

RAY TRACING ON A STREAM PROCESSOR

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

Timothy John Purcell

March 2004

© Copyright by Timothy John Purcell 2004
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.

Patrick M. Hanrahan
(Principal Adviser)

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.

William R. Mark
University of Texas, Austin

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

Approved for the University Committee on Graduate Studies.

Abstract

Ray tracing is an image synthesis technique which simulates the interaction of light with surfaces. Most high-quality, photorealistic renderings are generated by global illumination techniques built on top of ray tracing. Real-time ray tracing has been a goal of the graphics community for many years. Unfortunately, ray tracing is a very expensive operation. VLSI technology has just reached the point where the computational capability of a single chip is sufficient for real-time ray tracing. Supercomputers and clusters of PCs have only recently been able to demonstrate interactive ray tracing and global illumination applications.

In this dissertation we show how a ray tracer can be written as a *stream program*. We present a stream processor abstraction for the modern programmable graphics processor (GPU) — allowing the GPU to execute stream programs. We describe an implementation of our streaming ray tracer on the GPU and provide an analysis of the bandwidth and computational requirements of our implementation. In addition, we use our ray tracer to evaluate simulated GPUs with enhanced program execution models. We also present an implementation and evaluation of global illumination with photon mapping on the GPU as an extension of our ray tracing system. Finally, we examine hardware trends that favor the streaming model of computation. Our results show that a GPU-based streaming ray tracer has the potential to outperform CPU-based algorithms without requiring fundamentally new hardware, helping to bridge the current gap between realistic and interactive rendering.

Acknowledgements

I would like to thank my wife Jessica for all the love and support she has given me ever since I started graduate school. She has been my inspiration, and provided encouragement when research progress was slow. I'm especially grateful for her calming influence when I feel stressed, and for her enthusiasm for spending our down time together playing Nintendo. What more could you want from a wife?

Having Pat Hanrahan as an adviser has been an amazing experience. He was willing to take a chance on my research from the beginning, and has always pushed me to fill in that one last detail to elevate the level of my thinking and my work. I also admire and respect his integrity both as a researcher and as a person.

Thanks to Philipp Slusallek for organizing my first SIGGRAPH speaking opportunity and having me come visit his lab during the summer of 2001. I met some fantastic friends and collaborators there in Ingo Wald and Jörg Schmittler. I also gained the confidence and drive to continue my research while I was visiting — though there's not much else to do when you don't speak the language.

James Percy and Bob Drebin from ATI are responsible for the largest contiguous block of sleep deprived nights in my life to date. Nothing beats hacking away on a demo at 1am the night before your first SIGGRAPH paper talk. Pradeep Sen and Eric Chan also lost a lot of sleep to help make those demos shine.

I would also like to thank the many people I have associated with at NVIDIA over the past few years. David Kirk, Matt Papakipos, and Nick Triantos have generously provided hardware and driver support for me to abuse. NVIDIA has also generously provided my funding the last two years through their excellent fellowship program. Thank you.

Jim Hurley and Gordon Stoll were excellent mentors and managers during my time at Intel. I'm glad I had the chance to share ideas and learn a little bit about microprocessor research.

I have been fortunate to work with Ian Buck, Bill Mark, Pradeep Sen, Mike Cammarano, Craig Donner, Henrik Wann Jensen, Jörg Schmittler, and Ingo Wald as collaborators in my research efforts. I've also been lucky to be in a lab where everyone is willing and able to participate in shaping your research. I've had many fruitful discussions with just about everyone, in particular Kurt Akeley, Mike Houston, Greg Humphreys, John Owens, Matt Pharr, and Kekoa Proudfoot. Thanks to Ada Glucksman and Heather Gentner for always making sure I had money, and to John Gerth for keeping the lab running smoothly.

I'd like to thank my committee members Steve Kerckhoff and Marc Levoy for passing me on my oral exam and providing insightful feedback. And a special thanks goes to my readers Pat Hanrahan, Bill Mark, and Mark Horowitz for helping me get thoughts for this dissertation written coherently.

And last, but not least, thanks to my parents, family, and other friends that I've associated with throughout my graduate career. The encouragement, support, and board game playing has helped make graduate school a fantastic experience.

Contents

Abstract	iv
Acknowledgements	v
1 Introduction	1
1.1 Contributions	2
1.2 Outline	4
2 Background	5
2.1 Ray Tracing and Global Illumination	5
2.1.1 Parallel Ray Tracing	10
2.1.2 Interactive Ray Tracing	11
2.1.3 Ray Tracing Hardware	12
2.2 Stream Programming	13
2.3 Programmable Graphics Hardware	16
2.3.1 Near Term GPU Changes	19
2.3.2 Scatter	19
3 Streaming Ray Tracer Design	21
3.1 Stream Programming Model	21
3.1.1 Kernels	22
3.1.2 Streams	23
3.2 Streaming Hardware	24
3.3 Ray Tracing Kernels and Streams	24

3.3.1	Eye Ray Generator	27
3.3.2	Traverser	27
3.3.3	Intersector	28
3.3.4	Shader	28
4	Programmable Graphics Processor Abstractions	32
4.1	The Programmable Fragment Processor as a Stream Processor	32
4.1.1	Streams	34
4.1.2	Flow Control	36
4.2	Texture Memory as Memory	39
4.3	Summary	41
5	Ray Tracing on Programmable Graphics Hardware	43
5.1	Mapping Ray Tracing onto the GPU	44
5.1.1	Kernels	44
5.1.2	Memory Layout	45
5.1.3	Data Flow	46
5.2	Architecture Simulations	46
5.2.1	Simulation Methodology	47
5.2.2	SIMD vs. MIMD Architecture	50
5.2.3	Stream Buffer vs. Cache	55
5.2.4	Summary	56
5.3	Implementation Results	57
5.4	Discussion	61
5.4.1	Short-Term GPU Improvements	62
5.4.2	Long-Term GPU Design Changes	64
5.5	Conclusions	65
6	Photon Mapping on Programmable Graphics Hardware	66
6.1	Introduction	66
6.2	Photon Mapping on the GPU	68
6.2.1	Photon Tracing	69

6.2.2	Constructing the Photon Map Data Structure	70
6.2.3	The Radiance Estimate	75
6.2.4	Rendering	77
6.3	Results	77
6.3.1	Rendered Test Scenes	77
6.3.2	Kernel Instruction Use	80
6.3.3	SIMD Overhead	82
6.3.4	Interactive Feedback	82
6.4	Discussion and Future Work	83
6.4.1	Fragment Program Instruction Set	83
6.4.2	Memory Bottlenecks	84
6.4.3	Parallel Computation Model	84
6.4.4	Uniform Grid Scalability	85
6.4.5	Indirect Lighting and Adaptive Sampling	85
6.5	Conclusions	86
7	Ray Tracing and the Memory–Processor Performance Gap	87
7.1	The Memory–Processor Performance Gap	88
7.2	Streaming	89
7.2.1	Parallelism	90
7.2.2	Locality	91
7.3	CPU-based Ray Tracing	92
7.3.1	Single CPU Architectures	92
7.3.2	Multithreaded and Multiprocessor Architectures	93
7.3.3	Locality	95
7.4	Conclusion	95
8	Conclusions	97
8.1	Contributions	97
8.2	Final Thoughts	98
	Bibliography	100

List of Tables

5.1	Ray tracing kernel summary	48
5.2	Simulation pass breakdown and active rays	54
5.3	GPU ray tracing kernel summary	58
5.4	GPU ray tracing timings and pass breakdown	60
6.1	GPU photon mapping render times	80
6.2	Photon mapping kernel instruction use	81

List of Figures

2.1	The ray tracing algorithm	6
2.2	Examples of indirect illumination	7
2.3	Example acceleration structures	9
2.4	The stream programming model	14
2.5	The programmable graphics pipeline	16
2.6	The programmable fragment processor	18
3.1	The streaming ray tracer	26
3.2	Ray-triangle intersection kernel code	29
3.3	Ray tracing data flow diagrams	31
4.1	The programmable fragment processor as a stream processor	34
4.2	Multiple kernel invocations via multipass rendering	36
4.3	Dependent texture lookup as a pointer dereference	40
5.1	Ray tracing texture memory layout	45
5.2	Test scenes and statistics	49
5.3	Compute and bandwidth usage	51
5.4	Compute and bandwidth usage	52
5.5	Bandwidth consumption by data type	53
5.6	Bandwidth ratio with and without a texture cache	55
5.7	GPU ray tracing test scenes	59
5.8	CORNELL BOX test scene images	62
5.9	TEAPOTAHEDRON test scene images	62

5.10	QUAKE3 test scene images	63
6.1	Photon mapping system flow	69
6.2	A bitonic sort of eight elements	71
6.3	Cg code for bitonic merge sort	72
6.4	Texture memory layout for a grid-based photon map	73
6.5	Building a photon map with stencil routing	74
6.6	Computing the radiance estimate with the kNN-grid	76
6.7	Photon mapping test scenes	78
6.8	Caustic image over time	83
7.1	The memory–processor performance gap	88

Chapter 1

Introduction

Computer graphics have become commonplace in our daily lives. Nearly every modern high budget movie has some sort of digital special effects shot. Even small budget movies and local television advertisements tend to utilize some graphics technology. There are even several Saturday morning cartoon shows that are completely animated and rendered using a computer. Yet aside from a small handful of high production cost movies, even non-experts can tell when computer rendered special effects are used.

Why are computer rendered images easy to pick out? Several factors contribute to an effects shot, including modeling, animation, and rendering. If we focus just on rendering the answer is simple: time. When realistic imagery is required — for example when adding computer graphics effects into a live action film — the computational cost of generating the image increases. This cost increase prevents many films from achieving the visual fidelity necessary for seamless effects integration. It also prevents applications like games, that require fast rendering times, from having realistic imagery.

Most of the high-quality, photorealistic renderings made today rely on a rendering technique called ray tracing. Ray tracing simulates the interaction of light with surfaces, a process which is very computationally expensive. When the rendering does not have to happen interactively, such as a special effects shot for a movie, it is simply

a matter of compute time and resources to perform the simulations necessary to generate realistic images. When rendering speed matters, image quality is often traded off for increased speed. This trade off point varies from production to production and plays a large role in determining its achievable level of realism.

We are interested in bridging the gap between realistic and interactive graphics. State of the art ray tracers today can render scenes at several frames per second on a supercomputer or on a cluster of high-end PCs. Unfortunately games, CAD, and other single-user applications, can not rely on every person having a supercomputer or cluster to run them on. Instead, they exploit commodity graphics processors (GPUs) and render images with as high of quality as possible. Graphics processors have improved significantly in the past several years both in terms of speed and in terms of the type of shading they support. Most recently, they have exposed a fairly general programming environment which allows developers the freedom to write very realistic looking shaders.

In this dissertation we explore the reformulation of high quality rendering algorithms for interactive rendering. We show that ray tracing can be expressed as a streaming computation, and that high performance processors like GPUs are well suited for supporting streaming computations. We can abstract the GPU as a general stream processor. The combination of this reformulation and abstraction allows us to implement high quality rendering algorithms at interactive rates on commodity programmable graphics hardware. Furthermore, we will argue that ray tracing is most naturally and efficiently expressed in the stream programming model.

1.1 Contributions

This dissertation makes several contributions to the areas of computer graphics and graphics hardware design:

- We present a streaming formulation for ray tracing. Streaming is a natural way to express ray tracing. Modern high performance computing hardware is well suited for the stream programming model. The stream programming model

helps to organize the ray tracing computation optimally to take advantage of modern hardware trends.

- We have developed a stream processor abstraction for the programmable fragment engine found in modern graphics processors. This abstraction helps us to focus on our algorithm rather than on the details of the underlying graphics hardware. More broadly, this abstraction can support a wide variety of computations not previously thought of as being mappable to a graphics processor.
- We have also developed an abstraction for the GPU memory system. We show that dependent texturing allows texture memory to be used as a read only general purpose memory. We can navigate complex data structures stored in texture memory via dependent texture lookups. As with our stream processor abstraction, our memory abstraction has proved useful to a wide variety of algorithms.
- We have implemented and evaluated a prototype of our streaming ray tracer on current programmable graphics hardware. Our implementation runs at comparable speeds to a fast CPU-based implementation even though current GPUs lack a few key stream processor capabilities. The GPU-based implementation validates our ray tracing decomposition and GPU abstractions.
- We have extended our ray tracer to support photon mapping. Our implementation solves the sorting and searching problems common to many global illumination algorithms in the streaming framework. This implementation is an initial step in demonstrating that the streaming model is valid for advanced global illumination computations as well as simple ray tracing.
- We analyze how other ray tracing systems are optimized to take advantage of fundamental hardware trends. We show that these optimizations take advantage of the same trends that the streaming approach does, only in a much more cumbersome way than is exposed by the stream programming model. We conclude that high performance ray tracing is most naturally expressed in the stream programming model.

In this dissertation, we are primarily concerned with studying static scenes – scenes in which the light sources or eye position can move but scene geometry is fixed. Our analysis does not account for any scene preprocessing times. Efficient ray tracing of dynamic scenes is an active research area in the ray tracing community, and is beyond the scope of this thesis.

1.2 Outline

We begin in chapter 2 with a background discussion of ray tracing, stream programming, and modern programmable graphics hardware. We present our stream programming model and the streaming formulation for our ray tracer in chapter 3. We then present our abstraction of the programmable graphics processor as a stream processor in chapter 4.

We describe our implementation of a streaming ray tracer using programmable graphics hardware in chapter 5. Chapter 6 then describes how we have extended our ray tracer to perform photon mapping. In chapter 7 we evaluate the stream programming model in the context of fundamental hardware trends. We then examine how other ray tracing implementations have adapted to hardware trends and compare these with the stream programming model.

Finally, we suggest areas of future research, reiterate our contributions, and conclude this dissertation in chapter 8.

The core ideas for expressing ray tracing as a streaming computation and for abstracting programmable graphics hardware as a stream processor were summarized in work published at SIGGRAPH in 2002 [Purcell et al., 2002]. The photon mapping work of chapter 6 was published the following year at Graphics Hardware 2003 [Purcell et al., 2003].

Chapter 2

Background

This dissertation spans three different areas of graphics and computer architecture: ray tracing, stream programming, and graphics hardware. In this chapter we will provide the necessary background in each area. We will also look at the related state of the art work in each area.

2.1 Ray Tracing and Global Illumination

In the real world, light is emitted from light sources as photons. Photons interact with objects by scattering until they are absorbed by the environment, or captured by an imaging device such as a camera, sensor, an eye, etc. Ray tracing is an image synthesis technique that simulates this light interaction with the environment. We can render an image by placing a virtual sensor in a computer model and simulating the light emission from the light sources, tracking the light arriving at the sensor. This simulation is very expensive as only a small fraction of the total light particles emitted will actually find their way into the sensor to be recorded. We would have to simulate many particles in hopes of recording enough data to create a reasonable image.

Instead, we take advantage of a physical property known as reciprocity. That is, the path a light particle takes from a light to a sensor is reversible. We can therefore trace particles back from the sensor to the light source without breaking the physical

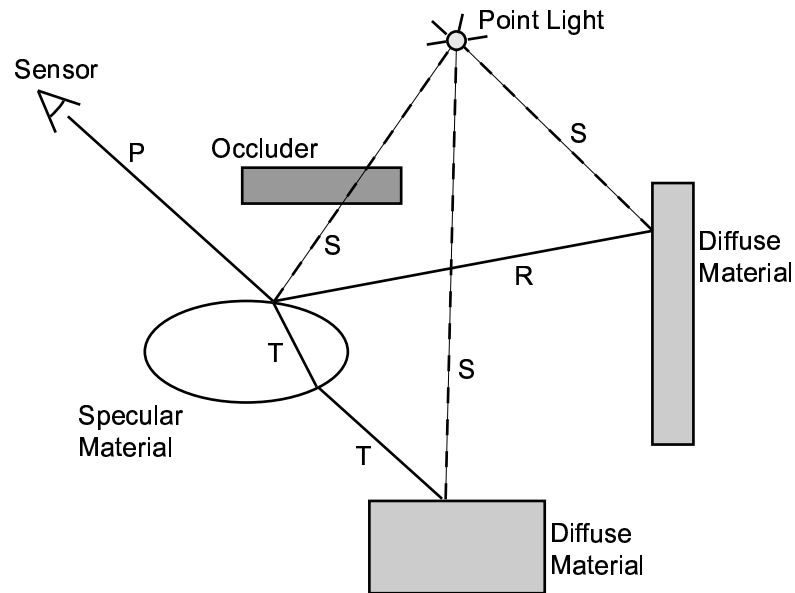


Figure 2.1: The ray tracing algorithm. Primary rays (P) proceed from the sensor into the scene. Transmissive (T) and reflective (R) rays are spawned at a specular surface. Each surface interaction also causes a shadow ray (S) to be sent toward the light source.

simulation. This process saves the computational costs of simulating particles that never reach the sensor. This process is called ray tracing, and was originally published by Whitted [1980]. Classic, or Whitted-style ray tracing captures the important features of a scene to help make it look real: shadows, reflection and refraction, and diffuse surface shading. Figure 2.1 shows some example rays propagated through a scene from the viewpoint of a camera.

Global Illumination

Classic Whitted-style ray tracing is perhaps the most basic global illumination algorithm. We call this a *global* illumination algorithm because the computed light interaction at a surface depends on other objects in a scene. For example, a shadowed surface has no light arriving at it and is rendered black, whereas a perfect mirror

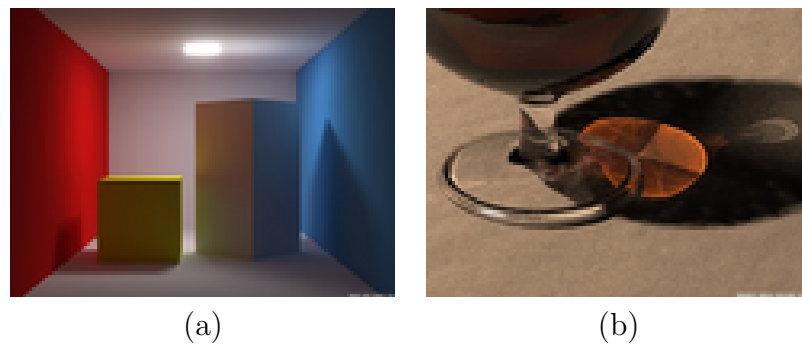


Figure 2.2: Examples of indirect illumination. (a) Diffuse color bleeding. The wall casts a bluish color on the tall block. Both blocks give the floor around them a yellow glow. (b) Caustics. Light focuses through the liquid in the glass and makes a caustic pattern on the floor. Images courtesy Henrik Wann Jensen.

surface gets its color from the objects it reflects. In contrast, a *local* illumination algorithm computes the interactions of light with a surface as though it were the only surface in the scene. We will examine a common local illumination algorithm, called feed-forward rendering, when we describe how modern graphics hardware computes images in section 2.3.

Whitted-style ray tracing simulates the effects of direct illumination, shadows, reflections, and refractions. However, it does not capture the effects of indirect lighting. Indirect lighting occurs when light bounces off of one surface to illuminate another. The two most common effects of indirect illumination are color bleeding and caustics. One common case where color bleeding occurs is when one diffuse surface reflects light onto another surface. The light reflected will be shifted toward the color of the reflecting surface, such that a red wall will illuminate a nearby surface with reddish light. Figure 2.2a shows the effects of color bleeding. A caustic is caused by light that has been reflected, refracted, or focused through another material. The result is a bright spot or band of light on a receiving surface. Figure 2.2b shows an example caustic. The effects of indirect illumination are subtle but important effects when rendering realistic scenes.

Researchers recognized the importance of the subtle global effects early on. Distribution ray tracing [Cook et al., 1984] added some random sampling to renderings

to account for several effects, including some limited color bleeding. Distribution ray tracing was extended and generalized with Kajiya's Monte Carlo path tracing [Kajiya, 1986]. This generalization makes it possible to account for all paths of light in a scene, given enough random samples. Radiosity methods also capture indirect illumination effects [Goral et al., 1984; Nishita and Nakamae, 1985]. We will not investigate radiosity methods in this dissertation since they compute scene illumination by solving large systems of linear equations instead of directly simulating light transport as is done in ray tracing.

One difficulty with using pure Monte Carlo path tracing to compute an image is the large number of samples required to make a smooth looking image. The random sampling can take a very long time to converge for areas of highly varying intensity (like caustics). One solution is to first trace rays out from light sources separately from rays from the sensor and then combine the paths of both tracings when creating the final images [Chen et al., 1991]. This technique has been refined and extended, resulting in the photon mapping technique commonly used for high quality renderings today [Jensen, 1996; 2001]. We will examine the details of this algorithm in more detail when we discuss mapping it to graphics hardware in chapter 6.

Acceleration Structure

At the center of most ray tracing based global illumination algorithms is the idea of following a ray (or particle) into a scene and finding the nearest surface interaction point (hit point or intersection point). It is important to efficiently cull away surfaces which the ray will not intersect. Otherwise, for very large scenes, a ray would test against millions of surfaces before finding the nearest one.

This culling is usually accomplished through a data structure called an acceleration structure. A spatial acceleration structure subdivides the scene into several smaller regions. These regions can be tested efficiently against each ray, and the geometry residing in those regions that the ray does not interact with can be safely ignored. An alternate approach is to use bounding volumes. Bounding volumes surround groups of complex objects in a simple shape. These simple shapes are tested against each

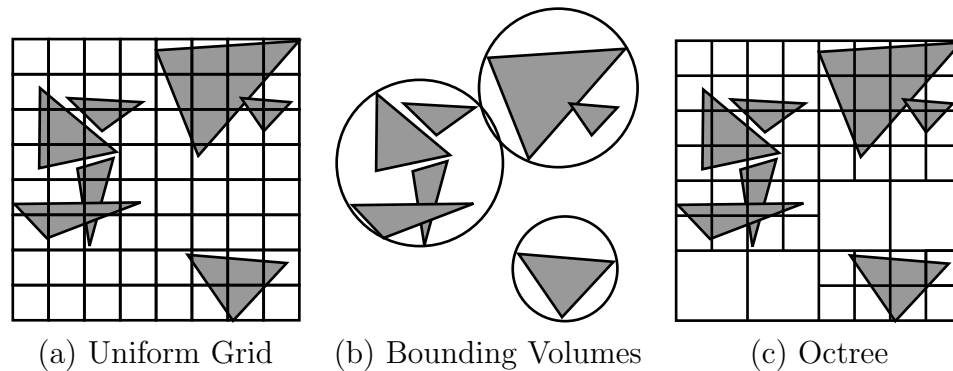


Figure 2.3: Example acceleration structures. (a) The uniform grid divides space into equal regions or voxels. (b) Bounding volumes group nearby objects into simpler enclosing volumes. These are often also arranged hierarchically. (c) The octree recursively divides space containing geometry into smaller regions. Notice the empty spaces are not subdivided further.

ray, and only if the ray intersects the bounding volume does the ray test against the enclosed geometry.

There are several different types of acceleration data structures explored in the literature. Figure 2.3 shows several different types of acceleration structures, including (a) the uniform grid [Fujimoto et al., 1986], (b) bounding volume hierarchy [Rubin and Whitted, 1980], and (c) octree [Glassner, 1984]. Havran et al. [2000] wrote an excellent study of several acceleration structures and their relative effectiveness.

The basic ray tracing algorithm changes only slightly with the addition of an acceleration structure. Before determining if a ray intersects any geometry in the scene, the ray is tested against the acceleration structure. The acceleration structure returns to the ray tracer the nearest region containing a subset of geometry. The ray tracer performs intersection calculations with this geometry. If the ray does not intersect any geometry in the subset, the structure is again queried and another set tested. This continues until an interaction is found, or the ray is determined not to intersect with any of the geometry in the scene.

Acceleration structures are typically built offline, and the construction cost is not included in the cost of ray tracing a scene. However, this cost approximation is only valid for static scenes – scenes where only the viewer or light sources change position.

