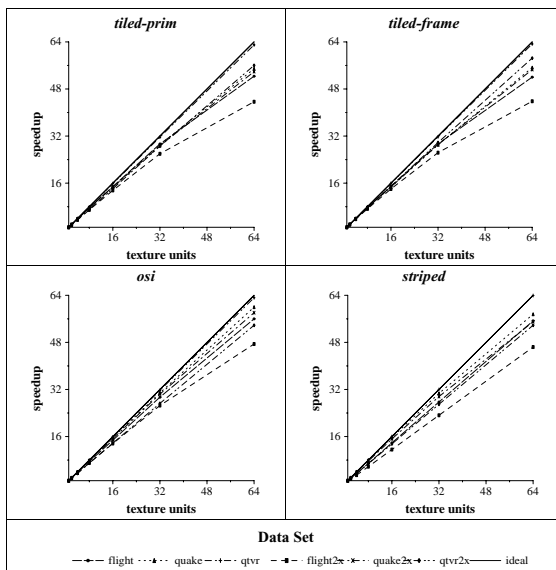


Figure 4.17: Speedup Graphs for Dedicated Texture Memory

These speedup graphs show speedup as a function of the number of texturing units for dedicated texture cache architectures. The curves on each graph show speedup for a different scene.

In Figure 4.18, we graph the speedup of a shared texture architecture. The speedup efficiencies realized are almost identical to a dedicated texture architecture at or below 16 texturing units. At higher numbers of texturing units, however, these speedup efficiencies are generally better than a dedicated architecture for the configurations that exhibited large load imbalance. This can be explained by the fact that in a shared texture architec-

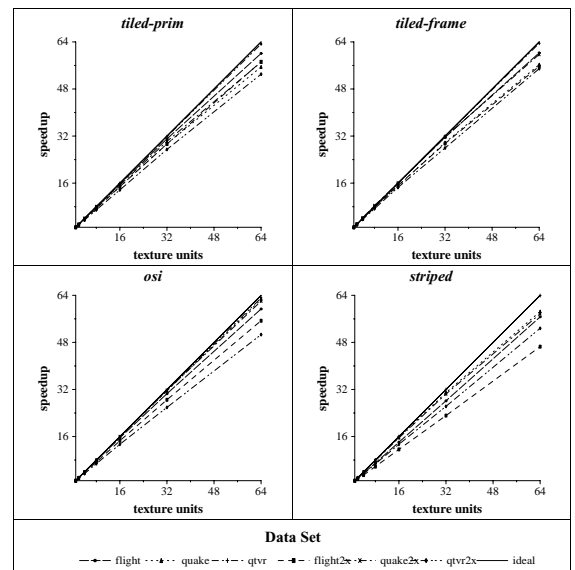


Figure 4.18: Speedup Graphs for Shared Texture Memory

These speedup graphs show speedup as a function of the number of texturing units for shared texture cache architectures. The curves on each graph show speedup for a different scene.

ture, the texturing unit with the highest miss rate spreads its memory requests across many texture memories, and thus performance becomes dependent on load imbalance amongst the texture memories, which is much lower than texturing unit imbalance. In effect, the busiest texture unit is able to steal memory bandwidth that would go unused in a dedicated texture memory system.

4.2.5 Texture Updates

One reason parallel texture caching is so straightforward to implement is that texture mapping is a read-only operation, except for texture downloads. This allowed us to use multiple caches on the same data without the need for a complex coherence protocol to detect changes in the texture memory caused by another texturing unit. However, any real system with a shared texture memory will have to deal with such texture updates. There are two potential hazards when a texture is updated. First, a texture read for a fragment generated by a polygon submitted before the texture update could read the new contents of the texture rather than the old contents. The converse could also occur when a fragment generated by a polygon submitted after a texture update reads texture values that are stale. These problems arise both because texture updates are large events that do not necessarily occur atomically, and, more simply, because the work being performed at any point in time by one texture unit (e.g., downloading a texture) is generally not tightly coupled with the work being performed by another texture unit (e.g., processing fragments).

