INCREASING RENDERING PERFORMANCE OF GRAPHICS HARDWARE

Justin Hensley

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2007

ABSTRACT

JUSTIN HENSLEY: Increasing Rendering Performance of Graphics Hardware
(Under the direction of Anselmo Lastra and Montek Singh)

Graphics Processing Unit (GPU) performance is increasing faster than central processing unit (CPU) performance. This growth is driven by performance improvements that can be divided into the following three categories: algorithmic improvements, architectural improvements, and circuit-level improvements. In this dissertation I present techniques that improve the rendering performance of graphics hardware measured in speed, power consumption or image quality in each of these three areas.

At the algorithmic level, I introduce a method for using graphics hardware to rapidly and efficiently generate *summed-area tables*, which are data structures that hold pre-computed two-dimensional integrals of subsets of a given image, and present several novel rendering techniques that take advantage of summed-area tables to produce dynamic, high-quality images at interactive frame rates. These techniques improve the visual quality of images rendered on current commodity GPUs without requiring modifications to the underlying hardware or architecture.

At the architectural level, I propose modifications to the architecture of current GPUs that add conditional streaming capabilities. I describe a novel GPU-based ray-tracing algorithm that takes advantage of conditional output streams to reduce the memory bandwidth requirements by over an order of magnitude times when compared to previous techniques.

At the circuit level, I propose a *compute-on-demand* paradigm for the design of high-speed and energy-efficient graphics components. The goal of the compute-on-demand paradigm is to only perform computation at the bit-level when needed. The compute-on-demand paradigm exploits the data-dependent nature of computation, and thereby obtains speed and energy improvements by optimizing designs for the common case. This approach is illustrated with the design of a high-speed Z-comparator that is implemented using asynchronous logic. Asynchronous or “clockless” circuits were chosen for my implementations since they allow for data-dependent completion times and reduced power consumption by disabling inactive components. The resulting circuit-level implementation runs over 1.5 times faster while on dissipating 25% the energy of a comparable synchronous comparator for the average case.

Also at the circuit-level, I introduce a novel implementation of *counterflow pipelining*, which allows two streams of data to flow in opposite directions within the same pipeline without the need for complex arbitration. The advantages of this implementation are demonstrated by the design of a high-speed asynchronous Booth multiplier. While both the comparator and the multiplier are useful components of a graphics pipeline, the objective of this work was to propose the new design paradigm as a promising alternative to current graphics hardware design practices.

In memory of my brother, Jeff Hensley. As children we didn't always get along, and I regret not being able to tell you how much you meant to me. I love you and I miss you.

ACKNOWLEDGMENTS

Ever since I can remember, I have always wanted to get my Ph.D. As a young child, I would often inform people that I was going to get my doctorate in aeronautical engineering. Since you are currently reading this dissertation, it should be fairly clear to you that I did not in fact get my Ph.D. in aeronautical engineering. If you are expecting a treatise on airplane design, you are gravely mistaken. Despite this minor change in plans, I have always known that I would be going to college, even from an early age. I believe that I have my parents to thank for this. There was never a question about whether I would go to college. My parents always supported me, and always wanted me to do my best, whatever it was that I chose to do. For their unwavering support, I would like to thank my parents.

My journey through college has been filled with some of the happiest times of my life, and some of the saddest times of my life. As an undergraduate, I was fortunate enough to live in a small dorm at the University of California at Davis. It is there that I met Ami, the love of my life, an amazing woman, who, when I asked her to marry me, said yes. She was willing to transfer universities in the middle of her veterinary medical program so that we could be together after the we got married. I know that it was difficult for Ami to leave her friends at Purdue to attend North Carolina State University, and for her sacrifice I am eternally grateful, since I know that I would not have been able to finish this dissertation without her support. For being there when it mattered most for me, thank you Ami. [1]

As an undergraduate, I originally started out in the biosystems engineering program at UC Davis. It did not take long, one quarter in fact, for me to decide that biosystem engineering was not for me. Suffice to say, the course that led to my decision involved learning about mass producing bean sprouts, and I have always known that I did ***not*** want to work in agriculture. Sorry dad. After filling out the requisite forms, I was soon majoring in computer science and

[1]note to reader: despite my wife's protestations, I did not remove the preceding paragraph

electrical engineering. When I got my first introduction to computer graphics, I was well on my way to becoming a computer architect, but after taking "Introduction to Computer Graphics" with Professor Ken Joy, I knew that I wanted to work with computer graphics. I would like to thank Professor Joy for introducing me to the field of computer graphics, and for his encouragement to attend UNC for my doctorate degree.

At UNC, I initially joined the "Office of the Future" research group, and began working on what was eventually named PixelFlex. As a member of OOTF I was advised by Herman Towles and Henry Fuchs. I am grateful for the guidance they gave me during my first two years at UNC, but as fate would have it one tragic day I received a phone call that my brother had died. In a single moment, my life had changed forever. While I know it is trite to say that, there are no other ways I can describe how I felt on that day and how I still feel.

My co-advisors Montek Singh and Anselmo Lastra have been invaluable to me. All of the research I have done since leaving the PixelFlex project has been under their supervision. For the past four years, the endpoint of my research has not always been sight; Anselmo and Montek have been there to help me through the rough patches. For their support and guidance, I would like to thank my co-advisors.

To the rest of my committee, Leonard, Mary, and Steve: I would like to thank you for your advice and guidance. To my friends at UNC, thank you for the support and friendship you have given me. Finally, I would also like to thank the many foosball players in the department who helped me keep my sanity.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

2D	Two-Dimensional
3D	Three-Dimensional
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
GPU	Graphics Processing Unit
HC	High-Capacity
HC-CF	High-Capacity Counterflow
HDR	High Dynamic Range
LDR	Low Dynamic Range
PC	Pre-charge

CHAPTER 1

Introduction

Graphics Processing Unit (Graphics Processing Unit (GPU)) performance is increasing faster than Central Processing Unit (Central Processing Unit (CPU)) performance, and is reported by some to be growing at a “Super-Moore’s Law” rate (Govindaraju et al., 2006). This growth is driven by performance improvements that can be divided into the following three categories.

- ***Algorithmic:*** Algorithmic improvements are often initially implemented at the application level, using the current capabilities of current hardware. The relatively recent introduction of programmable pipelines to commodity GPUs enables a wide variety of algorithms to be implemented without architectural changes.

- ***Architectural:*** There are situations where a new algorithm lends itself to an efficient, direct implementation in hardware that would require only minimal changes to the GPUs architecture. For example, various environment mapping techniques initially required the application developer to handle texture coordinate generation, whereas modern GPUs are able to automatically transform normal and reflection directions into texture coordinates.

  At the architectural level, enhancements designed to introduce new functionality or increase performance should be made without impacting the performance of legacy applications. This is an especially important property for devices destined for established commodity markets. For better or worse, the driving forces behind the commodity GPU market are video games, and it would be considered unacceptable for a newly released GPU to run slower on the current “game-de-jour” than the previous generation of products.

- ***Circuit:*** At the circuit level, it is possible to make dramatic changes to the underlying circuitry without modifying the GPU's architecture or the application programmer interface. Some of the possible benefits of circuit-level modifications include faster computation, lower energy-consumption, or improved yields when fabricating integrated circuits. For example, some modern hardware has moved from using standard-cell implementations to custom ASICs without requiring changes to the overall architecture. Also, over the lifetime of a specific architecture, multiple different fabrication processes might be used — e.g. a device might initially be implemented in a 90 nm process, but later be fabricated in a 65 nm process as the architecture and fabrication processes become more mature.

Performance improvements can be measured in several ways. In this work I use metrics related to the visual image quality, speed of computation, and energy-efficiency. The following sections give more detail about the contributions made in these three areas.

1.1 The Algorithmic Axis: Improving Rendering Quality with GPU-Based Summed-Area Tables and Extensions

At the algorithmic level, I will discuss several novel techniques that are capable of improving the visual quality of images rendered on current commodity GPUs without requiring modifications to the underlying hardware or architecture. In particular, I will describe a method for using graphics hardware to rapidly and efficiently generate *summed-area tables*, which are data structures that hold pre-computed integrals of a given image. Additionally, I will describe extensions to summed area tables that reduce their precision requirements and present multiple rendering techniques that take advantage of summed-area tables to produce high-quality images in real-time. My algorithm is able to compute a 256x256 summed-area table in less than 2ms, and render complex scenes, such as the one shown in Figure 1.2(a), with interactive glossy reflections at over 60 frames per second on relatively old hardware such as a Radeon X800XT PE.

The following quick tutorial is intended for readers unfamiliar with texture mapping and summed-area tables. Readers familiar with the topics can skip to Section 1.1.2.

1.1.1 Background

Texture mapping, a technique that is ubiquitous in computer graphics, is a process that enables simple rendered surfaces to appear to have complex surface properties. In its simplest form, texture mapping can be thought of as an image being used as a decal on the surface of an object. Introduced by Catmull (Catmull, 1974), texture mapping is a process whereby positions in three-space are mapped to an n-dimensional, parametric space, which allows for easy sampling of data stored in n-dimensional arrays such as images (2D) and volumetric data (3D).

As with any sampling technique, it is often necessary to filter or interpolate the data sampled from a texture map to attain visually pleasing images. In practice, it is cost-prohibitive to directly interpolate the texture map with anything more complex than linear interpolation. *Mipmaps* (Williams, 1983) address this issue by pre-computing multiple images, which are scaled-down versions of the original input image, and then performing trilinear interpolation between two levels of the mipmap.

First introduced by Crow (Crow, 1984), summed-area tables enable more general texture filtering than is possible with the commonly-used mipmapping technique. In particular, summed-area tables are able to more accurately reflect the data actually inside a box filter kernel at a specific location, whereas a nearby samples from a mipmap are limited to the average of a fixed set a pixels. When using a mipmap to perform more generalized filtering, this becomes apparent as visual defects in the filtered image. Although introduced at roughly the same time as mipmaps, summed-area tables have a more stringent precision requirement than an equivalently sized mipmap. Partly due to this limitation of summed-area tables, mipmapping became a more attractive technique to implement in early GPUs, and summed-area tables fell out of favor in real-time rendering. But, summed-area tables remained a useful technique in offline rendering applications such as the renders used in the film industry. The relatively recent availability of floating-point capable datapaths, and the associated increase in the available computational precision on commodity GPUs has facilitated a resurrection in

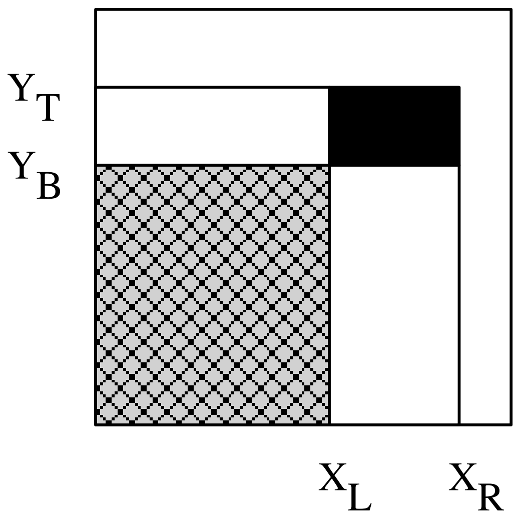
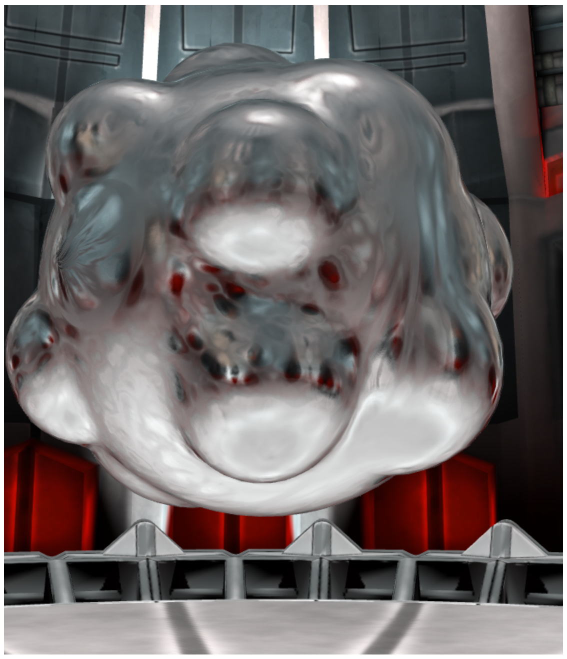


Figure 1.1: (after [Crow84]) An entry in the summed-area table holds the sum of the values from the lower left corner of the image to the current location. To compute the sum of the dark rectangular region, evaluate $T[X_R, Y_T] - T[X_R, Y_B] - T[X_L, Y_T] + T[X_L, Y_B]$ where T is the value of the entry at (x, y).

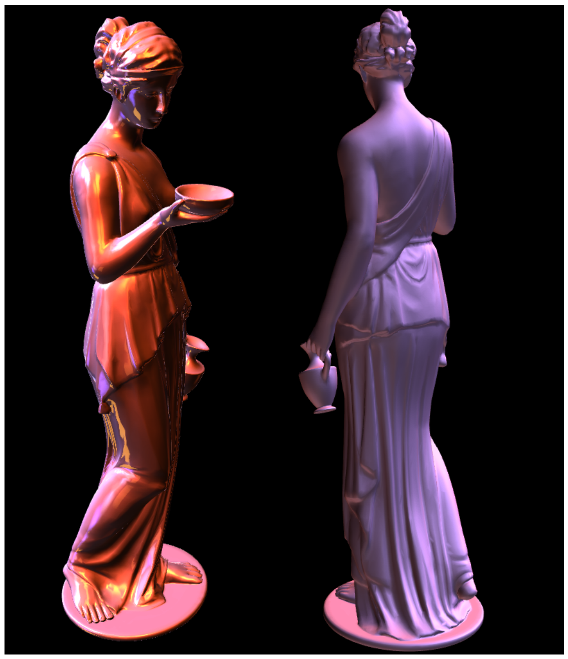
the use of summed-area tables in real-time graphics, despite their high precision requirements.

For an image, a summed-area table is a two-dimensional array where each entry in the array stores the discrete integral for each rectangular sub-image of the input image. Summed-area tables enable at a fixed computational cost the rapid calculation of the sum of the pixel values in an arbitrarily sized, axis-aligned rectangle. Figure 1.1 illustrates how a summed-area table is used to compute the sum of the values of pixels spanning a rectangular region. To find the integral of the values in the dark rectangle, we begin with the pre-computed integral from (0,0) to (x$_R$, y$_T$). We subtract the integrals of the rectangles (0, 0) to (x$_R$, y$_B$) and (0, 0) to (x$_L$, y$_T$). The integral of the hatched box is then added to compensate for having been subtracted twice. Finally, the average value of a group of pixels can be calculated by dividing the sum by the area.

Once generated, a summed-area table provides a means to evaluate a spatially varying box filter with a constant number of memory accesses. For example, the average value of a 100x100 pixel region of an image could be computed with only four accesses to a summed-area table, whereas if the region were to be directly filtered, it would take 10,000 accesses to the original image. Summed-area tables were later extended by Heckbert (Heckbert, 1986) to



(a) Glossy environmental reflections



(b) Approximate HDR image-based lighting

Figure 1.2: Image (a) shows an object rendered in real-time with an environment map filtered by a spatially varying filter. Image (b) shows A real-time rendering of a statue of Hebe, the Greek goddess of youth. All lighting calculations are only performed by sampling summed-area tables computed from the Grace Cathedral lightprobe (lightprobe courtesy of Paul Debevec). The left image of Hebe shows an approximation to the Phong BRDF, and the right image shows an approximation to a diffuse BRDF.

handle complex filter functions by taking advantage of the properties of convolution. These extensions will be described in more detail in Chapter 2.

1.1.2 Contributions

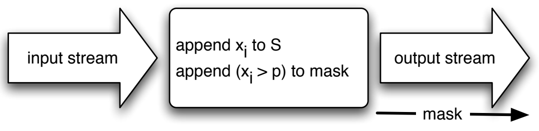
In this dissertation I present a method to rapidly generate summed-area tables that is efficient enough to allow multiple summed-area tables to be generated every frame while maintaining interactive frame rates. I extend the traditional summed-area table algorithm to reduce their precision requirements, which enables summed-area tables to easily be used on graphics hardware with relatively limited-precision, or alternatively, larger summed-area tables to be generated with visual artifacts being introduced. I demonstrate the applicability of spatially varying filters for real-time, interactive computer graphics through several different applications. Some example applications are the interactive rendering of dynamic glossy reflections

(Hensley et al., 2005), and dynamic image-based lighting (Hensley et al., 2006). Figure 1.2(a) shows an image captured from a real-time application that is rendering an object with spatially varying glossy environmental reflections, while Figure 1.2(b) shows an image captured from a real-time application that is rendering a statue illuminated using an approximate image-based lighting technique. Using the metrics described earlier, these contributions improve rendering performance in two key ways: (i) improving the visual quality of rendered images, and (ii) increasing the speed of computing summed-area tables.

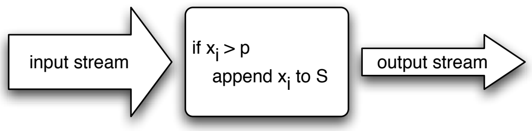
1.2 The Architectural Axis: Extending Graphics Architectures with Conditional Output Streams Increases Rendering Capabilities

As with most data-parallel architectures, conditional operations are difficult to handle efficiently on graphics processors. This inefficiency arises because data-parallel architectures typically execute the same operation on multiple elements of data at the same time. This mode of operation is often referred to as SIMD — Single Instruction, Multiple Data. Since all the elements are operated on by the same instruction, data that requires separate branch paths forces redundant execution. In the worst case each element requires a different path through the code, forcing sequential operation instead of parallel operation, which is clearly undesirable.

GPUs are just now gaining conditional operations but because of a GPU's SIMD nature, they are limited to either consecutive arithmetic operations with no branching, or the inefficient execution of all of the possible branch outcomes for all pixels that executed together. As an example, ATI's X1800 shader architecture (ATI Technologies, 2005) processes sixteen pixels in parallel. The X1800 is optimized to take advantage of the situation where all pixels branch the same way, otherwise both paths of the branch would require execution (although, this optimization only helps where there is large amount of locality to the branching pattern). For example, consider the following shader pseudo-code:



(a) Without conditional streams



(b) With conditional streams

Figure 1.3: Conditional operations with streaming architectures. Figure (a) shows how a mask must be used to prevent downstream compute kernels from operating on invalid data. The output stream is the same size as the input stream, and the mask has the same number of elements as the input stream. Figure (b) shows the same simple conditional operation with conditional output streams. In this situation, the output stream is only as large as it needs to be, no additional mask vector is needed, and processor utilization of downstream kernels is increased.

```
if pixel not in shadow then
        computeLighting()
else
        computeShadow()
end if
```

Assume that it takes X amount of time to execute *computeLighting()* and Y amount of time to execute *computeShadow()* for a *batch* of sixteen pixels. In the situation where all sixteen pixels in a batch are all not in shadow, then it will take X amount for time to process the pixels. Alternatively, if all the pixels are shadowed it will take Y amount of time to process the pixels. In the unfortunate situation where some of the pixels are shadowed and some are not, $X + Y$ amount of time must be spent to process the sixteen pixels.

As with any area of research it is often useful to examine how other researchers dealt with this issue in related fields. GPU architectures are sometimes referred to as *stream architectures*. Researchers have introduced the concept of conditional streams (Kapasi et al., 2000), which augment traditional streaming processors with the capability to conditionally read from input streams, and conditionally write to output streams.

1.2.1 Conditional Streams

Figure 1.3(a) shows how a simple conditional operation would be handled by a streaming architecture without conditional streams. The goal of the compute kernel is to filter all the values that are below p. Since there must be a 1-to-1 correspondence between the input and output streams, all of the input values must be copied to the output. A mask is also generated to inform downstream kernels which elements of the output stream are valid. Since the mask can disable processing of some elements, the stream processors will not be fully utilized.

Figure 1.3(b) shows how the same conditional operation would be handled with conditional output streams. In this situation, the kernel can conditionally write values that are greater than p to the output stream. Since the output stream is only as large as it needs to be, the downstream kernels will fully utilize the stream processor since the output stream is densely packed.

Conditional output streams enable the efficient implementation of if-else style branches, and have the advantage that they can be implemented with a negligible performance penalty, only several additional gate delays. Conditional streams allow for a dramatic increase in processor utilization, hence an increase in processing efficiency, and memory access coherence, and thereby increase in performance. Algorithms such as sorting, boosted Haar cascades (Viola and Jones, 2001) which are useful for object detection, and particle system simulations, in additional to many others both graphics and non-graphics related, would execute more efficiently if the GPU supported conditional streams.

1.2.2 Contributions

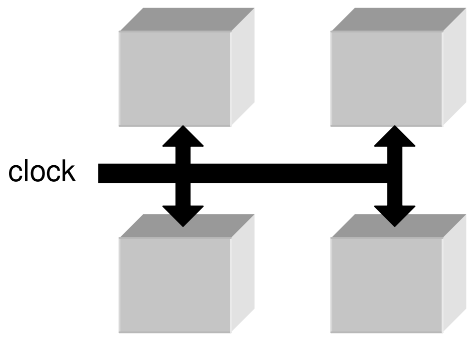
Conditional streams provide the opportunity to increase the rendering performance of graphics hardware by enabling the efficient implementation of high-quality rendering algorithms, and by making more generalized computations efficient. In this dissertation, I discuss augmenting the the architectures of graphics processors with conditional streams. A change of this type has been suggested by Popa (Popa, 2004) who proposed the use of conditional streams for compiling data-dependent control flows on SIMD GPUs. Additionally, while Direct3D 10 has introduced the concept of geometry shaders (Blythe, 2006), which also have a similar capability to generate compacted streams of data. Although there are no graphics chips capable of implementing this feature of geometry shaders efficiently at this time. Prior work has reported on using conditional streams in GPUs, it does not directly address the problem of ray tracing. In this dissertation, I will focus on the use of conditional output streams to accelerate a novel ray tracing algorithm. In particular, I describe a ray tracing algorithm that takes advantage of conditional streams that is able to reduce the required memory bandwidth by more than an order of magnitude when compared to previous work using GPUs for raytracing.

1.3 The Circuit Axis: Asynchronous Techniques for Improving the Efficiency and Performance of GPUs

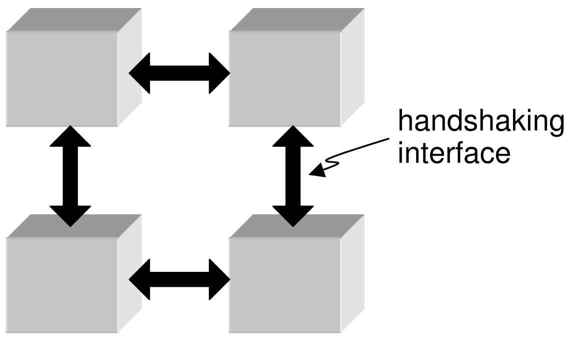
Current trends in micro-electronic design pose a challenge to synchronous systems: (i) high clock speed, (ii) large die area, (iii) handling worst-case delay in deep submicron processes (e.g. 90 *nm* and smaller), and (iv) managing large, complex designs. As a result, an alternative paradigm—*asynchronous* or "clockless" design—is becoming an increasingly attractive approach because of asynchronous logic's promise in reversing these negative trends (Berkel et al., 1999). As illustrated in Figure 1.4, instead of using global clocking, an asynchronous system uses *handshaking* between interacting components to achieve local synchronization.

Asynchronous design has potentially significant energy and performance benefits: lower energy consumption results due to elimination of the power wasted driving the clock, and by limiting switching activity to when and where needed (Berkel et al., 1999). Since local handshaking is used instead of a global clock for synchronization, asynchronous components can gain performance benefits by exploiting the data-dependency of computation completion times (Nowick et al., 1997; Rotem et al., 1999).

Since underlying circuit implementations can largely be decoupled from system level architecture, the circuit designer is basically free to use exotic techniques, such as clockless logic, while leaving the system architecture unchanged. For example, some modern CPUs, e.g. the Pentium 4, use asynchronous logic to implement their arithmetic units, while still



(a) A synchronous system, featuring centralized control



(b) An asynchronous system, with distributed control

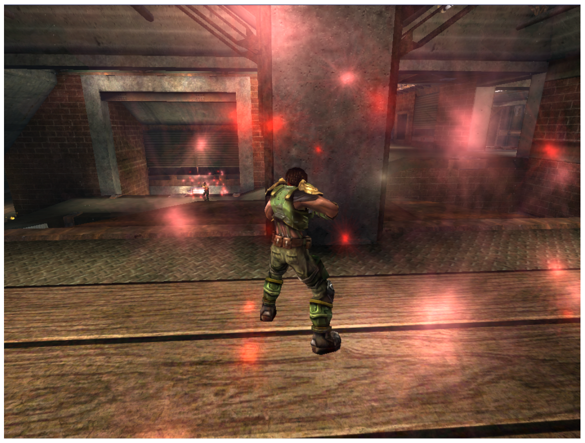
Figure 1.4: Synchronous and asynchronous system block diagrams.

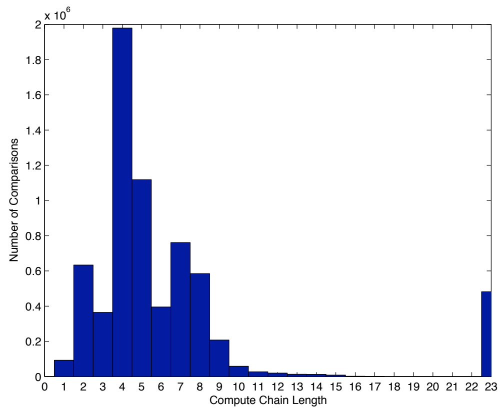
realizing a standardized architecture that appears unchanged at a high level. At the extreme, an entire micro-controller has been implemented with asynchronous circuits. This processor is a functional, drop-in replacement for standard synchronous micro-controller (Gageldonk et al., 1998).

1.3.1 Contributions

The work presented in the dissertation involving what we have termed the *circuit axis* makes use of asynchronous logic, and increases rendering performance by making graphics hardware more energy efficient, while operating faster for the average case. The dissertation introduces two novel concepts: (i) the *compute-on-demand* paradigm, whereby computation at the bit-level is on performed on an as-needed case, and (ii) a novel implementation of the counterflow pipeline architecture. In particular it extends the high capacity (HC) asynchronous pipelining style, which in turn uses dynamic logic, both of which will be briefly described below, and in more detail in Chapter 3.4.

Figure 1.5(a) shows a rendered image from the game *Unreal Tournament 2004* (Epic Games, 2004), and Figure 1.5(b) shows a histogram of the number of bits that actually need to be compared per depth comparison to render the image. In Chapter 3, I will present a novel z-comparator that is both energy-efficient and fast. The comparator gains its increased performance by taking advantage of average case performance, and introduces my *compute-on-demand* paradigm.


(a) Rendered frame



(b) Compute chain length

Figure 1.5: Image (a) shows a frame from Unreal Tournament 2004. The frame requires 6,768,766 comparisons of incoming fragments with the depth buffer. On average, only the 7.3 most significant bits are actually needed to resolve each comparison. Figure (b) shows the distribution of z-comparison compute chain length for the frame shown in (a).

Additionally, I describe an asynchronous Booth multiplier that uses a novel implementation of a counterflow architecture (Sproull et al., 1994) in a single pipeline. In a counterflow architecture, data flows in one direction and control information flows in the opposite direction. This counterflow architecture allows for shorter critical paths, and therefore higher operating speed.