Scenes where geometry moves around require an acceleration structure to be built each time an object moves. The design of efficient acceleration structures for dynamic scenes is an open area of research [Reinhard et al., 2000]. This dissertation focuses on ray tracing static scenes, with the acceleration structure built as a preprocessing step. Chapter 6 explores the construction of a photon map data structure, which is a step toward building an acceleration structure for ray tracing.

2.1.1 Parallel Ray Tracing

Ray tracing is a very computationally expensive algorithm. Fortunately, ray tracing is quite easy to parallelize. The contribution of each ray to the final image can be computed independently from the other rays. For this reason, there has been a lot of effort put into finding the best parallel decomposition for ray tracing. The vast body of parallel ray tracing work has recently been summarized [Chalmers et al., 2002]. We provide an brief introduction to some of the major design issues for parallel ray tracers. The reader is referred to that book for more details.

The simplest parallel ray tracers replicate the scene database for every processing node. For these systems, load balancing the ray tracing computation is the biggest challenge. However, if the entire scene is too large to fit on a single node or we do not wish to pay the scene replication overhead, then we have the additional challenge of distributing scene geometry and/or rays between the nodes.

When the scene database is replicated on all the nodes, or we are using a shared memory machine, load balancing is fairly straight forward. The ray tracing task is broken into chunks of work, usually by subdividing the screen into tiles. A master process assigns a tile to a client whenever that client needs more work to do. If the tile size is fixed, load imbalance can occur as not all tiles require equal amounts of computation. Instead, we can make the tile size decrease as the computation nears the end of a frame. This load balancing mechanism works quite well, and is discussed in more detail in [Parker et al., 1999a].

Without database replication, parallel ray tracing gets much more difficult. In this case the scene geometry is often split across multiple processors. If a processor

does not have the geometry required to perform the ray tracing computation, it must communicate the task to the appropriate processor(s). This communication can be expensive.

2.1.2 Interactive Ray Tracing

Even though ray tracing is trivially parallelized, very few interactive ray tracing systems exist. Interactivity requires the ray tracing system spend very little time on communication and synchronization. Simply adding processors does not necessarily increase the performance of a system unless it is properly engineered. Two types of systems have recently yielded interactive systems: shared memory supercomputers and clusters of commodity PCs. In both cases, these systems use a collection of standard microprocessors, each of which is designed for maximum performance on single threaded programs.

The first interactive ray tracers were realized on massively parallel shared memory supercomputers. These machines combine the fast inter-processor communication and high memory bandwidth necessary for interactive ray tracing. Muuss [1995] demonstrated interactive ray tracing of simple constructive solid geometry (CSG) scenes using an SGI Power Challenge Array. Parker et al. [1998; 1999a; 1999b] demonstrated a full-featured ray tracer, called *-Ray, with shadows, reflections, textures, etc. The system uses data structures tuned for the SGI Origin 2000 cache lines. Additionally, rays are distributed in varying sized chunks to processors for load balancing.

An alternative to the super computer is a cluster of commodity PCs. These systems are often more cost effective for the same amount of peak compute performance. However, clusters are limited by having lower communication bandwidth and higher communication latency. Wald et al. [2001; 2002] demonstrated a system for interactive ray tracing using a cluster of commodity PCs called RTRT. *-Ray has recently been ported to run on a cluster of PCs [DeMarle et al., 2003], but our discussion of *-Ray will focus on the supercomputer version.

RTRT and *-Ray have much in common. In each, the data structures are carefully tuned to match the cache line sizes of the underlying machine (Intel Pentium III CPUs for early versions of RTRT, Pentium 4 for more recent versions).

RTRT also takes advantage of the SSE execution unit [Intel, 2004] available on Pentium processors. The SSE unit is essentially a separate math unit on the Pentium chip that can execute multiple math operations in parallel. The execution is SIMD (single instruction multiple data), meaning that the simultaneous execution must be for the same instruction, but can be over different data. For example, the SSE unit can perform four multiplies or four adds at the same time, but not two adds and two multiplies together. RTRT gathers multiple rays together and tries to execute most of the ray tracing computation on the SSE units. This strategy provides a nice computational speedup since the SSE math units are faster than the standard math unit on the chip and also work in parallel.

Both *-Ray and RTRT share several common characteristics: highly efficient acceleration structures, processor specific data structure layout for improved caching, and efficient code parallelization. Both systems are so tuned to the underlying hardware that they are effectively non-portable to another platform without serious rewriting. In chapter 7 we will show that the stream programming model leads to code that can be automatically tuned to the underlying processor. That means that with proper language support, our stream formulation for ray tracing could be ported to any system and, in theory, only require recompiling to achieve an efficient implementation.

2.1.3 Ray Tracing Hardware

One of the earliest examples of special purpose ray tracing hardware is the LINKS-1 system [Nishimura et al., 1983]. This system was not what we commonly think of as special purpose hardware as it consisted of several general purpose Intel 8086 nodes networked together. However, the system was designed as a ray tracing machine and is an interesting precursor to the cluster ray tracing systems mentioned previously.

Most of the special purpose ray tracing hardware developed in the past few years can be found on volume rendering boards. The VIRIM [Günther et al., 1995],

VIZARD [Knittel and Straßer, 1997], and VolumePro [Pfister et al., 1999] boards all implement ray casting and some limited shading calculations. These renderers do not take into account shadows, reflections, or refractions.

The most recent piece of ray tracing hardware was designed to render high quality ray traced images, but not in real-time. The AR250 [Hall, 1999] and AR350 [Hall, 2001] from Advanced Rendering Technologies performed the full ray tracing calculation including shadows, reflections, and refractions in a custom hardware implementation. These systems could render scenes much faster than standard software based systems at the time, but never pushed into interactive rendering.

Finally, the most recent research into custom ray tracing hardware is a system called SaarCor [Schmittler et al., 2002]. SaarCor is based on the Saarland RTRT interactive ray tracer [Wald et al., 2001], and effectively implements that system in silicon. Simulations indicate the system could provide ray tracing at interactive frame rates with fewer transistors than commodity GPUs require. As of early 2004, early prototypes of the chip are being implemented on FPGAs at Saarland University.

2.2 Stream Programming

Stream programming and streaming processors have recently become popular topics in computer architecture. The main motivation for stream processor development is that semiconductor technology is at a point where computation is cheap and bandwidth is expensive. Stream processors are designed to exploit this trend by exploiting both the parallelism and locality available in programs. The result is machines with higher performance per dollar [Khailany et al., 2000]. To this end, stream processors provide hundreds of arithmetic processors to exploit parallelism, and a deep hierarchy of registers to exploit locality.

The stream programming model constrains the way software is written such that locality and parallelism are explicit within a program. These constraints allow compilers to automatically optimize the code to take advantage of the underlying hardware. Of course, stream processors require sufficiently parallel computations to achieve this

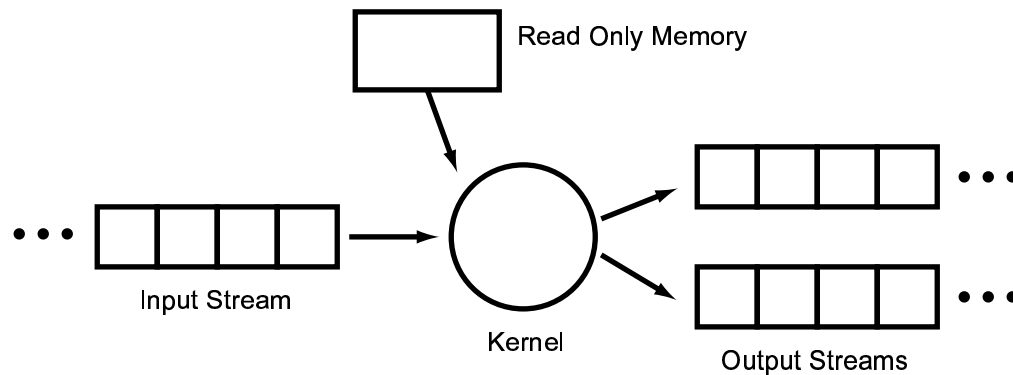


Figure 2.4: The stream programming model. Kernels are functions which can be operated on in parallel. Kernels process records from input streams, and place resulting records on output streams. Kernels may also be able to read from a global read-only memory.

higher performance. We will investigate the benefits of the stream programming model further in chapter 7.

The stream programming model is based on *kernels* and *streams*. A kernel is a function that is going to be executed on over a large set of input records. A kernel loads an input record, performs computations on the values loaded, and then writes an output record. The stream programming model is illustrated in figure 2.4.

The more computation a kernel performs, the higher its *arithmetic intensity* or locality, and the better a stream processor will perform on it. Streams are the sets of input and output records operated on by kernels. Streams are what connect multiple kernels together. Media processing such as MPEG decoding and signal processing kernels often found in DSPs are large target applications for stream processors.

There have been a variety of general purpose stream processors designed, each with a corresponding language implementing the stream programming model associated with the machine. There are also various polymorphic computing architectures like Smart Memories [Mai et al., 2000] and TRIPS [Sankaralingam et al., 2003] which can be configured as stream processors or as traditional cache-based processors. We will focus our discussion on the architectures based around stream processing.

An early example of a modern stream processor is the MIT RAW machine [Waingold et al., 1997; Taylor et al., 2002]. The RAW machine is made up of several simple processors on a single chip. The overriding idea of the RAW architecture is to expose the low level details of the architecture to the compiler. StreaMIT [Gordon et al., 2002] is the special purpose stream programming language associated with RAW.

The Imagine processor [Khailany et al., 2000] is a streaming processor made up of several arithmetic units connected to fast local registers and an on-chip memory called a stream register file. Imagine provides a bandwidth hierarchy with relatively small off-chip memory bandwidth, larger stream register file bandwidth, and very large local register file bandwidth. Programs written in the stream programming model can be scheduled for the processor such that they mainly use internal bandwidth instead of external bandwidth. Imagine is programmed using StreamC and KernelC — programming languages for streams and kernels that are a subset of C. These languages force programs to be written in a stream friendly manner, and are more general purpose than the StreaMIT language for the RAW processor. However, the underlying Imagine architecture is still exposed to the programmer when writing a stream program.

Finally, there is the Merrimac streaming supercomputer [Dally et al., 2002]. Merrimac is a large scale multi-chip streaming computer. Merrimac is programmed in a language called Brook [Buck, 2004]. Brook is like StreamC and KernelC as it is an augmented subset of C designed for stream programming. However, one big difference between Brook and StreamC/KernelC is that Brook does not expose the details of the underlying architecture to the programmer. This means that programs written in Brook can be recompiled (instead of rewritten) for other stream machines.

Perhaps the most relevant target that Brook supports is GPUs. A BrookGPU program [Buck et al., 2004] can compile to run on a standard Intel processor, or one of several different graphics processors (such as the NVIDIA GeForce FX or ATI Radeon parts described in the next section). The ray tracing approach presented in this dissertation was recently reimplemented in BrookGPU in only a couple of days time. We will examine the advantages of stream programming for ray tracing further in chapter 7.

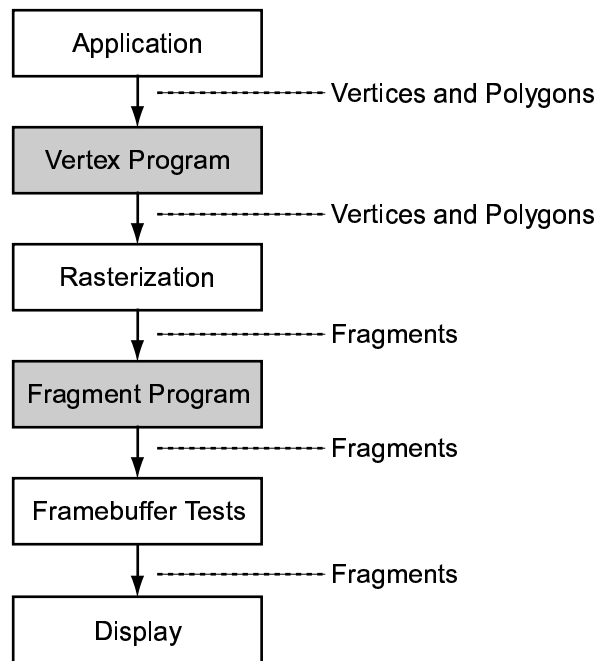


Figure 2.5: The programmable graphics pipeline. The gray boxes show the stages where the programmable vertex and fragment engines are located. The types of data passed between each stage of the pipeline are shown by the dotted lines.

2.3 Programmable Graphics Hardware

Ray tracing is not the only way to generate computer images. In fact, the graphics processor found in nearly every desktop PC uses a very different algorithm. Recall that a ray tracer computes what parts of a scene are visible at each pixel by following a ray for each pixel into the scene. In a sense, the image is computed pixel by pixel. Graphics processors use nearly the opposite algorithm. Images are computed by drawing each object in the scene, and keeping track of the closest object for every pixel in the image. Once all the objects are drawn, the correct image is displayed.

Figure 2.5 shows what a modern programmable graphics pipeline looks like. The scene is fed into the hardware as a sequence of triangles, with colors and normals at the vertices. The vertex program stage is generally used to transform the vertices from model coordinates to screen coordinates using matrix multiplication. The rasterizer

takes the transformed triangles and turns them into fragments — essentially pixels in memory rather than on the display device. In addition, the rasterizer interpolates the values of each vertex between other vertices in the triangle so each fragment has a color and normal value. Fragments then pass through a fragment program stage. This stage is generally used to modify the color of each fragment with texture mapping or other mathematical operations. Fragments are finally compared against the stored value in a depth buffer. Those fragments that are nearer than the stored value are saved and displayed, while others are discarded.

The programmable vertex and fragment engines found on today’s graphics chips, such as the NVIDIA GeForce FX 5900 Ultra [NVIDIA, 2003a] and the ATI Radeon 9800 Pro [ATI, 2003] execute user-defined programs and allow fine control over shading and texturing calculations. Each stage is programmable through OpenGL ARB extensions [ARB, 2003c; 2003b], vendor specific extensions [NVIDIA, 2002; 2003b], or the DirectX 9 API [Microsoft, 2003]. We will be primarily interested in the programmable fragment pipeline for this dissertation; it is designed to operate at the system fill rate (approximately 4 billion fragments per second).

The programming model for the programmable fragment engine is shown in figure 2.6. Most GPUs have a set of several parallel execution units that implement the fragment engine. However, the exact number of parallel units is not exposed to the programmer. Fragment programs are written in a 4-way SIMD assembly language [ARB, 2003b; NVIDIA, 2003b], which includes common operations like add, multiply, dot product, and texture fetch. Fragment programs are not allowed to perform data dependent branching — the hardware may use SIMD parallelism to implement multiple fragment processors. All of the operations in a fragment program are performed in floating point.

GPUs allow memory fetches through texture lookups. They also permit what is known as a dependent texture lookup. A dependent texture fetch is simply a texture fetch at an address that has been computed, unlike a standard texture fetch where the address is determined by interpolated texture coordinates. This feature is useful, for example, to compute a bumped reflection map for a surface. One texture fetch will

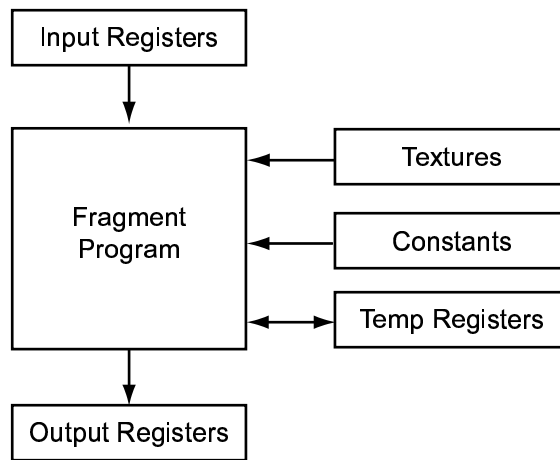


Figure 2.6: The programmable fragment processor. A fragment program can read data from input registers, constants, texture memory, and temporary registers. Temporary registers store intermediate calculations, and the output registers store the final color values for the fragment.

grab the perturbed normal at a point from a normal texture map, and that normal can then be used to index into a reflection map [Kirk, 2001].

An additional feature provided by modern graphics hardware is a specialized reduction operator called the `NV_OCCLUSION_QUERY` extension [NVIDIA, 2003c]. An occlusion query simply returns a count of the number of fragments that were drawn to the framebuffer between the start and end of the query. This capability can be used to accelerate rendering by drawing simple objects in place of more complex ones (e.g. bounding boxes instead of true geometry). The occlusion query will return a zero value if the object is not visible, meaning the complex geometry does not need to be drawn.

Unfortunately, unlike CPUs, GPUs currently do not virtualize their resources. Instead, limits are set on how many operations and how many memory references can be used in a given program. Fragment programs are currently limited to 64 or 1024 instructions, depending on the specific chip and API being used. Only 16 or 32 registers are available to use in a program, and as few as four levels of dependent texture fetches are supported. Additionally, many GPUs allow only a single output

value to be computed per fragment. The specific resource limits depend on the vendor, card, and extension used to program the GPU.

2.3.1 Near Term GPU Changes

Our ray tracing and photon mapping implementations discussed in subsequent chapters use the DirectX 9 class hardware discussed previously. As the hardware continues to evolve, we expect that several features missing from the fragment programming model will be added:

- Data dependent branching in fragment programs
- Increased program length
- Increased dependent texture fetching limits
- Multiple outputs

Our analysis of the ray tracing system presented in chapter 5 evaluates the impact of some of these improved capabilities.

2.3.2 Scatter

Nearly as important as features that are available on the GPU are features that are not available. There are several such missing features, but perhaps the most important one is the scatter operation for fragment programs. Simply stated, scatter is the ability to compute an output address to write fragment data to. GPUs provide the ability to perform random memory reads through dependent texture fetches. A scatter would be similar to computing a dependent texture write. This lack of a scatter capability restricts the types of algorithms that can be run efficiently on the GPU. We will explore several algorithms that could be much more efficient with a scatter operation (such as sorting). Other algorithms do not seem possible to implement without scatter (such as building acceleration structures). We will soon see graphics architectures that allow a multipass scatter operation by feeding the framebuffer data

from one rendering pass back through the pipeline as primitive data. Unfortunately, a single pass scatter operation is not on the horizon for future graphics architectures.

Chapter 3

Streaming Ray Tracer Design

In this chapter, we describe how we decompose ray tracing for the stream programming model. We first define the stream programming model we use for this thesis. We'll then briefly enumerate the underlying hardware support we expect. Finally, we describe the way we decompose ray tracing for the stream programming model.

3.1 Stream Programming Model

In section 2.2 we saw several examples of stream processors and languages that implement the stream programming model. When we designed our streaming ray tracer, we did not target a specific programming language or architecture. Instead, the design was based on our assumption that we would have a processor that was a cross between Imagine and a GPU. We designed the ray tracer assuming much the same functionality that can be expressed in KernelC and StreamC. Brook [Buck et al., 2004] is the closest language to what we envisioned. Since our programming model is slightly different, we will enumerate what we assumed it could support. The programming model we present is idealized. We present an evaluation of two different architectures that implement this programming model in chapter 5.

3.1.1 Kernels

Recall that a kernel is essentially a specialized function call. Kernels are expected to be invoked on many different records all requiring the same processing. In the stream programming model, kernels are designed to be executed in parallel. In particular, kernels must be stateless — the results of a kernel call can not depend on a previous invocation of that kernel by another record.

The kernels in our streaming ray tracer are allowed to access read-only global memory. This is a different approach than taken by KernelC. KernelC required all memory access to be through streams. Global memory could be accessed by building an index stream — essentially a stream of memory addresses. That stream would fetch the data from memory, and the data could be used in an input stream to another kernel. This style of global memory access puts a large burden on the application programmer to split kernels into several pieces whenever global memory is required and the addresses are unknown at compile time. Instead, we allow read-only access to global memory directly in a kernel. This capability ends up being strictly a programmer convenience, as kernels could be split at memory references automatically by the underlying compiler.

Our kernels are also able to perform data dependent branches. This capability is important to ray tracing for two reasons. First, ray tracing finds the nearest visible point by marching rays through an acceleration structure, intersecting geometry as it goes. If we write ray tracing as a single kernel, the kernel must be able to change between traversing the structure and intersecting geometry. Second, each part of the acceleration structure can hold a different amount of geometry. We can not determine before kernel invocation how many intersection tests need to be done. We will examine the impact of branching at the end of chapter 5.

Finally, kernels can perform both integer and floating point arithmetic operations. Additionally, kernels support data types of both float and integers up to four elements wide and provide short vector math operations for those types. Much of the ray tracing calculation is vector math. GPUs operate on these four wide types natively, and they simplify many calculations that we have to do. Kernels support standard

arithmetic operations on these data types plus corresponding graphics operations like dot product.

3.1.2 Streams

A stream is a set of records that a kernel operates over. Generally speaking, a kernel takes a stream of records as input and produces a stream (or streams) of records as output. Kernels are allowed to only have a single input stream. This stream may contain data generated by several different kernels, but only the single input stream is visible to the kernel. By only allowing a single stream, we do not have to figure out what to do when two input streams have different length. This restriction takes away some of the flexibility of our stream programming model, but this case does not come up for a ray tracing computation and is thus not an issue.

We do allow kernels to generate multiple output streams, and output to these streams may be conditional. A kernel may write to zero or more of its possible destination streams. Conditional streams are especially useful for a ray tracer. During shading, a ray can generate several shadow rays, along with reflection and refraction rays. However, not all surfaces generate all types of rays. Conditional output allows a complex shading model to fit naturally into the streaming framework. Additionally, rays can exit the bounds of a scene during rendering. These rays contribute nothing to the final image. Kernels can discard these rays by not writing them to the output stream, avoiding bandwidth and computation overhead for useless rays.

Finally, our programming model has a modest ordering requirement for streams. If a kernel has no conditional output, then streams are required to remain in order. That is, the first element of the input stream generates the first element of the output stream. With a conditional output, this requirement is relaxed. Split streams do not have to maintain relative order.

3.2 Streaming Hardware

In this section, we want to briefly describe the desired hardware for implementing our stream programming model. In chapter 5 we will present results from running our ray tracer on simulated hardware like this, as well as simulated hardware much more like a GPU. We'd like to use our stream model to show potential benefits of new GPU features. Our model assumes:

- 32-bit \times 4-wide floating point math units and registers. The ray tracer will be working primarily with 3- and 4-vector quantities (i.e. color, vertex position, ray direction and origin). As such the native data type should be a four element vector. In addition, common graphics operations like dot product and cross product should be very fast.
- MIMD execution units. Our stream programming model allows data dependent branching within a kernel. We want MIMD (Multiple Instruction Multiple Data) execution units to implement the branching in hardware efficiently.
- On-chip stream buffer. The data structures passed between kernels can be quite large, and depending on the granularity of the computation can happen quite often. An on-chip stream buffer minimizes off-chip bandwidth.
- Fast local cache for read-only global data. Geometry, materials, and the acceleration data structure are among the read only data in our system.

We assume some fairly basic hardware capable of implementing our stream abstraction. The primary goal of this dissertation is to show that the stream programming model is a good model for ray tracing, not to describe the proper hardware to implement that model.

3.3 Ray Tracing Kernels and Streams

The challenge is then to map ray tracing onto our streaming model of computation. This is done by breaking the ray tracer into kernels. These kernels are chained

together by streams of data. In this section, we show how to reformulate ray tracing as a streaming computation.

We need to make some decisions up front about the type of scenes we will be able to ray trace before we can formulate the ray tracer for streaming. We need to define the type of geometric primitives our ray tracer can handle, whether to support static or dynamic scenes, and the type of acceleration structure we will use.

We assume that all scene geometry is represented as triangles. Ray tracers often can render scenes made of several different geometric primitives. The literature is full of ray-object intersection algorithms (see *An Introduction to Ray Tracing* [Glassner, 1989] for a good survey). However, we believe focusing only on triangles is valid for several reasons. First, graphics hardware supports only triangle rendering. Other surfaces like splines, spheres, and other shapes can be specified, but they are transformed into triangles before rendering anyway. Modeling programs and scanning software produce models made of triangle meshes. Finally, ray tracing is much simpler, and in many cases more efficient [Wald et al., 2001], when only a single primitive is allowed. For streaming computations, this restriction means all rays can be handled by the same small set of kernels which simplifies the data flow of the system.