One solution to these hazards is to force texture updates to occur atomically and to introduce a strict ordering of operations between the texturing units during texture update. This is most naturally expressed by performing a barrier operation across all of the graphics system at the start of the texture update to ensure that previous accesses to the old texture have completed. Then, a second barrier operation at the end of the texture update ensures that the new texture is in memory before any new fragments access it. The barriers could be implemented either in hardware via a shared signal among the texturing units, thus allowing the rest of the pipeline to make progress, or in the software driver, forcing a flush of the graphics pipelines. Additionally, texturing units must flush stale data from their caches in conjunction with texture updates.

4.3 Conclusion

In this chapter, we have demonstrated two key components to making scalable texture mapping hardware. First, we have shown that any amount of latency may be hidden by a proper texture prefetching architecture, even in conjunction with caching. The complete lack of data dependencies between different fragments for the texturing operation, as noted in Section 2.1, allows us to transform the texture memory access problem into one of bandwidth rather than latency. Applying this idea in general, we see that while latency is the critical factor in systems with a high degree of data dependency (e.g., microprocessors), bandwidth is the critical factor in systems with low amounts of data dependency (e.g., texturing). Second, we have shown that parallel texture caching works given the proper choice of algorithms and parameters. In particular, it is important to consider the effects of parallelization algorithms on texture caching. Algorithms that lead to good load balancing in fragment work (e.g., the *striped* architecture) also reduce the amount of texture locality significantly, crippling texture caches. Moderate granularity of work balances this tradeoff well. Even with this tradeoff, we noticed a surprisingly large amount imbalance in texture load at high degrees of parallelism (e.g., *tiled* on the *flight* dataset). Besides the obvious benefit of scaling the amount of texture memory available, a shared texture memory architecture balances this load among the texture memories significantly.

Chapter 5

Conclusion

In this thesis, we have examined scalability in graphics architectures from a novel point of view and provided ways of scaling two aspects of graphics architectures that have been ignored by previous work. We will review these contributions and describe several areas for future work.

In Chapter 2, we examined parallelism in graphics architectures. The graphics API defines the instruction set of a virtual graphics machine. Compared to a microprocessor architecture, we saw that graphics instructions define a much greater amount of computation: each instruction that specifies a triangle requires hundreds of floating-point operations, and each of the potentially millions of fragments generated by a triangle requires hundreds of fixed-point operations. Furthermore, few data dependencies exist within and between these graphics instructions, allowing graphics architectures to exploit large amounts of parallelism. The sorting taxonomy classifies the ways existing architectures have tried to exploit this parallelism, and the large variety in types of parallel architectures is related to the flexibility afforded by this parallelism.

In Section 2.2, we examined scalability in graphics architectures. Traditionally, researchers only looked at triangle rate and pixel rate in determining the scalability of graphics architectures. However, by only concentrating on these metrics, many researchers overlook other important aspects of scalability. In this dissertation, we presented a

novel set of quantitative (input rate, triangle rate, pixel rate, texture memory, display) and qualitative (mode semantics, frame semantics, ordering semantics) metrics on which graphics architectures may be compared, and we described techniques for scaling two of the metrics that have been ignored by previous work: input rate and texture memory. In fact, by using the techniques described in this dissertation in conjunction with a novel multi-sort algorithm, Pomegranate is able to achieve full scalability in all five quantitative metrics while satisfying all the qualitative semantic constraints required by OpenGL.