While using asynchronous logic in graphics hardware would require dramatic changes at the circuit-level, it would not require (or prevent), changes at the architectural or algorithmic levels.

1.4 Thesis Statement

Using multiple techniques at the circuit, architectural, and algorithmic levels, it is possible to increase the rendering performance of graphics hardware, where rendering performance can be defined as either increasing the energy efficiency of computation, increasing the speed of computation or increasing the visual quality of rendered images.

- Efficient construction of summed-area tables on commodity graphics hardware makes possible dynamic, real-time glossy environmental reflections and dynamic real-time image-based lighting without requiring changes to the underlying architecture.

- Conditional output streams facilitate an implementation of ray tracing that reduces memory-bandwidth, and allows for high-quality, geometrically correct reflection, refraction, and shadows.

- Asynchronous design techniques, increase energy efficiency, decrease area usage, and increase throughput of basic components used in graphics pipelines.

1.5 Major Contributions

This dissertation presents research at the circuit, architectural, and algorithmic levels that improves the energy efficiency of graphics processors, increases the speed of the computations, and increases the visual quality of rendered images.

My research contributions include along the three axes are:

- **Algorithm Axis**
  - ***Efficient construction of summed-area tables:*** I describe a method using graphics hardware to rapidly generate summed-area tables that is efficient enough to allow multiple tables to be generated every frame while maintaining interactive frame rates. Several possible applications of using summed-area tables in interactive graphics are presented.
  - ***Offset summed-area tables:*** I propose a technique that alleviates the precision requirements needed in the construction and use of summed-area tables by offsetting the input image by a constant value. This method improves precision in two ways: (i) there is a 1-bit gain in precision because the sign bit now becomes useful, and (ii) the summed-area function becomes non-monotonic, and therefore the maximum value reached has a relatively lower magnitude, thereby significantly increasing precision by lowering the dynamic range needed to store a summed-area table.
  - ***Fast image-based lighting using summed-area tables:*** I present a method to rapidly generate higher-order summed-area tables — e.g., a summed-area table

of a summed-area table — that is efficient enough to allow multiple tables to be generated every frame while maintaining interactive frame rates. I demonstrate using higher order summed-area tables to approximate reflections generated using a Phong BRDF and high dynamic range environment maps.

- **Architecture Axis**
  - ***Novel ray tracing algorithm using conditional streams:*** I propose a novel streaming ray casting algorithm. The algorithm uses conditional output streams to reduce memory bandwidth and increase processor utilization when compared to previous methods. The algorithm is able to reduce memory bandwidth by over an order of magnitude compared to the most efficient method presented so far. One possible use for our proposed technique is to implement hybrid rendering algorithms that use standard z-buffering techniques to generate the first hits from the camera view, and then use ray tracing to generate geometrically correct reflections and shadows.
- **Circuit Axis**
  - ***Compute-on-demand paradigm for asynchronous circuits:*** I introduce the notion of compute-on-demand as a design principle for fast and energy-efficient graphics hardware. The key idea is to exploit the data-dependent nature of computation, and to obtain speed and energy improvements by optimizing the design for the common case, instead of assuming worst-case operation. An asynchronous or clockless circuit style is used to facilitate this paradigm. In particular, only those portions of compute blocks are activated that are actually required for a particular operation, thereby saving energy and reducing critical delays.
  - ***Novel conterflow pipeline approach:*** I propose a novel implementation of counterflow pipelining which has significant advantages compared with previous implementations; it eliminates the need for complex synchronization and arbitration required between the two distinct data streams in the original counterflow implementation. This feature allows shorter critical paths, and therefore higher

operating speed. To demonstrate my counterflow methodology, I introduce a novel multiplier organization, in which the data bits flow in one direction, and the Booth commands are piggybacked on the acknowledgments flowing in the opposite direction.

1.6 Dissertation Organization

The remainder of this dissertation is organized as follows:

Chapter 2 discusses increasing rendering performance at the algorithmic level. First, I present background information on summed-area tables, pre-filtering environment maps, and image-based lighting is presented. Next, a method to rapidly construct summed-area tables using graphics hardware is described. Then, a technique to improve the precision requirements of summed-area tables, called offset summed-area tables is presented. Example applications are then described. Next, a method to construct higher-order summed-area tables in real-time is presented. Finally, a dynamic, real-time approximate image-based lighting algorithm.

Chapter 3 describes an architectural extension to commodity graphics processors that would improve processor utilization during execution o conditional operations. The chapter begins by presenting background information on conditional output streams, and GPU-based ray tracing algorithms. Next, as an example of the benefits of conditional streams, a novel ray tracing algorithm is presented. Then the performance of the algorithm is discussed.

Chapter 4 covers techniques used to improve graphics hardware efficiency and performance at the circuit level. It begins with an overview of asynchronous logic, with a particular emphasis on the High Capacity (HC) pipelining style. Next, a novel asynchronous z-comparator is presented which introduces and illustrates the compute-on-demand paradigm. Then two asynchronous Booth multipliers are presented to demonstrate the proposed counterflow pipelining style. Finally, an experiment designed to examine the relationship between pipeline complexity and average-case performance is discussed.

Chapter 5 concludes my dissertation with a summary of my contributions and discusses possible future work.

CHAPTER 2

Increasing Rendering Performance Along the Algorithmic Axis

In this chapter, I present techniques that increase rendering quality on current commodity GPUs. In the next section background information will be presented along with related work. Next, my method for summed-area table construction will be discussed in detail. Additionally, a novel modification to standard summed-area tables, which I have term offset summed-area tables, will be presented. Finally multiple novel algorithms are presented which use summed-area tables to increase the quality of renderings. All of the algorithms presented in this chapter depend on the ability to rapidly generate multiple summed-area tables at interactive rates.

In (Kautz et al., 2000), Kautz et al. presented a method for real-time rendering of glossy reflections for static scenes. They rendered a dual-paraboloid environment map and pre-filtered it in an offline process. Instead of pre-filtering, my algorithm creates a summed-area table for each face of a dual-paraboloid map on the fly, and uses them to filter the environment map at run time. This enables real-time, interactive environmental glossy reflections for dynamic scenes.

2.1 Background

In this section, I present background information on several techniques used as the basis for the research presented in this chapter, including environment mapping, summed-area tables, and high-dynamic range images. Then I discuss several related techniques that use mipmaps instead of summed-area tables to perform image filtering.

2.1.1 Reflection and Environment Mapping

Reflection and environment mapping (Blinn and Newell, 1976) are a set of techniques that are useful for approximating reflection and refraction without having to resort to using ray tracing, which can be expensive or even impossible due to architectural limitations on current graphics hardware.

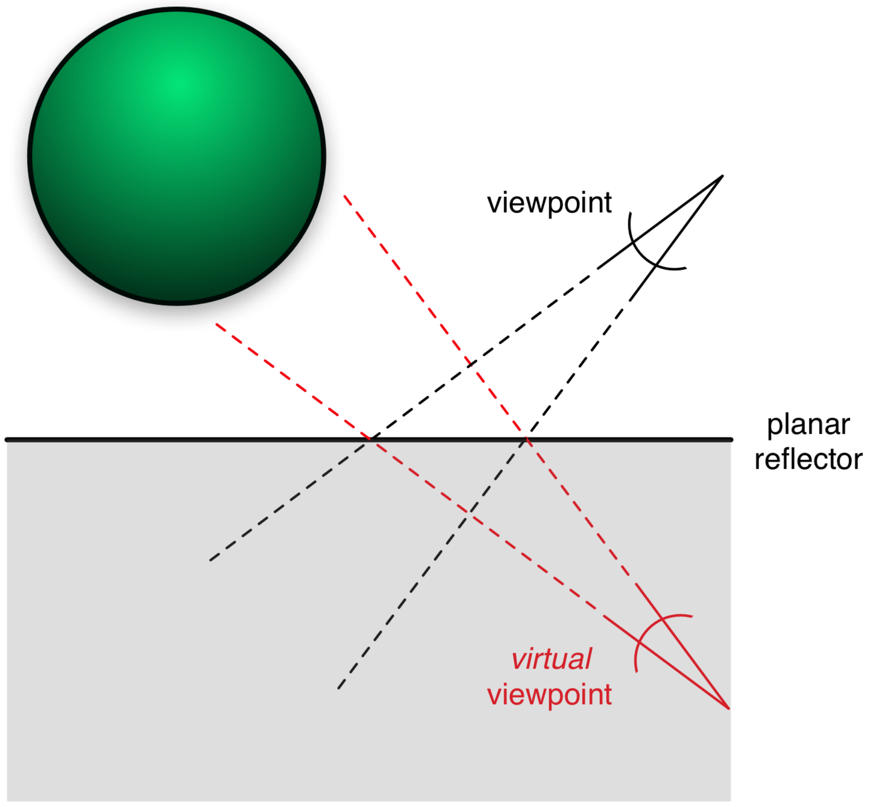


Figure 2.1: Planar reflection mapping. A virtual camera is placed opposite the planar reflector from the actual camera to generate planar reflections.

2.1.1.1 Planar Reflection Mapping

Planar mirror reflections are generated by a relatively simple process whereby the scene is rendered from the view point of a virtual camera that is located on the opposite side of the planar reflector from the actual camera (Figure 2.1). On modern graphics hardware this procedure is accomplished using additional rendering passes, and rendering the results into a texture for use in the final rendering pass. This technique has the benefit of generating physically correct reflections, but suffers from performance issues when there are a large number of planar reflectors, since the scene must be re-rendered separately for each reflector.

2.1.1.2 Cube Mapping

Environment mapping typically assumes that the reflective object, the reflector, is surrounded by a shell, such as a sphere (spherical mapping) or more commonly a cube (cube mapping), and then the environment is projected onto the surrounding shell using the center of the shell as the center of projection. For cube mapping, this process simply requires the scene to be rendered six times, once for each face of the cube. Then during the final rendering pass, the normal to the reflector and the view direction are used to compute a reflection direction which is used to lookup values from the *cubemap*. Modern graphics chips include support for automatically sampling the correct cube map face given a direction in 3-space.

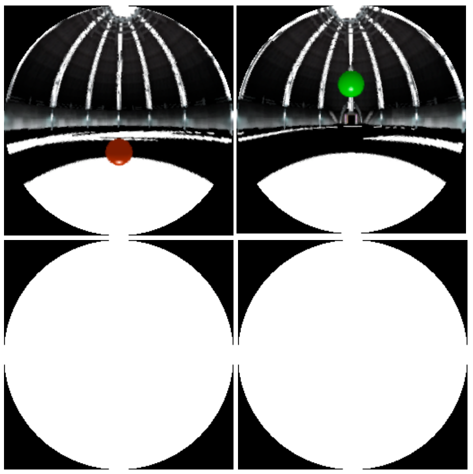
Most environment mapping techniques make the simplifying assumption that the reflected environment is far from the reflector. When the reflector is relatively close to the environment, or if the center of the reflector is not near the center of the environment map, geometric distortions may be visible since the map has been generated with the center of an enclosing surface as the center of projection.

2.1.1.3 Dual-Paraboloid Mapping

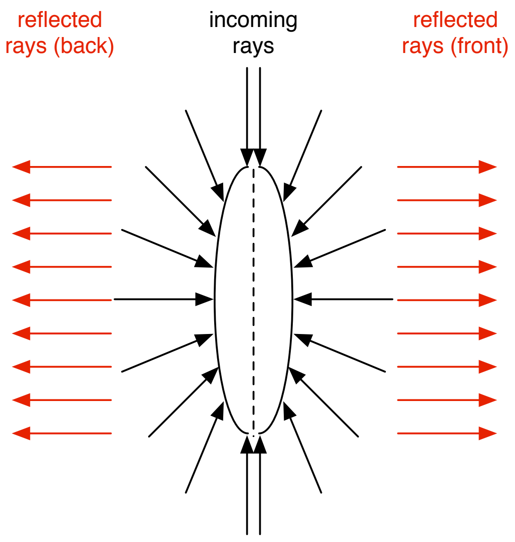
Dual-paraboloid environment mapping (Heidrich and Seidel, 1998) is a technique that stores an environment map in two textures, each of which stores half of the environment as reflected by a parabolic mirror (see Figure 2.2). Typically the alpha channel of each dual-paraboloid map face stores a circular mask that indicates whether a pixel contains relevant data.

A direct mapping from a cube map to a dual-paraboloid map is given by Blythe (Blythe, 1999). The following High Level Shader Language (HLSL) code perfoms the mapping for the front face of the dual-paraboloid map.

```
samplerCube tCube; // the cubemap to covert to a dual-paraboloid map
```



(a) Example dual-paraboloid map. The left column shows the *front* map, and the right column shows the *back* map.



(b) Diagram of a dual-paraboloid map projection. The reflected rays are parallel to each other.

Figure 2.2: Dual-paraboloid maps.

```
float 4 main (float2 inUV : TEXCOORD0 /* quad texcoord from 0..1 */ )
    : COLOR
{
    float2 uv = 2.0 * inUV - 1.0;  // scale and bias into -1..1 range
    float3 dir;                    // lookup direction for cubemap

    // convert front dual-paraboloid face texture coordinate to 3D
    // direction
    dir.x = 2.0*uv.x;
    dir.y = 2.0*uv.y;
    dir.z = -1.0 + dot( uv, uv );
    dir /= (dot( uv, uv ) + 1.0);

    // compute circular mask for alpha channel
    float alpha = (dot( uv, uv ) < 1.0 ) ? 1.0 : 0.0;
    // look up cubemap texture sample and multiply with alpha mask
    return float4(texCUBE( texCUBE(tCube, dir).rgb, 1.0) * alpha;
}
```

A similar shader is used to compute the values for the back map. An alternative to converting 2D texture coordinates to a 3D direction vector in the shader is to pre-compute the conversion and store it in a lookup texture. Then at run-time, an indirect texture lookup is performed during generation of a dual-paraboloid map.

2.1.2 High-Dynamic Range Images

Images in computer graphics are typically represented using low-dynamic range (LDR) values, since the archetypal display can display only a relatively limited dynamic range (the intensity range between black and white). Current commodity LCD displays typically only have a dynamic range on the order of 1000 to 1, whereas real world data will have a dynamic range several orders of magnitude larger.

Instead of using an integer value to represent the intensity of each color channel of an image, high-dynamic range (HDR) images typically use floating point values to represent the intensity of each channel, and allows for more realistic lighting and rendering effects. The availability of commodity graphics processors with single precision floating point native data paths, makes processing HDR data an attractive proposition that can dramatically increase the quality of rendered imagery. Given the limited capabilities of current display technologies, HDR data does have to be mapped to a LDR data via a process called *tone mapping.* Tone mapping is a separate topic from the material covered in this chapter, and will not be discussed further in this thesis.

2.1.3 Image-Based Lighting

Image-based lighting (IBL), a technique introduced by Debevec (Debevec, 1998; Debevec, 2002), enables synthetic objects to be rendered into *real* scenes with realistic lighting. This dramatically increases the perceived realism of the synthetic objects. As presented by Debevec, IBL uses a *lightprobe*, which is simply an HDR environment map of real world data.

Lightprobes are typically captured from multiple images taken of a mirrored sphere, which allows radiance data to be captured in all directions. Once a lightprobe is generated, ray tracing is used to compute the incident illumination from the image-based lighting environment on each of the synthetic surfaces in the rendered scene (see Figure 2.3).

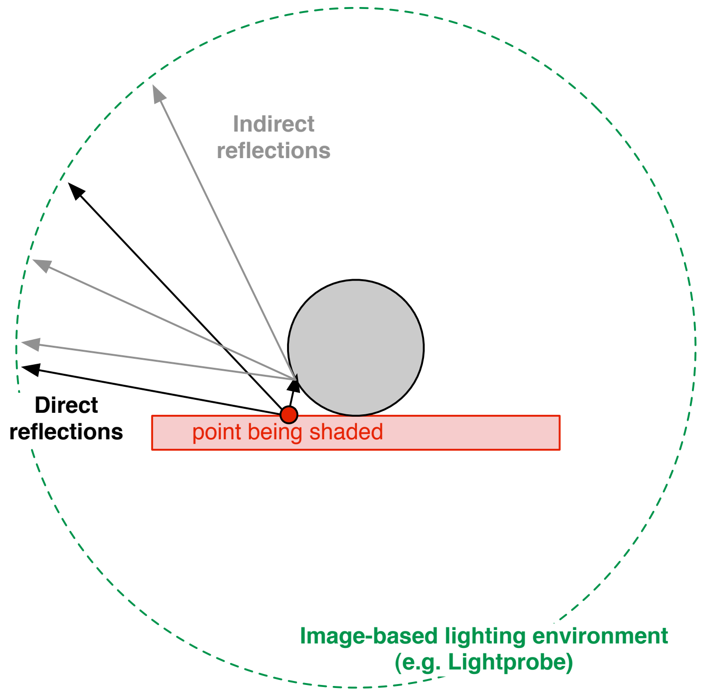


Figure 2.3: Image based lighting.

Assuming that normalized coordinates are used to access the lightprobe, directions in 3-space can be generated by rotating an normalized vector pointing in the direction of the -z-axis by θ and ϕ, where θ and ϕ are given by

$$\begin{aligned} u &= [-1,1], v = [-1,1] \\ \theta &= arctan(v/u) \\ \phi &= \pi * \sqrt{(u^2 + v^2)} \end{aligned}$$

Using these relationships, it is a relatively simple task to generate HDR cube and dual-paraboloid environment maps from real world data.

When a local-lighting model is used, an approximation of IBL is to simply convolve a synthetic object's BRDF with a lightprobe, thereby approximating the lighting that the synthetic object would have received had it been located at the position where the lightprobe was taken. The use of ambient occlusion (Pharr, 2004) further increases the quality of this approximation.

2.2 Summed-Area Tables

As described in Section 1, summed-area tables enable the rapid calculation of the sum of the pixel values in an arbitrarily sized, axis-aligned rectangle at a fixed computational cost. Figure 1.1 illustrates how a summed-area table is used to compute the sum of the values of pixels spanning a rectangular region. To find the integral of the values in the dark rectangle, we begin with the pre-computed integral from (0,0) to (x_R, y_T). We subtract the integrals of the rectangles (0, 0) to (x_R, y_B) and (0, 0) to (x_L, y_T). The integral of the hatched box is then added to compensate for having been subtracted twice.

The average value of a group of pixels can be calculated by dividing the sum by the area. Crow's technique amounts to convolution of an input image with a box filter. The power lies in the fact that the filter support can be varied at a per pixel level without increasing the cost of the computation. Unfortunately, since the value of the sums (and thus the dynamic range) can get quite large, the table entries require extended precision. The number of bits of precision needed per component is

$$P_s = log_2(w) + log_2(h) + P_i$$

where w and h are the width and height of the input image. P_s is the precision required to hold values in the summed-area table, and P_i is the number of bits of precision of the input. Thus, a 256x256 texture with 8-bit components would require a summed-area table with 24 bits of storage per component.

Another limitation of Crow's summed-area table technique is that it is only capable of implementing a simple box filter. This is because only the sum of the input pixels is stored; therefore it is not possible to directly apply a more complex filter by weighting the inputs.

2.2.1 Higher-Order Summed-Area Tables

In (Heckbert, 1986), Heckbert extended the theory of summed-area tables to handle more complex filter functions. Heckbert made two key observations. The first is that a summed-area table can be viewed as the integral of the input image, and the second that the sample function introduced by Crow was the same as the derivative of the box filter function. By taking advantage of those observations and the following convolution identity

$$f \otimes g = f'^{n} \otimes \int^{n} g$$

it is possible to extend summed-area tables to compute higher order filter functions, such as a Bartlett filter, or even a Catmull-Rom spline filter. The process is essentially one of repeated box filtering. Higher order filters approach a Gaussian, and exhibit fewer artifacts such as the blockiness associated with box-filtering.

For instance, Bartlett filtering requires taking the second-order box filter, and weighting it with the following coefficients:

$$f = \begin{array}{rrr} 1 & -2 & -1 \\ -2 & 4 & -2 \\ 1 & -2 & -1 \end{array}$$

Unfortunately, a direct implementation of the Bartlett filtering example requires 44 bits of precision per component, assuming 8-bits per component and a 256x256 input image.

In general, the precision requirements of Heckbert's method can be determined as follows:

$$P_s = n * (log_2(w) + log_2(h)) + P_i$$

where w and h are the width and height of the input texture, n is the degree of the filter function, P_i is the input image's precision, and P_s is the required precision of the n^{th}-degree summed-area table (Heckbert, 1986).

2.2.2 Related Techniques

Various techniques (Ashikhmin and Ghosh, 2002; Yang and Pollefeys, 2003) have been presented that combine multiple samples from different levels of a mipmap to approximate filtering. Ashikhmin and Ghosh approximate simple blurry reflections by using multiple samples from a mipmapped environment map instead of pre-filtering the environment map (Ashikhmin and Ghosh, 2002). By using multiple samples, they are able to approximate various simple BRDFs and blur the environment map on the fly, giving objects the appearance of having glossy BRDFs. Yang and Pollefeys use the same approach to assist in performing depth correlation on a pair of stereo images. They take multiple samples from the mip-map and sum them together to approximate a smooth filter functions.

These techniques suffer from several problems. First, a small step in the neighborhood around a pixel does not necessarily introduce new data to the filter; it only changes the weights of the input values. Second, when the inputs do change, a large amount of data changes at the same time, due to the mipmap, which causes noticeable artifacts. Demers et al. (Demers, 2004) added noise in an attempt to make the artifacts less noticeable; although, the visual quality of the resulting images was noticeably reduced.

2.2.3 Efficient Summed-Area Table Generation on GPUs

This section presents one of the major contributions of this thesis. In particular I present a technique to rapidly generate summed area tables on GPUs. In order to efficiently construct summed-area tables, I borrow a technique, called recursive doubling (Dubois and Rodrigue, 1977), often used in high-performance and parallel computing. Using recursive doubling, a parallel gather operation amongst n processors can be performed in only $log_2(n)$ steps, where a single step consists of each processor passing its accumulated result to another processor.

In a similar manner, the method presented uses the GPU to accumulate results so that only O(log n) passes are needed for summed-area table construction. To simplify the following description, I assume that only two texels, *texture elements*, can be read per pass. Later in the discussion I explain how to generalize the technique to an arbitrary number of texture reads per pass.

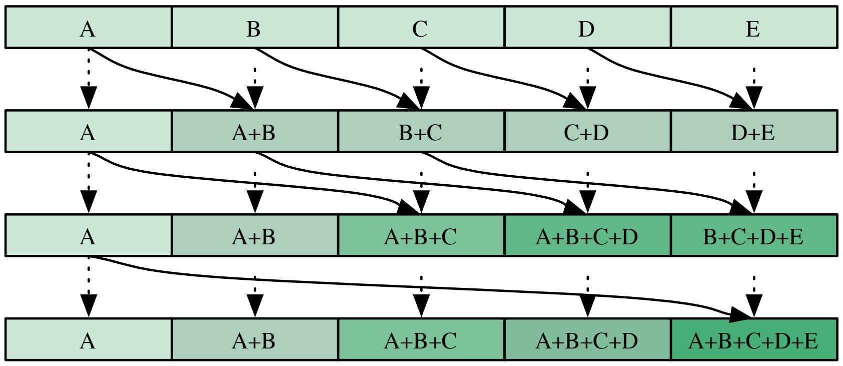


Figure 2.4: The recursive doubling algorithm in 1D. On the first pass, the value one element to the left is added to the current value. On the second pass, the value two elements to the left is added the current value. In general, the stride is doubled for each pass. The output is an array whose elements are the sum of all of the elements to the left, computed in O(log n) time.

The algorithm proceeds in two phases: first a horizontal phase, then a vertical phase. During the horizontal phase, results are accumulated along scan lines, and during the vertical phase, results are accumulated along columns of pixels. The horizontal phase consists of n passes, where $n = ceil(log2(image\ width))$, and the vertical phase consists of m passes, where $m = ceil(log2(image\ height))$.

For each pass a screen-aligned quad is rendered that covers all pixels that do not yet hold their final sum. This prevents pixels that have already computed their final value from wasting precision resources. The input image is stored in a texture named t_A. In the first pass of the horizontal phase two texels are read from t_A: the one corresponding to the pixel currently being computed and the one to the immediate left. They are added together and stored into texture t_B.

For the second pass, the textures are swapped so that data is read from t_B and written to t_A. Now the fragment program adds the texels corresponding to the one currently being computed and the one two pixels to the left. t_A now holds the sum of four pixels.

The third pass repeats this scheme, now reading from t_A and writing to t_B and summing two texels four pixels apart, resulting in the sum of eight pixels in t_B. This progression continues for the rest of the horizontal passes until all pixels are summed up in the horizontal direction. Note that in pass i the leftmost 2^i pixels already hold their final sum for the horizontal phase and thus are not covered by the quad rendered in this pass. Next the vertical phase proceeds in an analogous manner. Figure 2.4 shows the horizontal passes needed to construct a summed-area table of a 4x4 image. The following pseudo-code summarizes the algorithm.

```
$t_A \Leftarrow InputImage$
$n \Leftarrow log_2(width)$
$m \Leftarrow log_2(height)$

// horizontal phase
$i \Leftarrow 0$
for $i < n$ do
        $t_B[x, y] \Leftarrow t_A[x, y] + t_A[x + 2^i, y]$
        $swap(t_A, t_B)$
        $i \Leftarrow i + 1$
end for

// vertical phase
$i \Leftarrow 0$
for $i < m$ do
        $t_B[x, y] \Leftarrow t_A[x, y] + t_A[x, y + 2^i]$
        $swap(t_A, t_B)$
        $i \Leftarrow i + 1$
end for

// Texture $t_A$ holds the result
```

In practice, reading more than two texels per fragment, per pass is possible, and this reduces the number of passes required to generate a summed-area table by at least a factor of two. The current implementation supports reading 2, 4, 8, or 16 texels per fragment, per pass. This allows trading per-pass complexity with the number of rendering passes required. Adding 16 texels per pass enables us to generate a summed-area table from a 256x256 image in only four passes, two for the horizontal phase, and two for the vertical phase. As shown later, adjusting the per-pass complexity helps in optimizing summed-area generation speed for different input texture sizes. The following is the pseudo-code to generate a summed-area table when r reads per fragment are possible.

```
$t_A \Leftarrow InputImage$
$n \Leftarrow log_r(width)$
$m \Leftarrow log_r(height)$

// horizontal phase
$i \Leftarrow 0$
**for** $i < n$ **do**
        $t_B[x, y] \Leftarrow t_A[x, y]+$
                $t_A[x + 1 * r^i, y]+$
                $t_A[x + 2 * r^i, y]+$
                $\cdots +$
                $t_A[x + r * r^i, y]$
        $swap(t_A, t_B)$
        $i \Leftarrow i + 1$
**end for**

// vertical phase
$i \Leftarrow 0$
**for** $i < n$ **do**
        $t_B[x, y] \Leftarrow t_A[x, y]+$
                $t_A[x, y + 1 * r^i]+$
                $t_A[x, y + 2 * r^i]+$
                $\cdots +$
                $t_A[x, y + r * r^i]$
        $swap(t_A, t_B)$
        $i \Leftarrow i + 1$
**end for**

// Texture $t_A$ holds the result
```

Note that near the left and bottom image borders the fragment program will fetch texels outside the image regions. To ensure correct summation of the image pixels, the texture units must be configured to use *clamp to border color* mode with the border color set to 0. This way texel fetches outside the image boundaries will not affect the sum. Alternatively, it is possible to render a single pixel black border around the input image and configure the texture units to use *clamp to edge* mode.