Our ray tracer is designed to render static scenes. We assume triangles are stored in an acceleration data structure before rendering begins. We will not consider the cost of building this data structure. Since this operation may be expensive, and may not map well to the stream programming model, this assumption implies that the algorithm described may not be efficient for dynamic scenes.

We also decided to use a uniform grid to accelerate ray tracing. There are many possible acceleration data structures to choose from: bounding volume hierarchies, bsp trees, k-d trees, octrees, uniform grids, adaptive grids, hierarchical grids, etc. We chose uniform grids for two reasons. First, many experiments have been performed using different acceleration data structures on different scenes [Havran et al., 2000]. From these studies no single acceleration data structure appears to be most efficient; all appear to be within a factor of two of each other. Second, uniform grids are particularly simple for hardware implementations. Accesses to grid data structures require constant time; hierarchical data structures, in contrast, require variable time

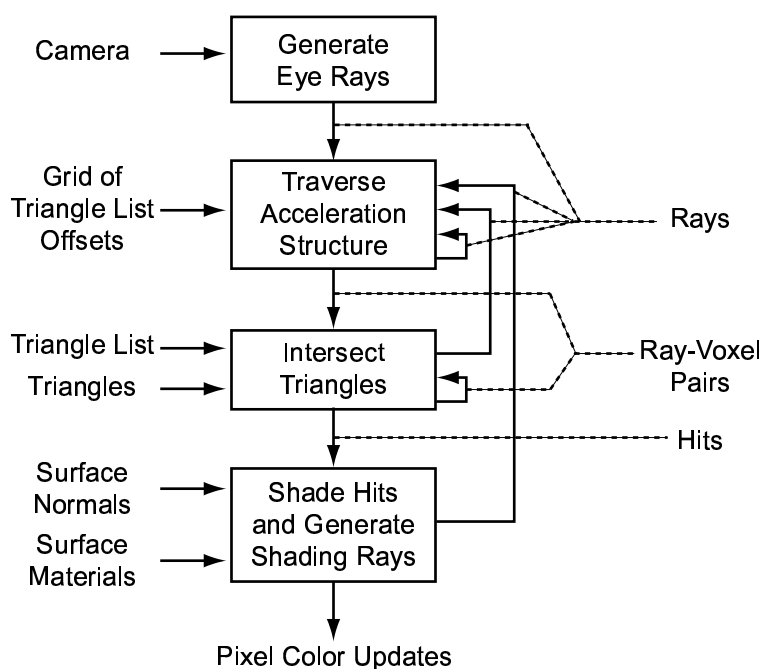


Figure 3.1: The streaming ray tracer. Ray tracing can be broken down into several core kernels, represented by boxes in this diagram: eye ray generation, acceleration structure traversal, triangle intersector, and a shader that computes colors and secondary rays. The inputs to each kernel are shown to the left of the box. The types of stream records passed between kernels are shown by the dotted lines.

per access and involve pointer chasing. Code for grid traversal is also very simple and can be highly optimized in hardware.

We have split the streaming ray tracer into four kernels: eye ray generation, grid traversal, ray-triangle intersection, and shading, as shown in figure 3.1. This split is not mandated by our stream programming model. Instead, it is a result of our desire to eventually run the ray tracer on a GPU which lack branching within a fragment program. We explain how we work around the lack of branching in chapter 4, and analyze the effects of the workaround in chapter 5.

The eye ray generator kernel produces a stream of viewing rays. Each viewing ray is a single ray corresponding to a pixel in the image. The traversal kernel reads the stream of rays produced by the eye ray generator. The traversal kernel steps a

ray through the grid until the ray encounters a voxel containing triangles. The ray and voxel address are output and passed to the intersection kernel. The intersection kernel is responsible for testing a ray with all the triangles contained in a voxel. The intersector has two types of output. If ray-triangle intersection (hit) occurs in that voxel, the ray and the triangle that is hit is output for shading. If no hit occurs, the ray is passed back to the traversal kernel and the search for voxels containing triangles continues. The shading kernel computes a color. If a ray terminates at this hit, then the color is written to the accumulated image. Additionally, the shading kernel may generate shadow or secondary rays; in this case, these new rays are passed back to the traversal stage. The following sections detail the implementation of each ray tracing kernel.

3.3.1 Eye Ray Generator

The eye ray generator is the simplest kernel of the ray tracer. Given camera parameters, including viewpoint and a view direction, it computes an eye ray for each screen pixel. The kernel is invoked for each pixel on the screen, generating an eye ray for each. The eye ray generator also tests the ray against the scene bounding box. Rays that intersect the scene bounding box are processed further, while those that miss are terminated.

3.3.2 Traverser

The traversal stage searches for voxels containing triangles. The first part of the traversal stage sets up the traversal calculation. The second part steps along the ray enumerating those voxels pierced by the ray. Traversal is equivalent to 3D line drawing and has a per-ray setup cost and a per-voxel rasterization cost.

We use a 3D-DDA algorithm [Fujimoto et al., 1986] for this traversal. After each step, the kernel queries the grid data structure. If the grid contains a null pointer, then that voxel is empty and the ray is output into the stream of rays to be traversed again. If the pointer is not null, the voxel contains triangles. In this case, a ray-voxel

pair is output and the ray is marked so that it will be tested for intersection with the triangles in that voxel.

The traversal setup is performed once per ray. The main traversal kernel marches a ray one step through the voxel grid. This means each step a ray takes through a voxel is done as a separate iteration of the traversal kernel. At the end of the kernel, the ray is either output to the rays needing traversing stream or the rays to be intersected stream.

3.3.3 Intersector

The triangle intersection stage takes a stream of ray-voxel pairs and outputs ray-triangle hits. It does this by performing ray-triangle intersection tests with all the triangles within a voxel. If a hit occurs, a ray-triangle pair is passed to the shading stage. We compute ray-triangle intersections using the method described by Möller and Trumbore [1997]. A ray-triangle intersection kernel can be found in figure 3.2.

Because triangles can overlap multiple grid cells, it is possible for an intersection point to lie outside the current voxel. The intersection kernel checks for this case and treats it as a miss. Note that rejecting intersections in this way may cause a ray to be tested against the same triangle multiple times (in different voxels). It is possible to use a mailbox algorithm to prevent these extra intersection calculations [Amanatides and Woo, 1987], but mailboxing is difficult to implement efficiently when multiple rays are traced in parallel.

As with the traversal stage, we have implemented the ray-triangle intersection kernel to perform a single ray-triangle intersection. Each ray is cycled through the intersection kernel for each triangle in the voxel. After all the triangles have been intersected, the decision is made whether to send the ray to the traversal or shading kernel.

3.3.4 Shader

The shading kernel evaluates the color of the surface intersected by the ray at the hit point. Shading data consists of vertex normals and vertex colors for each triangle


```

// ro, rd are ray origin and direction
// list_pos contains the triangle list entry
// h is current best hit
float4 IntersectTriangle( float3 ro, float3 rd, int list_pos, float4 h ){
    float tri_id = memfetch( list_pos, trilist );
    float3 v0 = memfetch( tri_id, v0 );
    float3 v1 = memfetch( tri_id, v1 );
    float3 v2 = memfetch( tri_id, v2 );
    float3 edge1 = v1 - v0;
    float3 edge2 = v2 - v0;
    float3 pvec = Cross( rd, edge2 );
    float det = Dot( edge1, pvec );
    float inv_det = 1.0/det;
    float3 tvec = ro - v0;
    float u = Dot( tvec, pvec ) * inv_det;
    float3 qvec = Cross( tvec, edge1 );
    float v = Dot( rd, qvec ) * inv_det;
    float t = Dot( edge2, qvec ) * inv_det;
    bool validhit = select( u >= 0.0, true, false );
    validhit = select( v >= 0.0, validhit, false );
    validhit = select( u+v <= 1.0, validhit, false );
    validhit = select( t < h[0], validhit, false );
    validhit = select( t >= 0.0, validhit, false );
    return select( validhit, float4(t, u, v, tri_id), h );
}

```

Figure 3.2: Code for the ray-triangle intersection kernel.

and is stored in memory much like triangle data. The hit information that is passed to the shader includes the triangle number. We access the shading information by a simple lookup for the particular triangle specified.

By choosing to generate different shading rays, we can implement several flavors of ray tracing using our streaming algorithm. Figure 3.3 shows a simplified flow diagram for ray casting, Whitted-style ray tracing, path tracing, and shadow casting, along with an example image produced by our system for each flavor.

The shading kernel optionally generates shadow, reflection, refraction, or randomly generated rays. These secondary rays are placed in the stream of rays processed by the traverser. Each ray is also assigned a weight, so that when it is finally

terminated, its contribution to the final image may be simply added into the image [Kajiya, 1986]. This technique of assigning a weight to a ray eliminates recursion and simplifies the control flow.

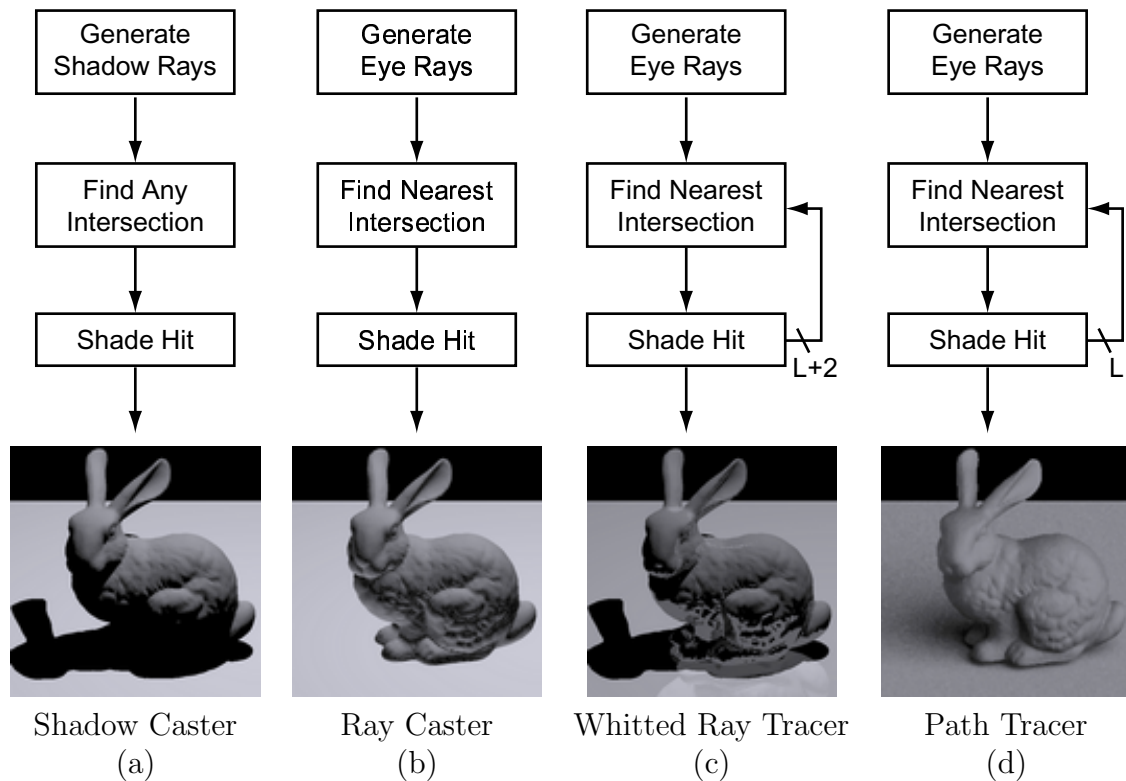


Figure 3.3: Ray tracing data flow diagrams. The algorithms depicted are (a) shadow casting, (b) ray casting, (c) classic Whitted-style ray tracing, and (d) path tracing. Shadow casting takes initial visibility information generated outside our system and traces shadow rays from those visible surface locations. For ray tracing, each ray-surface intersection generates $L + 2$ rays, where L is the number of lights in a scene, corresponding to the number of shadow rays to be tested, and the other two are reflection and refraction rays. Path tracing randomly chooses one ray bounce to follow and the feedback path is only one ray wide. Our path tracer only generates a single eye ray per pixel. The Stanford bunny image was generated by each algorithm.

Chapter 4

Programmable Graphics Processor Abstractions

In the previous chapter, we described the stream programming model and a streaming formulation for ray tracing. In this chapter we show how we can abstract the programmable fragment engine in current GPUs as a stream processor. The GPU will serve as our implementation substrate for the streaming ray tracer (chapter 5) and our extension to global illumination via photon mapping (chapter 6).

We will present two abstractions for the GPU: one for multipass rendering through the fragment engine and another for the memory subsystem. These abstractions are key for our ray tracing and photon mapping implementations, and have been used by other researchers using the GPU for general purpose computation. At the end of this chapter, we examine the impact of GPU limitations on the efficiency of our algorithm.

4.1 The Programmable Fragment Processor as a Stream Processor

We would like to be able to use the computational power of the GPU to perform the ray tracing computation. Unfortunately, as we saw in chapter 2, ray tracing is a very

different type of computation than what GPUs were built to do. Even with the high amount of programmability available on the GPU, it is not obvious how to map all the pieces of a ray tracer onto the GPU pipeline. In this section, we show how to think about the fragment processor as a limited general purpose stream processor.

We discussed the basic flow for feed-forward rendering on a GPU in chapter 2. This task is the one the GPU is designed for. The basic idea is that the GPU renders objects by turning them into fragments. After all objects are drawn, the fragment closest to the viewer is displayed on the screen. Our goal is to use the fragment engine as a stream processor. As such, we will generally ignore the rest of the graphics pipeline and focus on feeding the appropriate data to the fragment engine.

The first step then is to feed streams to the fragment engine to process. The fragment engine processes fragments that are generated by the rasterizer. These fragments come from geometry that has been transformed by the vertex program portion of the pipeline. Our goal is to execute a computation at every screen pixel (the ray tracing computation). A screen aligned square is the simplest geometry that will generate a fragment for every screen pixel. If we disable the pipeline operations that happen after fragment program execution (like depth test, stencil test, alpha blend, etc), the result displayed on the screen is the result of the computation performed by the fragment program.

Conceptually, this framework is all we need to get a ray tracer to work. We simply write a fragment program that expresses the entire ray tracing computation for a pixel, draw a square, let the fragment program execute over all the fragments, and come away with a ray traced image. This is exactly how we would write a ray tracer for a fully general MIMD stream processor. Unfortunately, current GPUs have several limitations that make this approach impossible including limits on program length, limits on dependent texture lookups, and the lack of data dependent branching.

We can overcome most of the limitations of the GPU with multipass rendering techniques. We split the ray tracing kernel into several smaller kernels, with one kernel per major ray tracing operation as we did in chapter 3. After we finish one rendering pass, we store output stream data into texture memory. We can then run another rendering pass (by drawing another square) with a different fragment program that

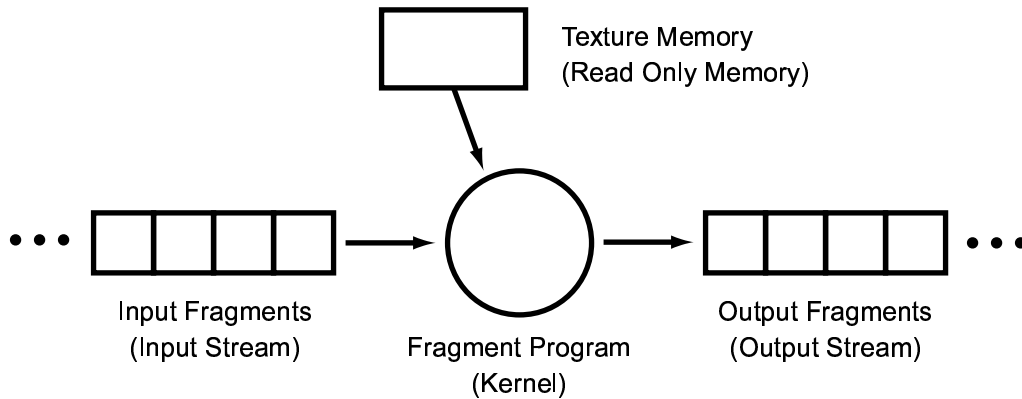


Figure 4.1: The programmable fragment processor as a stream processor. The figure shows the GPU-centric term and the corresponding stream processing term in parentheses.

reads the stored data from the previous pass back from texture memory and continues the computation. We do this until the computation is finished.

Our basic stream processor abstraction is fairly straight forward. Figure 4.1 shows the inputs and outputs for a programmable fragment processor. Input, output, and functional unit is labeled with its graphics name (e.g. fragment program) and its stream programming name (e.g. kernel). We treat the fragments produced by the rasterizer as an input stream, the fragment program as a kernel, and the fragments written to the framebuffer as an output stream.

The following sections expound on our abstraction. We will not address the straight forward mapping between kernels and fragment programs further. Instead we will focus on the mapping of fragments and textures to streams, and on flow control.

4.1.1 Streams

As we already mentioned, the set of fragments produced by the rasterizer constitute an input stream to a kernel. The rasterizer generates a bunch of data associated with each fragment called interpolants. These data include values such as screen position of the fragment, color, and a set texture coordinates. They are called interpolants

because the values are specified at geometry vertices and the rasterizer interpolates the values between vertices for each fragment.

Interpolants are a compact way to specify an input stream of data to a kernel. However, the number of interpolants is limited and stream data can not always be computed via interpolation. For example, when an input stream is an arbitrary collection of data it is impossible to generate through interpolation.

When simple interpolation fails, we instead store stream data in texture memory. The rasterizer still produces an input stream of fragments that initiate computation, however the interpolants now describe where to fetch data from texture memory. If data is stored in texture memory aligned with the square we use to initiate computation, we can use the interpolation hardware to give us the addresses to fetch from. Note that this texture fetch is not a dependent texture fetch as the texture coordinates are provided by the rasterizer. The only change in the kernel is that it must initiate the memory fetch.

This method of storing stream data in texture memory is also used during multipass rendering. In this case, the output stream of one kernel is copied into texture memory. When the rasterizer generates fragments from the next square, the fragments have the appropriate texture coordinates to fetch the results from the previous kernel. This process is illustrated in figure 4.2. If the data being read does not fit into a single texture, we may need to read multiple times. Similarly, we may need to write multiple times by repeating the rendering pass for each output.

The copying of stream data into texture memory happens after all the fragments have been processed from a rendering pass. The hardware issues a barrier operation, and the framebuffer contents are copied into texture memory as read-only. If framebuffer memory and texture memory are physically the same on the hardware, this operation can be performed with a pointer renaming instead of a true copy. The operation can also be as expensive as a copy to the host CPU and re-download of texture data. In either case, we end up using texture memory to store stream data.

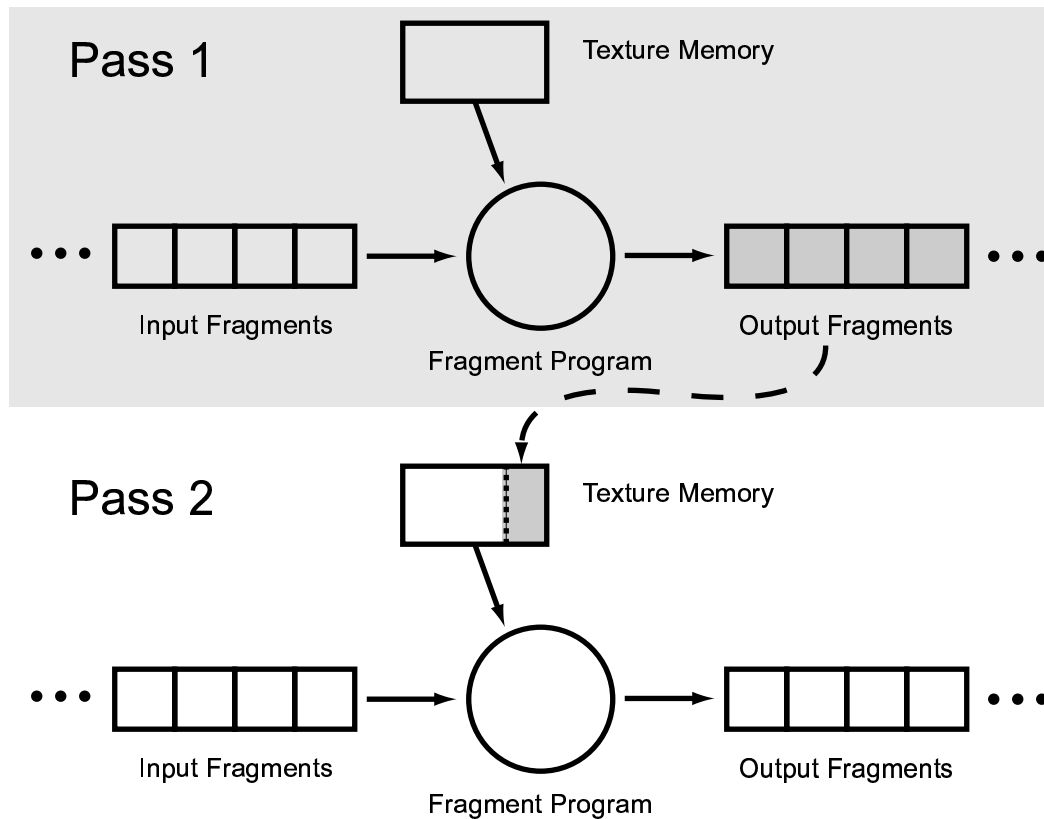


Figure 4.2: Multiple kernel invocations via multipass rendering. This figure shows how multipass rendering techniques fit into the stream programming model. After a rendering pass, the set of output fragments are conceptually copied into read-only memory. They can now act as an input to the following fragment program.

4.1.2 Flow Control

The most difficult part of mapping the stream programming model to current graphics hardware is flow control. Looping and flow control decisions are made on the host. Static flow control decisions simply require binding the fragment programs in the proper order. Dynamic flow control requires a bit more work. Dynamic flow control can happen when when we need to loop over data or when we want to implement conditional output streams.

Current graphics hardware does not allow dynamic flow control within a fragment program. Despite this limitation, programs with loops and conditionals can be

mapped to graphics hardware using the multipass rendering technique presented by Peercy et al. [Peercy et al., 2000]. To implement a conditional using their technique, the conditional predicate is first evaluated using a sequence of rendering passes, and then a stencil bit is set to true or false depending on the result. The body of the conditional is then evaluated using additional rendering passes, but values are only written to the framebuffer if the corresponding fragment’s stencil bit is true.

Although their algorithm was developed for a fixed-function graphics pipeline, it is the inspiration for our algorithm for dynamic flow control on a programmable graphics pipeline. To explain our algorithm, we will examine how we can implement data dependent looping. Assume we have a fragment program that executes the body of our loop called `bodyfp`. We add a single additional output value to `bodyfp` that represents the “state” of the computation (i.e. whether the computation is finished or not). Immediately after executing `bodyfp`, we execute a second fragment program called `donefp`. `donefp` reads the state value written by `bodyfp` and issues a KILL instruction if the state indicates the computation is done for that fragment. The KILL prevents that fragment from being drawn to the framebuffer. `donefp` is surrounded by an occlusion query [NVIDIA, 2003c], which counts the number of fragments that get drawn to the framebuffer. The host runs `bodyfp` and `donefp` within a loop that terminates when the value returned by the occlusion query is zero. When the occlusion query returns zero, all the fragments executing `bodyfp` have their state output set to done, and the loop body is done.