In Chapter 3, we described a novel parallel immediate-mode graphics interface. By introducing synchronization commands into the API, ordering between multiple graphics streams could be explicitly constrained. Since synchronization is done between graphics streams, an application thread is able to continue issuing graphics commands even when its graphics stream is blocked—this allows the application to specify, fully in parallel, a scene that requires even an exact ordering. Several implementations of the parallel API were described, and with the Pomegranate implementation, we demonstrated that ordering does not constrain performance. Even on data sets that required a total ordering of primitives, we were able to demonstrate a speedup efficiency of 99% using 64 graphics streams, allowing the hardware implementation to achieve 1.10 billion triangles per second, largely due to 64 GB per second of input bandwidth from the parallel interface. Given that we are now at the point where a graphics card costing less than a couple of hundred dollars can render triangles faster than the system can specify them, the parallel API is critical for scaling graphics beyond even the cheapest of graphics architectures.

The parallel API provides a new paradigm for writing parallel graphics applications, but, in general, parallel applications are difficult to write. Many graphics algorithms that need an immediate-mode interface are limited by application computation speed (e.g., [Sederberg & Parry 1986, Hoppe 1997]), and parallelizing the graphics commands alongside the computation is straightforward. However, there are two other uses of the parallel API that are of special interest because they present the application programmer with a simple serial interface. Scene graph libraries such as Performer [Rohlf & Helman 1994] are parallel libraries that traverse, cull, and issue scenes on multiple processors on behalf

of a serial application. Pipeline parallelism is currently used to distribute different tasks among different processors, but Performer is limited on most applications by the single processor responsible for the issuing of the graphics commands. The parallel API can be used to write such libraries in a homogeneous, scalable fashion. A second novel use of the parallel API is to write a “compiler” that can automatically parallelize the graphics calls of a serial graphics application. Recent advances in compiler technology allow automatic parallelization of regular serial applications [Amarasinghe et al. 1996], and extending this work to encompass graphics applications would be an interesting research direction.

In Chapter 4, we presented two key contributions that allow for scalable texture mapping subsystems. Two aspects of texture mapping must be scaled: performance and amount of texture data. Multiple texturing units with caches accessing a shared texture memory across a scalable network served as the basis of such a scalable subsystem. In such a subsystem, two main challenges were addressed.

First, in Section 4.1, we presented and analyzed a prefetching texture cache architecture for a single texturing unit. This is critical to a shared texture memory system because of the highly variable latencies. We designed the architecture to take advantage of the distinct memory access patterns of texture mapping. In particular, caching works well because mip mapping guarantees small strides across texture memory. Furthermore, because no dependencies exist in the stream of read-only texture accesses, arbitrary amounts of latency may be tolerated by the prefetching architecture that separates the cache tags from the cache data temporally. We demonstrated the architecture’s ability to hide the memory latency with a 97% utilization of the available hardware resources. The ability to tolerate memory latency is important in serial texture systems and parallel texture systems with dedicated texture memories because of the latencies of modern memories. Furthermore, such a system is critical for shared texture memory architectures that scale the amount of texture memory through texture caching because of the highly variable latencies introduced by a system with non-uniform memory access times.

In Section 4.2, we demonstrated that parallel texture caching is able to load balance texture memory requirements well without incurring excessive amounts of redundant work given the proper choice of various parameters. We demonstrated that parallel texture caching works well across nearly two orders of magnitude of parallelism, from a serial texture unit up to 64 texture units. An interesting discovery was that even though fragment work may be well balanced across texturing units, significant imbalance can exist in the texture bandwidth requirements. However, a shared texture memory in which data is interleaved balances this bandwidth across the multiple texture memories very well.

While this thesis demonstrated how two major aspects of a graphics system may be scaled, attaining ideal amounts of scalability requires the modification of the serial graphics system. For example, although WireGL transparently implements the parallel API on top of serial graphics architectures, a layer of software is responsible for implementing the parallel API. Compared to the hardware implementation of Pomegranate, this system resolves parallel API commands much more slowly, partially due to the fact that it is in software, and partially due to the fact that API commands are broadcast to the renderers. In the realm of scalable texturing, although the results of Chapter 4 demonstrate rasterization algorithms that are able to scale texturing performance based on unmodified serial graphics pipelines, the scalability of the amount of texture memory requires special support for a shared texture store in the graphics pipeline. In the microprocessor space, a minimal set of additions to a non-scalable microprocessor (e.g., cache coherency hardware) is the basis of scalability. Pomegranate demonstrates that one set of additions to a non-scalable graphics architecture allow for a fully scalable graphics architecture. An open question is whether or not these additions are minimal—if not, what is?