The algorithm presented has been implemented in both Direct3D and OpenGL, with similar results. Tables 2.1 and 2.2 summarize the Direct3D results. The OpenGL implementation uses a double buffered pbuffer to mitigate the cost of context switches. Instead of switching context between each pass, the implementation simply swaps the front and back buffers of the pbuffer. This allows us to efficiently *ping-pong* between two textures as results are accumulated. The Direct3D implementation simply uses two different render targets. If implemented at the driver level, similar to the way that automatic mip-map generation is done, the costs of the passes would be reduced even more.

2.2.4 Summed-Area Table Generation Performance

Table 2.1 shows the time required to generate summed-area tables of different sizes on a number of graphics cards using DirectX 9. For each card, and for each of the three input image sizes, we show the shortest time to generate a summed-area table along with the number of texels read per fragment per pass that gives the best result.

	Summed-area table size		
	256x256	512x512	1024x1024
Radeon 9800 XT[1]	3.1 ms (8)	14.2 ms (4)	70.1 ms (4)
Radeon X800XT PE[1]	1.4 ms (8)	7.3 ms (4)	36.2 ms (4)
Geforce 6800 Ultra[2]	4.3 ms (8)	32.4 ms (4)	95.3 ms (4)

Table 2.1: Shortest time to generate summed-area tables of different sizes. The number of samples per pass are given in parentheses. [1]$24 - bit\, floats$ [2]$32 - bit\, floats$

	Summed-area table size		
Samples/pass	256x256	512x512	1024x1024
2	2.3 ms	9.9 ms	44.3 ms
4	1.8 ms	7.3 ms	36.2 ms
8	1.4 ms	9.9 ms	45.6 ms
16	2.7 ms	12.4 ms	53.3 ms

Table 2.2: Time to generate summed-area tables of different sizes using different number of samples per pass on a Radeon X800XT Platinum Edition graphics card.

Table 2.2 shows performance based on input size and the number of samples per pass for one of the cards used in the tests. Benchmark results show that finding a good balance between the number of rendering passes and the amount of work performed during each pass is important for the overall performance of summed-area table generation. The optimal tradeoff between the number of passes and per-pass cost is largely dependent on the overhead of switching render targets, which typically causes a pipeline flush, and the design of the texture cache on the target platform.

Computing summed-area tables directly on the graphics card is better than performing this computation on the CPU for a two primary reasons. First, the input data is already present in GPU memory. Transferring the data to the CPU for processing and then back again would put an unnecessary burden on the bus and can easily become a bottleneck because many graphics drivers are unable to reach full theoretical bandwidth utilization when reading back data from the GPU (GPUBench, 2004). Moreover, moving data back and forth between GPU and CPU would break GPU-CPU parallelism because each processor would end up waiting for new results from the other processor.

2.3 Offset Summed-Area Tables

A key challenge to the usefulness of the summed-area table approach is the loss of numerical precision, which can lead to significant noise in the resultant image. This section first discusses the source of such precision loss and then presents my approach to mitigate this problem (this technique is also described in (Hensley et al., 2005)). Example images are provided that demonstrate how the approach achieves significant reduction in noise: up to 31 dB improvement in signal-to-noise ratios.

2.3.1 Source of Precision Loss

One source of precision loss could be the GPU's floating-point implementation: Current graphics hardware does not implement IEEE standard 754 floating point but, as shown by Hillesland (Hillesland and Lastra, 2004), current GPU implementations behave reasonably well, so this is not the primary source of numerical error.

The summed-area table approach can exhibit significant noise because certain steps in the algorithm involve computing the difference between two relatively large finite-precision numbers with very close values. This is especially true for pixels in the upper right portion of the image because the monotonically increasing nature of the summed-area function implies that the table values for that region are all quite high.

As an example, consider the images of Figure 2.5, which are 256x256 images with 8-bit components. The middle and right columns show the image after being filtered through an "identity filter," i.e., a 1-bit filter kernel that is ideally supposed to produce a resultant image that is a replica of the original image. To avoid loss of computational precision, a summed-area table with 24 bits of storage per component per pixel would be sufficient, since the maximum summed-area value at any pixel cannot exceed [256x256]x256. However, the summed-area table used in this example used 16 and 24 bit FP values. 16-bit floating point values are represented with one sign bit, 5 exponent bits, and 10 mantissa bits (s10e5), whereas 24-bit floating point values are represented with one sign bit, seven exponent bits, and 16 mantissa bits (s16e7). As a result, significant noise is seen in the filtered image, with worsening image

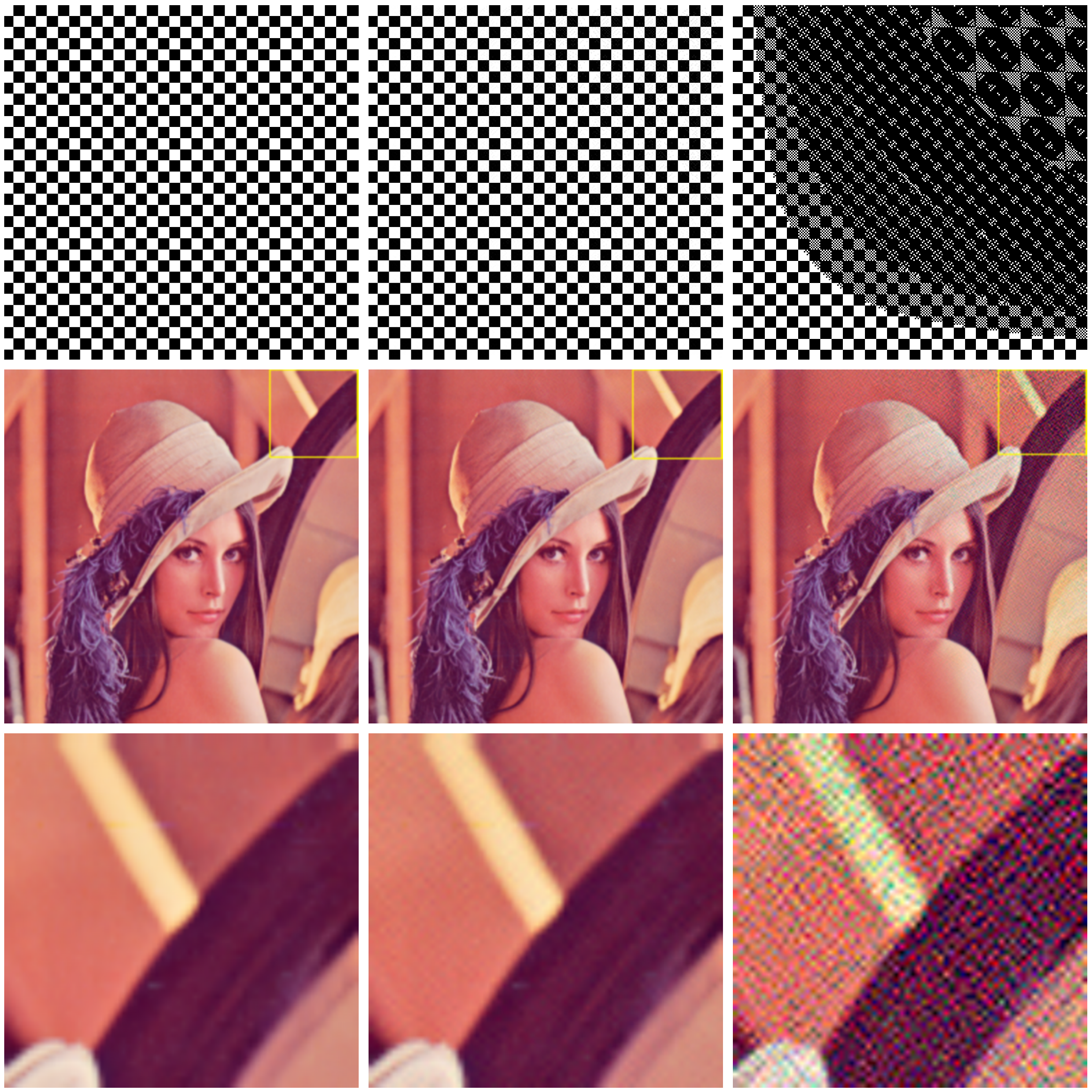


Figure 2.5: The left column shows the original input images, the middle column of images are reconstructions from summed-area tables (SATs) generated using our method, and the right column are reconstructions from SATs generated with the Crow's method. For the first row, the SATs are constructed using 16 bit floats, for the second row the SATs are constructed using 24 bit floats, and the final row shows a zoomed version of the second row (region-of-interest highlighted)

quality in the direction of increasing xy.

2.3.2 Using Signed-Offset Pixel Representation

Offset summed-area tables simply represent pixel values in the original image as *signed* floating-point values (e.g., values in the range -0.5 to 0.5), as opposed to the traditional approach that uses unsigned pixel values (from 0.0 to 1.0).

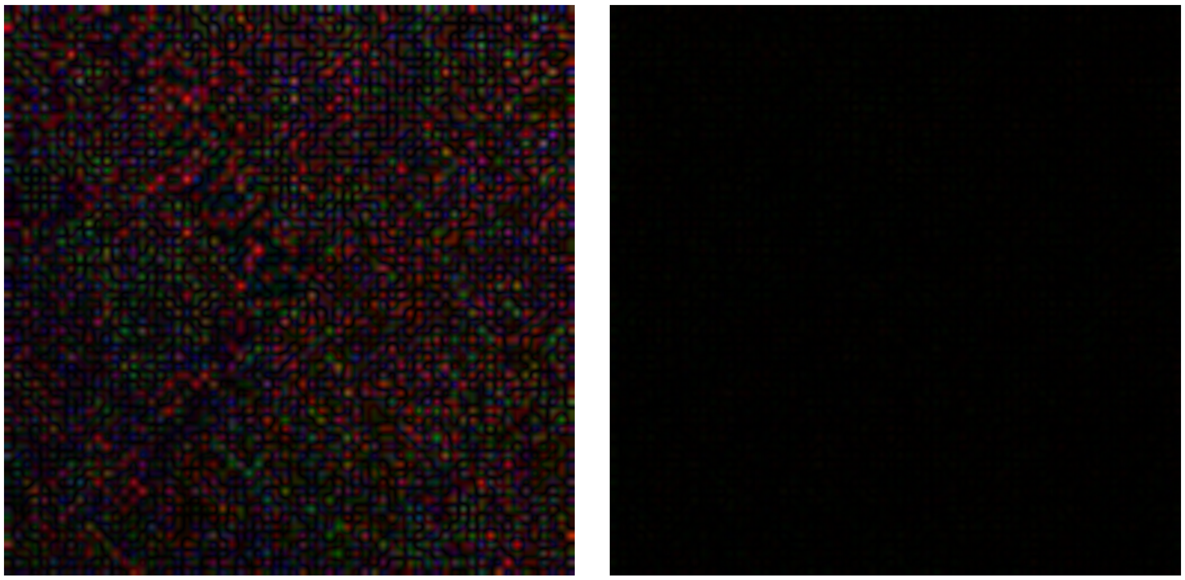
This modification improves precision in two ways: (i) there is a 1-bit gain in precision because the sign bit now becomes useful, and (ii) the summed-area function becomes non-monotonic, and therefore the maximum value reached has a relatively lower magnitude.

I have investigated two distinct methods for converting the original image to a signed-offset representation: (i) centering the pixel values around the 50% gray level, and (ii) centering them around the average image pixel value. The former involves less computational overhead and gives good precision improvement, but the latter provides even better results with modest computational overhead.

Centering around 50% gray level. This method modifies the original image by subtracting 0.5 from the value at every pixel, thereby making the pixel values lie in the -0.5 to 0.5 range. The summed-area table computation proceeds as usual, but with the understanding that the table entry at pixel position (x,y) will now be 0.5xy less than the actual summed-area value. The net impact is a significant gain in precision because the table entries now have significantly lower magnitudes, and therefore computing the differences yields a greater precision result.

Figure 2.5 demonstrates the usefulness of this approach. The first row shows three versions of a checkerboard. The image on the right, reconstructed from a traditional summed-area table, exhibits unacceptable noise throughout much of the image. In contrast, the middle image, generated by our method, shows no visibly perceptible errors.

Centering around image pixel average. While centering pixel values around the 50% gray level proved to be useful, an even better approach is to store offsets from the image's average pixel value. This is especially true of images such as Lena for which the image average can be quite different from 50% gray. For such images, centering around 50% gray could still result in sizable magnitudes at each pixel position, thereby increasing the probability that the


(a) Absolute-value of the difference between ground truth and a reconstruction using original summed-area tables.

(b) Absolute-value of the difference between ground truth and a reconstruction using my method (offset summed-area tables).

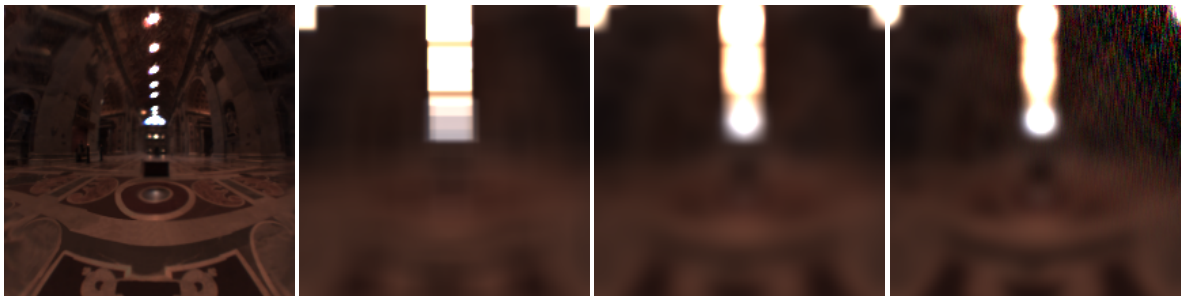
Figure 2.6: Summed-area table reconstruction error of the inset (bottom row) of Figure 2.5.

summed-area values could appreciably grow in magnitude. Centering the pixel values around the actual image average guarantees that the summed-area value is equal to 0 both at the origin and at the upper right corner (modulo floating-point rounding errors). Figure 2.6 shows the *error* images between a reconstruction using the original summed-area table algorithm, and the offset summed-area tables method presented in this dissertation.

The computational overhead of this approach is modest as the image average is easily computed in hardware using mip mapping.

2.4 Higher-Order Summed-Area Table Generation

Generating the interior pixels of a higher-order summed-area table is a simple matter of re-running the algorithm presented in Section 2.2.4 on the $n-1$ order summed-area table. Complicating the issue is that an essential portion of the image-based lighting algorithm presented later requires filtering large pixel regions, which increases the chance that a filter kernel will extend beyond the boundary of the image being filtered. Traditional filtering approaches handle such a situation by padding the image with selected values beyond its



(a) original image (b) $oSAT^1$ (box) (c) $oSAT^2$ (Bartlett) (d) $oSAT^3$ (cubic)

Figure 2.7: Comparison of images filtered using repeated offset summed-area tables. Image (a) shows the original input image, half of a dual-paraboloid map computed from the St. Peter's Basilica light probe (Debevec, 1998). Image (b) is (a) filtered with a box filter using a first order offset summed-area table. Image (c) is (a) filtered with a Bartlett filter using a second order offset summed-area table. Image (d) is (a) filtered with a cubic filter using a third order offset summed-area table. The error in the upper right corner of image (d) is the result of a loss of precision.

boundary pixels. One such padding is to extend the image with black pixels, while arguably, a more reasonable approach is to simply extend the boundary pixels.

There is a simple optimization that traditional approaches use, which is no longer applicable when computing higher-order summed-area tables. Traditionally, padding by extending boundary pixel values is actually achieved by re-mapping read accesses to the padded region back to the boundary pixels. Thus, no actual padding is done, but its effect is simulated by "clamping" the sampling coordinates to the boundary. However, for computing higher order summed-area tables, one cannot use clamping on the first-order summed-area table. In particular, the correct padding is not simply a replication of its boundary values; it is actually the integral of the underlying padded image.

Therefore, for filtering from a second-order summed-area table, instead of clamping, a simple, yet effective, approach is to actually pad the original image, and then do all computations on the padded image. This technique ensures that all higher-order summed-area tables are computed accurately, at an increased computational cost. In practice, this does not dramatically affect performance.

Image 2.7 shows a comparison of filtering a high-dynamic range (HDR) image with a first, a second and a third order summed area table. For the comparison, the first, second, and third order offset summed area tables of the St. Peter's Basilica Lightprobe were generated using 32-bit floating point arithmetic. For the filtered images, the width of the filter kernel is the same for the different filter orders. The first order filtered image, Figure 2.7(b), clearly shows blocking artifacts from the use of the box filter, while the third order image, Figure 2.7(d) suffers from a loss of precision. The blocking artifacts are greatly exacerbated by the HDR data from the input image, since as soon as bright spot enters the filter kernel, it overwhelms the rest of the data in the filter kernel's region of support. Figure 2.7(c) shows that filtering with a second order summed-area table offers a reasonable compromise by greatly reducing the blocking artifacts while limiting errors introduced from a loss of precision. Additionally, filtering with a second order summed-area table requires only nine memory accesses, while filtering with a third order summed-area table would require sixteen memory accesses.

2.5 Rendering Glossy Reflections with Summed-Area Tables

Since the technique presented in Section 2.2.2 is efficient enough to generate summed-area tables every frame (less than 2ms for a 256x256 input image on an X800XT PE), their use becomes feasible to generate real-time, interactive effects. This makes summed-area tables an attractive candidate for implementing techniques that approximate dynamic glossy reflections by filtering dynamically generated images.

2.5.1 Glossy Environmental Reflections

Figure 2.8 is an image of an object where the environment map has been filtered with a spatially varying filter function; in this case the filter support has been modulated by another texture. The image is rendered in real time, at a rate of over 60 frames per second on a Radeon X800XT PE with 24-bit floating-point arithmetic. The filter function, scene geometry and environment map can change every frame.

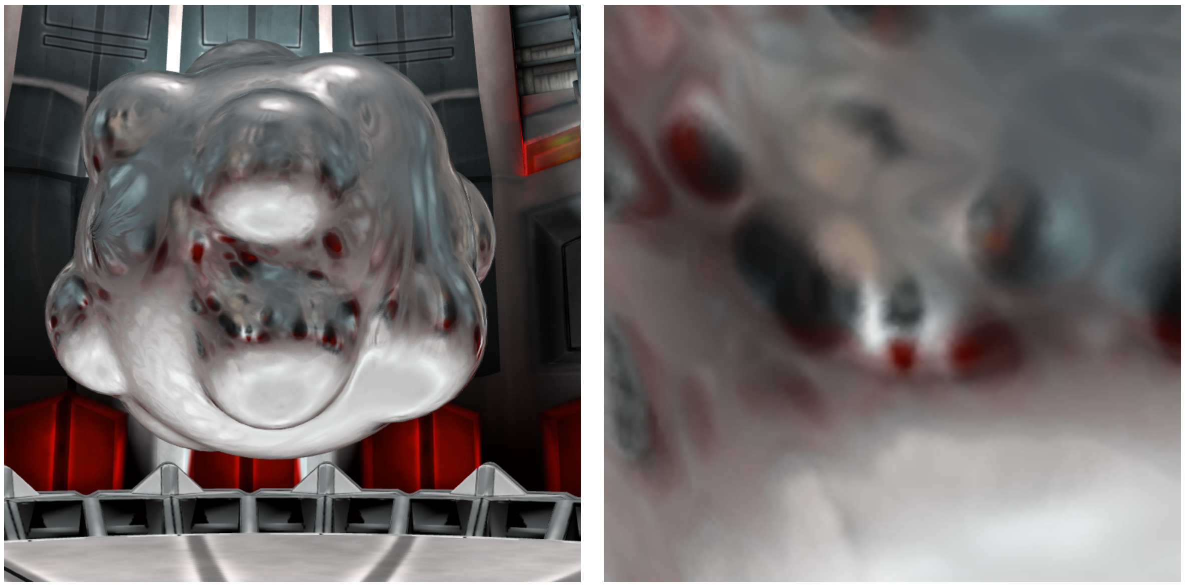
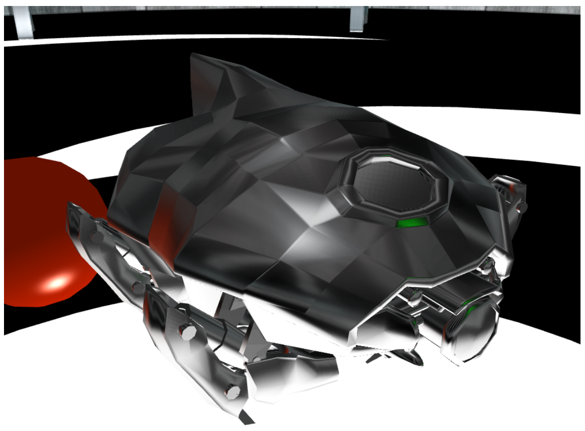
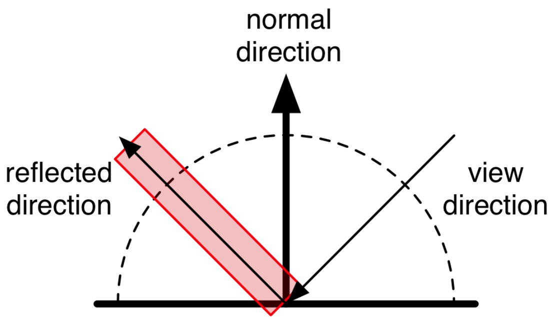

(a) Rendered image (b) close-up of reflections

Figure 2.8: An object rendered with glossy reflections by filtering a dynamic environment map with a spatially varying kernel size. Image (b) shows a close-up of the rendered image, and the spatially varying *glossiness*

There are several compelling reasons for using dual-paraboloid environment mapping over the more commonly used cube mapping. First, Kautz et al. showed that when filtering in image space, as opposed to filtering over a solid angle of a hemisphere, a dual-paraboloid environment map has lower error than a cube map or a spherical map. Second, it is only necessary to generate two summed-area tables as opposed to six summed-area tables (one per face of the cube). Finally, for large filters, a dual-paraboloid map requires data from only two textures, whereas it is possible that data might be required from all six faces of a cube map.

A gross approximation to a glossy BRDF is a simple box filter. A single box-filter evaluation takes four texture reads from the summed-area table. Two evaluations are required in the current implementation when a filter is supported by both the front and the back of a dual-paraboloid map, since at the time of implementation efficient *if-else* statements were not supported on GPUs. On hardware that has more optimized branching, it is possible to evaluate the filters for both maps only when necessary.


(a) Rendered image



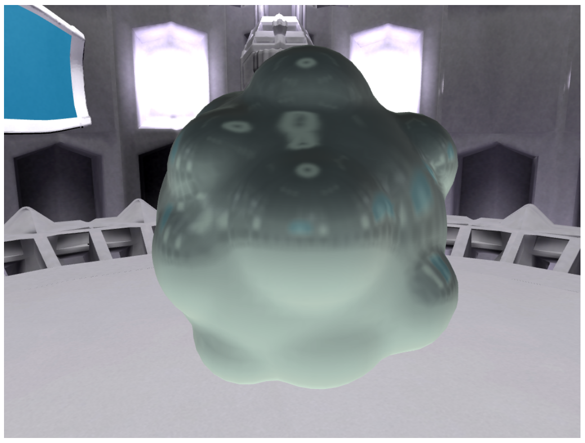
(b) single box filter

Figure 2.9: An object textured using four samples from a pair of summed-area tables generated from an environment map in real-time.

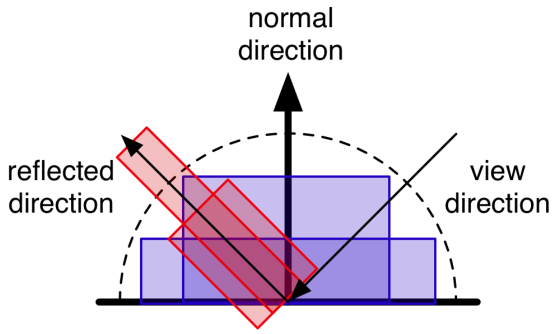
As is common when storing a spherical map in a square texture, our implementation uses the alpha channel to mark the pixels that are in the dual-paraboloid map. A pixel is considered to be in the map if its alpha value is one. The algorithm also uses the alpha value to count the area covered by the filter. After combining the result of the evaluation from the front and back maps, the alpha channel holds the total count of summed texels, which is then used to normalize the filter value.

The basic algorithm for rendering glossy environmental reflections follows.

```
$renderCubeMap()$
$generateDualParaboloidMapFromCubeMap()$
$generateSummedAreaTable(FrontMap)$
$generateSummedAreaTable(BackMap)$
$setupTextureCoordinateGeneration()$
$renderScene()$
$return$

$renderScene:$
for every fragment on reflective object do
```



(a) Rendered image



(b) stacked filters

Figure 2.10: A set of four box filters stacked to approximate a Phong BRDF.

```
        $front \Leftarrow evaluateSAT(FrontSAT, filter\_size)$
        $back \Leftarrow evaluateSAT(BackSAT, filter\_size)$

        // computer filter area
        $filtered.alpha \Leftarrow front.alpha + back.alpha$

        // combine front and back color
        $result \Leftarrow front + back$
        // divide by the area of the filter
        $result \Leftarrow result/filtered.alpha$
        $computeFinalColor(result)$
    **end for**
```

While the implementation presented creates a dual-paraboloid map from a cube map, it is possible to directly generate the dual-paraboloid map by using a vertex program to project the scene geometry as done in (Coombe et al., 2004), assuming that the introduced error by this method is acceptable.

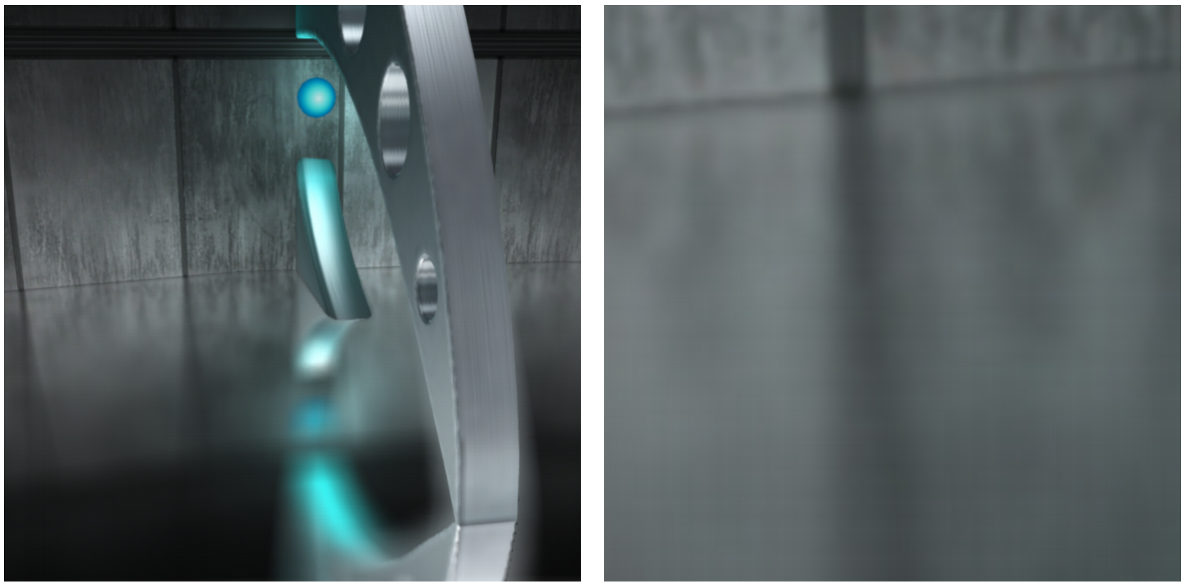

(a) Rendered image (b) close-up of reflections

Figure 2.11: An image illustrating the use of a summed-area table to render glossy planar reflections where the blurriness of an object varies depending on its distance from the reflector. The object-reflector distance is used to vary the filter size to sharpen the reflections as the object nears the planar reflector as can be seen in image (b), a close-up of where the floor meets the wall. As can be seen the sharpness of the reflections varies with the distance between the reflector and the wall.