This simple algorithm solves the problem of data dependent looping, but is not very efficient. When `bodyfp` executes, any fragments in the done state have to move their inputs to their outputs. These moves end up consuming computation and bandwidth resources that could be better used moving the computation forward. We can take advantage of early-z occlusion culling [Kirk, 2001] to make our looping much more efficient. Instead of using `donefp` to mask off framebuffer writes, we use it to modify the depth buffer value for each fragment based on the stored state value for that fragment. Fragments that are still executing set the depth buffer to one value ($z1$), while those that are done set it to another value ($z2$). The loop over `bodyfp` and `donefp` is performed on the host exactly as before with two small changes: we

wrap the occlusion query around `bodyfp` instead of `donefp` and we set the depth test to be `GLEQUAL`. When `bodyfp` executes, all the fragments are given a depth of $z1$. Fragments in the done state (with depth $z2$) are not updated. Early-z occlusion culling discards the fragments with depth $z2$ before they execute `bodyfp`, so they do not consume any fragment program resources. When all fragments have been discarded by early-z occlusion culling, the occlusion query count will be zero and the loop will terminate. This optimization is very fragile. Certain operations within a fragment program can disable early-z occlusion culling, and the granularity of the early-z occlusion culling cannot be controlled. This optimization also requires us to copy our input data to the output target of the program before rendering (essentially double buffering the data). With this copy operation, fragments discarded by early-z occlusion culling will maintain their original values and the rest of the fragments will update their values with the execution of `bodyfp`.

The mechanism we have described for performing data dependent looping can be used to simulate conditional output streams as well. We can use the “state” output to store a value indicating which conditional output stream a fragment belongs to. We assign a unique depth value to each possible output state. We can then execute a fragment program over the fragments of a particular output stream by using the `GLEQUAL` depth test as we did for data dependent looping. The challenge then, is to choose the best order to evaluate each conditional output stream. The order we evaluate the different output streams can impact the overall performance of the system.

Our method for choosing which kernel to run is motivated in part by Delany’s implementation of a ray tracer for the Connection Machine [Delany, 1988]. The traversal and intersection kernels of a ray tracer both involve loops and conditional outputs. There are various strategies for nesting the loops and choosing which kernel to execute. The simplest algorithm would be to step through voxels until any ray encounters a voxel containing triangles, and then intersect that ray with those triangles. However, this strategy would be very inefficient when run in parallel, since during intersection only one or a few rays will be active. Most rays will not yet have encountered a voxel with triangles. On a SIMD machine like the Connection Machine, this strategy

results in very low processor utilization. For graphics hardware, this strategy yields an excessive number of passes. The following is a more efficient algorithm:

```
generate eye ray()
while (any(active(ray))) {
    if (oracle(ray))
        traverse(ray)
    else
        intersect(ray)
}
shade(ray)
```

After eye ray generation, the ray tracer enters a while loop which tests whether any rays are *active*. Active rays require either further traversals or intersections; inactive rays have either hit triangles or traversed the entire grid. Before each pass, an oracle is called. The oracle chooses whether to run a traverse or an intersect pass. Various oracles are possible. The poorly performing algorithm described previously runs an intersect pass if *any* rays require intersection tests. A better oracle, first proposed by Delany, is to choose the pass which will perform the most work. This can be done by calculating the percentage of rays requiring intersection vs. traversal with occlusion queries. In our experiments, we found that performing intersections once 20% of the rays require intersection tests produced the minimal number of passes, and is within a factor of two to three of optimal for a SIMD algorithm executing a single kernel per rendering pass.

4.2 Texture Memory as Memory

Traditionally, texture memory has been used to store the textures that are applied to geometry in a scene. Conceptually, the rasterizer produced fragments which each had their own texture coordinates which served as indexes into a two dimensional region of memory containing color information. A major limitation of this technique is that the address to fetch from memory could not be computed, but was rather a property of the geometry being rasterized.

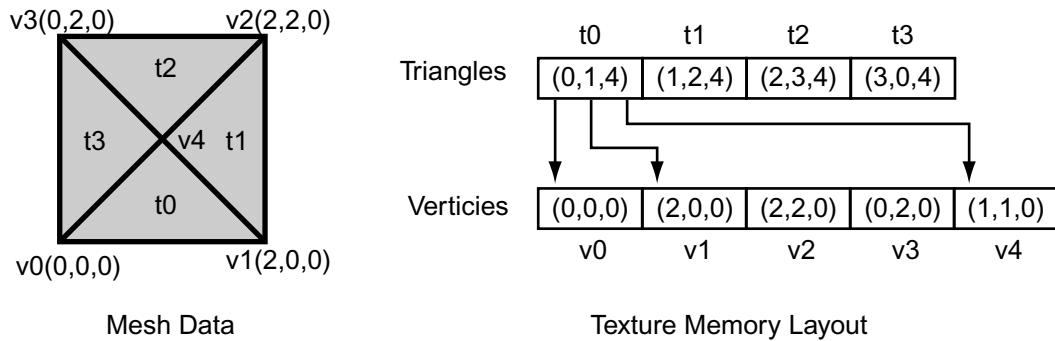


Figure 4.3: Dependent texture lookup as a pointer dereference. A simple four triangle mesh can be stored in two textures: one storing triangle vertex numbers, the other storing actual vertex data. A set of dependent texture fetches allows a fragment program to retrieve the vertex data for a given triangle.

Graphics processors have recently added the ability to perform what is known as a dependent texture lookup. Dependent texture fetching allows the address being fetched from texture memory to be computed by the fragment program. It also allows the results of a memory lookup to be used to compute another memory address. This feature was added to the graphics processor to enable interesting effects like reflective bump mapping to run as a single rendering pass [Kirk, 2001].

The side benefit of allowing dependent texture lookups is that we can use the texture memory subsystem as a general read-only memory. We are interested in performing general computations on the graphics hardware. Many algorithms involve complex data structures and lookup of elements within these structures. As an illustrative example, consider a simple triangle mesh data structure as shown in figure 4.3. To simplify the discussion, we will consider only 1D textures. With 1D texture mapping, the i -th element is stored at texture location i . The texture coordinate is the memory address.

Each triangle is stored as an RGB texture, with the R, G, and B channels holding the location in the vertex texture of vertex 0, 1, and 2 respectively. The triangle vertex texture is another RGB texture storing the 3D coordinates of the vertex, with the R, G, and B channels storing x, y, and z respectively. If we want to know the locations of the three vertices of a triangle, we simply perform three separate dependent texture

fetches. The first fetch uses the R channel of the triangle as the texture location for the first vertex. The second and third fetches use the G and B channels.

Unfortunately, the abstraction is not as clean as it appears with our simple 1D texture examples. Graphics processors currently have limited support for 1D texturing. Many data structures will overflow the maximum size of a 1D texture. Instead, we are forced to use 2D textures. Our general abstraction still works, it just requires address conversion to and from 1D, much like a virtual to physical memory translation. Simply put, we do all the address calculations in 1D and only convert to 2D when we need to do actual texture lookups. This conversion adds several instructions to our lookups, but functionally works fine. We do need to watch out for precision issues though. Current GPUs do not have integer data types, so we must be careful to always floor address calculations. Additionally, the number of bits dedicated to floating point mantissa varies from architecture to architecture. The number of mantissa bits limits the size of our 1D virtual address space.

In summary, our simple abstraction of texture memory enables us to load a complex data structure into memory and use fragment programs to navigate through it. More importantly, it allows us to think about texture memory on the GPU as simple read-only memory. This step is a key one to thinking about the GPU as a more general purpose processor. Rather than worry about texture management and texture coordinates, etc. we can think about memory and addresses.

4.3 Summary

We have shown how the programmable fragment engine for modern GPUs can be thought of as a stream processor. This abstraction, coupled with the presented abstraction of texture memory, makes it easy to see that the GPU can perform many general purpose computations not anticipated by their designers. Several other researchers have used the abstractions described here in other projects [Bolz et al., 2003; Goodnight et al., 2003]. The limited capabilities of current GPUs can make some aspects of streaming challenging to implement efficiently (such as conditional output

streams). The abstraction allows us to think about writing complex streaming algorithms instead of worrying about GPU-specific implementation details. As GPUs evolve toward more general computation models, the limitations will disappear and streaming algorithms will improve in efficiency. Labonte et al. [2004] present a detailed evaluation of the strengths and weaknesses of current GPUs when used as general purpose stream processors.

Finally, there is work being done to develop a language that makes general purpose stream programming on the GPU much easier. Of particular note is the BrookGPU programming language [Buck et al., 2004]. BrookGPU hides all details of the underlying graphics hardware from the programmer, and instead targets a general stream programming model, much like that described in chapter 3. Non-graphics programmers can write BrookGPU programs and take advantage of the powerful compute resources available in their GPUs.

Chapter 5

Ray Tracing on Programmable Graphics Hardware

In this chapter, we present the synthesis of our stream formulation of ray tracing with our stream processor abstraction of the GPU. The result is a ray tracing system that runs completely on the GPU. The system has performance comparable to CPU-based ray tracing systems. It also validates the stream programming model for the GPU and the stream decomposition of ray tracing. This implementation also opens up the possibility of real-time rendering that combines ray tracing with standard feed-forward pipeline rendering.

We begin by describing the changes we made to allow the ray tracer described in chapter 3 to run on the GPU. Next, we present simulation results of our ray tracer running on two different architectures that resemble modern GPUs. The simulations helped us evaluate the potential of a GPU-based ray tracer before any hardware was available. We then present results from our implementation of a ray tracer on the ATI Radeon 9700 Pro [ATI, 2002]. Finally, we discuss some of the short-term and long-term improvements to the GPU that could make ray tracing and general computation more efficient.

5.1 Mapping Ray Tracing onto the GPU

As we saw in chapter 4, the programmable fragment processor found in modern GPUs is effectively a stream processor. However, it is not quite as general as the stream processor described in chapter 3, so we need to make a few modifications to our stream formulation of ray tracing to implement ray tracing on the GPU. We need to change some of our kernels, memory layout, and flow control mechanisms.

5.1.1 Kernels

In general, the functionality provided by fragment programs is nearly equivalent to what the ray tracer was designed for. The two features lacking that we need to compensate for are the lack of integer operations and lack of conditional output streams.

Integer operations are especially important for memory addressing in our ray tracer. All of our data structures (discussed below in section 5.1.2) use integer addresses. Our ray tracer must be able to access specific triangles and voxels. Since the arithmetic units are floating point, we have to simulate integer operations with floating point operations when we need integer data types. Fortunately, integer operations are relatively easy to simulate. We can compute most integer operations by taking the `floor` of the result of a floating point operation. Integer modulus operations, however, require a few more operations including `frac` which returns the fractional part of a floating point number:

$$A \bmod B = \text{floor}(\text{frac}(A / B) * B);$$

The larger problem is the lack of conditional output streams. We saw in chapter 4 that we could simulate conditional streams by adding extra state to the output of the fragment program. Our ray tracer uses a state value that indicates whether a ray is traversing, intersecting, or shading. In general, adding the extra state output did not add significant cost to our kernels, but instead added more complexity to the flow control of the system.

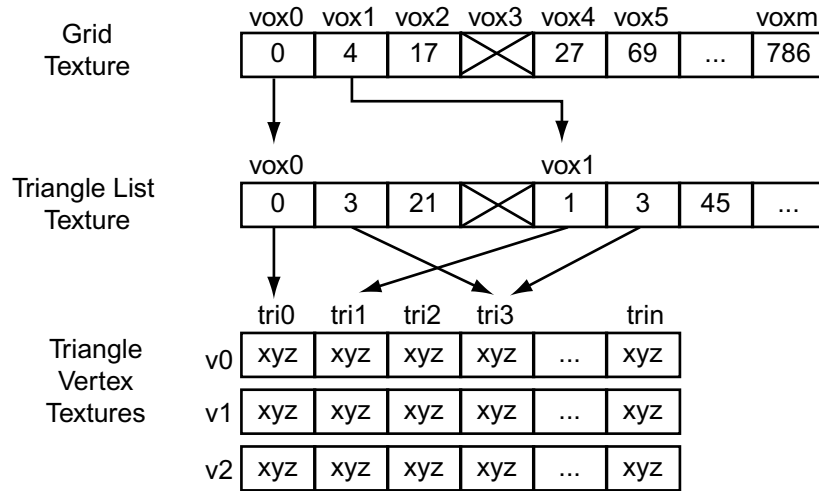


Figure 5.1: Texture memory layout for the streaming ray tracer. Each grid cell contains a pointer to the start of the list of triangles for that grid cell, or a null pointer if the cell is empty. The triangle lists are stored in another texture. Each entry in the triangle list is a pointer to a set of vertex data for the indicated triangle. Triangle vertices are stored in a set of three separate textures.

5.1.2 Memory Layout

We will use the abstraction for texture memory we developed in chapter 4 for navigating our regular grid acceleration structure. The precise memory layout is shown in figure 5.1. The uniform grid contains pointers to a delimited and compact list of triangles for each grid cell. This list is simply a list of pointers to actual triangle data, stored in yet another set of textures. For our implementation, we store each triangle separately and do not share vertex data. Shading data such as vertex normals and vertex colors are stored in a similar fashion to the vertex positions.

Texture memory is also used as the stream buffer that stores the data passed between different kernels in the system. This double use of texture memory is one major limitation of using the GPU for streaming computations. All the stream data will pass through the texture cache, basically evicting any other data in the cache and limiting the usefulness of the cache. Our simulations in section 5.2 quantify the impact of this effect.

5.1.3 Data Flow

Mapping the ray tracer to the GPU requires major changes to the ray tracer's data flow as compared to its ideal streaming implementation. In particular, we no longer have the stream buffer to transfer rays between kernels. Instead, we rely on texture memory and multipass rendering techniques to simulate stream buffer functionality. This comes at a cost, but allows us to implement our streaming ray tracer on the current generation of GPU. The cost is that we always have to read and write each stream element since there is no conditional in or out on the GPU. Conditional stream outputs would eliminate this need.

We simplify mapping our ray tracer by limiting the maximum depth to which rays can bounce to a value set before rendering begins, usually three or four bounces. This limit ensures we know the maximum size of the shade tree and can allocate texture memory appropriately. In practice this is a reasonable simplification. The contributions of rays to a final pixel color are small after just a few bounces.

Ray tracing performs a significant amount of looping over voxels and triangles while locating the nearest hit point. We can use the optimization discussed in section 4.1.2 to determine whether to enter into a traversal or intersection pass. The basic idea is to switch between intersection and traversal passes until all the rays are ready to shade. We then run a shading pass. Then, if we are doing reflections or shadows, we go back to the traversal and intersection passes until they finish. This sequence repeats for each level in the shade tree.

5.2 Architecture Simulations

We ran several high-level functional simulations of our streaming ray tracer before any DirectX 9 class GPUs were available for testing an implementation. The simulations gave us a rough idea of the performance we could expect from a GPU-based ray tracer. More importantly however, we were able to quantify the effects of the architectural differences between the idealized streaming ray tracer described in chapter 3 and the GPU-based streaming ray tracer described in this chapter.

We simulated two different architectures. We refer to one architecture as the MIMD architecture. The MIMD architecture allows data dependent looping within a kernel. This architecture executes the entire ray tracing computation within a single fragment program. This architecture can execute an arbitrary number of instructions in a single kernel invocation.

The other architecture we refer to as the SIMD architecture. For this architecture, kernels are not allowed data dependent looping. Instead, looping is controlled outside the kernel. The SIMD architecture is intended to closely resemble current GPUs. Our simulations also assume this architecture is capable of an operation similar to the early-z occlusion culling optimization discussed earlier. Before every kernel invocation, it performs a check on a small 8-bit buffer containing ray state information. If the state does not match the kernel that is being executed, that stream element is not processed, preventing computation and bandwidth waste. Our simulations account for the cost of both reading and writing this ray state information.

We did not simulate an on-chip stream buffer with either architecture. We knew ahead of time that the upcoming graphics hardware would require us to use the texture memory for stream data. Studies of on-chip stream buffers would be an interesting extension to the work presented in this dissertation.

5.2.1 Simulation Methodology

The SIMD and MIMD architecture simulators were written in C++. Each simulation run generates an image and tabulates several statistics. Example statistics include the average number of traversal steps taken per ray, or the average number of ray-triangle intersection tests performed per ray. The SIMD architecture simulator also tracks the number and type of rendering passes performed, as well as state-buffer activity. These statistics allow us to compute the cost for rendering a scene by using the cost model described later in this section.

To evaluate the computation and bandwidth requirements of our streaming ray tracer, we implemented each kernel as an assembly language fragment program. The assembly language implementation provides estimates for the number of instructions

Kernel	SIMD						MIMD			
	Instr. Count	Mem. Words			State		Instr. Count	Mem. Words		
		R	W	M	RS	WS		R	W	M
Generate Eye Ray	28	0	5	0	0	1	26	0	4	0
Traverse										
Setup	38	11	12	0	1	0	22	7	0	0
Step	20	14	9	1	1	1	12	0	0	1
Intersect	41	14	5	10	1	1	36	0	0	10
Shade										
Color	36	10	3	21	1	0	25	0	3	21
Shadow	16	11	8	0	1	1	10	0	0	0
Reflected	26	11	9	9	1	1	12	0	0	0
Path	17	14	9	9	1	1	11	3	0	0

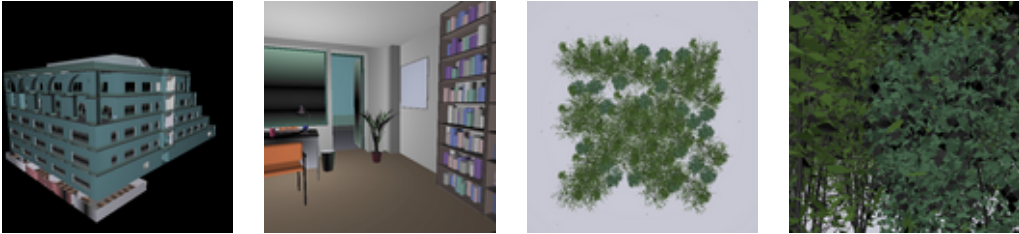
Table 5.1: Ray tracing kernel summary. We show the number of instructions required to implement each of our kernels, along with the number of 32-bit words of memory that must be read and written between rendering passes (R, W) and the number of memory words read from random-access textures (M). Two sets of statistics are shown, one for the SIMD architecture and another for the MIMD architecture. For the MIMD architecture, we also show the number of 8-bit state-buffer reads (RS) and writes (WS) for each kernel. State-buffer read overhead is charged for all rays, whether the kernel is executed or not.

required for each kernel invocation. We also calculate the bandwidth required by each kernel; we break down the bandwidth as stream input bandwidth, stream output bandwidth, and memory (random-access read) bandwidth.

Table 5.1 summarizes the computation and bandwidth required for each kernel in the ray tracer, for both the SIMD and the MIMD architectures. For the traversal and intersection kernels that involve looping, the counts for the setup and the loop body are shown separately. The MIMD architecture allows us to combine individual kernels; as a result the kernels are slightly smaller since some initialization and termination instructions are removed.

Using table 5.1, we can compute the total compute and bandwidth costs for the scene using a cost model:

$$C = R * (C_r + vC_v + tC_t + sC_s) + R * P * C_{state}$$



	Soda Hall Outside	Soda Hall Inside	Forest Top Down	Forest Inside
v	14.41	26.11	81.29	130.7
t	2.52	40.46	34.07	47.90
s	0.44	1.00	0.96	0.97
P	2443	1198	1999	2835

Figure 5.2: Statistics for our test scenes. Recall that v is the average number of voxels pierced by a ray; t is the average number of triangles intersected by a ray; s is the average number of shading calculations per ray; and P is the total number of SIMD passes required to render the scene. Soda hall has 1.5M triangles and the forest has 1.0M triangles. We also used the Stanford bunny shown in figure 3.3. The Stanford bunny has 70K triangles, with $v = 93.93$, $t = 13.88$, $s = 0.82$, and $P = 1085$. Scene statistics were computed from 1024×1024 pixel renderings.

Here R is the total number of rays traced. C_r is the cost to generate a ray; C_v is the cost to walk a ray through a voxel; C_t is the cost of performing a ray-triangle intersection; and C_s is the cost of shading. P is the total number of kernel invocations, and C_{state} is the cost of reading the state-buffer. The total cost associated with each stage is determined by the number of times that kernel is invoked. This number depends on scene statistics: v is the average number of voxels pierced by a ray; t is the average number of triangles intersected by a ray; and s is the average number of shading calculations per ray. The MIMD architecture has no state-buffer checks, so C_{state} is zero for that architecture. The SIMD architecture must pay the state read cost for all rays over all kernel invocations.

We used several large models as test scenes for our simulations, shown in figure 5.2, along with the Stanford bunny scene from chapter 3.

- Soda Hall is a relatively complex model that has been used to evaluate other real-time ray tracing systems [Wald et al., 2001]. It has walls made of large

polygons and furnishings made from very small polygons. This scene has high depth complexity.

- The forest scene includes trees with millions of tiny triangles. This scene has moderate depth complexity, and it is difficult to perform occlusion culling. We analyze the cache behavior of shadow and reflection rays using this scene.
- The Stanford bunny was chosen to demonstrate the extension of our ray tracer to support shadows, reflections, and path tracing as shown in chapter 3.

Figure 5.2 also includes the statistics that are used in our cost model. Each scene was rendered at 1024×1024 pixels with simple shading. No shadow, reflection, or refraction rays were traced.

Choosing an optimal grid resolution for scenes is difficult. A finer grid yields fewer ray-triangle intersection tests, but leads to more traversal steps. A coarser grid reduces the number of traversal steps, but increases the number of ray-triangle intersection tests. We attempt to keep voxels near cubical shape, and specify grid resolution by the minimal grid dimension acceptable along any dimension of the scene bounding box. For the bunny, our minimal grid dimension is 64, yielding a final resolution of $98 \times 64 \times 163$. For the larger Soda Hall and forest models, this minimal dimension is 128, yielding grid resolutions of $250 \times 198 \times 128$ and $581 \times 128 \times 581$ respectively. These resolutions allow our algorithms to spend equal amounts of time in the traversal and intersection kernels.

5.2.2 SIMD vs. MIMD Architecture

We evaluate the differences between our two proposed architectures in two areas: resource consumption and the percentage of active rays during each rendering pass.

Computation and Bandwidth Consumption

When measuring resource consumption, we want to know how much bandwidth and computation is required to render a frame. These numbers tell us whether our algorithm is *compute limited* or *bandwidth limited*. A compute limited algorithm needs to

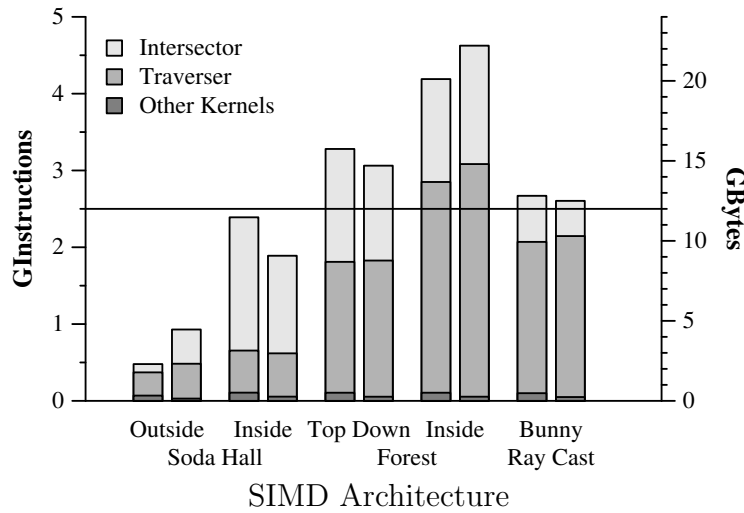


Figure 5.3: Compute and bandwidth usage for our scenes using the SIMD architecture. Each column shows the contribution from each kernel. Left bar is compute, right is bandwidth. The horizontal line represents the peak per-second bandwidth and compute performance of a Radeon 9700 Pro. All scenes were rendered at 1024×1024 pixels.

perform more arithmetic operations per second than the architecture can deliver. A bandwidth limited algorithm requires more data transfer per second than is available. With current hardware trends, it is easier to provide more arithmetic capacity to an architecture than it is to increase the bandwidth.