Bibliography

- [Akeley 1993] Akeley, K. [1993]. RealityEngine Graphics. *Computer Graphics (SIGGRAPH 93 Proceedings)*, 27, 109-116.
- [Akeley & Jermoluk 1988] Akeley, K. & Jermoluk, T. [1988]. High-Performance Polygon Rendering. *Computer Graphics (SIGGRAPH 88 Proceedings)*, 22, 239-246.
- [Amarasinghe et al. 1996] Amarasinghe, S., Anderson, J., Wilson, C., Liao, S., Murphy, B., French, R., Lam, M., & Hall, M. [1996]. Multi-processors from a Software Perspective. *IEEE Micro*, 16(3), 52-61.
- [Anderson et al. 1997] Anderson, B., MacAulay, R., Stewart, A., & Whitted, T. [1997]. Accommodating Memory Latency In A Low-Cost Rasterizer. *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, 97-102.
- [Beers et al. 1996] Beers, A., Agrawala, M., & Chaddha, N [1996]. Rendering from Compressed Textures. *Computer Graphics (SIGGRAPH 96 Proceedings)*, 30, 373-378.
- [Blinn 1988] Blinn, J. [1988]. Me and My (Fake) Shadow. *IEEE Computer Graphics and Applications*, 8(1), 82-86.
- [Buck et al. 2000] Buck, I., Humphreys, G., & Hanrahan, P. [2000]. Tracking Graphics State for Networked Rendering. *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*.
- [Chen 1995] Chen, S. [1995]. QuickTime VR: An Image-Based Approach to Virtual Environment Navigation. *Computer Graphics (SIGGRAPH 95 Proceedings)*, 29, 29-38.
- [Chen et al. 1998] Chen, M., Stoll, G., Igehy, H., Proudfoot, K., & Hanrahan, P. [1998]. Simple Models of the Impact of Overlap in Bucket Rendering. *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, 105-112.

- [Cox et al. 1998] Cox, M., Bhandari, N., & Shantz, M. [1998]. Multi-Level Texture Caching for 3D Graphics Hardware. *Proceedings of the 25th International Symposium on Computer Architecture*.
- [Crisp 1997] Crisp, R. [1997]. Direct Rambus Technology: The New Main Memory Standard. *IEEE Micro*, Nov / Dec, 18-28.
- [Crockett 1994] Crockett, T. [1994]. Design Considerations for Parallel Graphics Libraries. *Proceedings of the Intel Supercomputer Users Group 1994*.
- [Deering 1995] Deering, M. [1995]. Geometry Compression. *Computer Graphics (SIGGRAPH 95 Proceedings)*, 29, 13-20.
- [Deering & Nelson 1993] Deering, M. & Nelson, S. [1993]. Leo: A System for Cost Effective 3D Shaded Graphics. *Computer Graphics (SIGGRAPH 93 Proceedings)*, 27, 101-108.
- [Dijkstra 1968] Dijkstra, E. [1968]. Cooperating Sequential Processes. *Programming Languages*, 43-112.
- [Duato et al. 1997] Duato, J., Yalmanchili, S., & Ni, L. [1997]. *Interconnection Networks*, IEEE Computer Society Press.
- [Eldridge et al. 2000] Eldridge, M., Igehy, H., & Hanrahan, P. [2000]. Pomegranate: A Fully Scalable Graphics Architecture. *Computer Graphics (SIGGRAPH 2000 Proceedings)*, 34.
- [Evans & Sutherland 1992] Evans & Sutherland Computer Corporation. [1992]. *Freedom 3000 Technical Overview*, Technical Report.
- [Eyles et al. 1997] Eyles, J., Molnar, S., Poulton, J., Greer, T., Lastra, A., England, N., & L., Westover [1997]. PixelFlow: The Realization. *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, 57-68.
- [Foley et al. 1990] Foley, J., van Dam, A., Feiner, S., & Hughes, J. [1990]. *Computer Graphics: Principles and Practice*. Addison-Wesley, Second Edition.
- [Gettys & Karlton 1990] Gettys, J. & Karlton, P. [1990]. The X Window System, Version 11. *Software—Practice and Experience*, 20(S2), 35-67.
- [Gortler et al. 1996] Gortler, S., Grzeszczuk, R., Szeliski, R., & Cohen, M. [1996]. The Lumigraph. *Computer Graphics (SIGGRAPH 96 Proceedings)*, 30, 43-54.
- [Hakura & Gupta 1997] Hakura, Z. & Gupta, A. [1997]. The Design and Analysis of a Cache Architecture for Texture Mapping. *Proceedings of the 24th International Symposium on Computer Architecture*.