More complex filter functions can be constructed at the cost of more texture reads by *stacking* multiple box filters on top of each other. The stacked boxes approximate the shape of smoother filters. For a single summed-area table, each filter in the stack requires eight texture reads (four reads for the box filter in the front map and four reads for the box filter in the back map). So a complex filter created from a stack of four box filters would perform thirty-two texture reads per fragment.

Both OpenGL and Direct3D provide a means to automatically generate texture coordinates based on the normal direction and reflection direction. By combining box filters generated from both the reflection direction and the normal direction, it is possible to compute an approximation of the Phong BRDF. Figure 2.11(a) shows an image generated using a stack of two large box filters centered on the normal direction to approximate the diffuse component of the Phong BRDF and a stack of two smaller box filters centered on the reflection direction to approximate the specular component.

2.5.2 Glossy Planar Reflections

Since the summed-area table enables filtering with arbitrary support, it is relatively easy to render glossy reflections where the blurriness of an object varies depending on the distance of the reflected object from the reflector. This effect is often seen when an object is placed on a glossy table top. The object's reflection is much sharper where the object and table top meet than elsewhere. Figure 2.11 shows an image where the floor is a glossy reflector, and the blurriness of the reflection depends on the object's distance from the floor.

Figure 2.11(b) shows a close up of the rendering that is accomplished by augmenting the standard planar reflection algorithm to implement glossy reflections. The pass for rendering the reflected scene from the virtual viewpoint outputs both the color and the distance to the reflection plane to a texture. A summed-area table is generated from the color data. Then the planar reflector is rendered from the summed-area table, using the previously saved distance to modulate the filter width.

2.6 Depth-of-Field and Glossy Translucency

Since summed-area tables allow the efficient filtering of images with spatially varying filter kernels, other effects besides glossy reflections are possible. This section discusses two: a depth-of-field effect based translucency.

2.6.1 Depth-of-Field

In (Greene, 2003), Greene presents a technique to render an image with a depth-of-field effect using summed-area tables. His summed-area table generation technique is problematic since it requires that a texture be read from and written to at the same time. Unfortunately, graphics hardware — due to its parallel streaming architecture — makes no guarantees about the execution sequence of read-modify-write operations.

In (Demers, 2004), a technique to render a depth-of-field effect was presented that used mip maps to approximate a simple filter. Because of the artifacts introduced by the mip-map

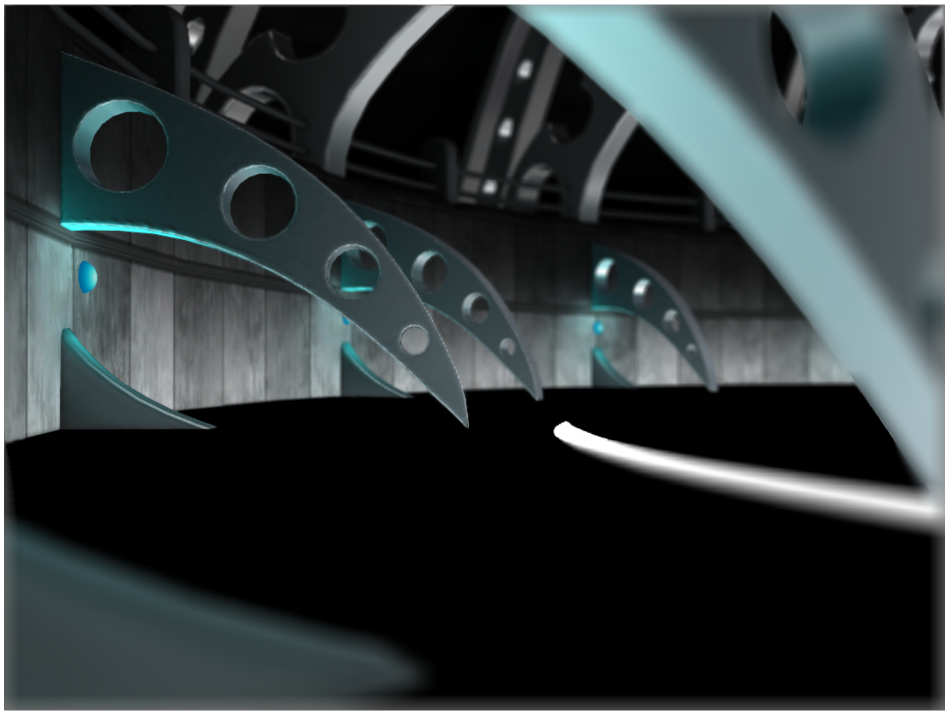


Figure 2.12: Simulated depth-of-field effect using summed-area tables.

filtering technique, the authors add noise to reduce the perceptible Mach bands.

Unlike mip maps, summed-area tables are able to average arbitrary rectangular sections of an image, allowing us to implement a real-time, interactive version of the depth-of-field effect, without having to add noise to mask filtering artifacts. However, our implementation does have the same drawbacks as other image filtering techniques for generating a depth-of-field effect, such as the bleeding of sharp in-focus objects onto blurry backgrounds. Figure 2.12 shows an image rendered with depth-of-field. This 1024x768 image renders at a rate of 23 frames per second with an ATI Radeon X800XT PE. Lower resolution versions render at higher frame rates.

The effect is accomplished by first rendering the scene from the camera's point-of-view and saving the color and depth buffers to texture memory. Next, a summed-area table is generated from the saved color buffer. As in (Demers, 2004), the depth buffer is used to determine the circle-of-confusion. Finally, a screen-filling quad is rendered, and a fragment program is used to blur the color buffer based on the circle-of-confusion.

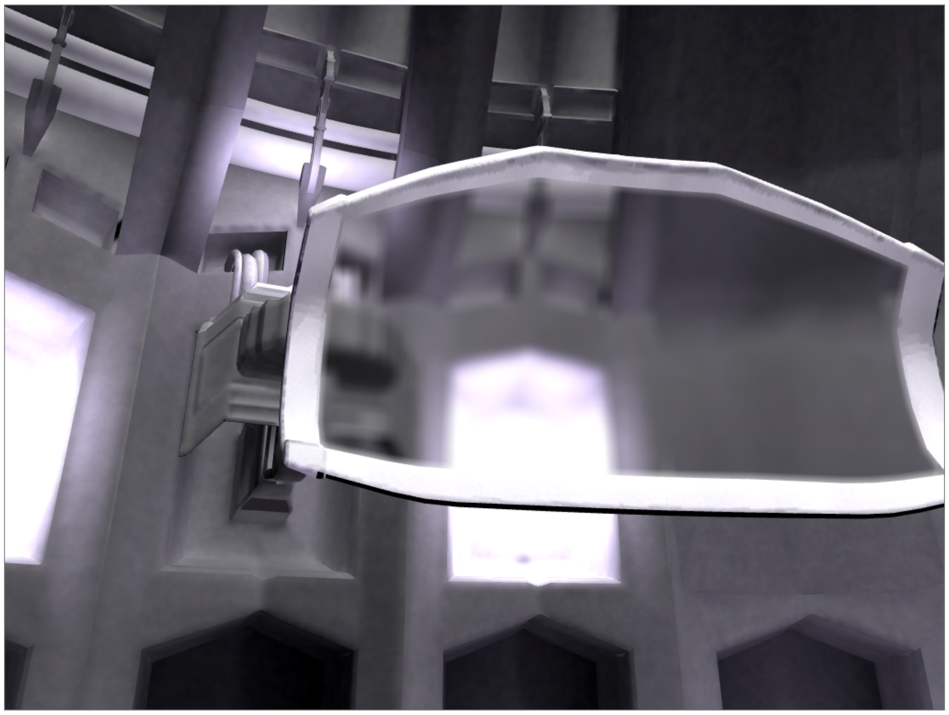


Figure 2.13: Example of translucency using a summed-area table to filter the view seen through the glass.

2.6.2 Translucency

Approaches to rendering translucent materials include those of (Arvo, 1995; Diefenbach, 1996). While these techniques are able to generate high-quality results, they are not able to handle dynamically changing environments in real time. Using summed-area tables it is possible to render real-time, interactive translucent objects. This technique can be used to render such effects as etched and milky glass. Figure 2.13 shows a scene with multiple translucent objects.

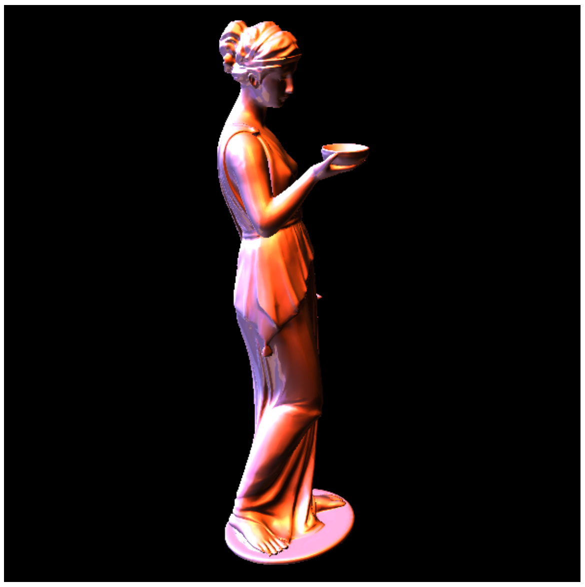
The rendering steps necessary for dynamic translucency are as follows.

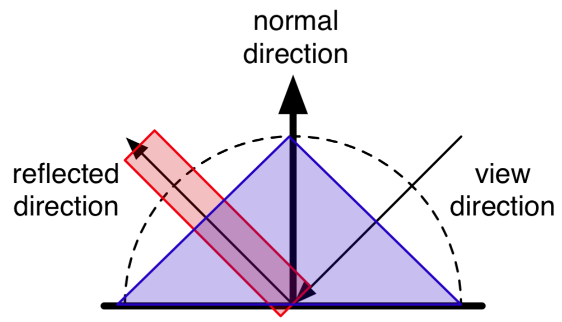
```
// Update the lighting environment
$texture \Leftarrow renderSceneWithoutTranslucentObjects()$
$SAT \Leftarrow generateSummedAreaTable(texture)$
$setupTextureCoordinateGeneration()$
$renderScene()$
```

```
$return$

$renderScene:$
for every translucent fragment do
        // transmap holds spatially varying filter kernel size
        $filter\_size \Leftarrow sampleTexture(transmap, u, v)$
        $result \Leftarrow evaluateSAT(FrontSAT1, 1, filter\_size)$

        // divide by the area of the filter
        $filtered \Leftarrow result/result.alpha$

        $computeFinalColor(filtered)$
end for
```

2.7 Approximate HDR Image-Based Lighting

By using higher-order offset summed-area tables, it is possible to filter a dual-paraboloid high-dynamic range environment map in constant time, irrespective of the filter size. Figure 2.14 shows the image of a model rendered with an approximate Phong BRDF computed dynamically using offset summed-area tables. The diffuse component is approximated by sampling a second order summed-area table with a large filter kernel in the direction of the normal (shown as the large triangle in Figure 2.14(b)), and the specular component is approximated by sampling a first order summed-area table with a small filter kernel in the direction of the reflection direction (shown as the thin rectangle in Figure 2.14(b)). The second order filter is used for the diffuse component to prevent noticeable artifacts from box filtering, shown in Figure 2.15. The first order filter is used for the specular component to increase performance, since it only requires four texture reads, and to limit the effect of


(a) Rendered image



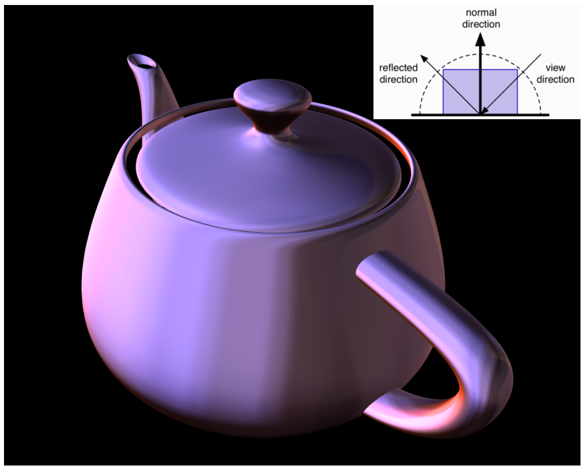
(b) filters used

Figure 2.14: A rendering of Hebe, with an approximate image-based lighting computed from two sets of summed area tables whose results are summed together. One 2nd order summed-area table to approximate diffuse lighting, and one 1st order summed-area table to approximate specular lighting. Image (b) represents the filters used.

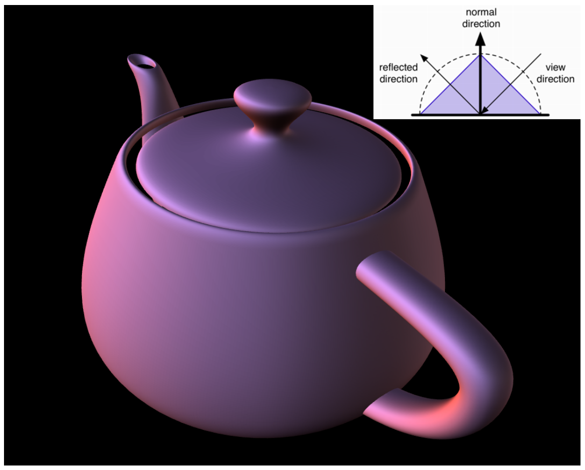
precision loss due to the use of single precision floating point textures.

The rendering steps necessary for fully dynamic image-based lighting are as follows.

```
// Update the lighting environment
$renderCubeMap()$
$generateDualParaboloidMapFromCubeMap()$
$FrontSAT1 \Leftarrow generateSummedAreaTable(FrontMap)$
$FrontSAT2 \Leftarrow generateSummedAreaTable(FrontSAT1)$
$BackSAT1 \Leftarrow generateSummedAreaTable(BackMap)$
$BackSAT2 \Leftarrow generateSummedAreaTable(BackSAT1)$
$setupTextureCoordinateGeneration()$
$renderScene()$
$return$
```



(a) Box filter



(b) Bartlett filter

Figure 2.15: Approximating a diffuse BRDF using summed area tables. Image (a) shows an approximation to a diffuse BRDF using a simple box filter. Image (b) shows an approximation using a Bartlett filter, which clearly eliminates the banding artifacts seen in Image (a).

```
$renderScene$ :
**for** every fragment on object **do**
        $front1 \Leftarrow evaluateNthSAT(FrontSAT1, 1, filter\_size)$
        $front2 \Leftarrow evaluateNthSAT(FrontSAT2, 2, filter\_size)$
        $back1 \Leftarrow evaluateNthSAT(BackSAT1, 1, filter\_size)$
        $back2 \Leftarrow evaluateNthSAT(BackSAT2, 2, filter\_size)$

        // computer filter area
        $filtered1.alpha \Leftarrow front1.alpha + back1.alpha$
        $filtered2.alpha \Leftarrow front2.alpha + back2.alpha$

        // combine front and back color
        $result1 \Leftarrow front1 + back1$
        $result2 \Leftarrow front2 + back2$

        // divide by the area of the filter
        $filtered1 \Leftarrow result1/result1.alpha$
        $filtered2 \Leftarrow result2/result2.alpha$

        $computeFinalColor(filtered1, filtered2)$
    **end for**
```

2.8 Conclusion

This chapter presented several rendering techniques that use the ability to generate summed-area tables efficiently on the GPU. Additionally, the techniques take advantage of the precision improvements provided by offset summed-area tables. While offset summed-area tables can dramatically increase the precision limitations of summed-area tables, the do not eliminate them.

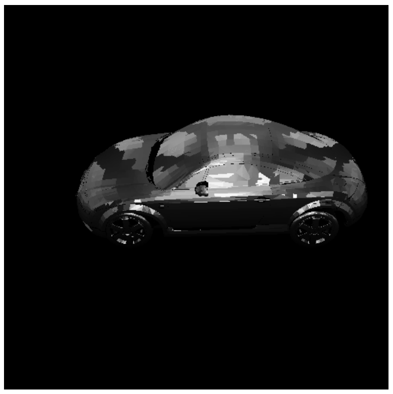
CHAPTER 3

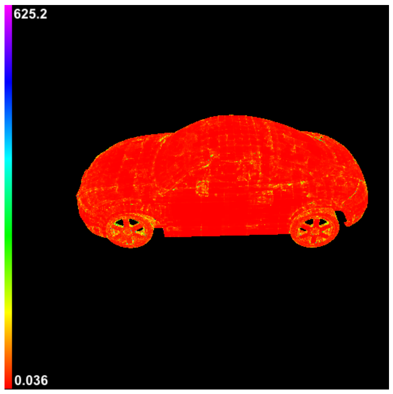
Increasing Graphics Hardware Performance Along the Architectural Axis

The second axis to be considered is the architectural axis. I discuss how graphics hardware that incorporates hardware support for conditional output streams can be used to implement an efficient ray casting algorithm. Furthermore, I discuss some of the issues involved with adding conditional streams to current commodity graphics hardware. Conditional output streams are an architectural extension that attempt to solve a critical shortcoming of most data-parallel machines such as GPUs: the inefficient execution of control-flow if statements. Typically, SIMD machines will execute both sides of a branch on each processor, causing load imbalances and wasted work. The novel streaming ray casting algorithm increases coherence by testing as many rays as possible against a particular node in the acceleration data structure before going on to the next node. I show that this method can reduce memory bandwidth requirements by more than forty times when compared to current ray tracing techniques.

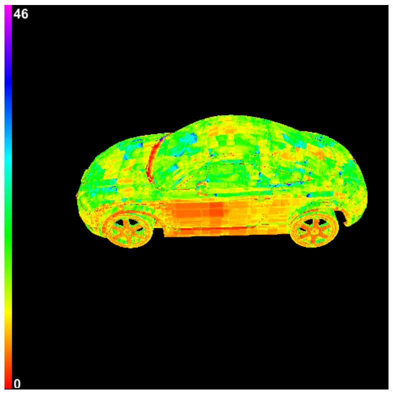
As on most data parallel architectures, conditional operations are difficult to handle on graphics processors. GPUs are just now gaining conditional operations but because of their SIMD processing nature, they are limited to either straight arithmetic operations with no branching, or inefficient execution of both sides of a branch. Although, the ATI X1800 shader architecture (ATI Technologies, 2005) is optimized to take advantage of the situation when all pixels in a processing group branch the same way, so that both branches do not need to be executed.

Researchers in streaming architectures have introduced the concept of conditional streams (Kapasi et al., 2000), which augment traditional streaming processors with the capability to conditionally read from input streams, and conditionally write to output streams. While


(a) Ray traced reflections



(b) Stream length metric



(c) Ray *restarts*

Figure 3.1: Ray traced second generation rays using conditional output streams. Image (a) shows reflections on a car placed inside the Sponza scene. For simplicity, the pixels are shaded with the distance to the closest intersection point. Image (b) shows sum of $1/(\textit{stream length})$ for each pixel. The colormap is on the left of the image. Red pixels represent smaller values, and purple pixels represent larger values. As can be seen, most rays are processed in large streams. Image (c) shows the number of ray *restarts* needed for each pixel. The number of restarts is strongly correlated with the number of node traversals required to find the closest intersection point. For the image shown, the worst case pixel required forty-six restarts. Images were rendered at 512x512.

some prior work has been reported on using conditional streams in GPUs, it does not directly address the problem of ray tracing. In particular, Popa (Popa, 2004) has proposed the use of conditional streams for compiling data-dependent control flows for SIMD GPUs. Additionally, Direct3D 10 (Blythe, 2006) introduces the concept of geometry shaders, which also have a similar capability to generate compacted streams of data.

By implementing conditional output streams on GPUs, a completely new set of capabilities becomes available to applications developers. Many applications are not sensitive to the order in which the data is processed and can benefit from better performance using conditional operations. In this dissertation, I focus on a novel streaming ray casting algorithm. The algorithm uses conditional output streams to reduce memory bandwidth requirements and increase processor utilization when compared to previous methods (Foley and Sugerman, 2005). The new algorithm is able to reduce memory bandwidth by over forty times compared to the most efficient method presented so far. One possible use for my proposed technique is to implement hybrid rendering algorithms that use standard z-buffering techniques to generate the first hits from the camera view, and then use ray tracing to generate geometrically correct reflections and shadows.

3.1 Background

The rather dramatic increases in the speed of processors over the past several decades has not been matched in DRAM devices. This has led to an ever increasing performance gap to the point where modern CPUs waster a majority of their cycles waiting for data to return from memory. While modern DRAM devices have relatively high bandwidth when accessing data in large sequential blocks, initiating accesses to non-adjacent memory locations typically incurs a long latency between the request and the arrival of the new data. As processor speed increases, this speed mismatch becomes increasingly difficult to hide. Processor architects have had to develop ever more ingenious ways to overcome the access latency, since general-purpose processors typically execute workloads that have relatively random access patterns. To help prevent processors from idling waiting for data from DRAM, modern processors have extremely large caches that smooth data access times. Caches allow data to be accessed in large, contiguous blocks, an operation that is more optimal for accessing DRAMs than random memory accesses. Despite this, conventional wisdom says that modern day general purpose processors spend most of their time waiting for data to return from the memory subsystem.

Because of the design practicalities involved with building large DRAM chips, it is necessary to sub-divide larger DRAM modules into several banks, each of which is further subdivided into blocks called pages. The first access to a page, often referred to as ”opening a page”, is a relatively long latency operation, but once a page is *opened* a large amount of data can either be written or read at high speed. Typically, only one page is allowed to be open per bank. Typically, DRAM modules allow concurrent accesses to multiple banks to mitigate the cost. So for example, if page X was open in bank A, and data needed to accessed in page Y of bank A, page X would have to be closed, and then page Y could be opened. The process of closing a page and opening a different page within the same bank takes a large number of cycles. Because of these constraints, it is in the architect’s best interest to access the maximum possible amount of useable data per page access.

3.1.1 Streaming Architectures

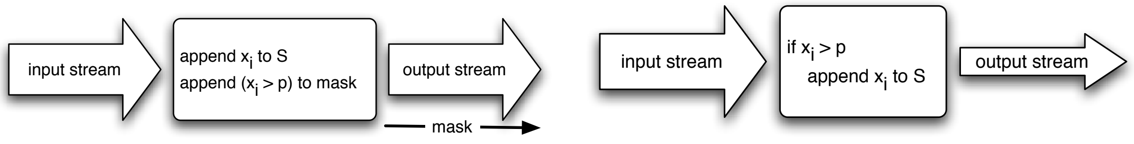
CPU architects have spent a large amount of time developing paradigms designed to overcome the widening performance gap between processor and memory performance. Once such class of architectures are stream processors. Stream processors read and write data in large sequential chunks, called streams. Each element of a stream is operated on by a small set of computational operations called a kernel. A stream can be pipelined through multiple kernels to implement complex functionality. Because stream processors access memory in very coherent chunks, they can take advantage of the high bandwidth of modern DRAMs without being adversely affected by the long latency needed to initially open DRAM pages. Stream processors are extremely adept at processing what are considered traditional DSP applications, such as image and signal processing where the same operation is executed on every piece of data in a stream, but as with most data-parallel architectures, conditional operations are inefficient.

3.1.2 Conditional Streams

As mentioned before, strictly data-parallel architectures typically have problems handling data-dependent operations. Kapasi et al. introduced the concept of conditional streams for stream processors as part of the Imagine stream processor architecture (Kapasi et al., 2000). As a part of his Master's thesis, Popa (Popa, 2004) discusses using conditional streams for SIMD GPUs in the context of compiling shaders that exhibit data-dependent control flows.

Figure 3.2(a) shows how a simple conditional operation would be handled by a streaming architecture without conditional streams. The goal of the kernel is to filter all the values that are below p. Since there must be a 1-to-1 correspondence between the input and output streams, all of the input values must be copied to the output. A mask is also generated to inform downstream kernels which elements of the output stream are valid. Since the mask disables processing of some elements, the down stream processor will not be fully utilized.

Figure 3.2(b) shows how the same conditional operation would be handled with conditional output streams. In this situation, the kernel can conditionally write values that are greater than p to the output stream. The output stream is only as large as it needs to be, so there



(a) Without conditional streams

(b) With conditional streams

Figure 3.2: Conditional operations with streaming architectures. Figure (a) shows how a mask must be used to prevent downstream kernels from operating on invalid data. The output stream is the same size as the input stream, and the mask has the same number of elements as the input stream. Figure (b) shows the same simple conditional operation with conditional output streams. In this situation, the output stream is only as large as it needs to be, no additional mask vector is needed, and processor utilization of downstream kernels is increased.

is no wasted space. Additionally, downstream kernels will fully utilize the stream processor since the output stream is densely packed with actually useful data as opposed to sparsely packed data with masking.

3.1.3 Ray Tracing on GPUs

There have been several commodity based GPU ray tracer implementations (Purcell et al., 2002) (Karlsson and Ljungstedt, 2004) (Christen, 2005) (Foley and Sugerman, 2005). The original work by Purcell used a uniform grid triangle bins as its acceleration data structure. Unfortunately, uniform grids do not gracefully handle scenes with a non-uniform distribution of geometric primitives. A large study of ray tracing acceleration data structures by Havran (Havran, 2001) shows that kd-trees handle a wide variety of scenes well. The obvious kd-tree traversal mechanism requires the use of a per-ray stack which is problematic on modern graphics hardware. Foley and Sugerman developed a GPU ray tracer that used kd-trees instead of uniform grids. They presented two separate stack-less traversal algorithms, kd-restart and kd-backtrack.

The kd-restart algorithm modifies the basic kd-tree traversal algorithm by advancing the t_{min}, t_{max} range when rays fall through leaf nodes (Foley and Sugerman, 2005). The rays then *restart* their traversal at the root node of the kd-tree. While this causes some nodes of the data structure to be read more times than necessary, it removes the need for a per-ray stack.

3.2 Architectural Modifications

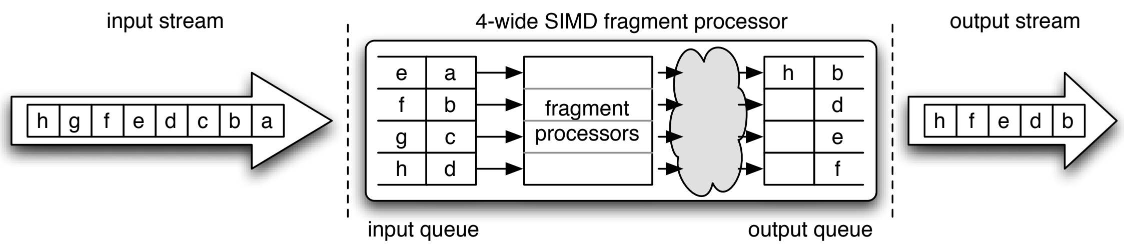


Figure 3.3: Conditional output streams for graphics hardware. Individual elements of the input stream are loaded in parallel by the fragment processors. This is a similar access pattern to stepping through a one dimensional texture. After some amount of computation, the fragment processors test a set of conditions and output only a subset of fragments. The resulting stream is then compacted into a new output stream. Note that ordering is not necessarily preserved in the output stream, but this example is shown ordered for a clearer exposition.

Figure 3.3 gives an overview of how streams would be accessed by the fragment processors. Individual elements of the input stream are loaded in parallel by the fragment processors. This is very similar to stepping through a one-dimensional texture. After some amount of computation, the fragment processors test a set of conditions and only output a subset of fragments. The resulting stream is then compacted into a new output stream.