Using our cost model and measured scene statistics, we can calculate the estimated the cost of rendering each scene in terms of bandwidth and computation. Figures 5.3 and 5.4 show the number of instructions and the bandwidth required to render a single frame of each scene. The consumption is broken down by kernel type. The horizontal line shows what a Radeon 9700 Pro is able to deliver per second as a reference point (approximately 2.5G 4-way SIMD instructions/s in its fragment processor and roughly 12 GB/s of memory bandwidth).

These graphs show that traversal and intersection kernels dominate the cost of rendering a scene, as is expected with our simple shading model. In addition, we can see that the compute and bandwidth requirements of ray tracing on the SIMD architecture match the design parameters of the Radeon 9700 Pro. That is, the

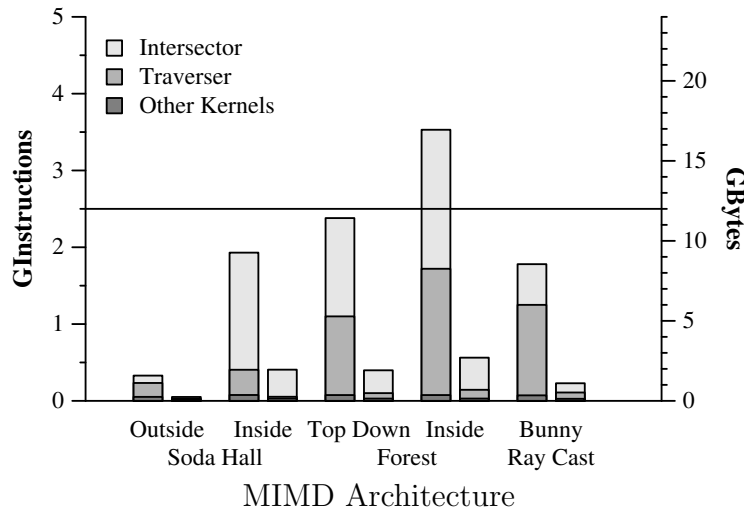


Figure 5.4: Compute and bandwidth usage for our scenes using the MIMD architecture. Each column shows the contribution from each kernel. Left bar is compute, right is bandwidth. The horizontal line represents the peak per-second bandwidth and compute performance of a Radeon 9700 Pro. All scenes were rendered at 1024×1024 pixels.

compute to bandwidth ratio is approximately the same as the design parameters of the Radeon 9700 Pro. Ray tracing on the MIMD architecture, however, is severely compute limited. It requires a small fraction of the bandwidth that ray tracing on the SIMD architecture requires.

Figure 5.5 shows the bandwidth measurements broken down by data type instead of by kernel. The graph shows that, as expected, all of the bandwidth required by the MIMD architecture is for reading voxel and triangle data structures from memory. The SIMD architecture, conversely, uses most of its bandwidth for writing and reading intermediate values to and from texture memory between passes. Similarly, saving and restoring these intermediates requires extra instructions. In addition, significant bandwidth is devoted to reading the state-buffer. This extra computation and bandwidth consumption is the fundamental limitation of the SIMD architecture.

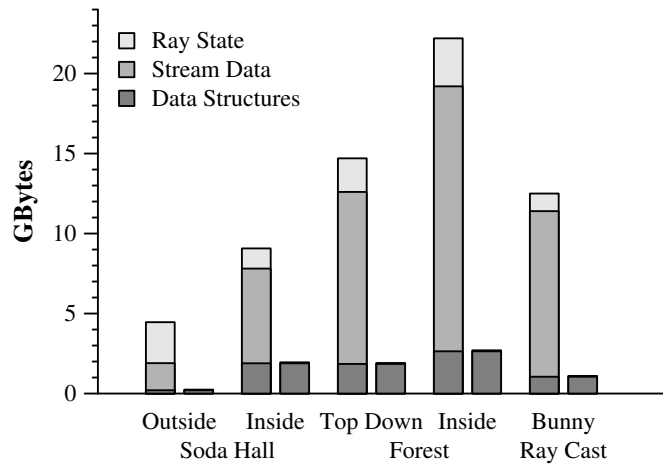


Figure 5.5: Bandwidth consumption by data type. Left bars are for the SIMD architecture, right bars for the MIMD architecture. Ray state is the bandwidth for reading and writing the 8-bit state-buffer. Stream data are data written to and read from texture memory between passes. Data structure bandwidth comes from read-only data: triangles, triangle lists, grid cells, and shading data. All scenes were rendered at 1024×1024 pixels.

Percentage of Active Rays per Rendering Pass

We also evaluate the average number of active rays per rendering pass for the SIMD architecture. Recall that rays may be in one of four states: traversing, intersecting, shading, or done. Between 6-10% of rays are active every pass in our test scenes, except for the outside view of Soda Hall, as shown in table 5.2. This viewpoint contains several rays that miss the scene bounding box entirely. As expected, the number of active rays is much lower since rays that miss the scene completely never become active during the rest of the computation. Although an average of 10% active rays may seem low, the fragment processor utilization is much higher since we are relying on early-z occlusion culling to discard inactive rays, freeing compute resources and bandwidth for active rays.

Table 5.2 also shows the maximum number of traversal steps and intersection tests that are performed per ray. Since the total number of passes depends on the worst case ray, these numbers provide lower bounds on the number of passes needed.

Scene	Per-Ray Maximum		Total SIMD Passes	Avg. Percent Active Rays
	Traversals	Intersections		
Soda Hall Outside	384	1123	2443	0.9%
Soda Hall Inside	60	1039	1198	6.1%
Forest Top Down	137	1435	1999	6.2%
Forest Inside	898	990	2835	6.8%
Bunny Ray Cast	221	328	1085	10.5%

Table 5.2: The total number of SIMD rendering passes (repeated here from figure 5.2) for each scene is bounded from below by the maximum number of traversal steps and intersection tests per ray, shown in this table. The average percentage of rays that are active during each rendering pass is shown in the last column.

Our SIMD algorithm interleaves traversal and intersection passes and comes within a factor of two to three of the optimal number of rendering passes. The naive algorithm, which performs an intersection as soon as any ray hits a full voxel, requires at least a factor of five times more passes than optimal on these scenes.

One way to reduce both the number of rendering passes and the bandwidth consumed by intermediate values in the SIMD architecture is to unroll the inner loops. We have presented data for a single traversal step or a single intersection test performed per ray in a rendering pass. If we instead unroll our kernels to perform four traversal steps or two intersection tests, all of our test scenes reduce their total bandwidth usage by 50%. If we assume we can suppress triangle and voxel memory references if a ray finishes in the middle of the pass, the total bandwidth reduction reaches 60%. At the same time, the total instruction count required to render each scene increases by less than 10%. With more aggressive loop unrolling the bandwidth savings continue, but the total instruction count increase varies by a factor of two or more between our scenes. These results indicate that loop unrolling can make up for some of the overhead inherent in the SIMD architecture, but unrolling still does not achieve the compute to bandwidth ratio obtained by the MIMD architecture.

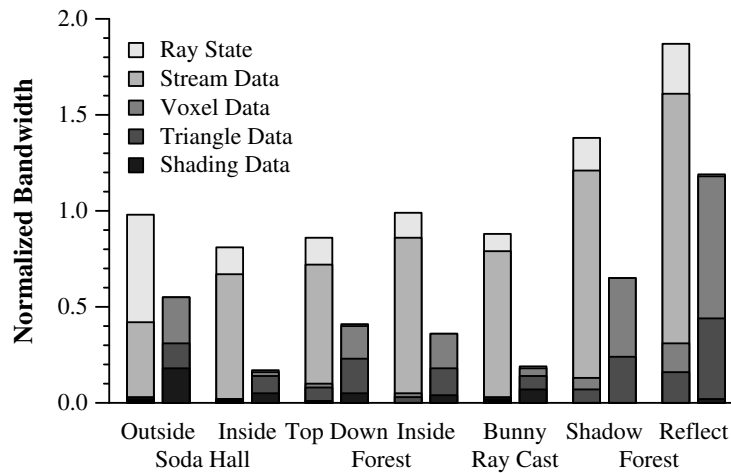


Figure 5.6: Ratio of bandwidth with a texture cache to bandwidth without a texture cache. Left bars are for the SIMD architecture, right bars for the MIMD architecture. Within each bar, the bandwidth consumed is broken down by data type. All scenes were rendered at 1024×1024 pixels.

5.2.3 Stream Buffer vs. Cache

Both the SIMD and the MIMD architecture simulators generate a trace file of the memory reference stream for processing by our texture cache simulator. In our cache simulations we used a 64 kB direct-mapped texture cache with a 48-byte line size. This line size holds four floating point RGB texels, or three floating point RGBA texels with no wasted space. The execution order of fragment programs affects the caching behavior. We execute kernels as though there were a single pixel wide graphics pipeline. It is likely that a GPU implementation will include multiple parallel fragment pipelines executing concurrently, and thus their accesses will be interleaved. Our architectures are not specified at that level of detail, and we are therefore not able to take such effects into account in our cache simulator.

Look again at figure 5.5. Notice that nearly all the bandwidth is consumed by stream buffer data. Figure 5.6 shows the bandwidth requirements when a texture cache is used. The bandwidth consumption is normalized by dividing by the non-caching bandwidth reported earlier. Inspecting this graph we see that the SIMD

architecture does not benefit very much from texture caching. Most of the bandwidth is being used for streaming data, in particular, for either the state-buffer or for intermediate results. Since this data is unique to each kernel invocation, there is no reuse. In contrast, the MIMD architecture utilizes the texture cache effectively. Since most of its bandwidth is devoted to reading shared data structures, there is reuse. If we examine the caching behavior of triangle data only, we see that a 96-99% hit rate is achieved by both the SIMD and the MIMD architectures. This high hit rate suggests that triangle data caches well, and that we have a fairly small working set size. These results indicate that if the SIMD architecture had a separate memory system for stream data (i.e. a stream buffer) and global read-only data, it would achieve a high overall cache utilization like the MIMD architecture.

Additionally, secondary rays do not cache as well as eye rays, due to their generally incoherent nature. The last two columns of figure 5.6 illustrate the cache effectiveness for secondary rays, measured separately from primary rays. For these tests, we render the inside forest scene in two different styles. “Shadow” is rendered with three light sources with each hit producing three shadow rays. “Reflect” applies a two bounce reflection and single light source shading model to each primitive in the scene. For the SIMD architecture, the texture cache is unable to reduce the total bandwidth consumed by the system. Once again the streaming data destroys any locality present in the triangle and voxel data. The MIMD architecture results demonstrate that scenes with secondary rays can benefit from caching. The system achieves a 35% bandwidth reduction for the shadow computation. However, caching for the reflective forest does not reduce the required bandwidth. Improving the coherence of reflected rays in our system is left as future work.

5.2.4 Summary

We have simulated two hypothetical GPUs: one based on SIMD fragment processing, the other based on MIMD fragment processing. The simulations of the ray caster on the SIMD architecture show a very good balance between computation and bandwidth. The ratio of instruction count to bandwidth matches the capabilities of a the

Radeon 9700 Pro. Expanding the traversal and intersection kernels to perform multiple traversal steps or intersection tests per pass reduces the bandwidth required for the scene at the cost of increasing the computational requirements. The amount of loop unrolling can be changed to match the computation and bandwidth capabilities of the underlying hardware. In comparison, the MIMD architecture consumes fewer instructions and significantly less bandwidth. As a result, the MIMD architecture is severely compute-limited based on today's GPU bandwidth and compute rates. The MIMD architecture will become more attractive in the future as the compute to bandwidth ratio on graphics chips increases with the introduction of more parallel fragment pipelines.

5.3 Implementation Results

We implemented our ray tracer on an ATI Radeon 9700 Pro graphics card. The Radeon 9700 Pro was the first DirectX 9 class GPU available. We have measured the peak performance of the fragment processor to be approximately 2.5G instructions/s and 12 GB/s of total memory bandwidth with Catalyst 3.10 drivers. Our ray tracing results were measured on a dual Pentium III 800 MHz machine with 1 GB RAM. The operating system was Microsoft Windows XP with Catalyst 2.3 drivers. The ray tracer was written using `ATI_FRAGMENT_PROGRAM` (which has been since deprecated in favor of `ARB_FRAGMENT_PROGRAM` [ARB, 2003b]).

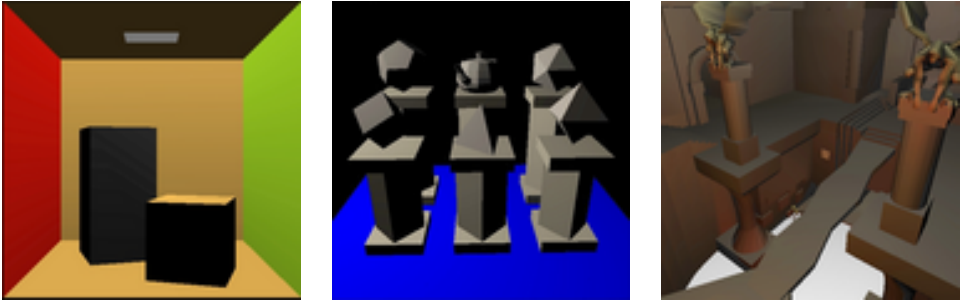
Our ray tracing code requires a custom OpenGL driver that allows us to take some shortcuts when coding the OpenGL code surrounding our fragment programs. Specifically, we are allowed to read from a texture that is bound as an output target. This capability allows us to avoid double buffering our data, conserving texture memory space. This is normally a dangerous optimization since a fragment program could lookup a texture value that has changed (a read after write hazard). Fortunately, the textures we modify in this way are used as streambuffer data — stream data that gets modified by a kernel is only ever read within that same kernel, avoiding read after write hazards. We are also allowed to have more than four renderable buffers within

Kernel	Instr. Count	Texture	
		R	W
Generate Eye Ray	27	0	3
Traverse			
Setup	32	2	4
Step	43	7	4
Intersect	59	9	2
Shade			
Color	48	9	4
Shadow	48	9	1
Reflected	46	8	4

Table 5.3: Ray tracing kernel breakdown for our Radeon 9700 Pro implementation. R is the number of 32-bit RGBA textures read, and W is the number of 32-bit RGBA output channels written per rendering pass.

the same rendering context. This means we don't ever have to switch rendering contexts, which Bolz et al. [2003] showed can be quite expensive. We can specify which four of our several buffers are to be mapped as output targets within the OpenGL code. We have not yet migrated our code to `ARB_FRAGMENT_PROGRAM`, and hence can not take advantage of any performance tuning available in newer drivers.

Our ray tracing kernels changed slightly from those used in simulation when we wrote them for the GPU. Table 5.3 shows the new cost of each kernel. We list the number of instructions, the number of textures read in, and the number of output channels used for each kernel. We were required to make all of our textures the same type, namely RGBA floating point textures (4 channels of 32-bit data each). This is partially due to the special OpenGL driver we used, and partially due to the constraints that the Radeon 9700 Pro has for writing to multiple render targets (textures) at the same time. The Radeon 9700 Pro supports writing to up to four textures at once per rendering pass. However, this comes at a price as all the data written must be the same data type, meaning four-component floating point for our application. This increases our bandwidth consumption considerably as compared to our simulations.



	CORNELL BOX	TEAPOTAHEDRON	QUAKE3
Triangles	32	840	35468
Voxels	64	9408	92220

Figure 5.7: Test scenes for the Radeon 9700 Pro ray tracer. Our ray tracer has been run with three scenes: the CORNELL BOX scene, the TEAPOTAHEDRON scene, and the QUAKE3 scene. These images were generated on the GPU using standard feed-forward rendering with diffuse shading. We show the triangle count for each scene along with the number of voxels in the acceleration structure we used for our experiments.

Another difference between our implementation and our simulations is that our GPU implementation utilizes the early-z occlusion culling mentioned in chapter 4. This means that for a given rendering pass, we do not actually know how many rays are actually being processed. This means we can not calculate the bandwidth and computation consumption for our rendered scenes. In addition, the OpenGL driver optimizes fragment programs, making any calculations rough upper bounds at best. Instead we provide a count of the number of traversal and intersection passes required to render each scene — numbers independent of the particular GPU and driver used.

We rendered several test scenes. Figure 5.7 shows each of these scenes rendered with standard OpenGL, along with the number of triangles and the grid resolution used to render each scene. Each scene was chosen to stress a different part of our ray tracing algorithm.

- The CORNELL BOX scene was used as our debugging scene and we also used it run a simple soft shadow algorithm. The soft shadows were generated by storing random numbers in a texture and using that texture when determining

	CORNELL BOX	TEAPOTAHEDRON		QUAKE3
Rays	S	E, S, R	E, R	S
Frame Rate	10.5 fps	1.0 fps	1.5 fps	1.8 fps
Traversals	18	123	79	146
Intersections	54	2350	1499	730

Table 5.4: Timings and pass breakdown of scenes rendered on the Radeon 9700 Pro at 256×256 pixels. Each scene traces eye rays (E), shadow rays (S), reflection rays (R), or a combination of rays depending on the shading model used. We show the frame rates and the number of traversal and intersection passes required to render the scenes from the viewpoint shown in figure 5.7.

shadow ray hit points on the light. This scene was also used to test out a hybrid rendering algorithm where the initial viewing ray hit positions were computed with an OpenGL feed-forward pass. Shadows were then added through a set of ray tracing passes.

- The TEAPOTAHEDRON scene was used to test a pure Whitted ray tracer. The scene is a simplified version of the scene found on the cover of *An Introduction to Ray Tracing* [Glassner, 1989]. For this scene, we cast eye rays, shadow rays, and reflection rays. The system is configured to allow us to turn off any or all of the secondary rays.
- The QUAKE3 scene stress tested our ray tracer with large geometric complexity. We also wanted to render a real game scene to test the performance in a real application. It was rendered with our shadow caster using the hybrid rendering scheme used for the CORNELL BOX scene.

A summary of performance results for each scene is found in table 5.4. Each scene is rendered at 256×256 pixels. We rendered each scene from a variety of viewpoints and with a variety of different shading techniques. The reported frame rates are the range of observed rates for the stated shading method as we moved the camera through the model. The table gives the type of rays traced into the scene, and provides counts for the number of times the traversal and intersection kernels

were called. The shade and traversal setup kernels are called once for each type of ray.

We measured both the peak ray-triangle intersection rate and the number of rays cast per second we were able to achieve on the GPU. We achieve a peak rate of 100M ray-triangle intersection tests per second. The Ray Engine [Carr et al., 2002], a GPU-based ray-triangle intersection engine that runs on the Radeon 8500 [ATI, 2001], achieved a peak rate of 114M ray-triangle intersection tests per second. These numbers compare favorably to the peak rate of 20M intersections per second achieved on an 800 MHz Pentium III [Wald et al., 2001]. More recent results show a peak rate of around 120M intersections per second on a 3.0 GHz Pentium 4 [Wald, 2004].

The peak ray-triangle intersection rate shows that the GPU is very good at large, compute intensive calculations. A better measure for a ray tracing system is the actual number of rays that can be processed per second. This number includes all the steps from creating a ray to computing a color for it. We have observed a range between 300K and 4M rays/s for our test scenes with our system. This again compares favorably to rates between 800K and 7.1M rays/s on a 2.5 GHz Pentium 4 [Wald et al., 2003] that do not include shading. When simple shading is included, the Pentium 4 performance drops to between 1.8M and 2.3M rays/s. These numbers indicate that the streaming formulation for ray tracing can produce high performance code — especially considering the GPU is lacking several features that could boost performance even further.

Figures 5.8, 5.9, and 5.10 show some screen captures of our system running with our test scenes. Figure 5.8 shows the CORNELL BOX scene rendered with our hybrid shadow caster. Figure 5.9 shows the TEAPOTAHEDRON scene rendered with our full Whitted ray tracer. Finally, figure 5.10 shows the QUAKE3 scene, again rendered with our hybrid shadow caster.

5.4 Discussion

In this section, we discuss some near term and longer term improvements to the GPU that would improve the performance of our ray tracer.

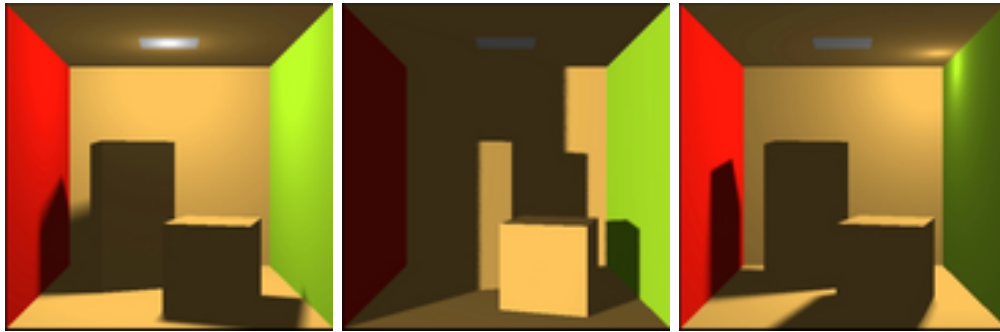


Figure 5.8: CORNELL BOX test scene with ray traced soft shadows on the GPU. This scene was rendered with a shadow caster, with the initial visible hit point found using the feed-forward pipeline. The shadows were added by a set of ray tracing passes.

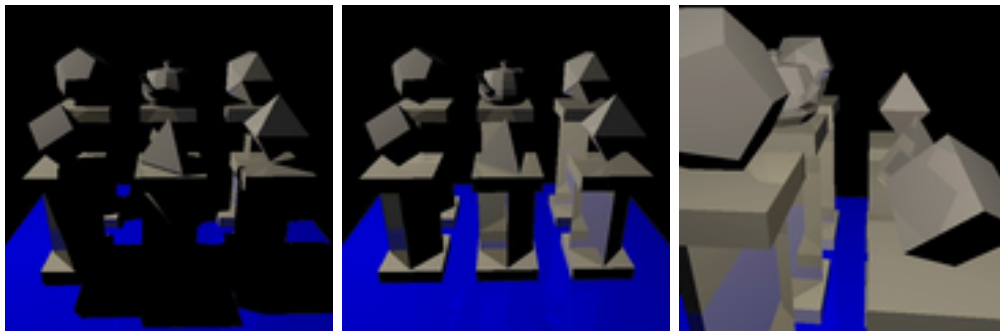


Figure 5.9: TEAPOTAHEDRON scene ray traced with shadows and reflections on the GPU. The right images are rendered with reflections only. This was rendered using a set of rendering passes that implement Whitted-style ray tracing.

5.4.1 Short-Term GPU Improvements

One of the most important improvements that can be made to the Radeon 9700 Pro is to the internal floating point precision. The Radeon 9700 Pro uses a s16e7 24-bit internal format. This format usually works well, except when trying to use the floating point registers to perform integer addressing calculations. The 16-bit mantissa allows for only 131,072 unique integer values. That means we can only address a 256x512 texture through the integer calculations. This caps our scene complexity at 131K triangles and 131K voxels. Clearly, before complex scenes can be ray traced with



Figure 5.10: QUAKE3 rendered with standard feed-forward pipeline shading and shadows added through rendering passes implementing a shadow caster.

this hardware, the mantissa size needs to be increased. A hierarchical acceleration structure might allow for better use of the voxel address space, but we start to hit dependent texture limits in this case.

One potential fix would be to include true integer operations and data types. Integer data types would allow full 24-bit addressing — more than enough to address most scenes. In addition, integer operations such as integer divide and modulo arithmetic would save several operations prior to memory fetches. Inclusion of a full integer ALU in the pipeline would also allow bit operations. These operations could make the uniform grid much more memory efficient.

Finally, the special capabilities in the OpenGL implementation we used are just starting to make their way into public APIs. The Superbuffers extension [Doggett, 2003] allows a programmer to create several light-weight buffers that can be rendered into by a fragment program. These are an alternative to full off-screen frame buffers (p-buffers) when a full rendering context is not needed. Our driver allowed us to have

several auxiliary buffers within a single p-buffer context to prevent expensive context switch overhead.