- [Hanrahan 1997] Hanrahan, P. [1997]. *The Visual Computer*, Invited State-of-the-Field Talk, Supercomputing 1997.
- [Hennessy & Patterson 1996] Hennessy, J. & Patterson, D. [1996]. *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, Second Edition.
- [Hoppe 1997] Hoppe, H. [1997]. View-Dependent Refinement of Progressive Meshes. *Computer Graphics (SIGGRAPH 97 Proceedings)*, 31, 189-198.
- [Igehy et al. 1999] Igehy, H., Eldridge, M., & Hanrahan, P. [1999]. Parallel Texture Caching. *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, 95-106.
- [Igehy et al. 1998a] Igehy, H., Eldridge, M., & Proudfoot, K. [1998]. Prefetching in a Texture Cache Architecture. *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, 133-142.
- [Igehy et al. 1998b] Igehy, H., Stoll, G., & Hanrahan, P. [1998]. The Design of a Parallel Graphics Interface. *Computer Graphics (SIGGRAPH 98 Proceedings)*, 32, 141-150.
- [Intel 1998] Intel Corporation [1998]. *Accelerated Graphics Port Interface Specification*, revision 2.0.
- [Kilgard 1996] Kilgard, M. [1996]. *OpenGL Programming for the X Window System*, Addison-Wesley.
- [Kilgard 1997] Kilgard, M. [1997]. Realizing OpenGL: Two Implementations of One Architecture. *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, 45-56.
- [Kirk 1998] Kirk, D. [1998]. Unsolved Problems and Opportunities for High-Quality, High-Performance 3D Graphics on a PC Platform. *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, 11-13.
- [Kirkland 1998] Kirkland, D. [1998]. *Personal Communication*, Intergraph Corp.
- [Kubota 1993] Kubota Pacific Computer Inc. [1993]. *Denali Technical Overview*, Technical Report.
- [Lampert 1979] Lampert, L. [1979]. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9), 241-248.