Figure 3.4 shows a block diagram of the hardware that enables conditional output streams. It acts as a ”stream compaction” unit, and provides a mechanism for fragments to conditionally write to a render target, yet still provide a coherent stream to the memory interface unit. The 2-stage FIFO provides a stage area for data to be compacted before it is sent to the memory controller. Once one of the FIFOs fills, the data can be dispatched to be written to memory while the other FIFO is being backed. Two stages are needed for the situation where the output from the fragment processors would overflow a single-stage output FIFO. Additionally, since modern hardware supports multiple render targets, it is reasonable to allow multiple conditional output streams. While not shown in the figure, multiple 2-stage FIFOs would be needed to compact the different output streams for each render target. Another

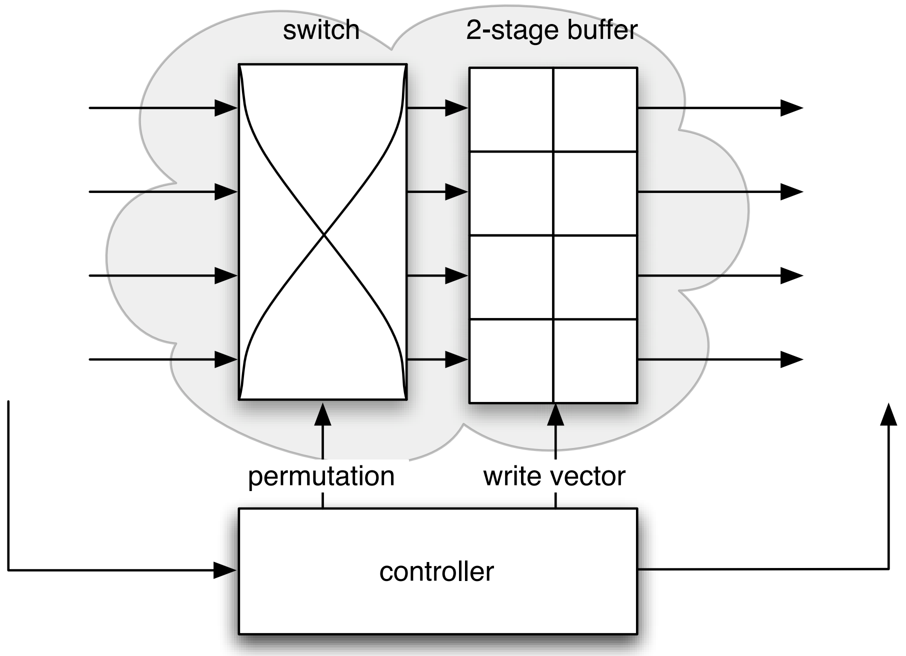


Figure 3.4: Additional hardware needed to implement conditional streams in graphics hardware. The switch permutes its input based on which processor outputs valid data. The output of the switch is written into a 2-stage buffer based on the *write vector* signal, which is provided by the controller (diagram adapted from Kapasi et al.).

possible solution presented by Popa (Popa, 2004) involves the use of a Beneš network.

3.2.1 Decreasing Memory Fragmentation

One drawback to using conditional output streams is the potential fragmentation of memory, since the output size is not known at the time of creation. The hardware must allocate enough space to hold the entire stream for situations where the entire input stream is written to memory. In cases where the input stream could be written to one of n separate streams, n output streams must be allocated, each with a size of the input stream. The special case of only two output streams allows for a simple optimization. In this situation, only one output buffer of size n needs to be allocated. One output stream writes from the beginning of the output buffer, and writes towards the end of the stream. The other output stream begins writing at the end of the buffer, and writes towards the beginning. Since no new fragments are introduced, the two output streams are guaranteed to fit exactly inside the space allocated. The major hardware modification that this optimization would require is the ability of the memory controllers to write memory in either the forward direction or the reverse direction, which should be a trivial modification.

3.3 Basic Streaming Ray Tracing Algorithm

At a high level, the streaming algorithm proceeds as follows. First the scene is rendered using the standard z-buffer algorithm. Reflection rays are then traced into the scene, and the rays are intersected with the scene using the streaming algorithm. Since ordering is not preserved inside the stream, the rays must then be scattered back to screen space, where the resulting fragments are shaded.

The goal of our ray tracing algorithm is to test as many candidate rays against each node in the acceleration data structure as few times as possible, thus conserving bandwidth. While the presented technique uses the kd-restart algorithm of Foley and Sugerman, other acceleration data structures and traversal algorithms are possible.

The ray tracing algorithm proceeds as follows.

1. Rays of given generation are all added to stream S_0. Since S_0 is the initial stream, the root kd-tree node, N_0, is assigned to be tested against S_0. S_0 is then inserted into the *work* queue.

2. A stream S_i is removed from the work queue, and its corresponding kd-tree node N_i is loaded. Unlike in a conventional recursive ray tracer, the rays in S_i are tested only against N_i

3. If the current node is an internal node, then the input stream is split into two separate streams. One stream S_{above} holds the rays *above* the current split plane, which will be tested against the kd-tree node N_{above}. The other stream S_{below} holds the rays *below* the current split plane, which will be tested against the kd-tree node N_{below}. The new output streams are then added to the *work* queue.

If the current node is a leaf node, the rays in S_i are tested against the node's geometry list. Rays that do not intersect any geometry are added to the *restart* stream $S_{restart}$. Rays that intersect geometry are added to a shade stream, which holds rays that are ready to be shaded.

4. The next stream S_{i+1} from the *work* queue is selected, along with the appropriate kd-tree node N_{i+1}. If there are no streams in the *work* queue, then the restart stream $S_{restart}$ is selected along with the root node of the kd-tree. The algorithm proceeds as previously described, until no streams are left.

All rays of the current generation have now been traced. Multiple generations of rays can be processed in this manner if recursive ray tracing is desired. If a unified shader architecture similar to ATI's Xenos (Doggett, 2005) is used, an output stream can be fed back as an input stream for the next iteration of the algorithm with out requiring a round-trip to and from memory. This further reduces bandwidth by eliminating the need to write every stream out to memory. Clearly, this optimization will be limited by resources, and large streams will have to be written to memory.

3.3.1 Hybrid Algorithm

The major benefit of this algorithm is its highly regular access pattern. As with most streaming algorithms, this one reads from and writes to memory in extremely coherent blocks. One potential drawback of this algorithm is that it may create numerous small streams. Foley and Sugerman found that the two major limits on performance for their algorithms were load balancing and the cost of recirculating data. Since the proposed hardware extensions enable true load balancing through conditional streams, a hybrid technique could use the proposed streaming algorithm to process the streams until the streams reach a lower size limit, at which point a modified version of Foley and Sugerman's algorithm that takes advantage of conditional streams instead of using masking would be used, by sorting fragments into separate streams, where each stream only contains fragments in a given state.

3.3.2 Results

Figure 3.1 shows the result of rendering images using our new streaming ray tracing algorithm. In Figure 3.1(a), our algorithm is only used to generate the reflections on the surface of a TT model inside the Sponza scene (both models are distributed as part of PBRT (Pharr and Humphreys, 2004)). This image mimics the situation where the primary ray intersections are calculated using standard z-buffering techniques. Physically correct reflections are then generated using our ray tracing algorithm.

Figure 3.1(b) visualizes the sum of the metric $1/stream\,length$. The color map, shown on the left of the image, ranges from small values, represented by red, to large values, represented by purple. As can be seen, most of the image is red, which indicates that most pixels are processed in relatively large streams. In this image, the worst-case pixel is in the front wheel of the TT model where the spokes meet, and the ray must be restarted forty-six times.

During the rendering of the TT, the average stream size when processing internal nodes was 23.6 rays per stream. The overhead of processing streams with an average size of 23 elements should be roughly the same overhead as that of rendering a triangle with 23 pixels, which is reasonable. The average stream size when processing leaf nodes was 5.3, which means that each triangle of the model was, on average, tested against 5 rays at a time. This result agrees with previous results that have shown that there is a surprising amount of coherence available when ray tracing reflections (Wald et al., 2001). In another test scene, a perfectly reflective Sponza, the average stream size for the fifth bounce was still over thirty rays.

Figure 3.1(c) shows the number of restarts needed to render each pixel; the number of restarts is strongly correlated with the number of kd-tree node traversals needed to render each pixel. As before, the color map is on the left side of the image. The worst case pixel requires 46 restarts and 1324 node traversals, and is again located in the front wheel of the TT model. The minimum number of traversals needed to render the image was 21, and the average number of traversals over the entire image was 259.

Table 3.1 summarizes the results of a comparison of our technique with Foley and Sugerman's. In an effort to normalize the estimates with regard to image and scene size, the bandwidth is presented as bytes per ray per triangle. Using estimates calculated from data

Scene	number of triangles	Bandwidth per (triangle * pixel * frame)
TT in Sponza	**371,544**	**0.022**
Foley - kitchen	110,561	0.52
Foley - robot	71,708	0.89

Table 3.1: Comparison of our technique with the results presented by Foley and Sugerman. All images were rendered at 512x512.

presented by Foley and Sugerman, their algorithm requires 0.52 bytes of data per triangle per pixel to be transferred during the process of rendering a single frame of the kitchen scene and 0.089 bytes of data to render a frame of the robot scene. The images rendered for Figure 3.1 only required 0.022 bytes per triangle per pixel, a decrease in bandwidth by a factor of 23 to 40 times. By using conditional streams, memory bandwidth is dramatically reduced when compared to previous techniques. This is due to two factors. Since masking is not used, the output streams only contain *active* rays, which eliminates the need to recirculate useless data and increases processor utilization. Additionally, conditional streams allow us to load balance the fragment processor by eliminating useless computation. The second factor is that our ray tracing algorithm reduces memory bandwidth by attempting to maximize the number of rays that are tested at once against each kd-tree node, thus reducing the number of times that the node has to be read from memory.

Besides the cost of shading, there are three separate computation costs. The first is the cost of intersecting the rays with the acceleration data structure. The image shown in Figure 3.1 requires 12,704,461 individual ray traversal tests. Recall that only the reflected rays are traced. This constitutes 95% of the intersection tests needed to render the scene. The second cost is associated with the actual ray triangle tests, and the test scene requires only 691,846 ray/leaf node tests (these are the other 5%). The third cost is associated with the need to *scatter* the unsorted computed fragments to the correct location in the framebuffer for shading and display.

To estimate the computational cost of traversing the kd-tree and computing ray-triangle intersections, OpenGL Shading Language (GLSL) shaders were tested on a 3.2 GHz Pentium IV with an ATI X1900XT graphics card. The shaders do not implement conditional streams, since there is no support in current hardware, but instead emulate the functionality as closely

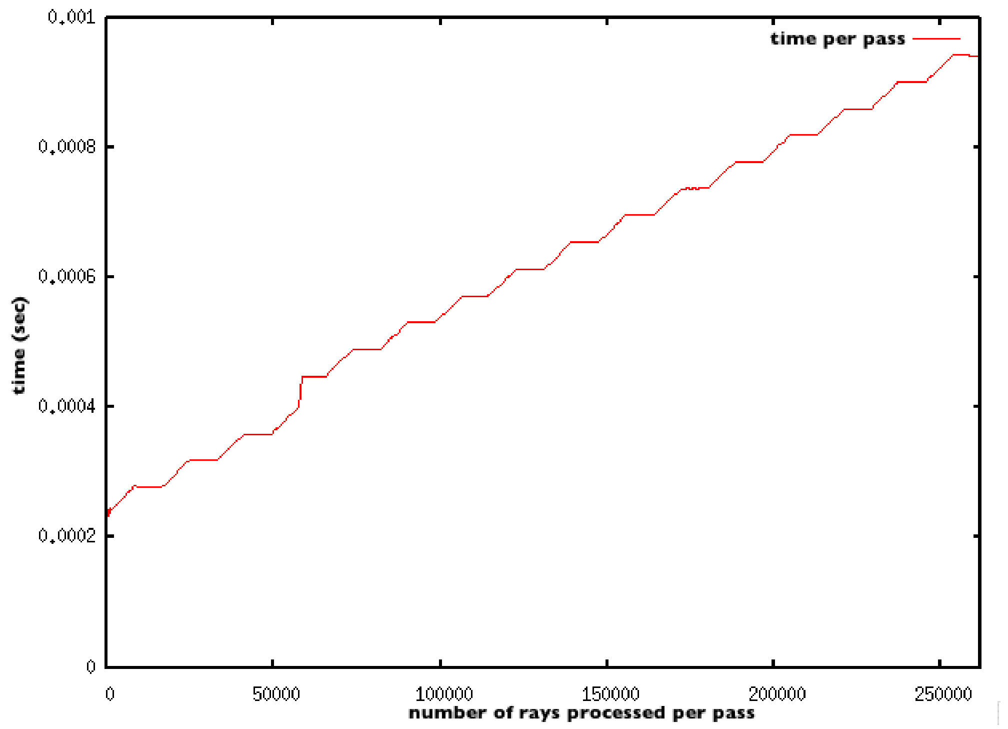


Figure 3.5: Processing time per pass for kd-tree traversal GLSL shader. Data captured on an ATI X1900XT

as possible. Figure 3.5 shows the time it takes to traverse a single node of the kd-tree for a given number of rays using the traversal shader. For 262,144 rays (a 512x512 image), the shader is able to run at approximately 1,000 passes per second. This conservative test shows that the time to render a 4 element stream is only four times faster than the time to process a 262,144 element stream, which is clearly undesirable. Figure 3.6 shows the effective cost of processing of single ray in a stream as the stream size is varied. Because of the overhead associated with rendering a single small stream, the effective processing time is longer for short streams versus long streams. This can be alleviated by processing multiple small streams together in a single pass. For the car inside the Sponza test scene, 12,704,461 individual ray traversal tests need to be run and, assuming that all the streams are bundled together in a pass, completely traversing the kd-tree for the example scene could be performed approximately 10 times per second. Bundling the streams together using the memory optimization presented in Section 3.2.1 is relatively straightforward, but would decrease memory access coherence, since multiple kd-tree nodes not stored adjacently would have to read at the same time. However, a drop in memory coherence can be hidden by processing enough computational threads at the same time to effectively hide memory latency, a technique already used in modern GPUs.

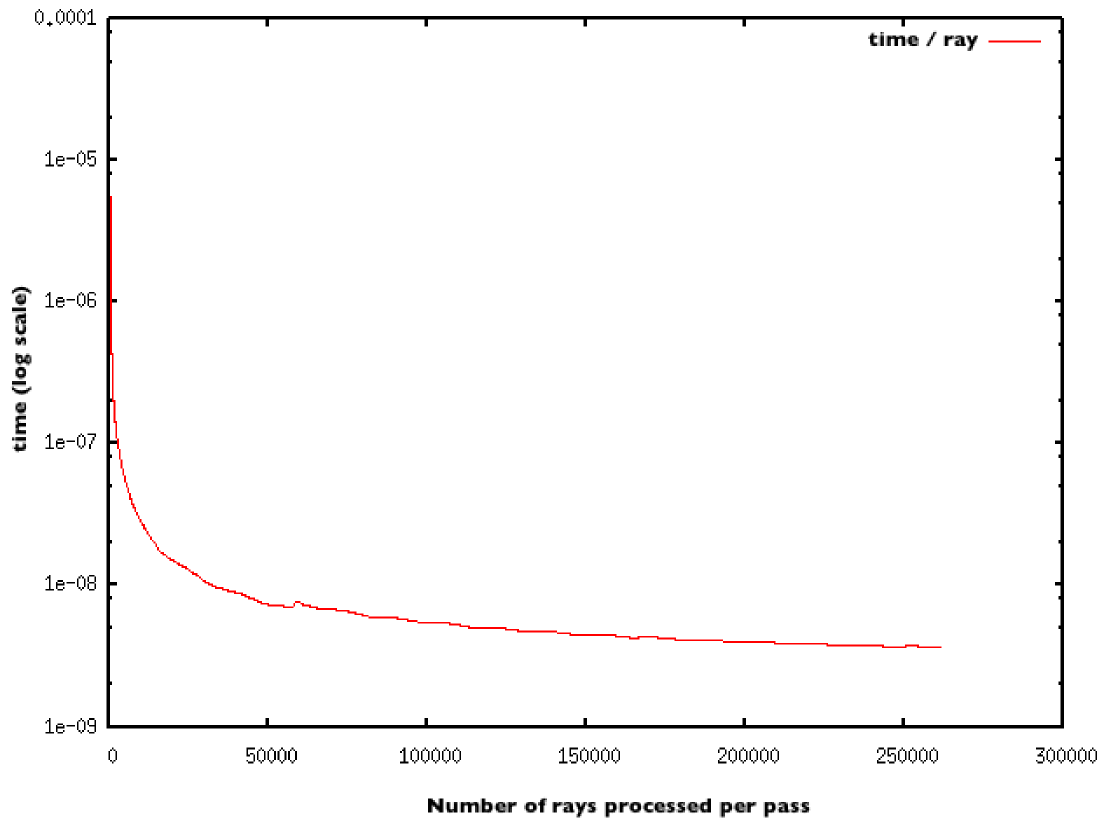


Figure 3.6: Effective processing time per ray for traversal shader. Data captured on an ATI X1900XT

To estimate the computational cost of traversing leaf nodes in the kd-tree and testing ray-triangle intersections, another GLSL shader was written. Dependent texturing is used to reference sets of 3 indices contained in an index texture. The indices are then used to reference the actual vertices of the triangle that are contained in yet another texture. For the triangle intersection code, memory bandwidth is clearly the limiting factor due to the number of texture reads performed versus the amount of computation that needs to be performed. For my test scene, the kd-tree leaf nodes contain an average of 3.5 triangles. For situation where all 512x512 rays are each being tested against 4 different triangles, the X1800XT is still able to test each ray against the 4 distinct triangles at a rate of 434 fps. Since only

Image size	Without scatter (FPS)	With scatter (FPS)	Ratio (with/without)
512x512	831	774	93%
1024x1024	212	192	90%

Table 3.2: Scatter performance using GL_POINTS with a vertex shader on an ATI X1900XT. On current hardware a scatter only causes a slight drop in performance versus pure point rendering.

five percent of the kd-tree intersection tests involve leaf nodes and geometry, the ray-triangle intersection tests are not the limiting factor.

The final step of the ray-tracing algorithm requires the stream of shader fragments to be *scattered* to the correct location of the frame buffer. Tests using GLSL show that the X1900XT can perform a scatter on a fully randomized 512x512 element stream approximately 774 times a second. While this wastes $3/4^{ths}$ of the shader resources since GL_POINTS are used instead of small quads, the scatter operation needs to be performed only once per rendered frame, so it is clearly not the limiting factor. Additionally, some commodity graphics cards support limited scatter operations, albeit without API access in OpenGL or Direct3D.

3.4 Conclusion

I have presented a way to use conditional streaming on a GPU to implement a novel streaming ray-casting algorithm that lowers memory bandwidth and increases processor utilization when compared to current GPU based techniques. One of the biggest drawbacks of the algorithm presented is possible fragmentation of memory, and the overhead associated with processing small streams, although these can be mitigated with techniques presented in this chapter. Additionally, the algorithm implements recursive ray tracing via tail-recursion, which forces data recirculation between ray generations.

CHAPTER 4

Increasing Graphics Hardware Performance Along the Circuit Axis

The third axis on which performance improvements can be made is at the circuit level. The primary aspect of performance addressed by my work along this axis involves energy efficiency which is not only important for mobiles devices, but also for "desktop" devices. At one end of the spectrum, it is important to be energy efficient to maximize the battery life of mobile devices, while at the other end of the spectrum, energy efficiency is important since there are practical physical limits on the amount of power that a chip can dissipate. In this chapter, I present a design paradigm for creating energy efficient graphics hardware components. The design paradigm is illustrated via two different concrete examples.

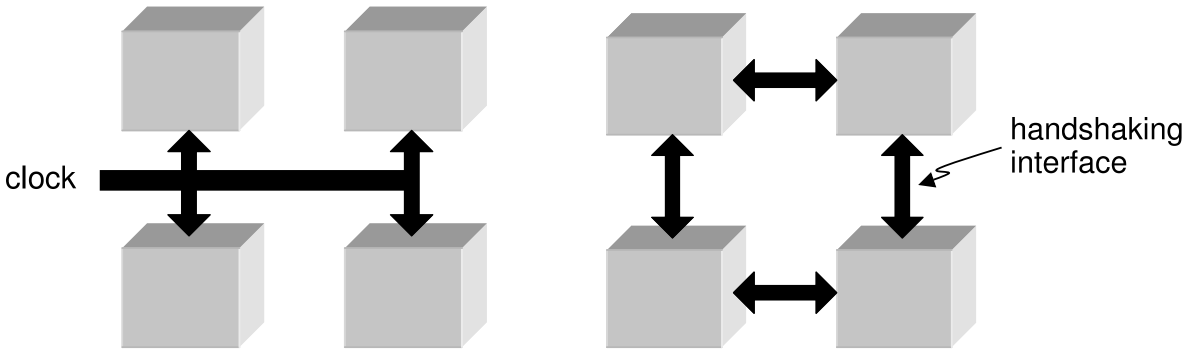
The key idea behind the "compute-on-demand" paradigm is to exploit the data-dependent nature of computation, and to obtain speed and energy improvements by optimizing the design for the common case, instead of assuming worst-case operation. An asynchronous or clockless circuit style is used to facilitate this paradigm. In particular, only those portions of the computation blocks are activated, i.e. using energy, that are actually required for a particular computational operation, thereby saving energy and reducing critical delays. The first example is a z-comparator that takes advantage of data-depencies to reduce energy consumption and average-case latency.

The second example illustrates a novel implementation of a counterflow organization (Sproull et al., 1994) that eliminates the need for complex synchronization and arbitration. This design allows shorter critical paths, and therefore higher operating speed. As an example of our counterflow methodology, we introduce a novel multiplier organization, in which the data bits flow in one direction and the commands are piggybacked on the acknowledgments flowing in the opposite direction. Because of the reduction in power, this implementation is suitable for use in the graphics cores of mobile devices.

The chapter is organized as follows. First background on asynchronous circuits is given. Next, the z-comparator is described in detail. Finally, the Booth multiplier is described in detail.

4.1 Background

As mentioned in Chapter 1, trends in semiconductor devices are posing significant challenges to globally clocked design: (i) distribution of a multi-GigaHertz clock across large dies, (ii) handling of multiple timing domains (e.g. GPUs typically have multiple clocking domains: 2D engine clock, 3D engine clock, and memory clock), (iii) overcoming worst-case performance, (iv) limiting wasteful clock power dissipation (e.g. a modern CPU will waste almost half of its power budget just driving the clock **??**), and (v) interfacing with arbitrary environments.



(a) A synchronous system, featuring centralized control

(b) An asynchronous system, with distributed control

Figure 4.1: Synchronous and asynchronous systems (figure adapted from (Singh, 2001)).

As a result of these problems, an alternative approach—*asynchronous* or "clockless" design—is becoming an increasingly attractive (Berkel et al., 1999). As shown in Figure 4.1, instead of using a global clock, an asynchronous system uses *handshaking* between interacting components to achieve local synchronization.

Since asynchronous design eliminates the wasteful clock power and limits circuit activity to when and where necessary, it has potential significant energy and performance benefits. Performance benefits result because typical asynchronous components are capable of exploiting the data-dependency of completion times (Nowick et al., 1997; Rotem et al., 1999). The designs presented in this chapter use asynchronous logic to increase energy-efficeincy and performance. The next few sections present background information on asynchronous logic (adapted from (Singh, 2001)).

4.1.1 Advantages of Asynchronous Design

Asynchronous design offers several potential advantages:

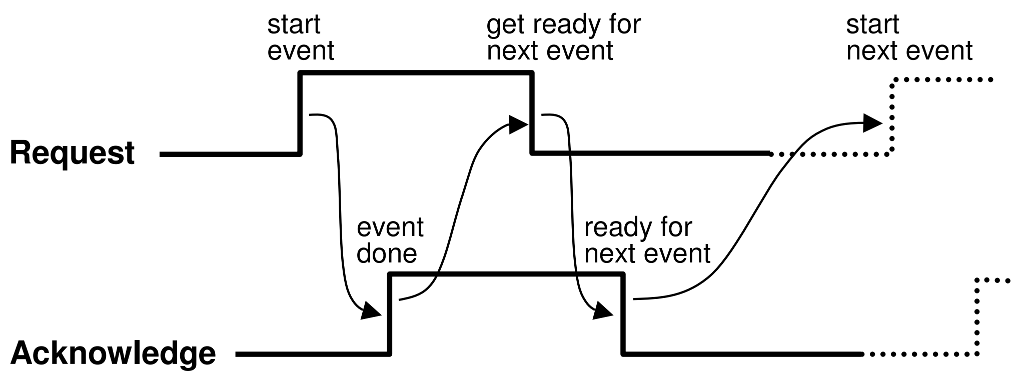
- *Higher performance.* Unlike synchronous systems, which are constrained to worst-case operation, asynchronous systems can potentially obtain average-case performance by taking advantage of data-dependent completion times (Nowick, 1996; Nowick et al., 1997; Rotem et al., 1999; Benes et al., 1998).

- *Lower power.* Asynchronous circuits can provide dramatic reduction in power consumption due to two factors: (i) elimination of clock power dissipation, and (ii) exhibiting switching activity only "on demand." Well-designed asynchronous components will only exhibit switching when new data arrives. This is provides the savings of clock gating (Gowan et al., 1998; Tiwari et al., 1998) at the gate level automatically.

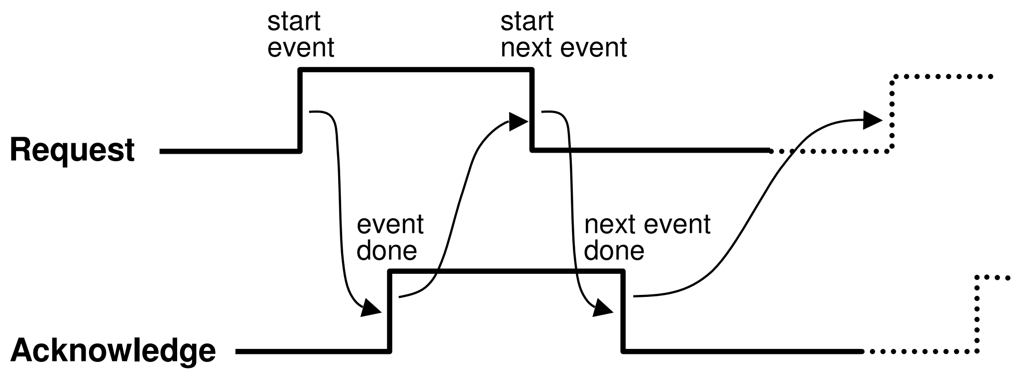
- *Greater modularity and design reusability.* The lack of a global timing constraints potentially makes asynchronous designs more modular and flexible. This allows design reuse, since well-designed asynchronous components with standardized interfaces can easily be connected together (Berkel et al., 1999). This makes asynchronous logic an attractive methodology for implementing designs that easily need to be retargeted to multiple market segments, such as GPUs.

4.1.2 Asynchronous Design Background

The two most basic decisions that must be made when designing an asynchronous systems are determining the type of *handshaking*, or control signaling, to use, and the type of *data*



(a) A four-phase handshake scheme



(b) A two-phase handshake scheme

Figure 4.2: Examples of handshake schemes

encoding, or data representation, to use.

4.1.2.1 Control Signaling

Two of the commonly-used handshake protocols include *four-phase* and *two-phase.* I describe only the four-phase protocol in detail since that is the signaling protocol used in the work described later in the chapter.