5.4.2 Long-Term GPU Design Changes

There are two ways that GPUs could evolve that would enable ray tracing to be much more efficient. The first is to add efficient MIMD processing to the fragment pipeline. We saw in section 5.2.2 that MIMD cores make much better use of their resources than SIMD cores for ray tracing. This difference is due to the large amounts of looping and branching in the ray tracing computation.

The second way to make ray tracing much more efficient on a GPU would be to add a stream buffer. Without a stream buffer, a program that requires multiple kernels suffers from inefficient use of both bandwidth and computation resources. Bandwidth gets wasted when data that should live in the stream buffer is passed through the texture cache. The stream data effectively flushes the cache, evicting other potentially useful data.

Additionally, without a stream buffer, the GPU has to rely on early-z occlusion culling to prevent fragments from being processed when they are in the wrong state. This culling currently happens at a granularity larger than a single pixel (in fact, it appears to happen in 4x4 pixel blocks and larger). This means several pixels that have not changed value during the rendering pass read their inputs and write their outputs. This is a waste of both computation and bandwidth resources.

A stream buffer with conditional outputs would not only allow computation and bandwidth to be used more effectively, but would ease programmer burden. For the ray tracer, rays that required more traversal or intersection would be fed into the right streams automatically. The programmer would not have to worry about masking out rays in the wrong state for a kernel execution.

5.5 Conclusions

Through simulation and implementation, we have demonstrated how to map a ray tracer onto a streaming processor. We have shown that programmable GPUs are starting to support a general streaming model of computation. Algorithms such as ray tracing that can benefit from the stream programming model can leverage these processors. The result is high performance code competitive with the best known CPU implementations. Simple changes to GPU architectures would make algorithms like ray tracing run even better.

Chapter 6

Photon Mapping on Programmable Graphics Hardware

6.1 Introduction

Global illumination is essential for realistic image synthesis in general environments. Effects such as shadows, caustics, and indirect illumination are important visual cues that add to the perceived realism of a rendered scene. Photon mapping [Jensen, 1996] is one of the more widely used algorithms, since it is very practical and capable of computing a full global illumination solution efficiently. It is a two-pass technique in which the first pass consists of tracing photons through the scene and recording their interaction with the elements in the scene in a data structure, the photon map. This photon map is used during the second pass, the rendering pass, to estimate diffuse indirect illumination as well as caustics. The illumination at a given point is estimated based on statistics, such as the density, of the nearest photons located in the photon map.

Global illumination algorithms such as photon mapping have traditionally relied on sophisticated software implementations and offline rendering. Using the GPU for computing a global illumination solution has not previously been possible due to the lack of floating point capability, as well as insufficient programmability. This has changed with the most recent generation of programmable graphics hardware such

as the ATI Radeon 9800 Pro [ATI, 2003] and the NVIDIA GeForce FX 5900 Ultra [NVIDIA, 2003a]. The programming model for these GPUs is still somewhat limited, mainly due to the lack of random access writes. This prevents efficient construction of most data structures and makes many common algorithms such as sorting difficult to implement efficiently. Nonetheless, several researchers have harnessed the computational power of programmable GPUs to perform computations previously run in software [Bolz et al., 2003; Carr et al., 2002; Harris et al., 2002; Krüger and Westermann, 2003; Larsen and McAllister, 2001; Purcell et al., 2002]. Similarly, we are interested in using GPUs to simulate global illumination using photon mapping.

Previous research on graphics hardware has explored the idea of simulating global illumination. Ma et al. [Ma and McCool, 2002] proposed a technique for approximate nearest neighbor search in the photon map on a GPU using a block hashing scheme. Their scheme is optimized to reduce bandwidth on the hardware, but it requires processing by the CPU to build the data structure. Carr et al. [Carr et al., 2002] and Purcell et al. [Purcell et al., 2002] used the GPU to speed up ray tracing, as described in chapter 5. They also simulated global illumination using path tracing. Unfortunately, path tracing takes a significant number of sample rays to converge and even with the use of GPUs it remains a very slow algorithm.

Recently, Wald et al. [Wald et al., 2002] demonstrated that photon mapping combined with instant radiosity could be used to simulate global illumination at interactive rates on a Linux cluster. They achieve interactive speeds by biasing the algorithm and by introducing a number of limitations such as a highly optimized photon map data structure, a hashed grid. By choosing a fixed search radius *a priori*, they set the grid resolution so that all neighbor queries simply need to examine the 8 nearest grid cells. However, this sacrifices one of the major advantages of the *k*-nearest neighbor search technique, the ability to adapt to varying photon density across the scene. By adapting the search radius to the local photon density, Jensen's photon map can maintain a user-controllable trade off between noise (caused by too small a radius yielding an insufficient number of photons) and blur (caused by too large a search radius) in the reconstructed estimate.

We present a modified photon mapping algorithm that runs entirely on the GPU. We have changed the data structure for the photon map to a uniform grid, which can be constructed directly on the hardware. In addition, we have implemented a variant of Elias’s algorithm [Cleary, 1979] to search the grid for the k -nearest neighbors of a sample point (kNN-grid). This is done by incrementally expanding the search radius and examining sets of grid cells concentrically about the query point. For rendering, we have implemented a stochastic ray tracer, based on a fragment program ray tracer like that introduced in chapter 5. We use recursive ray tracing for specular reflection and refraction [Whitted, 1980] and distributed tracing of shadow rays to resolve soft shadows from area lights [Cook et al., 1984]. Finally, our ray tracer uses the kNN-grid photon map to compute effects such as indirect illumination and caustics.

Our implementation demonstrates that current graphics hardware is capable of fully simulating global illumination with progressive and even interactive feedback to the user. To compute various aspects of the global illumination solution, we introduce a number of GPU based algorithms for sorting, routing, and searching.

6.2 Photon Mapping on the GPU

The following sections present our implementation of photon mapping on the GPU. Section 6.2.1 briefly describes the tracing of photons into the scene. Section 6.2.2 describes two different techniques for building the photon map data structures on the GPU. Section 6.2.3 describes how we compute a radiance estimate from these structures using an incremental k -nearest neighbor search. Finally, section 6.2.4 briefly describes how we render the final image. A flow diagram for our system is found in figure 6.1.

Most of our algorithms use the programmable fragment engine as a stream processor. For every processing pass, we draw screen sized quad into a floating point p-buffer, effectively running an identical fragment program at every pixel in the 2D buffer. This setup is common among several systems treating the GPU as a computation engine [Bolz et al., 2003; Carr et al., 2002; Purcell et al., 2002]. When computing the radiance estimate, however, we tile the screen with large points, enabling us to

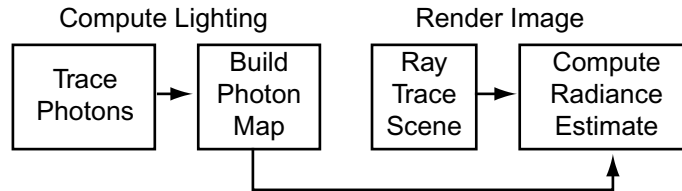


Figure 6.1: Photon mapping system flow. Photon tracing and photon map construction only occur when geometry or lighting changes. Ray tracing and radiance estimates occur at every frame.

terminate certain tiles sooner than other tiles. The benefits of tiling are examined further in section 6.3.

6.2.1 Photon Tracing

Before a photon map can be built, photons must be emitted into the scene. The process of tracing eye rays and tracing photons from a light source is very similar. The most important difference is that at each surface interaction, a photon is stored and another is emitted. Much like tracing reflection rays, this takes several rendering passes to propagate the photons through several bounces. Each bounce of photons is rendered into a non-overlapping portion, or *frame*, of a photon texture, while the results of the previous pass are accessed by reading from the previous frame. The initial frame is simply the positions of the photons on the light source, and their initial random directions. The direction for each photon bounce is computed from a texture of random numbers.

Not all photons generated are valid; some may bounce into space. Current GPUs do not allow us to selectively terminate processing on a given fragment. Instead, GPUs provide a KILL instruction that prevents a fragment value from being written to the framebuffer. Since we are rendering to a separate frame of the photon texture for each bounce, KILL does not save any texture space. Instead, we explicitly mark photons as valid or invalid.

6.2.2 Constructing the Photon Map Data Structure

The original photon map algorithm uses a balanced k -d tree [Bentley, 1975] to store photons. While this structure makes it possible to quickly locate the nearest photons at any point, it requires random access writes to construct efficiently. Instead, we use a uniform grid for storing the photons. In this section we present two different techniques for building a uniform grid photon map. The first method sorts photons by grid cell using bitonic merge sort. This step creates an array of photon indices where all photons in a grid cell are listed consecutively. Binary search is then used to build an array of indices to the first photon in each cell (see figure 6.4 for an example of the resulting data structure). To reduce the large number of passes this algorithm requires, we propose a second method for constructing an approximate photon map using the stencil buffer. In this method, we limit the maximum number of photons stored per grid cell, making it possible to route the photons to their destination grid cells with a single pass using a vertex program and the stencil buffer.

Fragment Program Method — Bitonic Merge Sort

One way to index the photons by grid cell is to sort them by cell and then find the index of the first photon in each cell using binary search.

Many common sorting algorithms require the ability to write to arbitrary locations, making them unsuitable for implementation on current GPUs. We can, however, use a deterministic sorting algorithm for which output routing from one step to another is known in advance. Bitonic merge sort [Batcher, 1968] has been used for sorting on the Imagine stream processor [Kapasi et al., 2000], and meets this constrained output routing requirement of the GPU.

Bitonic merge sort is a parallel sorting algorithm that allows an array of n processors to sort n elements in $O(\log^2 n)$ steps. Each step performs n comparisons and swaps. The algorithm can be directly implemented as a fragment program, with each stage of the sort performed as one rendering pass over an n pixel buffer. Bitonic sort is illustrated graphically in figure 6.2 and the Cg [Mark et al., 2003] code we used

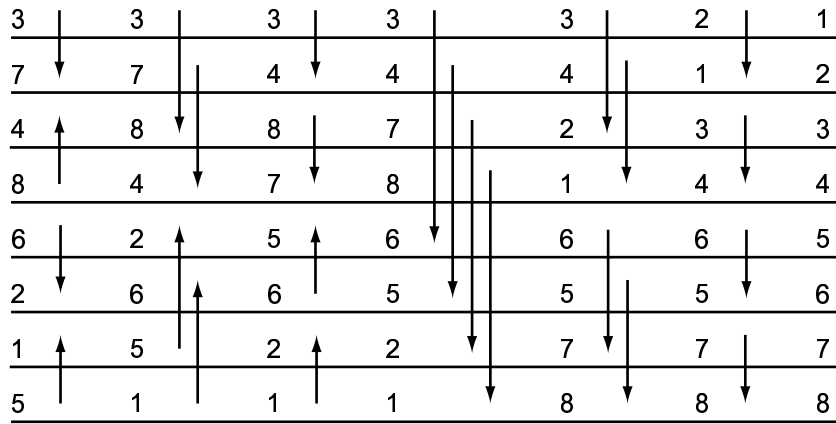


Figure 6.2: Stages in a bitonic sort of eight elements. The unsorted input sequence is shown in the left column. For each rendering pass, element comparisons are indicated by the arrows. Elements at the head and tail of the arrow are compared. The smaller element is placed at the tail of the arrow and the larger element is placed at the head of the arrow. The final sorted sequence is achieved in $O(\log^2 n)$ passes.

to implement it is found in figure 6.3. The result of the sort is a texture of photon indices, ordered by grid cell.

Once the photons are sorted, binary search can be used to locate the contiguous block of photons occupying a given grid cell. We compute an array of the indices of the first photon in every cell. If no photon is found for a cell, the first photon in the next grid cell is located. The simple fragment program implementation of binary search requires $O(\log n)$ photon lookups for each cell. Because there is no need to output intermediate results, all of the photon lookups can be unrolled into a single rendering pass. An example of the final set of textures used for a grid-based photon map is found in figure 6.4.

Sorting and indexing is an effective way to build a compact, grid-based photon map. Unfortunately, the sorting step can be quite expensive. Sorting just over a million photons (1024×1024) would require 210 rendering passes, each applied to the full 1024×1024 buffer. Each compare and swap operation requires two texture reads and one texture write, which makes sorting very bandwidth intensive as well.

```

fragout_float BitonicSort( vf30 In, uniform samplerRECT sortedplist,
                          uniform float offset, uniform float pbufinfo,
                          uniform float stage, uniform float stepno )
{
    fragout_float dst;
    float2 elem2d = floor(In.WPOS.xy);
    float elem1d = elem2d.y*pbufinfo.x + elem2d.x;
    half csign = (fmod(elem1d, stage) < offset) ? 1 : -1;
    half cdir = (fmod(floor(elem1d/stepno), 2) == 0) ? 1 : -1;
    float4 val0 = f4texRECT( sortedplist, elem2d );
    float adr1d = csign*offset + elem1d;
    float2 adr2d = convert1dto2d(adr1d, pbufinfo.x);
    float4 val1 = f4texRECT( sortedplist, adr2d );
    float4 cmin = (val0.y < val1.y) ? val0 : val1;
    float4 cmax = (val0.y > val1.y) ? val0 : val1;
    dst.col = (csign == cdir) ? cmin : cmax;
    return dst;
}

```

Figure 6.3: Cg code for the bitonic merge sort fragment program. The function `convert1dto2d` maps 1D array addresses into 2D texture addresses.

Vertex Program Method - Stencil Routing

The limiting factors of bitonic merge sort are the $O(\log^2 n)$ rendering passes and the $O(n \log^2 n)$ bandwidth required to sort the emitted photons. To support global illumination at interactive rates, we would prefer to avoid introducing the latency of several hundred rendering passes when generating the photon map. We would also prefer an algorithm that is less bandwidth hungry. To address these problems, we have developed an alternate algorithm for constructing a grid-based photon map that runs in a single pass and only requires $O(n)$ bandwidth.

We note that vertex programs provide a mechanism for drawing a `glPoint` to an arbitrary location in a buffer. The ability to write to a computed destination address is known as a scatter operation. If the exact destination address for every photon could be known in advance, then we could route them all into the buffer in a single pass by drawing each photon as a point. Essentially, drawing points allows us to solve a one-to-one routing problem in a single rendering pass.

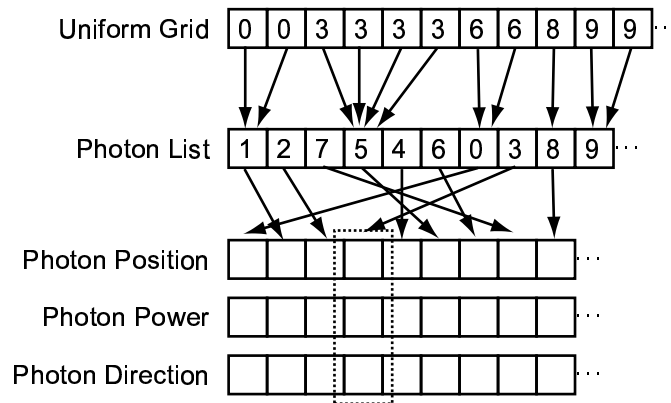


Figure 6.4: Resultant textures for a grid-based photon map generated by bitonic sort. The uniform grid texture contains the index of the first photon in that grid cell. The photon list texture contains the list of photon indices, sorted by grid cell. Each photon in the photon list points to its position, power, and incoming direction in the set of photon data textures.

Our task of organizing photons into grid cells is a many-to-one routing problem, as there may be multiple photons to store in each cell. However, if we limit the maximum number of photons that will be stored per cell, we can preallocate the storage for each cell. By knowing this “texture footprint” of each cell in advance, we reduce the problem to a variant of one-to-one routing.

The idea is to draw each photon as a large `glPoint` over the entire footprint of its destination cell, and use the stencil buffer to route photons to a unique destination within that footprint. Specifically, each grid cell covers an $m \times m$ square set of pixels so each grid cell can contain at most $m \times m$ photons. We draw photons with `glPointSize` set to m which when transformed by the vertex program will cause the photon to cover every possible photon location in the grid cell. We set the stencil buffer to control the location each photon updates within each grid cell by allowing at most one fragment of the $m \times m$ fragments to pass for each drawn photon. The stencil buffer is initialized such that each grid cell region contains the increasing pattern from 0 to $m^2 - 1$. The stencil test is set to write on equal to $m^2 - 1$, and to always increment. Each time a photon is drawn, only one fragment passes through, but the entire $m \times m$ region of the stencil buffer increments. This causes the next photon

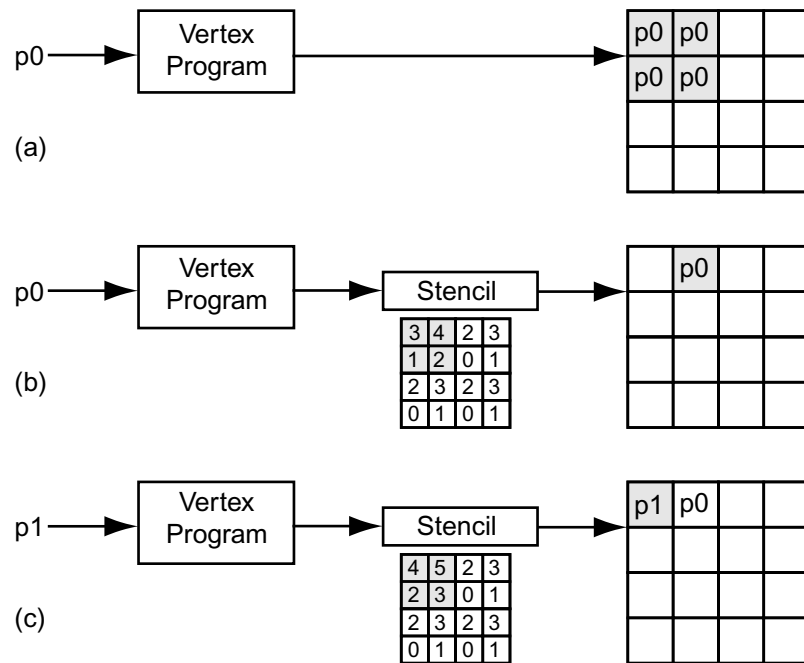


Figure 6.5: Building the photon map with stencil routing. For this example, grid cells can hold up to four photons, and photons are rendered as 2×2 points. Photons are transformed by a vertex program to the proper grid cell. In (a), a photon is rendered to a grid cell, but because there is no stencil masking the fragment write, it is stored in all entries in the grid cell. In (b) and (c) the technique we use is illustrated: the stencil buffer controls the destination written to by each photon.

drawn to a new location in the grid cell. This allows efficient routing of up to the first m^2 photons to each grid cell. This process is illustrated in figure 6.5.

We generally use a 1024×1024 stencil buffer with m set to 16, leaving 65,536 available grid cells (enough for a 40^3 grid). In regions of high photon density, many more photons than can be stored will map to a single grid cell. To reduce the artifacts of this method, we redistribute the power of the surplus photons across those that are stored. Note that the stencil buffer maintains a count of how many photons were destined for each grid cell, and we assume that all our stored photons have roughly the same power. Hence, we can scale the power of the stored photons by the ratio between the number of photons destined for a cell and the number actually stored. This redistribution of power is an approximation, but the potential performance benefits

of the fast routing method make it worthwhile. The idea of redistributing power of some photons to limit the local density of photons stored is discussed more generally in Suykens and Willems [Suykens and Willens, 2000].

By capping the number of photons stored per cell instead of using a variable length list, we can use a vertex program to route photons to grid cells in a single rendering pass. There are two main drawbacks to this method. First, the photons must be read from the photon texture and drawn as points, which currently requires a costly readback to the CPU. Second, the preallocation of storage for each grid cell limits the method's flexibility and space-efficiency. Redistribution of power is needed to represent cells containing more than m^2 photons, and space is wasted for cells with fewer photons (including empty cells).

6.2.3 The Radiance Estimate

To estimate radiance at a given surface location we need to locate the photons nearest to the location. For this purpose we have developed a k -nearest neighbors grid (kNN-grid) method, which is a variant of Elias's algorithm for finding the k -nearest neighbors to a sample point in a uniform grid [Cleary, 1979]. First, the grid cell containing the query point is explored, and all of its photons are examined. As each photon is examined, it will either be added to the running radiance estimate, or rejected. A photon is always rejected if it is outside a predefined maximum search radius. Otherwise, rejection is based on the current state of the search. If the number of photons contributing to the running radiance estimate is less than the number requested, the power of the new photon is added to the running estimate and the search radius is expanded to include that photon. If a sufficient number of photons have already been accumulated, the search radius no longer expands. Photons within the current search radius will still be added to the estimate, but those outside will be rejected.

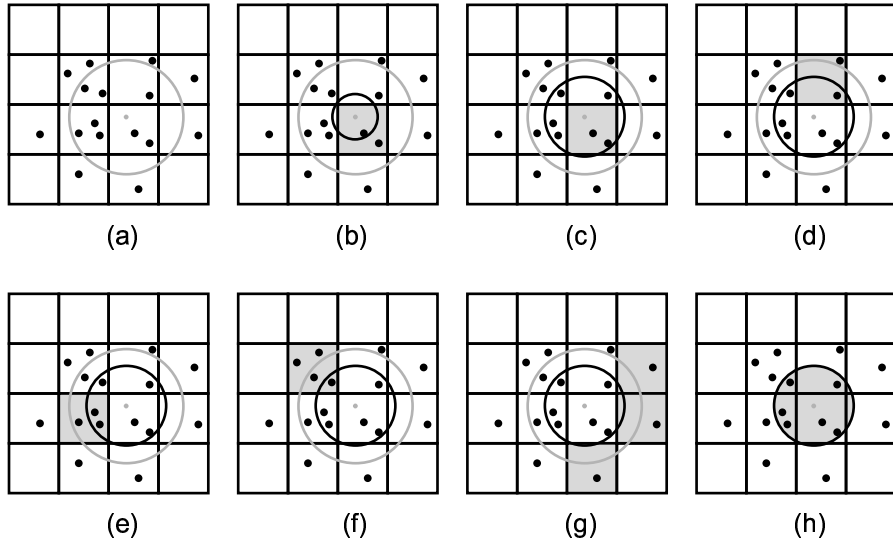


Figure 6.6: Computing the radiance estimate with the kNN-grid. For simplicity, this figure is shown in 2D. We implement a full 3D search in our system. For this example, four photons are desired in the radiance estimate. The initial sample point and the maximum search radius are shown in (a). The first grid cell searched (shaded in (b) and (c) contributes two photons and expands the search radius. The next cell searched (d) has one photon added to the radiance estimate, and the other rejected since it is outside the predefined maximum search radius. The photon outside the search radius in (e) is rejected because the running radiance estimate has the requested number of photons, causing the search radius to stop expanding. The cell in (f) contributes one photon to the estimate. None of the other cells searched in (g) have photons that contribute to the radiance estimate. The final photons and search radius used for the radiance estimate are shown in (h).

Grid cells are explored in concentric sets centered about the query point. The photon search continues until either a sufficient number of photons have been accumulated, or a predefined maximum search radius has been reached. Figure 6.6 illustrates the kNN-grid algorithm.

The kNN-grid always finds a set of nearest neighbor photons – that is, all the photons within a sphere centered about the query point. It will find at least k nearest photons (or as many as can be found within the maximum search radius). This means that the radius over which photons are accumulated may be larger than it

is in Jensen's implementation [Jensen, 2001], which uses a priority queue to select only the k -nearest neighbors. Accumulating photons over a larger radius could potentially introduce more blur into our reconstructed estimates. In practice, however, we have not observed any degradation in quality.

6.2.4 Rendering

To generate an image we use a stochastic ray tracer written using a fragment program. The output of the ray tracer is a texture with all the hit points, normals, and colors for a given ray depth. This texture is used as input to several additional fragment programs. One program computes the direct illumination using one or more shadow rays to estimate the visibility of the light sources. Another program invokes the ray tracer to compute reflections and refractions. Finally, we use the kNN-grid fragment program described in the previous section to compute the radiance estimates for all the hits generated by the ray tracer. We display the running radiance estimate maintained by the kNN-grid algorithm, providing progressively better global illumination solutions to the viewer.