- [Laudon & Lenoski 1997] Laudon, J. & Lenoski, D. [1997]. The SGI Origin: A ceNUMA Highly Scalable Server. *Proceedings of the 24th Annual Symposium on Computer Architecture*.
- [Levoy & Hanrahan 1996] Levoy, M., & Hanrahan, P. [1996]. Light Field Rendering. *Computer Graphics (SIGGRAPH 96 Proceedings)*, 30, 31-42.
- [Lorensen & Cline 1987] Lorensen, W. & Cline, H. [1987]. Marching Cubes: A High-Resolution 3D Surface Reconstruction Algorithm. *Computer Graphics (SIGGRAPH 87 Proceedings)*, 21, 163-169.
- [McMillan & Bishop 1995] McMillan, L., & Bishop, G. [1995]. Plenoptic Modeling: An Image-Based Rendering System. *Computer Graphics (SIGGRAPH 95 Proceedings)*, 29, 39-46.
- [Molnar 1995] Molnar, S. [1995]. The PixelFlow Texture and Image Subsystem. *Proceedings of the 10th Eurographics Workshop on Graphics Hardware*, 3-13.
- [Molnar et al. 1992] Molnar, S., Eyles, J., & Poulton, J. [1992]. PixelFlow: High-Speed Rendering Using Image Composition. *Computer Graphics (SIGGRAPH 92 Proceedings)*, 26, 231-240.
- [Molnar et al. 1994] Molnar, S., Cox, M., Ellsworth, D., & Fuchs, H. [1994]. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, 14(4), 23-32.
- [Montrym et al. 1997] Montrym, J., Baum, D., Dignam, D., & Migdal, C. [1997]. InfiniteReality: A Real-Time Graphics System. *Computer Graphics (SIGGRAPH 97 Proceedings)*, 31, 293-302.
- [Mowry 1999] Mowry, T. [1999]. *Personal Communication*. Carnegie Mellon University.
- [Mueller 1995] Mueller, C. [1995]. The Sort-First Rendering Architecture for High-Performance Graphics. *1995 Symposium on Interactive 3D Graphics*, 75-84.
- [Neider et al. 1993] Neider, J., Davis, T., & Woo, M. [1993]. *OpenGL Programming Guide*. Addison-Wesley.
- [Nishimura & Kunii 1996] Nishimura, S. & Kunii, T. [1996]. VC-1: A Scalable Graphics Computer with Virtual Local Framebuffers. *Computer Graphics (SIGGRAPH 96 Proceedings)*, 30, 365-372.
- [Porter & Duff 1984] Porter, T. & Duff, T. [1984]. Compositing Digital Images. *Computer Graphics (SIGGRAPH 84 Proceedings)*, 18, 253-259.

- [Regan et al. 1999] Regan, M., Miller, G., Rubin, S., & Kogelnik, C. A Real-Time Low-Latency Hardware Light-Field Renderer. *Computer Graphics (SIGGRAPH 99 Proceedings)*, 33, 287-290.
- [Rohlf & Helman 1994] Rohlf, J. & Helman, J. [1994]. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. *Computer Graphics (SIGGRAPH 94 Proceedings)*, 28, 381-395.
- [Samanta et al. 1999] Samanta, R., Zheng, J., Funkhouser, T., & Li, K. [1999]. Load Balancing for Multi-Projector Rendering Systems. *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, 107-116.
- [Scheifler & Gettys 1986] Scheifler, R. & Gettys, J. [1986]. The X Window System. *ACM Transactions on Graphics*, 5(2), 79-109.
- [Sederberg & Parry 1986] Sederberg, T. & Parry, S. [1986]. Free-Form Deformation of Solid Geometric Models. *Computer Graphics (SIGGRAPH 86 Proceedings)*, 20, 151-160.
- [Segal & Akeley 1992] Segal, M. & Akeley, K. [1992]. *The OpenGL Graphics System: A Specification*, <http://www.opengl.org>.
- [Torborg & Kajiya 1996] Torborg, J. & Kajiya, J. [1996]. Talisman: Commodity Real-Time 3D Graphics for the PC. *Computer Graphics (SIGGRAPH 96 Proceedings)*, 30, 57-68.
- [Vartanian et al. 1998] Vartanian, A., Béchenec, J., & Drach-Temam, N. [1998]. Evaluation of High Performance Multicache Parallel Texture Mapping. *Proceedings of the 12th ACM International Conference on Supercomputing*, 289-296.
- [Voorhies et al. 1988] Voorhies, D., Kirk, D., & Lathrop, O. [1988]. Virtual Graphics. *Computer Graphics (SIGGRAPH 88 Proceedings)*, 22, 247-253.
- [Williams 1983] Williams [1983]. Pyramidal Parametrics. *Computer Graphics (SIGGRAPH 83 Proceedings)*, 17, 1-11.