Four-Phase Handshaking. A four-phase handshake scheme uses four events per handshake, as shown in Figure 4.2(a) (Seitz, 1980; Birtwistle and Davis, 1995). Normally, only the first two of these events actually communicate useful information: a *request* and an *acknowledge*; the last two events only serve to return the signals back to their deactivated state ("return to zero").

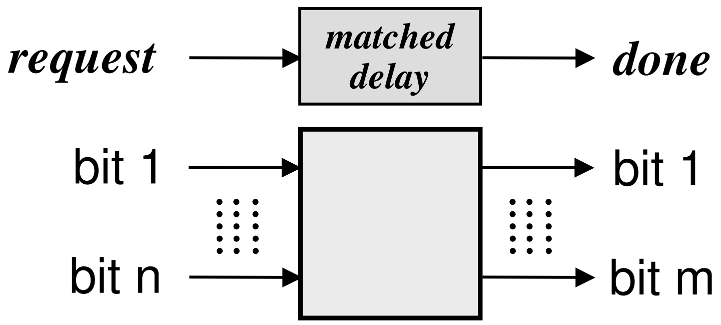


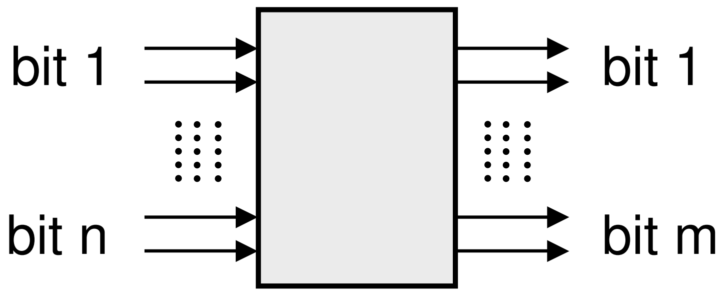
Figure 4.3: A bundled data function block

4.1.2.2 Data Representation

In asynchronous systems, numerous techniques have been proposed to encode data in order to communicate two vital pieces of information: The first is the validity of the data and the second is the actual data itself. Two of the most commonly used schemes are *single-rail*, or bundled data, and *dual-rail* data encoding schemes (Davis and Nowick, 1995).

Bundled Data. Bundled data encoding uses one wire for each bit of the datapath, and an additional wire to indicate validity of data, which is often used as a signal indicating arrival of new data and is often term a "request". To insure correct operation, the request must arrive *after* the data is valid. This constraint can often be met by adding an inline *matched delay* that is designed to delay the request signal until the data is valid and stable.

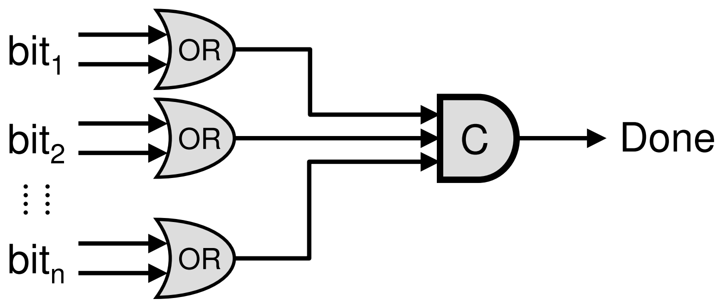
Dual-Rail Encoding. An alternative technique, dual-rail encoding, uses two wires for every data bit. The additional encoding cost allows the data validity and data to be represented together in a unified way (Williams, 1992; Davis and Nowick, 1995), as shown in Figure 4.4(a). Since each data value is represented by two wires, each *bit* can take on four values. The combinations 01 and 10 are used to communicate "1" and "0" values respectively for the data bit. The value 00 is used to represent invalid data, and is sometimes referred to as a "bubble" or "spacer". The final combination 11 is not used and is not an allowed state. In order to detect whether the dual-rail data is valid or not, *completion detectors* are often used.



(a) A dual-rail function block

Dual-rail code	Meaning
00	Data not available (reset spacer)
01	Valid 1
10	Valid 0
11	unused

(b) Dual-rail codes and their meaning



(c) A dual-rail completion detector using OR-gates and a Müller C-element

Figure 4.4: A dual-rail data encoding scheme

4.1.2.3 Completion Detection

Figure 4.4(c) shows a completion detector for a four-phase dual-rail datapath. The individual data bit validity, or non-validity, is checked by OR'ing their two rails together. This generates a vector of values defining which bits are valid. The vector is then fed into a Müller *C-element* (Sutherland, 1989), which generates the data validity signal. A C-element is a standard asynchronous state-holding element that asserts its output when all its inputs are high — all data bits valid — and asserts a low value when all its inputs are low — all data bits invalid. If the inputs are not all equal, the C-element maintains its state. (Upon initialization, the C-element is typically reset low.)

4.1.3 Dynamic Logic

Traditional datapaths are often designed using static CMOS gates, where each gate is composed of complementary pull-up and pull-down networks (Weste and Eshraghian, 1993). Since this requires two complementary sets of gates, a relatively large amount of chip area is used. Additionally, due to the device physics involved, *e.g.,* hole motility, the pMOS transistors used in the pull-up tend to be fairly large and slow devices. An alternative logic style that eliminates the need for the pMOS pull-up networks is called *dynamic,* or *precharge,* logic (Weste and Eshraghian, 1993). Unlike static gates with pull-up networks that are controlled by logic inputs, a dynamic gate has a single pull-up transistor driven by an external control input. Because of its its high-performance potential, dynamic logic is being increasing used in speed-critical portions of modern chips, *e.g.,* arithmetic and logic units (ALU's).

4.1.3.1 Energy Efficiency of Dynamic Logic

Dynamic logic was chosen as the circuit implementation style for our implementations because it can potentially help reduce energy consumption in several ways. First, the lower loading exhibited by dynamic gates can help reduce the switching capacitance. Second, and more significantly, dynamic gates are critical to the concept of *"compute-on-demand"*: the gates use an extra control input, which provides the ability to start and stop computation. As will be shown in Section 4.2, the control of the gate's operation can help obtain dramatic power reduction by preventing switching activity in those parts of a system that are not needed for a particular operation. Finally, as shown in the following subsection, with careful sequencing of control, it is possible to pipeline a dynamic datapath without the need for storage elements between pipeline stages, thereby eliminating wasteful energy dissipation in pipeline latches or registers.

4.1.3.2 Structure and Implementation

Figure 4.6 shows the structure of a general dynamic gate. Much like static logic, a dynamic gate has a pull-down network made of nMOS transistors. However, there is no pull-up network; instead, there is a single pull-up transistor ("precharge device"). Typically, there is also

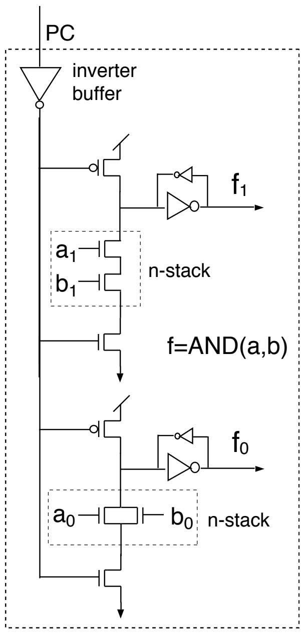


Figure 4.5: A dual-rail AND gate in dynamic logic

an additional nMOS transistor, in series with the pull-down network ("evaluation device" or "foot"). Both the precharge and evaluation devices are controlled by an external input, called PC, which stands for "precharge control." There also two inverters near the output of the gate, one for generating the correct polarity of the output ("output inverter"), and a weaker one that provides feedback to stabilize the output ("keeper").

4.1.3.3 Operation

A dynamic gate has two phases of operation—*precharge* (or *reset*) and *evaluation*—controlled by the PC input. The gate alternates between these phases.

Precharge occurs when PC is asserted *low.* A low PC switches the precharge device on, causing the wire labeled "dynamic node" to go high. As a result, the output inverter resets the gate output to low. The foot device ensures that the pull-down path is cut off during precharge, to avoid the possibility of a short-circuit between the voltage supply and ground.

The evaluation phase occurs when PC is de-asserted *high.* A high PC disables the precharge device, and enables the foot device; this action is called *precharge release.* At this point, the pull-down network processes its inputs, and produces a value that is inverted to form the

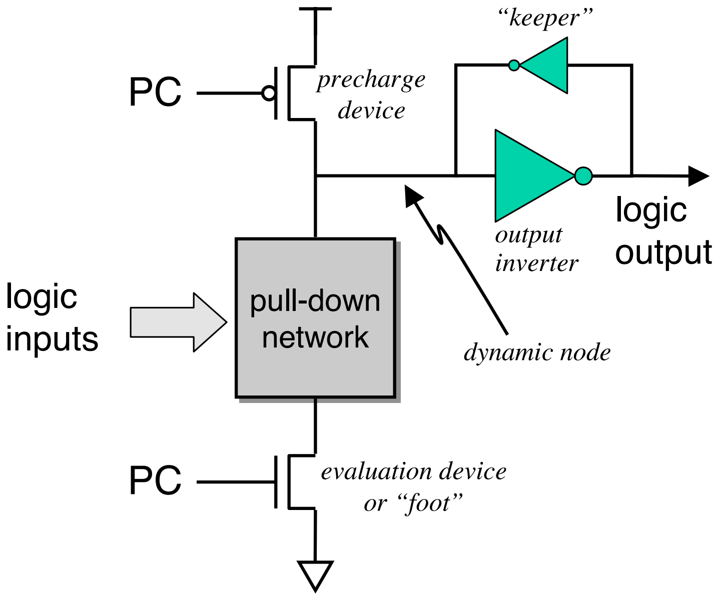


Figure 4.6: A dynamic logic gate (figure adapted from (Singh, 2001)).

gate output. In particular, if the input values are such that the pull-down network becomes conducting, the dynamic node is discharged, and the gate output makes a transition from low to high. For other input values, the output stays low. It is important to note that, once the gate output has changed from low to high, the output stays high *even if the gate inputs are de-asserted.* The output stays high because de-asserting the inputs only cuts off the pull-down network. For the output to reset, the precharge control itself must be asserted.

4.1.4 Asynchronous Pipelining

There are several well-known approaches for asynchronous pipelined circuit implementation. These approaches can be classified along many different dimensions, *e.g.,* static (Sutherland, 1989; Singh and Nowick, 2001) or dynamic logic datapaths (Williams and Horowitz, 1991; Singh and Nowick, 2000b; Singh and Nowick, 2000a); four-phase or two-phase handshaking (Day and Woods, 1995); and timing-independent (robust) (Lines, 1998) or high-performance (Sutherland and Fairbanks, 2001; Schuster et al., 2000).

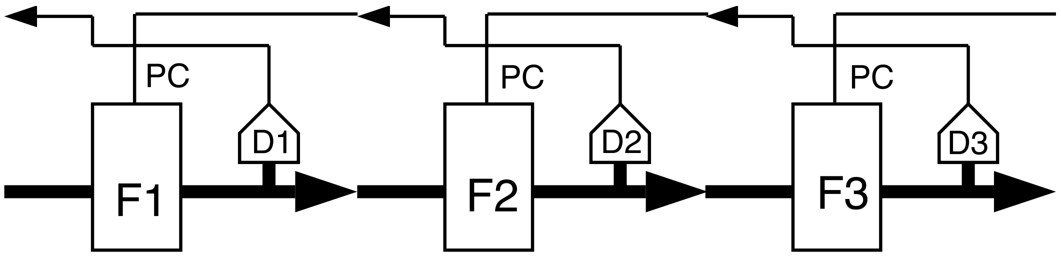


Figure 4.7: Block diagram of a PS0 pipeline

4.1.4.1 Overview

A simple asynchronous pipelining approach introduced by Williams and Horowitz (Williams and Horowitz, 1991; Williams, 1992) referred to as PS0, where "P" stands for "precharge," indicating that the datapath is implemented using dynamic logic; "S" refers to its *s*imple control implementation; and "0" indicates the absence of explicit storage elements in the pipeline. Storage is achieved inside the dynamic logic stages simply by appropriate sequencing of control. The PS0 style is described first because it is arguably the simplest dynamic logic pipeline style reported in literature.

4.1.4.2 Williams' PS0 Pipeline Style

This section provides background on Williams' PS0 pipeline approach.

PS0 Pipeline Structure. Figure 4.7 shows a three-stage fragment of Williams' PS0 pipeline. Each pipeline stage is composed of a dual-rail function block and a completion detector. The completion detectors indicate validity or absence of data at the outputs of the associated function block.

Each function block is implemented using *dynamic logic* gates (Weste and Eshraghian, 1993). The precharge/evaluate control input, PC, of each stage is tied to the output of the next stage's completion detector. Since a precharge logic block can hold its data outputs even when its inputs are reset, it also provides the functionality of an implicit latch. Therefore, a PS0 stage has no explicit latch. Figure 4.5 shows how a dual-rail AND gate, for example, would be implemented in dynamic logic; the dual-rail pair, f_1 and f_0, implements the AND of the dual-rail inputs a_1a_0 and b_1b_0.

Each completion detector verifies the completion of every computation and precharge of its associated function block.

PS0 Pipeline Protocol. The sequencing of pipeline control is quite simple. Stage N is precharged when stage $N+1$ finishes evaluation. Stage N evaluates when stage $N+1$ finishes reset. (Of course, the actual evaluation will commence only after valid data inputs have also been received from stage $N-1$.) This protocol ensures that consecutive data tokens are always separated by reset tokens (or "spacers").

The complete cycle of events for a pipeline stage is derived by observing how a single data token flows through an initially empty pipeline. The sequence of events from one evaluation by stage 1 to the next is: (i) Stage 1 evaluates, then (ii) stage 2 evaluates, then (iii) stage 2's completion detector detects completion of evaluation, and then (iv) stage 1 precharges. At the same time, after completing step (ii), (iii) stage 3 evaluates, then (iv) stage 3's completion detector detects completion of evaluation and initiates the precharge of stage 2, then (v) stage 2 precharges, and finally, (vi) stage 2's completion detector detects completion of precharge, thereby releasing the precharge of stage 1 and enabling stage 1 to evaluate once again. Thus, there are six events in the complete cycle for a stage from one evaluation to the next.

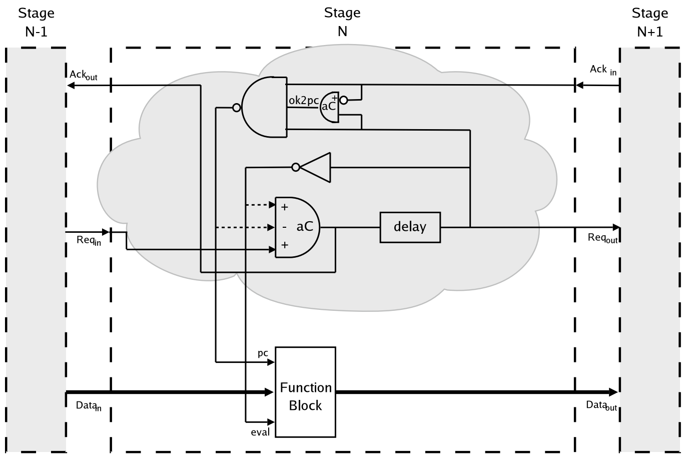
PS0 Pipeline Cycle Time and Latency. The complete cycle for a pipeline stage, traced above, consists of 3 evaluations, 2 completion detections and 1 precharge. The analytical pipeline cycle time, T_{PS0}, therefore is:

$$T_{\mathrm{PS0}} = 3 \cdot t_{\mathrm{Eval}} + 2 \cdot t_{\mathrm{CD}} + t_{\mathrm{Prech}}$$

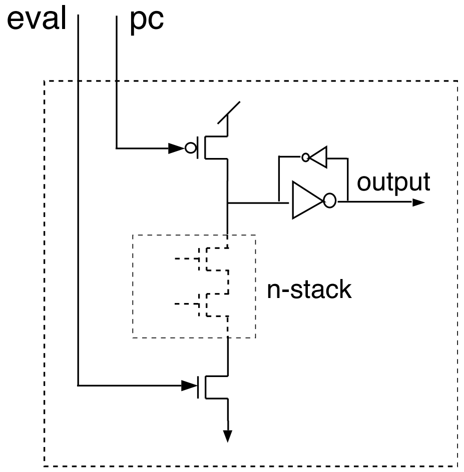
where, t_{Eval} and t_{Prech} are the evaluation and precharge times for each stage, and t_{CD} is the delay through each completion detector.

The per-stage forward latency, L, is defined as the time it takes the first data token, in an initially empty pipeline, to travel from the output of one stage to the output of the next stage. For PS0, the forward latency is simply the evaluation delay of a stage:

$$L_{\mathrm{PS0}} = t_{\mathrm{Eval}}$$



(a) Pipeline fragment showing three stages



(b) Details of a gate within a function block

Figure 4.8: The high-capacity (HC) pipeline style

4.1.5 Singh's High-Capacity Style Pipeline

The implementation style used for the multiplier design is based on the *high-capacity* (HC) asynchronous pipeline style (Singh and Nowick, 2000a; Singh et al., 2002). The HC style is reviewed here; Section 4.4 will present the enhancements to HC that were carried out to meet the design objectives of this portion of the dissertation.

4.1.5.1 Motivation

The HC style was chosen because of its area- as well as energy efficiency. In particular, the HC datapath uses dynamic logic and is *latchless, i.e.,* no explicit storage elements are used between pipeline stages. Instead, the dynamic function blocks themselves provide implicit storage capability through use of staticizers and by means of careful sequencing of control. The absence of latches translates into significant area and energy savings. Further, unlike some other asynchronous latchless dynamic styles (*e.g.,* PS0 (Williams, 1991)), HC does not require intervening "bubble" or "spacer" stages between data items, thereby keeping energy consumption and area low.

4.1.5.2 Overview

The key idea in the HC approach is one of decoupled control: the pull-up and pull-down of the dynamic gates are made separately controllable, as shown in Figure 4.8(b). Therefore, the precharge and evaluate controls can both be simultaneously de-asserted, allowing the gate to enter a special "isolate phase"—between evaluation and precharge—in which its output is protected from further input changes.

4.1.5.3 Structure

Figure 4.8(a) shows a block diagram of a high-capacity pipeline. Each stage consists of three components: *function block,* a *completion generator* and a *stage controller.* The function block is implemented using dynamic logic. It alternately evaluates and precharges, thereby alternately producing data tokens and reset spacers for the next stage. The completion generator indicates completion of the stage's evaluation or precharge. The third component, the stage controller, generates separate *pc* and *eval* signals which control the function block and the completion generator. Figure 4.8(b) shows one gate of a function block in a pipeline stage.

The *bundled data* scheme (Seitz, 1980; Davis and Nowick, 1995) is used to implement the asynchronous datapath. A control signal, *Req*, indicates arrival of new inputs to a stage when it is asserted; precharge of inputs is indicated when *Req* is de-asserted. For correct operation, a suitable matched delay must be inserted to ensure that *Req* arrives *after* the data inputs.

The *completion generator* is implemented using an asymmetric C-element, *aC* (Furber and Liu, 1996). The *aC*'s output, *Done*, is set when the stage has entered its evaluate phase (*eval* is high), and the previous stage has supplied valid data input (completion signal *Req* of previous stage is high). *Done* is reset simply when the stage precharges (*pc* asserted low).

The *stage controller* produces the control signals for the function block and the completion generator. It receives three inputs—the request from the previous stage, the delayed *Done* of the current stage, and the acknowledge from the next stage—and produces the two decoupled control signals, *pc* and *eval*.

4.1.5.4 Implementation

Figure 4.8(a) shows a complete implementation of the stage controller. The two outputs—*pc* and *eval*—and an internal state variable, *ok2pc*, are each implemented using a single gate. The 3-input NAND gate asserts *pc* when three conditions are met: the current stage has completed evaluation, the next stage has also completed its evaluation (indicated by a high ack), and these two stages contain the *same* data token (indicated by a high *ok2pc*). The state variable *ok2pc* is implemented using an asymmetric C-element as follows: *ok2pc* is set when Req_{in} is asserted high and *Done* is de-asserted low; *ok2pc* is reset when Req_{in} is de-asserted low.

An important feature of the HC protocol is that transitions on the *ok2pc* signal are designed to be off the critical path. In particular, while in Figure 4.8(a), *ok2pc* appears to add an extra gate delay to the control path to *pc*, this is not the case: the pipeline protocol allows *ok2pc* to be set in "background mode," so that *ok2pc* is typically set before T gets asserted. As a result, the critical path to *pc* has typically only one gate delay: from input T through the 3-input NAND gate, NAND3, to the output *pc*.

4.1.5.5 Operation

An HC pipeline stage simply cycles through three phases. After it completes its evaluate phase, it enters its isolate phase and subsequently its precharge phase. As soon as precharge is complete, it re-enters the evaluate phase again, completing the cycle.

The introduction of the isolate phase is the key to the new protocol. Once a stage finishes evaluation, it immediately *isolates* itself from its inputs by a self-resetting operation regardless of whether this stage is allowed to enter its precharge phase. As a result, the previous stage can not only precharge, but even safely evaluate the next data token, since the current stage will remain isolated. There are two benefits of this protocol: (i) higher throughput, since a stage N can evaluate the next data item even before stage $N + 1$ has begun to precharge; and (ii) higher capacity for the same reason, since adjacent pipeline stages are now capable of simultaneously holding distinct data tokens, without requiring separation by spacers.

The *ok2pc* state variable is critical to disambiguating between two pipeline states: one in which pipeline stages N and $N+1$ both contain valid data corresponding to the same data token, and the other in which N and $N+1$ contain valid data corresponding to distinct but consecutive data items. In the former case, the protocol ensures that *ok2pc* will be asserted, thereby enabling precharge of stage N. In the latter scenario, *ok2pc*'s value will be unset, thereby correctly preventing precharge of stage N.

4.1.5.6 Performance

If the evaluation and precharge times for a stage are denoted by t_{Eval} and t_{Prech}, and the delay through the NAND and aC elements by t_{NAND3} and t_{aC}, respectively, then the analytical cycle time of the pipeline is given by:

$$T_{\text{HC}} \; = \; t_{\text{Eval}} + t_{\text{Prech}} + t_{\text{aC}} + t_{\text{NAND3}} + t_{\text{INV}}$$

Also, a stage's latency is simply its evaluation delay:

$$L_{\text{HC}} \; = \; t_{\text{Eval}}$$

4.1.6 Power-Performance Trade-Off, and the $E\tau^2$ Metric

The freedom from stringent, hard-to-satisfy timing assumptions in asynchronous implementations greatly facilitates a trade-off between performance and power consumption.

In particular, an approach called *voltage scaling* can be used to reduce power consumption at the expense of some loss of performance. For instance, in lower-end power-aware applications, the performance of a system can often be reduced to a desired level by lowering the supply voltage. However, while the system throughput drops approximately linearly with voltage—within certain voltage limits—the drop in power consumption and radiated noise is more dramatic: they decrease as the square of the voltage.

In order to enable a fair comparison between different implementations of the same system, a composite power-performance metric is often used (Martin et al., 2001; Martin, 2001): *Energy·delay*^{2}, often written as $E\tau^2$. Here, E refers to the energy consumed per operation, and τ is the execution time per operation (or the *cycle time,* which is inverse of the *throughput*).

It is important to note that, in voltage scaling regimes, the $E\tau^2$ metric is more appropriate than simply the energy–delay product $E\tau$. The reason is that the $E\tau^2$ metric is fairly invariant to voltage scaling over a reasonable range of supply voltage, from about 25% below nominal voltage to about 50% above nominal (Martin et al., 2001). In contrast, the energy–delay product for a given implementation varies with the supply voltage, because of the quadratic impact on energy consumed but approximately an inverse linear impact on delay.

4.2 Compute-on-Demand Paradigm

This section introduces the notion of *compute-on-demand* as a design principle for fast and energy-efficient graphics hardware. The key idea is to exploit the data-dependent nature of computation, and to obtain speed and energy improvements by optimizing the design for the common case, instead of assuming worst-case operation. An *asynchronous* or "clockless" circuit style is used to facilitate this paradigm. In particular, only those portions of compute blocks are activated that are actually required for a particular operation, thereby saving energy. In addition, asynchronous components are typically capable of providing data-dependent completion times, thereby potentially obtaining speed improvements.

The design of a z-comparator is presented to illustrate the general compute-on-demand principle. By experimentation, I have determined that, on average, a typical depth comparison requires examination of many fewer bits than the typical 24 to 32 bits of the z value. For example, visibility for the complex frame in Figure 4.9 is determined by only comparing an average of 7.3 bits. Since a typical depth comparator compares all of the bits of z, it performs many unnecessary computations. That wastes energy, and potentially costs extra time. In contrast, the presented novel asynchronous comparator limits energy dissipation by only performing computation as required. To render the frame shown in Figure 4.9, the asynchronous comparator would dissipate 1/4th the energy of an equivalently sized synchronous comparator, while operating 1.67 times faster.

Arguably, only making the z-comparator fast and energy-efficient is not likely to result in any significant improvement in the speed or energy consumption of an entire graphics chip. However, the presented approach to designing the comparator holds promise for other parts of

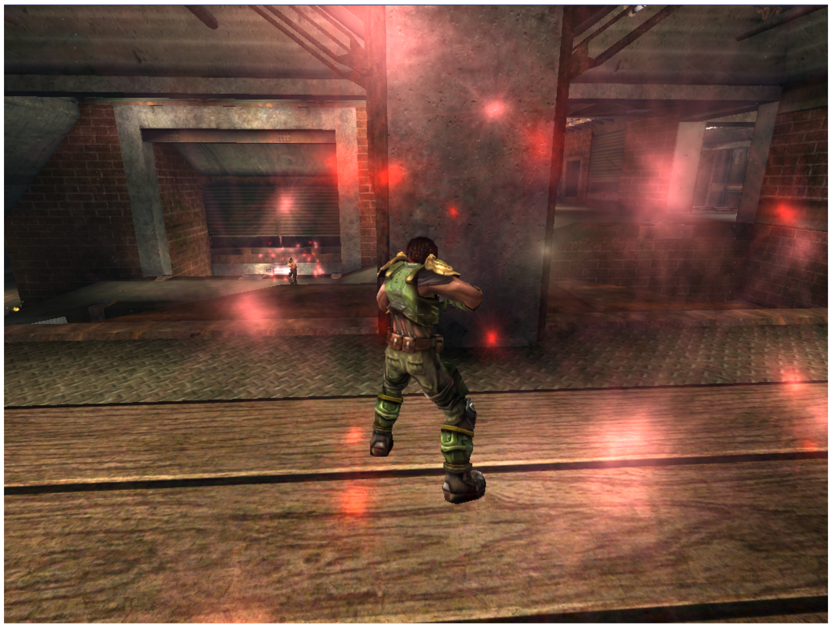


Figure 4.9: A frame from Unreal Tournament 2004. The frame requires 6,768,766 comparisons of incoming fragments with the depth buffer. On average, only the 7.3 most significant bits are actually needed to resolve each comparison.

the chip as well. For instance, certain arithmetic units can be constructed to take advantage of the fact that the entire precision of a number is not always needed (Ekanayake et al., 2005). Further, several asynchronous arithmetic blocks have been designed so as to obtain average-case cycle times and latencies, as opposed to the worst-case operation typical of synchronous components (Nowick et al., 1997; Rotem et al., 1999). Benefits of asynchrony have also been demonstrated in mixed synchronous-asynchronous pipelines (Singh et al., 2002). In sum, the comparator design is offered as an example to make the case that asynchronous circuits and the compute-on-demand paradigm are promising for next-generation graphics hardware.