6.3 Results

All of our results are generated using a GeForce FX 5900 Ultra and a 3.0 GHz Pentium 4 CPU with Hyper Threading and 2.0 GB RAM. The operating system was Microsoft Windows XP, with version 43.51 of the NVIDIA drivers. All of our kernels are written in Cg [Mark et al., 2003] and compiled with cgc version 1.1 to native fp30 assembly.

6.3.1 Rendered Test Scenes

In order to simplify the evaluation of the photon mapping algorithm we used scenes with no ray tracing acceleration structures. For each scene, we write a ray-scene intersection routine in Cg that calls ray-quadric and ray-polygon intersection functions for each of the component primitives. For these simple scenes, the majority of our

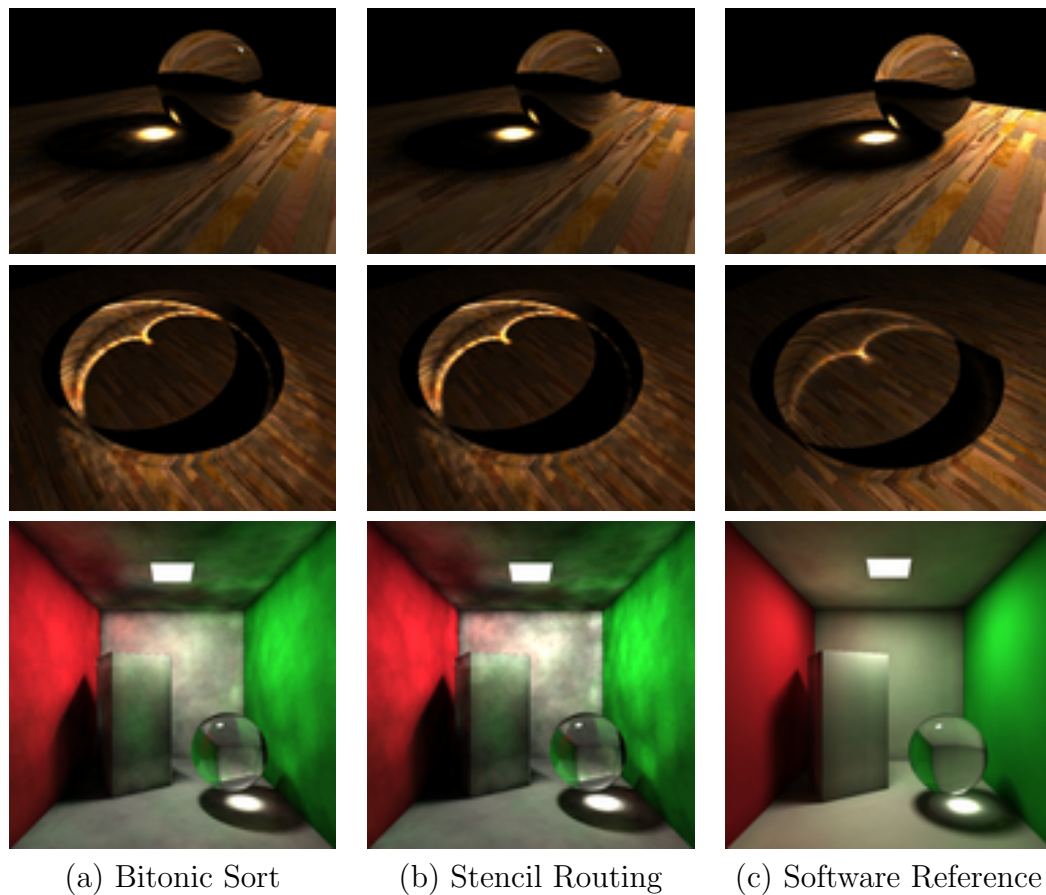


Figure 6.7: Test scene renderings. Both (a) and (b) were rendered on the GPU using bitonic sort and stencil routing respectively. Software renderings using Jensen’s algorithm are shown in (c) for reference.

system’s time is spent building the photon map, and computing radiance estimates. Very little time is spent on ray intersection. We will examine the performance impact of complex scenes later in section 6.4.

We have rendered several test scenes on the GPU using our photon mapping implementation. Figure 6.7 shows three sets of images of our test scenes. The first column shows the images produced by the GPU when using the kNN-grid on a photon map generated by bitonic sort. The second shows the results of using stencil routing and power redistribution when rendering the scenes. The third column shows a software reference image rendered with Jensen’s algorithm.

All of our test scenes are rendered with a single eye ray per pixel. The GLASS BALL and CORNELL BOX scenes have area lights which are randomly sampled by the ray tracer when computing shadows. The GLASS BALL scene samples the light source four times per pixel, and the CORNELL BOX scene samples the light source 32 times per pixel. The RING scene uses a point light source and only shoots one shadow ray per pixel.

The GLASS BALL scene was rendered at 512×384 pixels using a $250 \times 1 \times 250$ grid with 5,000 photons stored in the grid and 32 photons were sought for each radiance estimate. The RING scene was rendered at 512×384 pixels using a $250 \times 1 \times 250$ grid with 16,000 photons stored in the grid and 64 photons were sought for each radiance estimate. Finally, the CORNELL BOX scene was rendered at 512×512 pixels using a $25 \times 25 \times 50$ grid with 65,000 photons stored and 500 photons sought for each radiance estimate.

The rendering times for our test scenes vary between 8.1 seconds for the RING scene and 64.3 seconds for the CORNELL BOX scene. Table 6.1 summarizes the rendering times for the images, broken down by computation type.

The majority of our render time is spent computing the radiance estimates. The times listed in table 6.1 are for every pixel to finish computation. However, for our example scenes we find that the system reaches visual convergence (that is, produces images indistinguishable from the final output) after a much shorter time. In the GLASS BALL scene, a photon map built with bitonic sort will visually converge in 4 seconds — nearly four times as fast as the time listed for full convergence would suggest. This fast visual convergence happens for two reasons: First, dark areas of the scene require many passes to explore all the grid cells out to the maximum search radius, but few photons are found so the radiance estimate changes little. Second, bright regions have lots of photons to search through, but often saturate to maximum intensity fairly early. Once a pixel is saturated, further photons found do not contribute to its final color. Note that these disparities between visual and total convergence times are not observed when the photon map is built using stencil routing. Under that method, the grid cells contain a more uniform distribution of photons, and intensity saturation corresponds to convergence.

Scene Name	Trace Photons	Build Grid	Trace Rays	Radiance Estimate	Total Time
GLASS BALL	1.2 s	0.8 s	0.5 s	14.9 s	17.4 s
RING	1.3 s	0.8 s	0.4 s	6.5 s	9.0 s
CORNELL BOX	2.1 s	1.4 s	8.4 s	52.4 s	64.3 s

(a) Bitonic Sort

Scene Name	Trace Photons	Build Grid	Trace Rays	Radiance Estimate	Total Time
GLASS BALL	1.2 s	1.8 s	0.5 s	7.8 s	11.3 s
RING	1.3 s	1.8 s	0.4 s	4.6 s	8.1 s
CORNELL BOX	2.1 s	1.7 s	8.4 s	35.0 s	47.2 s

(b) Stencil Routing

Table 6.1: GPU render times in seconds for the scenes shown in figure 6.7, broken down by type of computation. Table (a) shows the times for the bitonic sort method, and table (b) shows the times for stencil routing method. Ray tracing time includes shooting eye rays and shadow rays. The GLASS BALL scene and RING scene were each rendered at 512×384 pixels. The CORNELL BOX scene was rendered at 512×512 pixels.

6.3.2 Kernel Instruction Use

A breakdown of how the kernels spend time is important for isolating and eliminating bottlenecks. The instruction breakdown tells us whether we are limited by computation or texture resources, and how much performance is lost due to architectural restrictions. Table 6.2 shows the length of each compiled kernel. These instruction counts are for performing one iteration of each computation (e.g. a single step of binary search or a single photon lookup for the radiance estimate). The table further enumerates the number of instructions dedicated to texture lookups, address arithmetic, and packing and unpacking of data into a single output.

We see at least 20 arithmetic operations for every texture access. It may be surprising that our kernels are limited by computation rather than memory bandwidth.

Kernel	Inst	TEX	Addr	Pack
Bitonic Sort	52	2	13	0
Binary Search	18	1	13	0
Rad. Estimate	202	6	47	41
Stencil Routing	42	0	25	0
Rad. Estimate	193	5	20	41

Table 6.2: Instruction use within each kernel. Inst is the total number of instructions generated by the Cg compiler for one iteration with no loop unrolling. Also shown are the number of texture fetches (TEX), address arithmetic instructions (Addr), and bit packing instructions (Pack).

Generally, we would expect sorting and searching to be bandwidth-limited operations. There are several factors that lead our kernels to require so many arithmetic operations:

- Limits on the size of 1D textures require large arrays to be stored as 2D textures. A large fraction of our instructions are spent converting 1D array addresses into 2D texture coordinates.
- The lack of integer arithmetic operations means that many potentially simple calculations must be implemented with extra instructions for truncation.
- The output from an fp30 fragment program is limited to 128 bits. This limit forces us to use many instructions to pack and unpack the multiple outputs of the radiance estimate in order to represent the components in the available space.

Our kernel analysis reveals the challenges of mapping traditional algorithms onto GPUs. For algorithms like sorting, the limited functionality of the GPU forces us to use algorithms asymptotically more expensive than those we would use on processors permitting more general memory access (e.g. we use the $O(n \log^2 n)$ bitonic merge sort instead of $O(n \log n)$ quicksort). In other cases, the limitations of the GPU force us to expend computation on overhead, reducing the effective compute performance.

In section 6.4, we discuss several possible architectural changes that would improve the performance of algorithms like photon mapping.

It should be noted that hand coding can still produce kernels much smaller than those generated by the Cg compiler. For example, we have hand coded a bitonic sort kernel that uses only 19 instructions instead of the 52 produced by Cg. However, we determined that the productivity benefits of using Cg during development outweighed the tighter code that could be achieved by hand coding. As the Cg optimizer improves, we anticipate a substantial reduction in the number of operations required for many of our kernels.

6.3.3 SIMD Overhead

Our radiance estimate kernel is run by tiling the screen with large points instead of with a single quad. Using the `NV_OCCLUSION_QUERY` extension, we are able to stop drawing a tile once all its pixels have finished their work. By terminating some tiles earlier than others, we are able to reduce the amount of SIMD overhead for our radiance estimate kernel.

This early termination of tiles substantially reduced the time required for our scenes to converge. We found tiling the screen with 16×16 points resulted in the largest improvements in convergence time. The `CORNELL BOX` scene saw the least improvement, with the time for the radiance estimate to fully converge dropping from 104 seconds to 52.4 seconds. Full convergence of the `GLASS BALL` scene was more dramatically affected, dropping from 102 seconds down to 14.9 seconds. These results are expected as the `CORNELL BOX` scene has a fairly uniform photon distribution but the `GLASS BALL` scene has high variance in photon density. We suggest ideas for more general ways to reduce SIMD overhead via a fine-grained “computation mask” in section 6.4.

6.3.4 Interactive Feedback

One advantage of the incremental radiance estimate is that intermediate results can be drawn directly to the screen. The images in figure 6.7 required several seconds to

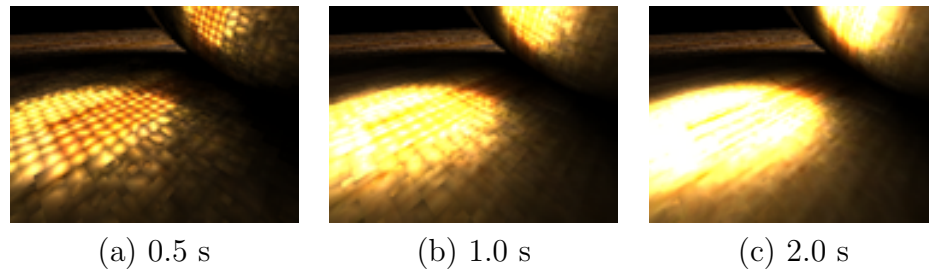


Figure 6.8: A detailed image of the GLASS BALL caustic over time. Reasonably high quality estimates are available much sooner than a fully converged solution.

fully converge. However, initial estimates of the global illumination are available very rapidly. Figure 6.8 shows various stages in the convergence of the radiance estimate for the full resolution GLASS BALL scene.

For smaller image windows, our system can provide interactive feedback. When rendering a 256×256 window, we can interactively manipulate the camera, scene geometry, and light source. Once interaction stops, the photon map is rebuilt and the global illumination converges in only one or two seconds.

6.4 Discussion and Future Work

In this section we discuss the limitations of the current system and areas for future work.

6.4.1 Fragment Program Instruction Set

The overhead of address conversion, simulating integer arithmetic, and packing is a dominant cost in many of our kernels. Addressing overhead accounts for nearly 60% of the cost of the stencil routing, and over 72% of the cost of the binary search. Similarly, the radiance-estimate kernels currently spend a third to a half of their instructions on overhead. Native support for integer arithmetic and addressing of large 1D arrays need not substantially complicate GPU design, but would dramatically reduce the amount of overhead computation needed in these kernels. Additionally,

providing multiple outputs would remove the need for aggressive packing of values in the radiance estimates. Even with the overhead eliminated from the radiance estimate kernels, they still execute several arithmetic instructions and would continue to benefit from increased floating point performance without being limited by memory bandwidth.

6.4.2 Memory Bottlenecks

Texture readback and copy can impose significant performance penalties. We have shown timings for renderings with tens of thousands of photons. The stencil routing performance is particularly affected by readback performance since we currently must readback the texture of photons in order to use them as input to the vertex processor. With a low number of photons, texture readback consumes about 10% of the photon map construction time. However, as the number of photons increases, the fraction of time dedicated to photon readback increases to 60% and more of the total map construction time. The Vertex Shader 3.0 specification found in the DirectX 9 API [Microsoft, 2003] supports displacement mapping, effectively permitting texture data to control point locations. We anticipate that similar functionality will appear as an extension to OpenGL, which would eliminate the need for readback in our stencil sort.

6.4.3 Parallel Computation Model

We mentioned in section 6.3 that we obtained a significant performance improvement by computing the radiance estimate by tiling the screen with large points instead of a full screen quad. Unfortunately, tiling is only practical when relatively few tiles are used and when pixels with long computation times are clustered so that they do not overlap too many tiles. One natural solution to reducing the SIMD overhead for pixels with varying workloads is what we call a “computation mask”. A user controllable mask could be set for each pixel in an image. The mask would indicate pixels where work has completed, allowing subsequent fragments at that location to be discarded immediately. This is essentially a user specified early-z occlusion query.

We observed a performance gain from two to ten using a coarse tiling, and believe that a computation mask with single pixel granularity would be even more efficient.

6.4.4 Uniform Grid Scalability

One issue associated with rendering more complex scenes is that the resolution of the grid used for the photon map needs to increase if we want to resolve illumination details. For sufficiently large scenes a high density uniform grid becomes too large to store or address on the GPU, and empty cells dominate the memory usage. One fix is to store the photons in a hash table based on their grid cell address [Wald et al., 2002]. High density grids no longer have empty cell overhead or addressability issues. Handling hash table collisions would add some overhead to the radiance estimate, however, as photons in the hash bucket not associated with the current grid cell must be examined and ignored. An additional problem for our stencil routing approach is that power redistribution becomes non-trivial.

6.4.5 Indirect Lighting and Adaptive Sampling

Our current implementation directly visualizes the photon map for indirect lighting and caustics. While this approach works well for caustics, the indirect lighting can look splotchy when few photons are used. A large number of photons are needed to obtain a smooth radiance estimate when the photon map is visualized directly. Instead, it is often desirable to use distributed ray tracing to sample incident lighting at the first diffuse hit point, and use the photon map to provide fast estimates of illumination only for the secondary rays. This final gather approach is more expensive, although the cost for tracing indirect rays can often be reduced using techniques like irradiance gradients [Ward and Heckbert, 1992] or adaptive sampling.

We have considered an adaptive sampling algorithm that initially computes a low resolution image and then builds successively higher resolution images by interpolating in low variance areas and tracing additional rays in high variance areas. Our initial studies have shown that this algorithm can reduce the total number of samples

that need to be computed by a factor of 10. However, such a scheme cannot be implemented effectively without support for a fine-grained computation mask like that described in section 6.4.3.

6.5 Conclusions

We have demonstrated methods for constructing a grid-based photon map, and for searching for at least k -nearest neighbors using the grid, entirely on the GPU. All of our algorithms are compute bound, meaning that photon mapping performance will continue to improve as next-generation GPUs increase their floating point performance. We have also proposed several refinements for extending future graphics hardware to support these algorithms more efficiently.

We hope that by demonstrating the feasibility of a global illumination algorithm running completely on graphics hardware, we will encourage GPU designers to improve support for these types of algorithms.

Chapter 7

Ray Tracing and the Memory–Processor Performance Gap

Processor designers must be aware of several technology trends that cause the optimal design for a computer architecture to change with time. In this chapter, we will focus on one trend that makes designing a faster processor difficult: the increasing performance gap between memory and the processor. In section 7.2, we show that the stream programming model results in programs that can be run on hardware architectures optimized to minimize the penalty for memory accesses — resulting in better overall utilization of VLSI resources. We will contrast programs written in the stream programming model with those written in the C-style sequential programming model in section 7.3 via examples from highly tuned software ray tracing systems. The conclusion is that programs written for the stream programming model, such as a streaming ray tracer, are more naturally matched to take advantage of hardware trends than programs written in the C-style sequential programming model.

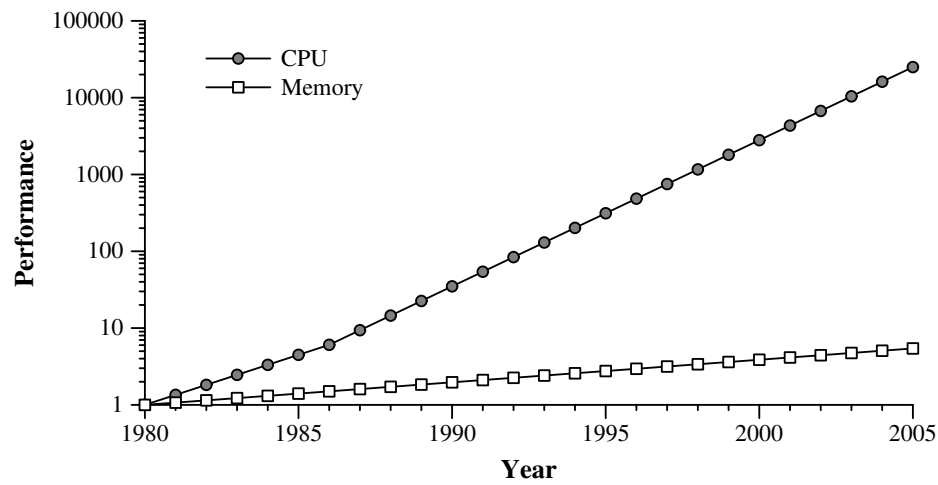


Figure 7.1: The memory–processor performance gap. This figure uses 1980 performance levels as a baseline. CPU performance increases at 35% per year through 1986 and then at 55% per year. Memory latency performance improves at 7% per year. Data from *Computer Architecture: A Quantitative Approach*, 3rd Edition by John L. Hennessy and David A. Patterson [2002].

7.1 The Memory–Processor Performance Gap

Over the past several years, there have been tremendous improvements in processor performance — doubling approximately every 18 months to two years. Memory chips have also seen improvements in capacity, bandwidth, and latency during this time. Unfortunately, memory latencies have not decreased at the same rate that processor performance has increased. Processor performance is increasing at a yearly rate of 55%, but the performance of memory latency is improving at a rate of only about 7% [Hennessy and Patterson, 2002]. Thus, the relative penalty for accessing memory increases with time. This trend, shown in figure 7.1, is known as the memory–processor performance gap. The challenge for chip designers today is to continue to increase processor performance while minimizing the impact of large memory access latencies.

There are two common ways that an architecture can deal with the memory–processor performance gap: parallelism and locality. At the most basic level, parallelism introduces multiple simultaneous computations. Examples of architectural

support for parallelism include multiple processors, multiple threads, instruction level parallelism through pipelining, superscalar processors, and speculative execution. Parallelism allows one task to execute while another is waiting for a memory request to return. Unfortunately, compilers can not usually automatically extract high levels of parallelism from arbitrary code. Instead, programmers often must provide hints to the compiler to achieve a high degree of parallelization.

The other way an architecture can minimize the effect of the memory-processor performance gap is to exploit locality. *Spatial locality* means that nearby memory addresses tend to be accessed close together in time. *Temporal locality* means that recently accessed memory is likely to be accessed again in the near future. Architectures have typically exploited both types of locality through memory hierarchies. Memory is typically organized in a hierarchical fashion, with the lowest level (registers) being fast, small, and close to the processor. Subsequently higher levels in the hierarchy are further away from the processor, larger, and have longer access times. Main system memory is the top of the hierarchy and generally has quite slow access times. Memory hierarchies attempt to exploit spatial locality by sending blocks of nearby memory when a single address is requested. Temporal locality is exploited simply by having the hierarchy: memory previously transferred from a higher level stays in the lower level until another access replaces it. Locality in a program does not happen automatically, however. As with parallelism, there are limits to the amount of locality a compiler and the underlying hardware can extract from a program without a programming model that explicitly expresses temporal and spatial locality.

7.2 Streaming

The stream programming model discussed in chapter 3 has several advantages over the standard C-style sequential programming model. First, it makes the parallelism in the code explicit. Second, it constrains the memory access pattern for stream accesses to exhibit both spatial and temporal locality. Finally, it makes performance tuning much less processor specific. This model leads to efficient code, and enables the use of

special stream hardware that efficiently uses VLSI resources. In this section we will examine how the parallelism and locality made explicit by the stream programming model are used by stream processors to overcome the memory-processor performance gap.

7.2.1 Parallelism

The parallelism in a program written in the stream programming model is explicit through the use of kernels. All stream data processed by a kernel can by definition be processed in parallel. The value of a kernel function for a given stream element is not allowed to depend on the results of any other stream element's evaluation of the kernel. This simple constraint allows the underlying hardware to exploit parallelism in several different ways.

- **Data parallelism.** Since the result of a kernel execution on a stream element can not depend on the results of that kernel executing on a neighboring stream element, processing order constraints are removed. Every stream element can be processed independently, possibly by a separate processing unit. The stream programming model makes no assumptions about the number of execution units that the underlying hardware has, so the level of parallelism can increase simply by adding more execution units. For example, graphics processors have exploited data parallelism to increase performance by adding more and more pipelines.
- **Task parallelism.** Each kernel is independent. That is, one kernel can begin executing as soon as its input stream has some elements. Taking advantage of task parallelism can require more complicated scheduling, but it means that more execution units can be added at any processing bottlenecks in a computation to gain a speed increase. Graphics processors exploit task parallelism by separating vertex processing from fragment processing. As applications have become more fragment intensive, GPUs have increased the number of fragment processors relative to vertex processors.

- **Latency hiding.** Perhaps the most interesting way that the stream programming model takes advantage of parallelism is through latency hiding. When a particular stream element needs to do an expensive memory fetch, its processing state can be swapped out into a delay queue and another element can begin processing. If the queue is long enough, then the outstanding memory request is guaranteed to return before the swapped out task reaches the front. This latency hiding technique has been used successfully in the graphics processor texture memory system [Torborg and Kajiyama, 1996; Anderson et al., 1997; Igehy et al., 1998]. The size of the delay queue needed to hide memory latency depends on the *arithmetic intensity* of the executing kernels. Arithmetic intensity is a measure of the amount of computation performed per word of memory fetched. Kernels with high arithmetic intensity will be inserted into the delay queue less often than low arithmetic intensity kernels. A stream architecture optimized for kernels with high arithmetic intensity can use a shorter delay queue than a stream architecture designed to hide the memory latency of kernels with low arithmetic intensity. The cost savings of a smaller delay queue means many stream processors are designed to achieve high performance only on kernels with sufficiently high arithmetic intensity.

7.2.2 Locality

As mentioned in the previous section, the most effective kernels exhibit high arithmetic intensity. We showed that having many parallel tasks allowed memory fetches to be amortized over several computational operations. The stream programming model also makes explicit the spatial and temporal locality of a program. This means the code can more easily be optimized to make more memory requests to fast memory instead of slow memory, further reducing the effect of the memory-processor performance gap. Spatial and temporal locality are exploited by the stream programming model as follows:

- **Spatial locality.** Kernels process input records in streams. Every element in a stream will be touched when a kernel is executed. The compiler can optimize

the data layout of the stream for the memory architecture of the underlying hardware. In particular, it can organize stream data to be in a contiguous block so that data transfers from memory can be performed at whatever granularity is most efficient for the hardware.

- **Temporal locality.** The data stored in local registers are very likely to be accessed again soon. Local registers for temporary variables in a kernel are accessed in essentially random patterns. However, the stream programming model guarantees that the temporaries only exist within a kernel and typically limits the number of registers.

7.3 CPU-based Ray Tracing

To date, CPUs have dealt with the memory-processor performance gap in a different way than stream processors. In this section we examine two interactive ray tracing systems: the Saarland RTTR system [Wald et al., 2001; 2002] and the Utah *-Ray system [Parker et al., 1998; 1999a; 1999b]. We will first see what a cache-based CPU with a single thread of execution can do to overcome the memory-processor performance gap. We will then examine both RTTR and *-Ray in our discussion of cache-based multithreaded and multiprocessor architectures.

7.3.1 Single CPU Architectures

In this section we examine cache-based single threaded architectures. We will ignore simultaneous multithreading [Tullsen et al., 1995] (e.g. Hyper-Threading [Marr et al., 2002]) in this discussion; we consider it to be a form of multithreading.

Parallelism

Traditional single threaded scalar processors like today's PCs have few options available for leveraging the available parallelism in an application. In general, the instruction sets of these processors require additional data parallel operations commonly found in vector processors to leverage application parallelism.

Modern PC processors provide four to sixteen vector SIMD instructions to improve performance on code that has fine-grained data parallelism. These SSE and SSE2 instruction sets [Intel, 2004] expose a limited amount of SIMD parallelism, but the SSE execution unit is exposed as a processor for vectors of a known, fixed length. The programming model for this unit does nothing to abstract the vector size. When the SSE unit vector length changes (i.e. becomes twice as wide), existing code has to be rewritten (and probably restructured) to take advantage of the wider unit.

Locality

In a single threaded system, the CPU relies on low-latency cache memory to reduce the effects of the memory-processor performance gap. Unlike a stream based architecture (or multithreaded architecture as we'll see in the next section), these architectures can not quickly swap tasks while waiting for memory. The cache is used as a low-latency read/write buffer so an application does not have to wait for the slow memory system with every request.

Applications can only achieve the highest performance levels if they are carefully tuned to the underlying memory system. Data structures have to be tuned to cache line sizes, and aligned to memory boundaries to avoid cache collisions. Without careful data structure management, the cache can actually degrade performance. Memory is loaded into the cache one line at a time. If a set of memory requests touches multiple elements from the same cache line, the overhead from fetching the line is amortized over all the accesses. However, sloppy access to the cache can cause more bandwidth transfer than the data requested. If a set of memory requests only touches a fraction of each cache line, more data is transferred than needed, wasting bandwidth.

7.3.2 Multithreaded and Multiprocessor Architectures

Both RTRT and *-Ray are ray tracing systems written to run on several processors. RTRT runs on a cluster of PCs connected by a fast network, and *-Ray is designed

to run on a shared memory SGI Origin 2000. Both systems are designed for a multi-processor implementation. The principles discussed here also apply to multithreaded architectures, as described by Levy [2002].

Parallelism

RTRT is built out of commodity single-threaded PCs. It exploits the parallelism we described in the previous section on single-threaded architectures. The cluster implementation of RTRT gives it, along with *-Ray, additional opportunities to exploit the parallelism found in ray tracing.

- **Data parallelism.** In a ray tracer, each ray can be processed independently from every other ray. Both RTRT and *-Ray separate the workload into chunks of independent rays to be processed. RTRT works on packets of rays simultaneously in the SSE core. In both systems, ray distribution to processing cores is handled at the software level by the programmer.
- **Task parallelism.** Neither *-Ray nor RTRT exhibit task parallelism. That is, each takes a ray from generation through shading on the same processor. This choice is made because of the high cost of moving data between processors. The RTRT system breaks computation up into small SSE kernels, but these kernels are not scheduled to run independently.
- **Latency hiding.** The RTRT system hides latency to remote memory by issuing network prefetches for batches of rays. RTRT has separate *software* threads (with relatively higher context switch times than hardware threads) for fetching data and for executing the ray tracing computation. The prefetch thread fetches data while the execution thread processes the batch of rays already on the local machine.

The same levels of parallelism in the ray tracing algorithm are available to traditional CPU-based systems as to a stream processor. The challenge on the CPU is to write code to take advantage of them. Both *-Ray and RTRT must stall their processors on L2 cache misses or when prefetching fails. They do not use parallelism

to cover the memory access latency. As such, both *-Ray and RTRT required massive engineering efforts and hand coding to achieve high performance.

7.3.3 Locality

Both *-Ray and RTRT use data structures carefully tuned to the underlying architectures to increase the spatial locality of their ray tracers. The *-Ray grid data structure was tuned to the cache of the R10000 processor and they optimized further for TLB hits. The data structures in RTRT were tuned to the Pentium III cache lines, and tuned to perform well with the SSE execution unit. They separated shading data from intersection data to reduce useless memory transfers and cache pollution.

Both systems also organized their systems to maximize temporal locality. Rays were assigned to processors in contiguous screen-space blocks. By blocking the computation in this manner, adjacently-processed rays are likely to access the same pieces of the acceleration structure and the same scene geometry. This means the processor caches can effectively reduce the performance impact of the memory access latency.

7.4 Conclusion

We have shown two different ways architectures can minimize the effects of the memory-processor performance gap: by exploiting parallelism and locality. We have also seen that the stream programming model and the traditional C-style programming model expose these methods quite differently.

The stream programming model exposes the parallelism and locality of an application through a restricted programming model consisting of kernels and streams. This approach allows a compiler to more easily generate optimal code for the underlying architecture. Applications written in the C-style programming model can also optimize for a given architecture. But this model places the burden of optimizing on the programmer, so that changes to the architecture can require major changes to application code. The stream programming model minimizes the amount of processor-specific tuning that a programmer must do to optimize code performance.

Stream processors are designed to efficiently implement the stream programming model. This does not mean that dedicated stream processors are the only architectures that a stream program can be efficiently compiled to. Traditional CPUs, multithreaded architectures, and multiprocessor architectures can also be targets for a stream program. However, a streaming architecture tends to make better use of VLSI resources than these other architectures when executing stream programs.

Streaming is not always the best programming model or architecture for a task. As we saw with sorting in chapter 6, the restrictions of the stream programming model can force us to use asymptotically slower algorithms for certain tasks. Sometimes, an algorithm may not have enough parallelism to stream at all. More research into writing algorithms for streaming may help us to deal with this problem. Another alternative is to use polymorphic architectures like Smart Memories [Mai et al., 2000] and TRIPS [Sankaralingam et al., 2003] which can be configured to implement the stream programming model or the C-style programming model. These architectures may not be as efficient as stream architectures on highly parallel code, but they are much more efficient at executing serial code. An architecture with both stream and serial processors, or a configurable architecture like these, may in fact be the architecture of choice for a general purpose processing machine.

Chapter 8

Conclusions

8.1 Contributions

This dissertation has made several contributions to computer graphics and graphics hardware design:

- We have shown how to efficiently implement ray tracing and photon mapping using the stream programming model. These algorithms are representative of the types of computation required by nearly every global illumination algorithm. We have shown that these algorithms map well to the stream programming model.
- We have shown how the programmable fragment processor of modern GPUs can implement the stream programming model. This model allows us to map our global illumination calculations, as well as other general purpose computations, onto the high-performance GPU. We have shown that our GPU-based implementation of global illumination algorithms have performance comparable to the fastest known CPU-based implementations. Furthermore, we have explained why we expect the performance of the GPU-based implementations to improve more rapidly with time than the CPU-based implementations.

- We have analyzed the performance of our algorithms running on graphics hardware. Our analysis provides GPU architects with insight into the costs associated with making these computations run in real time. We analyzed the performance of some architectural variations (such as MIMD fragment processors) to help guide the design of future architectures.

8.2 Final Thoughts

We have explored how to merge high performance graphics with high quality graphics. During the development of the work presented in this dissertation, an entire sub-field of computer graphics has emerged: general purpose computation on graphics hardware [GPGPU, 2003]. The work presented here is some of the earliest to take advantage of (and require) the full generality of modern graphics hardware.

We have proposed that the GPU can be thought of as a stream processor. This abstraction works for most applications, but the GPU hardware does not support this abstraction as well as it could. In fact, the stream programming model implemented by the GPU is far less general than it should be. The most glaring difficulty when programming the GPU is answering the question “*How do I perform data dependent computations?*” We have demonstrated that these computations can be performed relatively efficiently through specialized reductions (like `NV_OCCLUSION_QUERY`) and early fragment termination. However, the GPU does not currently implement this model as efficiently as it should and some applications suffer performance penalties accordingly.

Additionally, programming the GPU is not easy. Vendors have not provided the support tools necessary to transform the GPU into a viable general-purpose computing platform. High level languages, debuggers, and code profilers are among the most important platform development tools available to CPU programmers that GPU programmers must for the most part live without. We have several high level languages to choose from [Mark et al., 2003; Microsoft, 2003; ARB, 2003a], but no advanced tools to work with. Until the rest of these tools are readily available, general purpose GPU programming will not become mainstream.

Despite these issues, the GPU is very close to becoming a full high performance parallel co-processor. We have shown that global illumination algorithms like ray tracing and photon mapping can map onto a streaming architecture, and we have shown some ways the GPU can evolve into a general purpose stream processor. We may soon see the gap between realistic and interactive graphics disappear. We hope this work provides some inspiration and insight to facilitate this transition.

Bibliography

- [Amanatides and Woo, 1987] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*, pages 3–10, August 1987.
- [Anderson et al., 1997] Bruce Anderson, Rob MacAulay, Andy Stewart, and Turner Whitted. Accommodating Memory Latency In A Low-Cost Rasterizer. In *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 97–102, August 1997.
- [ARB, 2003a] OpenGL ARB. *ARB Shading Language Extension*, 2003. http://oss.sgi.com/projects/ogl-sample/registry/ARB/shading_language_100.txt.
- [ARB, 2003b] OpenGL ARB. *ARB_Fragment_Program Extension*, 2003. http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt.
- [ARB, 2003c] OpenGL ARB. *ARB_Vertex_Program Extension*, 2003. http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt.
- [ATI, 2001] ATI Technologies Inc. *Radeon 8500 Product Web Site*, 2001. <http://ati.com/products/radeon8500/radeon8500le128/index.html>.

- [ATI, 2002] ATI Technologies Inc. *Radeon 9700 Pro Product Web Site*, 2002. <http://ati.com/products/radeon9700/radeon9700pro/index.html>.
- [ATI, 2003] ATI Technologies Inc. *Radeon 9800 Pro Product Web Site*, 2003. <http://ati.com/products/radeon9800/radeon9800pro/index.html>.
- [Batcher, 1968] Kenneth E. Batcher. Sorting Networks and their Applications. *Proceedings of AFIPS Spring Joint Computing Conference*, 32:307–314, 1968.
- [Bentley, 1975] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.
- [Bolz et al., 2003] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Transactions on Graphics*, 22(3):917–924, July 2003.
- [Buck et al., 2004] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian and Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *submitted to ACM Transactions on Graphics*, 2004.
<http://graphics.stanford.edu/projects/brookgpu>.
- [Buck, 2004] Ian Buck. Merrimac Project Page, 2004.
<http://merrimac.stanford.edu/>.
- [Carr et al., 2002] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *Proceedings of the conference on*

- Graphics hardware 2002*, pages 37–46. Eurographics Association, 2002.
- [Chalmers et al., 2002] Alan Chalmers, Timothy Davis, and Erik Reinhard, editors. *Practical Parallel Rendering*. A K Peters, 2002. ISBN 156881-179-9.
- [Chen et al., 1991] Shenchang Eric Chen, Holly E. Rushmeier, Gavin Miller, and Douglass Turner. A Progressive Multi-Pass Method for Global Illumination. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, volume 25, pages 165–174, July 1991.
- [Cleary, 1979] John Gerald Cleary. Analysis of an Algorithm for Finding Nearest Neighbors in Euclidean Space. *ACM Transactions on Mathematical Software (TOMS)*, 5(2):183–192, 1979.
- [Cook et al., 1984] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed Ray Tracing. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 137–145, July 1984.
- [Dally et al., 2002] William J. Dally, Patrick Hanrahan, Mattan Erez, Timothy J. Knight, Francois Labont, Jung-Ho Ahn, Nuwan Jayasena, Ujval J. Kapasi, Abhishek Das, Jayanth Gummaraju, and Ian Buck. Merrimac: Supercomputing with Streams. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Computer Society Press, 2002.
- [Delany, 1988] H. C. Delany. Ray tracing on a connection machine. In *Proceedings of the 2nd international conference on Supercomputing*, pages 659–667. ACM Press, 1988.

- [DeMarle et al., 2003] D.E. DeMarle, S.G. Parker, M. Hartner, C. Gribble, and C.D. Hansen. Distributed Interactive Ray Tracing for Large Volume Visualization. In *IEEE Symposium on Parallel Visualization and Graphics*, pages 87–94, October 2003.
- [Doggett, 2003] Michael Doggett. Displacement Mapping. Game Developers Conference, 2003.
www.gdconf.com/archives/2003/Doggett_Michael.pdf.
- [Fujimoto et al., 1986] Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata. ARTS: Accelerated Ray-Tracing System. *IEEE Computer Graphics & Applications*, pages 16–26, April 1986.
- [Glassner, 1984] Andrew S. Glassner. Space Subdivision For Fast Ray Tracing. *IEEE Computer Graphics & Applications*, 4(10):15–22, October 1984.
- [Glassner, 1989] Andrew S. Glassner, editor. *An Introduction to Ray Tracing*. Morgan Kaufmann, 1989. ISBN 0122861604.
- [Goodnight et al., 2003] Nolan Goodnight, Cliff Woolley, Gregory Lewin, David Luebke, and Greg Humphreys. A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware. In *Graphics Hardware 2003*, pages 102–111, July 2003.
- [Goral et al., 1984] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modelling the Interaction of Light Between Diffuse Surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 213–222, July 1984.

- [Gordon et al., 2002] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *10th international conference on architectural support for programming languages and operating systems*, pages 291–303. ACM Press, October 2002.
- [GPGPU, 2003] GPGPU Website, 2003. www.gpgpu.org.
- [Günther et al., 1995] T. Günther, C. Poliwoda, C. Reinhart, J. Hesser, R. Männer, H.-P. Meinzer, and H.-J. Baur. VIRIM: A massively parallel processor for real-time volume visualization in medicine. *Computers & Graphics*, 19(5):705–710, September 1995.
- [Hall, 1999] Daniel Hall. The AR250: A New Architecture For Ray Traced Rendering. 1999 SIGGRAPH / Eurographics Workshop On Graphics Hardware - Hot 3D Session 2, 1999.
http://graphicshardware.org/previous/www_1999/presentations/ar250/index.htm.
- [Hall, 2001] Daniel Hall. The AR350: Today's Ray Trace Rendering Processor. 2001 SIGGRAPH / Eurographics Workshop On Graphics Hardware - Hot 3D Session 1, 2001.
http://graphicshardware.org/previous/www_2001/presentations/Hot3D_Daniel_Hall.pdf.
- [Harris et al., 2002] Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual simulation on graphics hardware. In

- Proceedings of the conference on Graphics hardware 2002*, pages 109–118. Eurographics Association, 2002.
- [Havran et al., 2000] Vlastimil Havran, Jan Prikryl, and Werner Purgathofer. Statistical Comparison of Ray-Shooting Efficiency Schemes. Technical Report TR-186-2-00-14, Institute of Computer Graphics, Vienna University of Technology, 2000.
- [Hennessy and Patterson, 2002] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2002. ISBN 1558605967.
- [Igehy et al., 1998] Homan Igehy, Matthew Eldridge, and Kekoa Proudfoot. Prefetching in a texture cache architecture. In *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 133–142, August 1998.
- [Intel, 2004] Intel Corporation. *IA-32 Intel Architecture Software Developers Manual*, 2004.
<http://www.intel.com/design/Pentium4/manuals/24547012.pdf>.
- [Jensen, 1996] Henrik Wann Jensen. Global Illumination using Photon Maps. In *Eurographics Rendering Workshop 1996*, pages 21–30, June 1996.
- [Jensen, 2001] Henrik Wann Jensen. *Realistic Image Synthesis using Photon Mapping*. A K Peters, 2001. ISBN 1568811470.
- [Kajiya, 1986] James T. Kajiya. The Rendering Equation. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, volume 20, pages 143–150, August 1986.

- [Kapasi et al., 2000] Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, and Brucek Khailany. Efficient conditional operations for data-parallel architectures. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 159–170. ACM Press, 2000.
- [Khailany et al., 2000] Brucek Khailany, William J. Dally, Scott Rixner, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, and Brian Towles. IMAGINE: Signal and Image Processing Using Streams. In *Hot Chips 12*. IEEE Computer Society Press, 2000.
- [Kirk, 2001] David Kirk. GeForce3 Architecture Overview, 2001. <http://developer.nvidia.com/docs/I0/1271/ATT/GF3ArchitectureOverview.ppt>.
- [Knittel and Straßer, 1997] Günter Knittel and Wolfgang Straßer. VIZARD: Visualization Accelerator for Realtime Display. In *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 139–147, August 1997.
- [Krüger and Westermann, 2003] Jens Krüger and Rüdiger Westermann. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. *ACM Transactions on Graphics*, 22(3):908–916, July 2003.
- [Labonte et al., 2004] Francois Labonte, Mark Horowitz, and Ian Buck. An Evaluation of Graphics Processors as Stream Co-processors. In *Submitted to ISCA 2004*, 2004.
- [Larsen and McAllister, 2001] E. Scott Larsen and David McAllister. Fast Matrix Multiplies using Graphics Hardware. In *Supercomputing 2001*, page 55, 2001.

- [Levy, 2002] Markus Levy. Tying Up a MIPS32 Processor With Threads. *Microprocessor Report*, pages 34–37, November 2002.
- [Ma and McCool, 2002] Vincent C. H. Ma and Michael D. McCool. Low latency photon mapping using block hashing. In *Proceedings of the conference on Graphics hardware 2002*, pages 89–99. Eurographics Association, 2002.
- [Mai et al., 2000] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart Memories: a modular reconfigurable architecture. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 161–171. ACM Press, 2000.
- [Mark et al., 2003] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Transactions on Graphics*, 22(3):896–907, July 2003.
- [Marr et al., 2002] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1), February 2002.
- [Microsoft, 2003] Microsoft Corporation. *DirectX Home Page*, 2003. <http://www.microsoft.com/directx/>.
- [Möller and Trumbore, 1997] Tomas Möller and Ben Trumbore. Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

- [Muuss, 1995] Michael John Muuss. Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models. In *Proceedings of BRL-CAD Symposium*, June 1995.
- [Nishimura et al., 1983] Hitoshi Nishimura, Hiroshi Ohno, Toru Kawata, Isao Shirakawa, and Koichi Omura. Links-1 - a parallel pipelined multimicrocomputer system for image creation. In *Proceedings of the 10th annual international symposium on Computer architecture*, pages 387–394. IEEE Computer Society Press, 1983.
- [Nishita and Nakamae, 1985] Tomoyuki Nishita and Eihachiro Nakamae. Continuous Tone Representation of Three-Dimensional Objects Taking Account of Shadows and Interreflection. In *Computer Graphics (Proceedings of SIGGRAPH 85)*, volume 19, pages 23–30, July 1985.
- [NVIDIA, 2002] NVIDIA Corporation. *NV_Vertex_Program Extension*, 2002. http://oss.sgi.com/projects/ogl-sample/registry/NV/vertex_program.txt.
- [NVIDIA, 2003a] NVIDIA Corporation. *GeForce FX 5900 Product Web Site*, 2003. http://nvidia.com/page/fx_5900.html.
- [NVIDIA, 2003b] NVIDIA Corporation. *NV_fragment_program Extension*, 2003. http://oss.sgi.com/projects/ogl-sample/registry/NV/fragment_program.txt.
- [NVIDIA, 2003c] NVIDIA Corporation. *NV_Occlusion_Query Extension*, 2003.

http://oss.sgi.com/projects/ogl-sample/registry/NV/occlusion_query.txt.

- [Parker et al., 1998] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. Interactive Ray Tracing for Isosurface Rendering. In *IEEE Visualization '98*, pages 233–238, October 1998.
- [Parker et al., 1999a] Steven Parker, William Martin, Peter-Pike J. Sloan, Peter S. Shirley, Brian Smits, and Charles Hansen. Interactive Ray Tracing. In *1999 ACM Symposium on Interactive 3D Graphics*, pages 119–126, April 1999.
- [Parker et al., 1999b] Steven Parker, Michael Parker, Yarden Livnat, Peter-Pike Sloan, Charles Hansen, and Peter Shirley. Interactive Ray Tracing for Volume Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):238–250, July 1999.
- [Percy et al., 2000] Mark S. Percy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive Multi-Pass Programmable Shading. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 425–432, July 2000.
- [Pfister et al., 1999] Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler. The VolumePro Real-Time Ray-casting System. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 251–260, August 1999.
- [Purcell et al., 2002] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray Tracing on Programmable

- Graphics Hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).
- [Purcell et al., 2003] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon Mapping on Programmable Graphics Hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.
- [Reinhard et al., 2000] Erik Reinhard, Brian Smits, and Chuck Hansen. Dynamic Acceleration Structures for Interactive Ray Tracing. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 299–306, June 2000.
- [Rubin and Whitted, 1980] Steven M. Rubin and J. Turner Whitted. A 3-Dimensional Representation for Fast Rendering of Complex Scenes. In *Computer Graphics (Proceedings of SIGGRAPH 80)*, volume 14, pages 110–116, July 1980.
- [Sankaralingam et al., 2003] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 422–433. ACM Press, 2003.

- [Schmittler et al., 2002] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR: a hardware architecture for ray tracing. In *Proceedings of the conference on Graphics hardware 2002*, pages 27–36. Eurographics Association, 2002.
- [Suykens and Willens, 2000] Frank Suykens and Yves Willens. Density Control for Photon Maps. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 23–34, June 2000.
- [Taylor et al., 2002] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrati, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpfen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22:25–35, March 2002.
- [Torborg and Kajiya, 1996] Jay Torborg and Jim Kajiya. Talisman: Commodity Real-time 3D Graphics for the PC. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 353–364, August 1996.
- [Tullsen et al., 1995] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 392–403. ACM Press, 1995.
- [Waingold et al., 1997] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee,

- Jank Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, 30:86–93, September 1997.
- [Wald et al., 2001] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001.
- [Wald et al., 2002] Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. Interactive Global Illumination using Fast Ray Tracing. In *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering*, pages 15–24, 2002.
- [Wald et al., 2003] Ingo Wald, Timothy J. Purcell, Jörg Schmittler, Carsten Benthin, and Philipp Slusallek. Realtime Ray Tracing and its use for Interactive Global Illumination. In *Eurographics State of the Art Reports*, 2003.
- [Wald, 2004] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Universität des Saarlandes, January 2004.
- [Ward and Heckbert, 1992] Gregory J. Ward and Paul Heckbert. Irradiance Gradients. In *Rendering Techniques 1992: 3rd Eurographics Workshop on Rendering*, pages 85–98, May 1992.

[Whitted, 1980]

Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.