4.2.1 Previous Work

Most relevant prior work to the presented comparator is by Knittel et al. (Knittel and Schilling, 1995), which introduces two comparator designs for use in a novel approach that folds z-comparisons into z-buffer storage itself.

Their first design is the most similar: the comparison proceeds from the MSB towards the LSB and, in certain cases, their design has data-dependent completion times. However, there are two key distinctions as well. Their design has data-dependent completion only when the result of the z-comparison is "true" (i.e., the new z-value is less than the old z-value); for the other cases, a "false" result is inferred after a *worst-case* delay. In contrast, the comparator is able to exploit data-dependence in all cases, thereby providing a potential speed advantage. The second difference is in the energy consumption. In particular, their design has a global enable signal, which must be broadcast to all bitslices, whereas the presented design asserts the enable for each bitslice only as needed, thereby conserving energy. Moreover, their design uses alternating stages that are dominated by nMOS and pMOS transistors; the p-type stages can represent significant capacitive loading. In contrast, the presented design uses domino logic, which is dominated by n-type devices only, thereby providing a further energy benefit.

Their second design is a modification of the first one to increase concurrency: a 32-bit comparator is decomposed into four 8-bit comparisons whose results are combined using appropriate priority. As a result, speed is improved approximately four-fold, at the cost of higher energy consumption. The comparator design could be similarly decomposed to achieve higher speed at the cost of energy. However, the relative merits highlighted above are likely to remain the same.

Another relevant approach is the energy-efficient comparator of Ponomarev et al. (Ponomarev et al., 2004), proposed for use in superscalar CPUs. Somewhat analogous to the presented "compute-on-demand" functionality, their design has a feature called "dissipate-on-match": their circuit consumes more energy when the operands match, and less for a mismatch. However, while the design examines only those bits that are necessary, their design still examines all bits in parallel. Moreover, their design is dominated by pMOS pass transistors, which imply increased loading, thereby wasting energy.

Most importantly, though, the comparator of (Ponomarev et al., 2004) has a significant limitation: it is useful only for testing equality of two operands, not for less-than or greater-than operations. Similarly, the comparator designs of (Wang et al., 2003) only check for equality. As a result, these designs are not suitable for use as a z-comparator. In contrast, the presented design provides all three comparisons (equal, greater-than or less-than).

4.2.2 Asynchronous Comparator

This section introduces the novel comparator, which generates "less-than," "equal-to," or "greater-than" for a pair of operands.

4.2.2.1 Comparator Architecture

Figure 4.10 shows the overall comparator architecture. The entire computation is bitsliced, with a partial result at each bit position evaluated by a function block implemented using dynamic logic. The precharge/evaluate control of each dynamic function block, labeled "eval" in Figure 4.10 ("PC" in Figure 4.6), is generated by the bitslice to its left.

Computation proceeds from left to right, most significant bit to least significant bit. The key idea is to have evaluation triggered in a bitslice if and only if all the bits to its left, i.e. the more significant bits, have been inspected and found to be identical in the two operands. Thus, this design uses the *smallest leftmost prefix* needed to evaluate the comparison.

An enabled bitslice compares the bits of the two operands at that position, and if they are not equal, it generates the "greater-than" (gt) or "less-than" (lt) output. The "greater-than" and "less-than" outputs of all the bitslices are OR'ed together using a tree of dynamic OR gates, to provide the appropriate result. These trees are in practice quite efficient because dynamic OR gates can typically have fan-ins as high as 6. If the comparison at a bitslice is "equal" (eq), an evaluation request is sent to the next bitslice in the chain, which then similarly evaluates the comparison at the next bit. If the rightmost, least-significant bit evaluates as "equal," then the result of the comparison is reported to be "equal."

4.2.2.2 Comparator Operation: Compute-on-Demand

The asynchronous comparator takes advantage of the fact that the entire width of the operands is not always needed to perform the computation. I have termed this feature *"compute-on-demand,"* since each bitslice only computes if its result is required. By preventing unnecessary partial-result evaluations, the asynchronous comparator limits its energy dissipation to a minimum. In addition, the latency of the comparator is data-dependent: easier comparisons are faster. If the remainder of the graphics pipeline can exploit the shorter

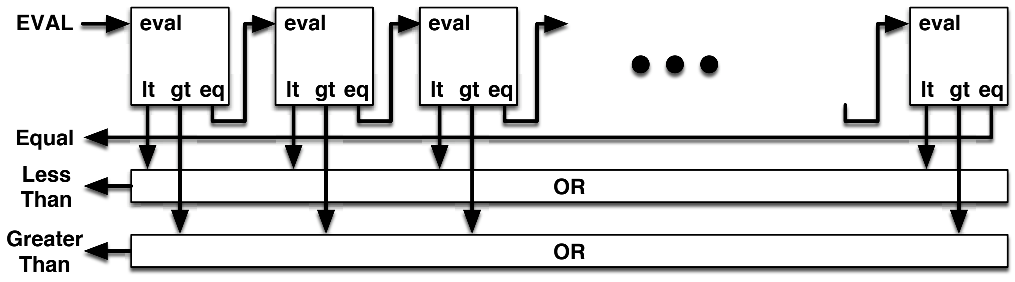


Figure 4.10: A novel "compute-on-demand" comparator

average comparator latency (compared with the worst-case latency of synchronous comparators), then the design also has speed benefits.

If the operands are completely random, then on average only three bits need to be compared to resolve a comparison, regardless of input width (Yun et al., 1997). This is because, as the evaluation proceeds from left to right, the probability that another bit must be inspected progressively falls by half.

In practice, however, the average number of bits inspected will be greater than three for operands that are not random. In particular, when the comparator is used in the z-compare unit, the operands will be incoming fragments whose depth values can exhibit some clustering.

The experiments on a variety of test scenes show that only 6–8 most significant bits are needed, on average, to perform the z-comparison for 24–32 bit depth values. Figure 4.9 shows a single random frame, rendered at a resolution of 1024x768, from the game *Unreal Tournament 2004*. A trace of all depth comparisons was generated using a modified version of the Mesa (Mesa3D, 2006) graphics library. For the frame shown, 6,768,766 comparisons were performed, and on average only the 7.3 most significant bits were needed to evaluate the z-comparisons.

4.2.3 Experimental Results

This section presents the results of electrical simulations of the new asynchronous comparator. To serve as the base case for comparison, a similar comparator was also designed using a clocked approach. Both were designed using the Cadence tool suite, and simulated using

Compute chain	Synchronous		Asynchronous		Async E/ Sync E	Async $E\tau^2$/ Sync $E\tau^2$
	delay, τ (ns)	$E(pJ)$	delay, τ (ns)	$E(pJ)$		
0	0.59	17.36	0.55	1.40	0.080	0.001
4	1.23	18.16	1.69	3.23	0.178	0.030
8	1.85	18.95	2.87	5.36	0.283	0.135
12	2.46	19.74	4.04	7.39	0.374	0.355
16	3.08	20.53	5.20	9.37	0.456	0.714
20	3.69	21.32	6.37	11.40	0.535	1.258
23	$\boxed{4.16}$	22.00	7.24	12.89	0.586	1.778

Table 4.1: 24-bit Comparator results

Spectre in a 0.18μm TSMC CMOS process, at 300K and 1.8V power supply.

For a fair comparison of the *energy efficiency* of the different implementations, a composite energy–performance metric must be used. In particular, the energy consumed per operation, multiplied by the square of the computation delay ($E\tau^2$) is an appropriate metric for systems in which voltage scaling can be used to trade off performance for energy savings (Martin et al., 2001; Martin, 2001).

Simulation Results. Each comparator was simulated with several different input values, and the computational latency and energy consumption were measured. Table 4.1 summarizes the results. The first column lists the number of bits after the MSB that were needed to be examined in order to generate the result, i.e., the length of the shortest leftmost prefix evaluated, excluding the MSB. The remaining columns provide the latency (τ) and energy consumed (E) for each design, along with the ratio of E and $E\tau^2$ for both.

The results clearly demonstrate the data-dependent nature of the comparison completion times. The shortest comparisons are roughly similar for the two implementations: 590 ps for synchronous and 550 ps for asynchronous. The longest comparisons take 4.16 ns and 7.24 ns, respectively. The asynchronous comparator is slower in the worst case, since each successive bitslice is enabled only once it is determined that further computation is required.

The advantage of the asynchronous implementation is quite clear: it truly exhibits variable computation delays. In contrast, the designer of the synchronous implementation will be forced to choose a clock time period that is long enough to accommodate the worst-case delay, 4.16 ns.

Depending upon the operand distribution, the asynchronous implementation can be sig-

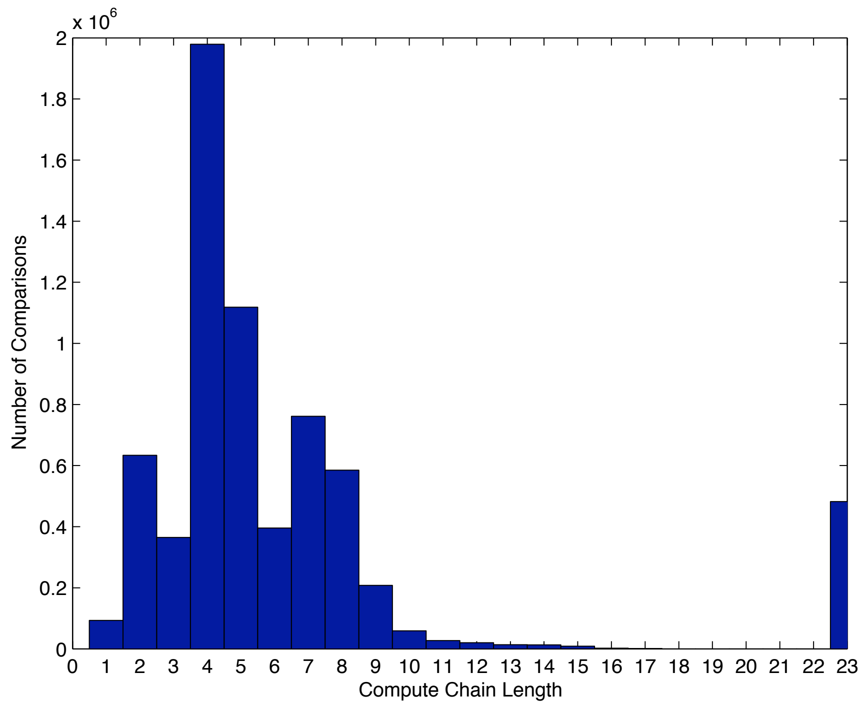


Figure 4.11: Distribution of z-comparison compute chain length for the frame shown in Figure 4.9

nificantly faster than the synchronous one. In particular, several real-world example scenes were analyzed, and I determined that the average compute chain lengths were in the range of 6–8 bits. Figure 4.11 shows the distribution of the compute chain length for the z-comparisons from the frame shown in Figure 4.9. For this frame, the eight most significant bits provide enough information to capture over 85% of the z-comparisons. Only rarely does the comparator need to look beyond 10 bits. Assuming this distribution, the asynchronous comparator would be able to evaluate the over 6 million comparisons 1.67 times faster (2.49 ns average latency) than the synchronous comparator (4.16 ns latency). Of course, the rest of the synchronous graphics pipeline must be capable of exploiting this latency improvement.

The energy advantage of the asynchronous comparator is even more significant. Assuming the distribution of data in Figure 4.11, the asynchronous comparator, on average, dissipates only $1/4^{th}$ the energy of the synchronous comparator. Interestingly, for the shortest compute chain, the asynchronous version is over 12 times more energy-efficient than the synchronous one. In the worst case, for the longest chain, the asynchronous design still dissipates 41% less energy.

4.3 High-Capacity Counterflow Pipelines

This section introduces the High-Capcity Counterflow (HC-CF) pipelining style. As an example of the benefits of HC-CF, an asynchronous radix-4 Booth multiplier architecture that is especially targeted to mobile devices, with the key objectives of high energy efficiency, small chip area, and design reusability is presented. Several emerging consumer electronic applications are likely to increasingly depend on the following capabilities: 3D graphics computation (*e.g.,* cell phones (Kameyama et al., 2003), handheld game consoles), digital signal processing (*e.g.,* portable audio players), and cryptographic processing (*e.g.,* smartcards). In each of these application domains, multiplication is a fundamental operation.

It is important to note the distinction between energy efficiency and power efficiency. The former represents the energy consumed per operation (*e.g.,* nano-Joules/op), whereas the latter refers to energy consumed per unit time (*e.g.,* milli-Watts). For applications where battery lifetimes are critical, energy efficiency is the more relevant metric. On the other hand, power efficiency is more relevant for those desktop or high-performance applications where heat dissipation or supply current are the limiting factors.

The presented multiplier is of the iterative radix-4 Booth type, implemented using asynchronous circuits. An iterative implementation was chosen, as opposed to a combinational array type, for higher area efficiency. A Booth implementation was chosen so as to uniformly handle signed as well as unsigned operands. However, a minor modification to the controller can easily transform the design into a simple (*i.e.,* non-Booth) iterative multiplier. Finally, an asynchronous circuit style was chosen because of its high energy efficiency (Berkel et al., 1999; Markoff, 2001; Tristram, 2001). In particular, asynchronous circuits have the advantage of demand-driven switching activity, effectively providing the benefits of fine-grain clock gating for free. In addition, the greater robustness to timing variations allows an asynchronous circuit to more easily exploit voltage scaling as a technique to further conserve energy.

Besides demonstrating the benifits of HC-CF, the multiplier has several interesting features:

- ***Counterflow Organization:*** A novel multiplier organization is introduced, in which the data bits flow in one direction, and the Booth commands are piggybacked on the acknowledgments flowing in the opposite direction. The presented counterflow organization has significant advantages compared with the counterflow pipeline of Sproull et al. (Sproull et al., 1994) because it eliminates the need for complex synchronization and arbitration the latter requires between two distinct data streams. This feature allows shorter critical paths, and therefore higher operating speed.

- ***Merged Arithmetic/Shifter Unit:*** An architectural optimization is introduced, which merges the arithmetic operations and the shift operation into the same function unit, thereby obtaining significant improvement in area, energy and speed.

- ***Overlapped Execution:*** The entire design is pipelined at the bit-level, which allows overlapped execution of multiple iterations of the Booth algorithm, including across successive multiplications. As a result, both the cycle time per Booth iteration, as well as the overall cycle time per multiplication are significantly improved.

- ***Modular Design:*** The design is quite modular, which allows the implementation to be scaled to arbitrary operand widths, without the need for gate resizing, and without incurring any overhead on iteration time. In particular, Booth commands are relayed from one stage to the next, instead of being broadcast across the entire width of the operand, thereby allowing for a constant Booth iteration time regardless of operand widths. Further, a *sentinel*-based approach is used to determine the termination condition (*i.e.,* all multiplier bits have been consumed), instead of using a counter. As a result, the Booth controller implementation becomes independent of operand widths.

- ***Precision-Energy Trade-Off:*** Finally, the architecture can be easily modified to allow dynamic specification of operand widths, *i.e.,* successive operations of a given multiplier implementation could operate upon different word lengths. This feature could potentially facilitate a dynamic trade-off between computation precision and energy consumption.

4.3.1 Related Work

Several asynchronous multipliers has been reported in literature, both array as well as iterative. It was shown in (Wang et al., 2001) that array multipliers not only have lower latencies, but may also have better energy efficiency than iterative multipliers. (Bartlett and Grass, 1999) presents a novel design of a bundled-data array concurrent multiply-accumulator unit, which reduces power consumption by eliminating unnecessary evaluation of certain partial products (*i.e.,* those corresponding to a zero value for the multiplier bit). By taking advantage of data-dependent evaluation times, their design was able to improve average throughput by 14% when compared to an equivalent synchronous design.

A number of iterative multipliers have been introduced recently. In (Killpack et al., 2001), an area-efficient low-power multiplier is described for use in a hearing aid. (Kearney and Bergmann, 1997) targets both array and iterative multiplication, and is able to show a 20% improvement for a bundled-data self-timed multiplier compared to an equivalent synchronous one.

Several iterative implementations increase the operating speed by processing more than one multiplier bit per iteration. For example, (Kim, 1999) reported a 32x32-bit iterative modified-Booth multiplier, using a new 4-phase asynchronous handshaking scheme. Their design uses two CSA adders and two 2-bit Booth encoders to reduce the number of iterations by half. A high-throughput iterative multiplier is presented in (Shin et al., 2001), which produces the product in $n/4$ iterations; however, it takes more than twice the area of a shift-and-add iterative multiplier.

Recently, (Efthymiou et al., 2004a) has proposed several multiplier implementations, including both the original radix-2 Booth algorithm, as well as the radix-4 Booth algorithm. The key novelty of their radix-2 implementation is that it is able to exploit data dependency to speed up its operation: it skips over arbitrarily long runs of ones and zeros in the multiplier operand, instead of performing sequential single shifts. However, due to the added complexity, its iteration time is actually longer, and therefore it exhibits performance advantages only for certain corner cases. Their radix-4 implementation does not exploit data dependency, but still obtains fairly good operating speed: 1.2 ns cycle time per Booth iteration, in 0.18μ

technology.

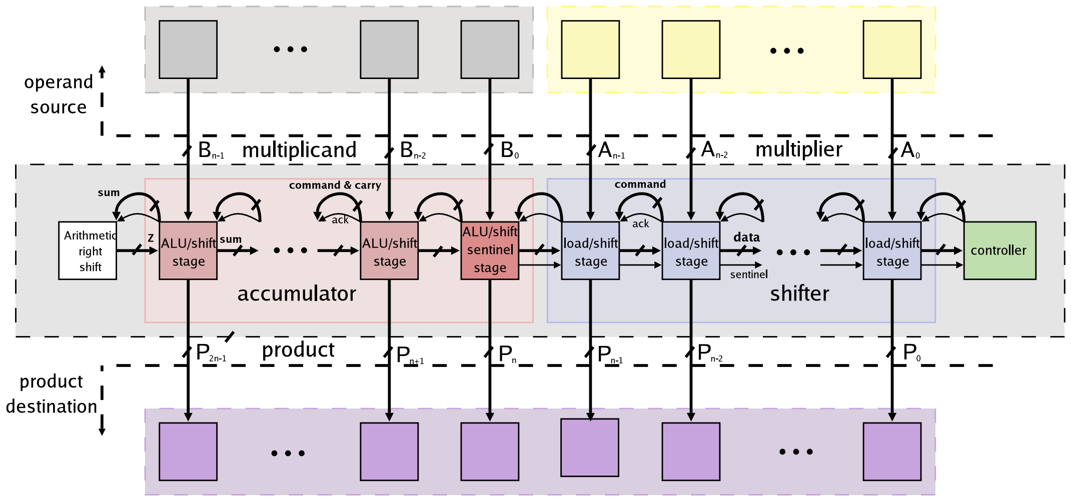


Figure 4.12: The counterflow Booth multiplier

In this section, the principal comparison of the presented multiplier will be to the radix-4 implementation of (Efthymiou et al., 2004a), and the radix-2 implementation of (Hensley et al., 2004).

4.3.2 Multiplier Design

The following sections present the new multiplier design. Section **??** presents the overall architecture, highlighting a novel counterflow organization. Next, Section 4.3.4 discusses the operation of the multiplier, which performs overlapped execution of multiple Booth iterations. Finally, Section 4.3.5 presents some of the key details of the implementation, including a novel asynchronous pipeline handshake style which builds upon the HC style of Section 4.1.5.

4.3.3 Multiplier Architecture

4.3.3.1 Overview

Figure 4.12 shows the overall architecture of the Booth multiplier. The new multiplier has a novel counterflow organization: the Booth commands (*i.e.,* add, add 2x, subtract, subtract 2x, and shift) are bit-level pipelined, *i.e.,* relayed from one bit to another. In contrast, existing iterative organizations involve a broadcast of the command to all bits, which makes those designs less scalable than mine. Another feature of the design is the folding together of the ALU and shifter units, resulting in a simple linear pipeline with area and energy advantages. Finally, the multiplier allows overlapped execution of multiple iterations of the Booth algorithm, including across successive multiplications.

The design of the multiplier is now presented in detail.

4.3.3.2 Novel Counterflow Organization

The multiplier has a *counterflow* organization: data and commands flow in opposite directions. In particular, data bits flow from left to right in the pipeline, whereas commands generated by the Booth controller (*i.e.,* add, add 2x, subtract, subtract 2x, or shift) flow from right to left. Wherever carry bits are generated, as a result of an add or subtract command, they are embedded in, and considered part of, the command itself and relayed to the stage on the left.

The flow of data and commands is interlocked to achieve correct operation. In particular, a data bit flows through a processing stage (*i.e.,* moves right from its input side to its output side) only after that stage has received the associated Booth command. Similarly, a command is relayed from a stage to its left neighbor only after it has interacted with valid data.

My counterflow approach allows data and command to transform each other when they interact. In particular, when data is evaluated by a pipeline stage, the actual operation performed on it depends not only on the functional implementation of the stage, but also on the command received by the stage. Similarly, this approach permits a command to be arbitrarily transformed by the data it interacts with, before it is relayed to the left neighbor. In this particular implementation of a Booth multiplier, the command transformation applies to the carry bits which are embedded within the command: the carry bits are replaced by the new carry bits that are generated when the Booth command interacts with the incoming data.

A key novelty of this architecture is that it performs overlapped execution of multiple iterations of the Booth algorithm. In particular, each command that is inserted by the Booth controller into the right end of the counterflow pipeline effectively performs one iteration of the Booth algorithm as it flows from right to left through the pipeline. However, the bit-level pipelined architecture enables multiple commands to be simultaneously "in flight," thereby effectively allowing multiple iterations of the algorithm to be overlapped. For instance, the lower significant bits of the accumulated result, near the right end of the pipeline, can commence a subsequent Booth iteration by interacting with a later command, while the higher significant bits are still waiting to complete earlier commands.

***Comparison with Counterflow Pipeline of Sproull* et al.** It must be emphasized that the presented counterflow pipeline organization is quite different from another counterflow organization proposed by Sproull et al. (Sproull et al., 1994). In particular, the architecture in (Sproull et al., 1994) uses two distinct pipelines to carry two different data streams in opposite directions, and introduces interlocks to allow the two streams to interact. A drawback of their approach is that arbiters are required between the two pipelines to ensure that corresponding data packets in the two streams do not "skip past" each other, leading to significant implementation complexity and also non-determinism in the system's operation. In contrast, my approach simply "piggybacks" commands on top of the acknowledge signals already required for asynchronous handshaking, and thereby does not suffer from these drawbacks.

4.3.3.3 Architectural Optimization: Folding Arithmetic Unit into Shifter

An architectural optimization was used to obtain significant improvement in area, speed, as well as power consumption: the arithmetic operations (*i.e.,* add and subtract) and the shift operation were merged into the same function unit.

Figure 4.13 shows the block diagram of a folded ALU/shift stage. Each such stage has three input sources and two output destinations. The first input stream, representing the current accumulated result at that bit position (labeled Z), enters the block from the left. The second input is applied to the top of the stage, and represents the corresponding constant multiplicand bit (labeled B). The third stream represents the Booth commands, along with embedded input carry bits (labeled C_{in}), and is accepted from the right. As a result of the command, the stage generates the new accumulated bit, and communicates it to the right, effectively causing a shift operation as well. The stage also produces a second output, which consists of the Booth command that was just executed, along with the *new* value of the carry bit (labeled C_{out}); this second result is communicated to the left.

The operation of the new ALU/shift stage is quite simple. Whenever it receives a Booth command from its right neighbor, and data from its left neighbor, it performs the command on the data, and transfers the result to its right neighbor. Thus, every command effectively causes a shift operation as well. If the command processed was a *shift* command, then the data is simply passed along unmodified; otherwise, for *add* and *subtract* commands, the multiplicand (B) and input carry (C_{in}) bits are combined with the data (Z) bit to generate the results.

The folding in of the ALU into the shifter has several advantages: (i) lower area, because no explicit shifter unit is required; (ii) faster operation, because the results of an arithmetic operation are immediately available for a subsequent arithmetic operation (in the next stage), thereby allowing shorter iteration times; and (iii) better energy efficiency because overall there is less movement of data, and hence fewer transistors are switched.

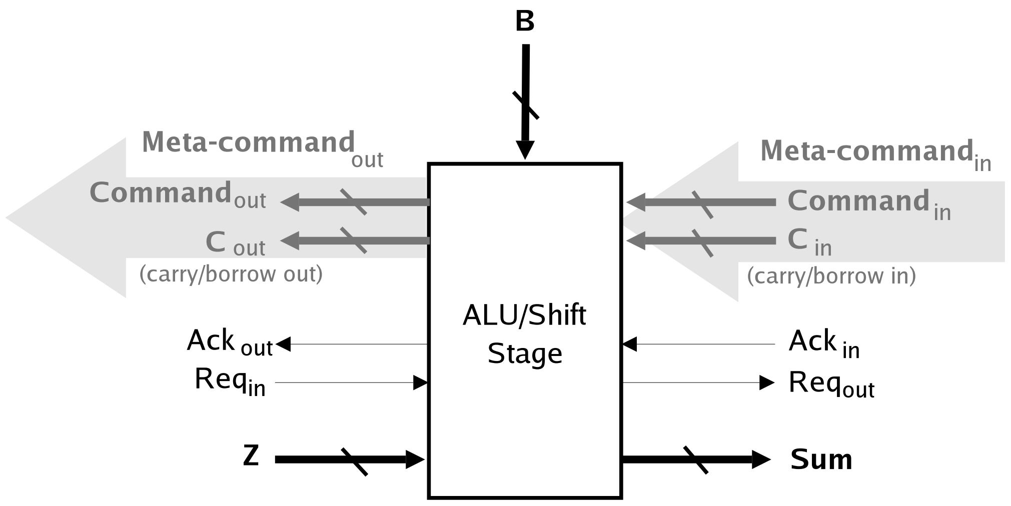


Figure 4.13: The merged arithmetic/shift unit

4.3.3.4 Command Representation

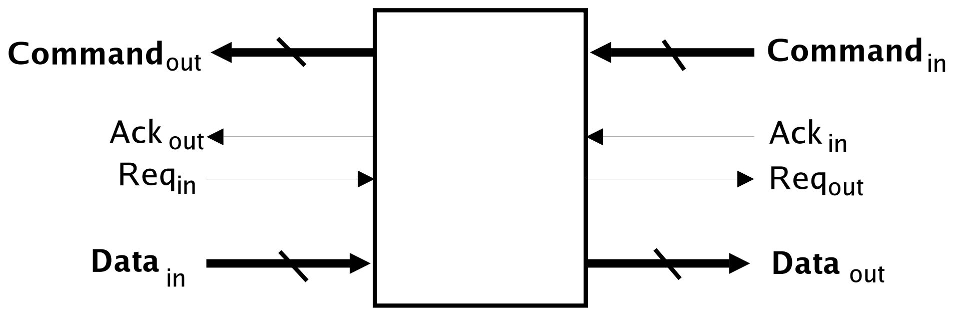
The commands that are generated by the Booth controller are represented using a 1-of-6 (*i.e., one-hot*) encoding that is delay-insensitive. The six representable commands are *initialize*, *add*, *add 2x*, *subtract*, *subtract 2x*, and *shift*. Besides being delay-insensitive, the encoding has two advantages: (i) no decoding circuitry required, and (ii) good energy efficiency, because each command causes switching activity on only one wire.

When a command reaches the left half of the multiplier pipeline (*i.e.,* the rightmost ALU/shift stage), the 1-of-6 encoding for the command is augmented by a 1-of-2 code to allow a carry bit to be also carried within the command (see Figure 4.13), making each command a 8-bit value (labeled "Meta-command" in the figure).

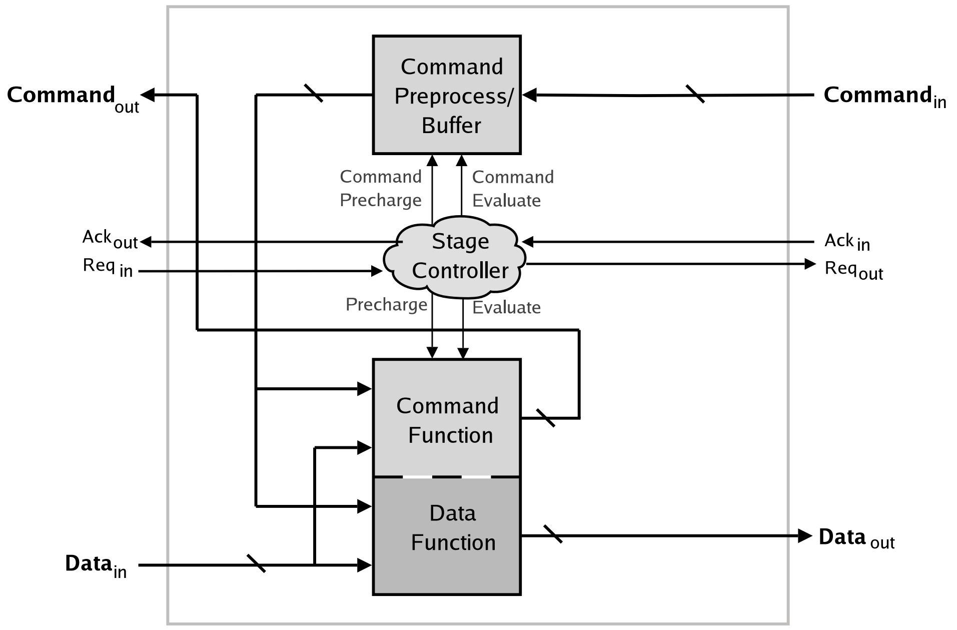
4.3.3.5 Data Representation

Each data bit is also represented using a 1-of-n encoding. In the left half of the pipeline, a 1-of-2 (or "dual-rail") encoding is used: one wire represents the logic "1" value, another wire represents the logic "0" value. In the right half of the pipeline, however, a 1-of-3 encoding is necessary because an additional value must be represented: a *sentinel* value, which encodes a terminating condition. When a stage contains the sentinel, stages to the right hold the remaining bits of the multiplier. Once the sentinel reaches the controller, the controller senses the terminating condition, and stops issuing further Booth commands. Subsequently, once the results of the multiplication have been consumed, the multiplier is ready to process its next set of operands.

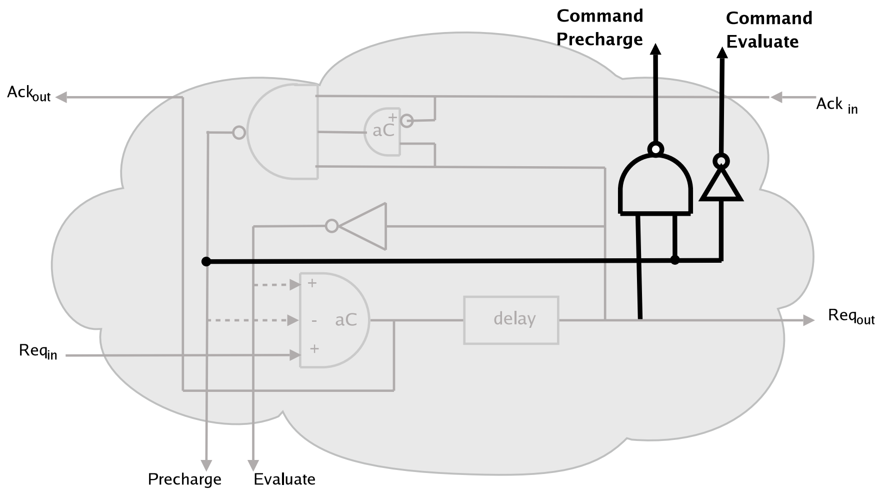
There are two significant benefits of using a sentinel-based encoding: (i) the design of the Booth controller is greatly simplified, *i.e.,* no counter is required, and (ii) the length of the multiplier can be dynamically specified by providing the sentinel in the appropriate bit position. While the current implementation uses a fixed position for the sentinel, Section **??** outlines how the implementation can be easily modifed to provide the ability to dynamically specify the length of both the multiplicand and the multiplier, independent of each other. This approach can potentially facilitate a *dynamic trade-off* between energy consumption and the precision of computation.



(a) Block diagram



(b) Detailed view



(c) Stage controller

Figure 4.14: Pipeline Handshake Controller

4.3.4 Operation

Computation proceeds in three phases: initialization, execution, and termination.

4.3.4.1 Initialization

The controller starts computation by issuing the *initialize* command. This command effectively copies the multiplier operand into the right half of the multiplier pipeline (see Figure 4.12). In particular, when a stage in the right half of the pipeline (*i.e.,* a load/shift stage) receives the *initialize* command, it loads the multiplier value at that bit position from the A input on its top, and passes the same command to its left neighbor.

When the *initialize* command reaches the sentinel stage, it causes that stage's output to get initialized to the sentinel value. This value marks the position immediately to the left of the most significant bit of the multiplier, and represents a termination flag that the Booth controller can sense.

Finally, as the *initialize* command reaches each ALU/shift stage, it causes both the sum and the carry output of the stage to be initialized to the logic "0" value, effectively clearing the contents of the accumulator so that computation may begin.

Actually, the *initialize* command serves another purpose which was omitted from the discussion here: to indicate completion of the *previous* computation. This function is explained later in Section 4.3.4.4, when initialization for the next round of computation is discussed.

4.3.4.2 Execution

After the initialization, the controller generates successive Booth commands: *add*, *add 2x*, *subtract*, *subtract 2x*, or *shift*. Each command corresponds to one iteration of the Booth algorithm. Since the operations are pipelined, multiple commands could be flowing through the pipeline, effectively causing multiple iterations of the Booth algorithm to be executed concurrently.

The load/shift stages in the right half of the multiplier pipeline interpret each of these commands as a *shift* command, and cause their contents to shift one position to the right.

When the command reaches an ALU/shift stage, that stage performs an arithmetic operation if specified, and a shift operation. The stage also generates a carry out that is bundled with the Booth command and relayed to its left neighbor.

4.3.4.3 Termination

When the sentinel reaches the Booth controller, the computation terminates, and the controller stops issuing further commands. The controller's internal state and history bit are cleared, and it prepares for the next set of operands. Strictly, though, at this point, there can be some commands that are still flowing throwing the pipeline. The *initialization* phase of the next iteration is used to handle this situation, as described below.

4.3.4.4 Initialization (next round of computation)

Upon termination, the Booth controller re-initializes the multiplier by generating a new *initialize* command. In addition to what was described above in Section 4.3.4.1, the *initialize* command also serves the purpose of ensuring that all prior Booth commands that are still flowing through the pipeline are correctly completed before the multiplication result is read.

In particular, when the *initialize* command reaches any pipeline stage, it causes that stage to copy the output of its left neighbor onto its P output, which represents the final product value for that radix-4 digit position, in dual-rail form (see Figure 4.12). Taken together, $P_{2n-1} \cdots P_0$ represents the result of the multiplication, where n is the number of radix-4 digits in the operands (*i.e.,* $2n$-bit operands). Even though the lower significant product bits are produced earlier than the higher significant ones, the dual-rail encoding of P_i ensures that the completion of the computation and validity of the result are correctly and robustly indicated.

4.3.4.5 Overlapped Execution of Consecutive Computations

Just as successive iterations of the *same* computation are executed in an overlapped fashion, the implementation allows an overlap between the last (or last few) iterations of one computation, with the first (or first few) iterations of the *next* computation. In particular, the Booth controller immediately commences issuing new Booth commands for the next round of computation, even though one or more commands for the previous computation may still be flowing leftward through the pipeline.

This ability to overlap successive computations is quite advantageous, resulting in significantly reduced latency for the multiplier. The results of the experiments indicate that the benefit is around a 60% reduction in latency.

4.3.5 Implementation

4.3.5.1 Pipeline Handshake Circuits

The pipeline handshake circuits used in the multiplier implementation are based closely on the HC style of (Singh and Nowick, 2000a) (see Section 4.1.5), but include a significant enhancement to enable bi-directional communication, which in turn is critical to enabling the counterflow organization.

Figure 4.14(a) shows the top-level view of the new pipeline stage. Compared with Figure **??**, the new stage has an extra input and an extra output: it receives $Command_{in}$ from its right neighbor, along with Ack_{in}, and similarly produces $Command_{out}$ for its left neighbor, along with Ack_{out}.

Figure 4.18(a) shows the internal organization of a generic stage of the new pipeline. There is a key difference with respect to the HC pipeline of Section 4.1.5: a buffer must be added to store the incoming command, because the command arrives at the start of a precharge phase, but it will be needed in a subsequent evaluation. During that evaluation, the command is combined with the input data to not only produce output data, but also to generate a new command for the left neighbor.

As an optimization, some preprocessing of the command is performed when it is stored in the command buffer: *e.g.,* since operand B is always available even if A has not yet been received from the left neighbor, the command could actually be combined with B to produce an intermediate result. As a result, the main function block, labeled “Data Function,” only needs to combine the intermediate result with operand A, resulting is reduced complexity for the function block. This optimization has the benefit of speeding up the critical forward path through the pipeline; the command buffering and preprocessing is off of the critical path.

Finally, Figure 4.18(b) shows the actual modification made to the HC handshake circuit. The new handshake circuit produces two additional outputs—*command precharge* and *command evaluate*—which serve as the control signals for the command buffer. The figure highlights the additional gates needed to generate the new signals. The function block's precharge signal triggers the evaluation of the command buffer, causing the incoming command to be stored. The command buffer is subsequently precharged only after the command is "consumed" by the function block upon its next evaluation. The new inverter and NAND gate directly implement this functionality.

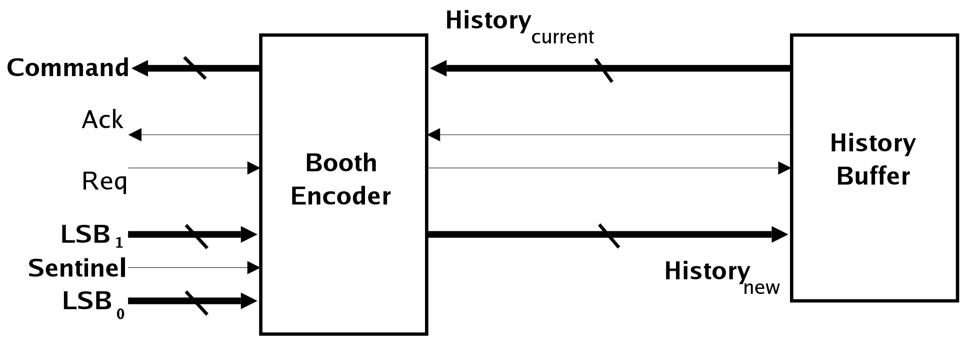


Figure 4.15: Booth Controller

		History bit	
LSB1	LSB0	0	1
0	0	*shift*	*add*
0	1	*add*	*add 2x*
1	0	*subtract 2x*	*subtract*
1	1	*subtract*	*shift*
-	sentinel	*init*	*init*

Figure 4.16: Booth commands

4.3.5.2 Booth Controller

Figure 4.15 shows the block diagram of the Booth controller. The history buffer stores a copy of the most recently examined multiplier bit. This history bit is communicated to the Booth encoder stage as a command. The Booth encoder processes this history bit along with the two new least significant multiplier bits—LSB1 and LSB0—and generates the appropriate

Booth command, as summarized in the truth table of Figure 4.16. This truth table is directly implemented using a dynamic logic function block.

4.3.6 Spice Simulation Results

The performance of the multiplier has been quantified through Spice simulations in a 0.18μm TSMC process, at 1.8V nominal supply voltage. The design takes 640–650ps per Booth iteration, regardless of the operand widths, thereby demonstrating the scalability of the presented approach. The overall computation time varies linearly with the width of the operand: *e.g.,* about 2.6 ns for 8-bit operands, and approximately 10.4ns for 32-bit operands. The energy consumed per multiplication varies as approximately square of the operand width, as expected: 0.11nJ for 8-bit and 1.42nJ for 32-bit multiplications. Finally, simulations performed at reduced supply voltages of 1.5V and 1.0V demonstrated that the multiplier operates correctly at lower voltages, and is able to trade off some performance for even higher energy efficiency.

Compared with several asynchronous Booth multiplier designs reported recently (Efthymiou et al., 2004b), this iteration time represents a nearly *two-fold* (1.95x) increase in throughput over the fastest radix-4 Booth multiplier reported in (Efthymiou et al., 2004b). Although the multiplier consumes more energy per multiplication than the one in (Efthymiou et al., 2004b), it actually has a superior energy-delay^2 ($E\tau^2$) product: 9.65 nJ $\cdot$ (ns)^2 compared with 14.8 nJ $\cdot$ (ns)^2. Experimental results indicate that the new multiplier outperforms the design of (Hensley et al., 2004): 3.3x higher throughput is obtained in the new design while consuming 2.2x lower energy consumption.

4.3.7 Dynamic Precision-Energy Trade-Off

This section outlines an approach to achieving a dynamic trade-off between computation precision and energy consumption in the multiplier architecture. Only a brief sketch of this idea is provided here; experimental validation is part of ongoing work.

Figure 4.17 shows the overall architecture proposed for handling variable width operands. There are several key differences between this picture and Figure 4.12.

First, the variable-precision architecture employs full ALU/shift units in each multiplier stage. In contrast, in Figure 4.12, the right half of the multiplier only had simpler shift units, with no ALU capability. This modification is required because the right half of the multiplier pipeline could now be involved in ALU computations for short operands.

Second, two sentinels are now needed: one to demarcate the MSB-end of the multiplier, and another to indicate the MSB-end of the multiplicand. The two operands are now supplied packed together as a single word.

Third, the function blocks in the ALU stages, which generate/relay the command for their left neighbor need a slight modification. In particular, when an add or subtract Booth command enters the multiplier from the right end, it is initially regarded by all the ALU units it traverses as simply a shift command. However, as the Booth command passes the first sentinel, its interpretation is changed to that of the actual add or subtract operation. As a result, the subsequent ALU stages it traverses while flowing leftward regard it as an actual ALU operation. Finally, when the command reaches the second sentinel, the ALU stage acts simply as an arithmetic right-shift unit, and the command is not relayed to any of the stages to the left.

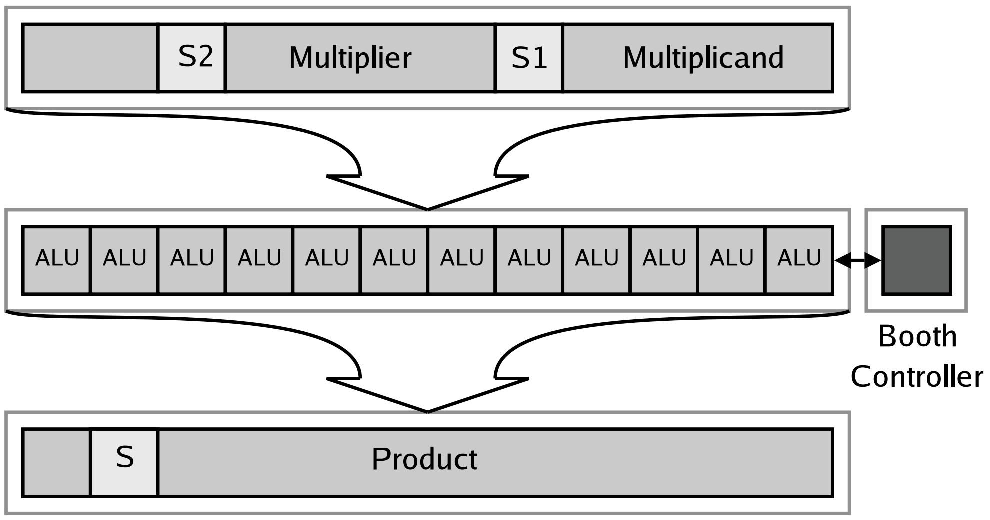


Figure 4.17: The variable-precision Booth multiplier architecture

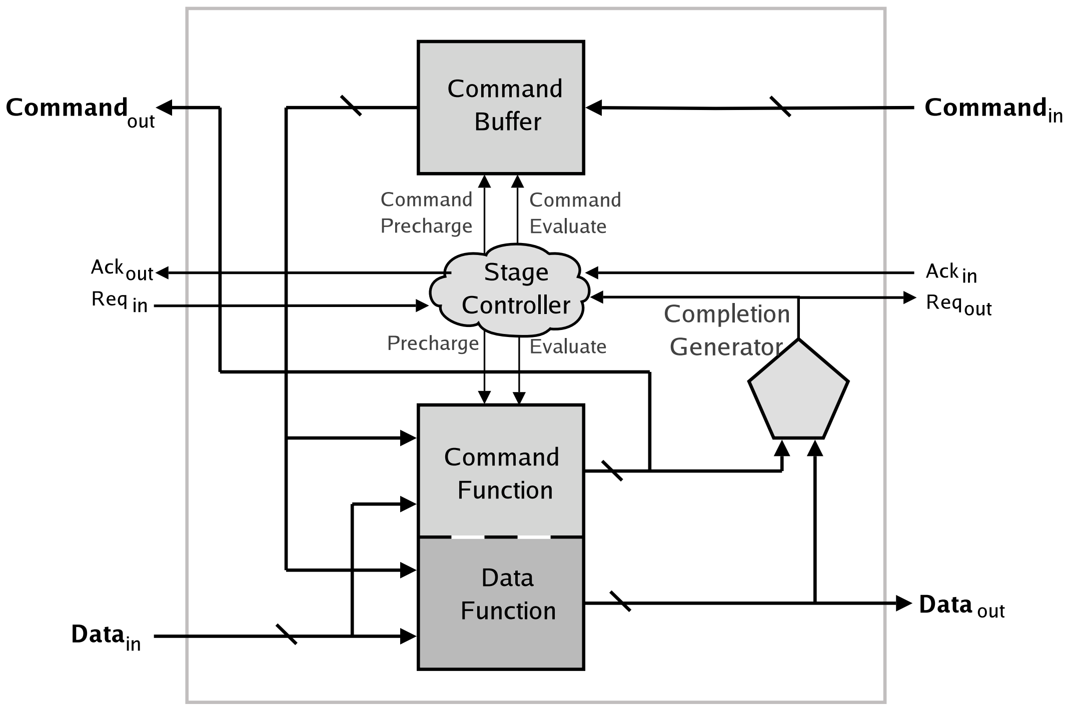
The final difference is that the product also contains a sentinel to indicate its MSB-end.

With these modifications, the multiplier architecture is capable of handling dynamically-varying operand widths, thereby giving a powerful architectural tool to allow the system to dynamically trade computation precision for energy efficiency.

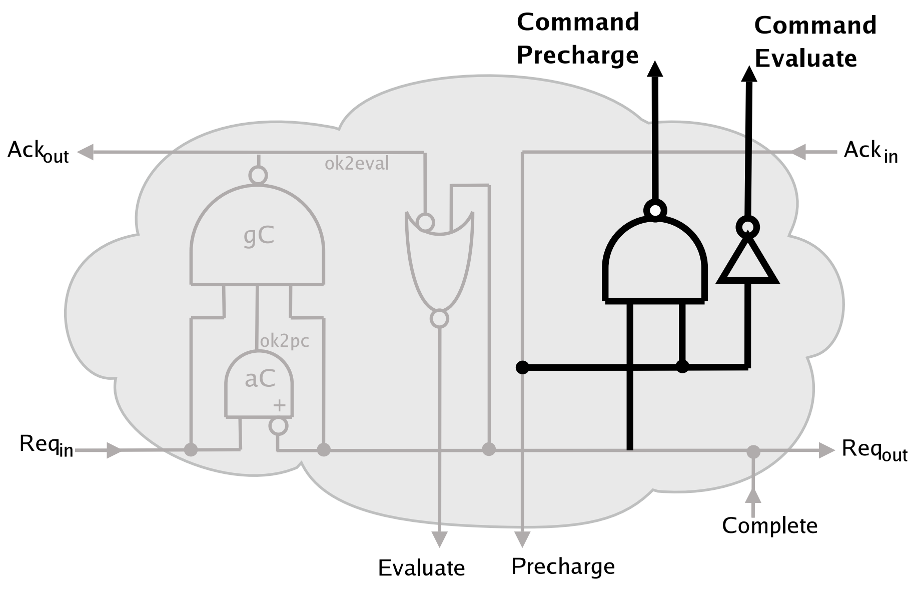
4.4 Dual-Rail High-Capacity Counter-flow Pipelines

A relatively simple modification allows HC-CF become more delay insensitive by using dual-rail signaling. Figure 4.18(a) shows the internal organization of a generic stage of the new pipeline. There are two key differences with respect to the HC pipeline. First, completion information is no longer generated using a matched delay. Instead, a robust delay-insensitive completion generator is used. Since the datapath is encoded using a 1-of-n code, completion is easily detected: when all rails of a bit a are reset, precharge is indicated; when exactly one rail is set, completion of evaluation is indicated. The second modification is the addition of a buffer to store the incoming command, because the command arrives at the start of a precharge phase, but it will be needed in a subsequent evaluation. During that evaluation, the command is combined with the input data to not only produce output data, but also to generate a new command for the left neighbor.

Finally, Figure 4.18(b) shows the actual modification made to the HC handshake circuit. The new handshake circuit produces two additional outputs—*command precharge* and *command evaluate*—which serve as the control signals for the command buffer. The figure highlights the additional gates needed to generate the new signals. The function block's precharge signal triggers the evaluation of the command buffer, causing the incoming command to be stored. The command buffer is subsequently precharged only after the command is "consumed" by the function block upon its next evaluation. The new inverter and NAND gate directly implement this functionality, as before.



(a) Detailed view



(b) Stage controller

Figure 4.18: Pipeline Handshake Controller

CHAPTER 5

Summary and Conclusion

The continuing exponential increase in graphics processor performance is attributable to developments along three distinct yet non-orthogonal axes. In the following sections, I discuss in detail the dissertation contributions and future work for each axis.

5.1 Algorithmic Axis

Algorithmic changes are often initially implemented at the application level, and typically use the current capabilities of the hardware.

At the algorithmic level, I discussed several techniques that are capable of improving the visual quality of images rendered on current commodity GPUs without requiring modifications to the underlying hardware or architecture. In particular, I described a method to generate summed-area tables rapidly using graphics hardware, improvements to summed-area tables, and several rendering techniques that take advantage of summed-area tables to increase the quality of imagery rendered in real-time.

5.1.0.1 Contribution

In this dissertation, I presented research contributions for improving the efficiency and performance of graphics processors along the algorithmic axis. My specific research contributions include:

- ***Efficient construction of summed-area tables:*** I developed a method to rapidly generate summed-area tables using graphics hardware, which is efficient enough to allow multiple tables to be generated every frame and used for a multitude of rendering effects while maintaining interactive frame rates. Several novel applications of using summed-area tables in interactive graphics are presented, such as the real-time rendering of interactive glossy reflections.

- ***Offset summed-area tables:*** I developed a technique that alleviates the precision requirements needed in the construction and use of summed-area tables by offsetting the input image data by a constant value. This method improves precision in two ways: (i) there is a 1-bit gain in precision because the sign bit now becomes useful, and (ii) the summed-area function becomes non-monotonic, and therefore the maximum value reached has a relatively lower magnitude.

- ***Fast image based lighting using summed area tables:*** Finally, I present a method to rapidly generate higher-order summed-area tables — e.g., a summed-area table of a summed-area table — that is efficient enough to allow multiple tables to be generated every frame while maintaining interactive/ frame rates. These higher order summed-area tables are then used to approximate reflections of high-dynamic range environments maps onto a surface exhibiting a Phong BRDF.

5.1.0.2 Future Work

For future work along the algorithmic axis, I would like to quantify how closely a set of summed box and Bartlett filters can approximate an arbitrary BRDF, and develop a set of criteria to generate the filter stack that best represents a given BRDF. While the techniques presented in this dissertation substantially reduce the precision requirements of summed-area tables, work is needed on techniques to reduce them even further. Future work in this area involves reducing the precision requirements of offset summed-area tables by extending the offsets to non-constant values, and taking advantage of better branching support in future shader architectures to reduce the need to sample both the front and back dual-paraboloid maps for all fragments.

5.2 Architectural Axis

That said, there are situation where the new algorithms will lend themselves to easy implementation in hardware. For example various environment mapping techniques initially required the application developer to handle texture coordinate generation, whereas modern GPUs are able to automatically transform normal and reflection directions into texture coordinates. At the architectural level, changes introducing new functionality should be made without reducing the performance of legacy applications, an important property for devices intended for established commodity markets.

I described a novel streaming ray tracer algorithm that uses a conditional output stream and that lowers memory bandwidth and increases processor utilization when compared to current GPU based techniques. While enabling conditional output streams would require minor changes to the architecture of current GPUs, the benefits would far out weigh the costs. This leverages current graphics hardware and is an alternative to developing a completely separate architecture suitable only for ray tracing.

5.2.0.3 Contribution

In this dissertation I presented research that has produced contributions for improving the efficiency and performance of graphics processors along the architectural axis. In particular, my specific research contributions include:

- ***Novel ray tracing algorithm:*** I described a novel streaming ray casting algorithm. The algorithm uses conditional output streams to reduce memory bandwidth and when compared to previous methods increases processor utilization. The algorithm reduces memory bandwidth by over forty times when compared to the most efficient method presented so far. My proposed technique could be used to implement hybrid rendering algorithms that use standard z-buffering techniques to compute primary visibility, and then use ray tracing to generate geometrically correct reflections and shadows.

5.2.0.4 Future Work

I would like to further develop algorithms that can take advantage of conditional output streams. For example, in an all-GPU particle system algorithm particles could be efficiently removed from the simulation. Further, I plan to develop a streaming k-nearest neighbor search algorithm that takes advantage of conditional streams.

5.3 Circuit-Level Axis

It is possible to make dramatic changes at the circuit-level without modifying the GPU's architecture or the application programmer interface. For example, modern hardware has moved from using standard-cell implementations to customs ASICs without requiring changes to the underlying architecture.

At the circuit level, I developed a methodology that extends the high capacity asynchronous pipelining style with a novel implementation of the counterflow organization. Additionally, I described the *compute-on-demand* paradigm for arithmetic circuits, and how asynchronous logic could be leveraged to improve the performance and energy efficiency of commodity graphics processors.

5.3.0.5 Contribution

I developed techniques that have produced contributions for improving the efficiency and performance of graphics processors along the circuit-level axis. In particular, my research contributions include:

- ***Novel conterflow implementation:*** I developed a counterflow organization that eliminates the complex synchronization and arbitration between the two distinct data streams required by the traditional method. My technique allows for shorter critical paths, and, therefore higher operating speed. As an example of my counterflow methodology, I describe a novel multiplier organization, in which the data bits flow in one direction, and the Booth commands are *piggybacked* on the acknowledgments floating in the opposite direction.

- ***Compute-on-demand paradigm:*** I described the notion of compute-on-demand as a design principle for fast and energy-efficient graphics hardware. The key idea is to exploit the data-dependent nature of computation, and to obtain speed and energy consumption improvements by optimizing the design for the common case, instead of assuming worst-case operation. An asynchronous or clockless circuit style is used to facilitate this paradigm. In particular, only those portions of compute blocks are activated that are actually required for a particular operation, thereby saving energy.

5.3.0.6 Future Work

I presented a compute-on-demand paradigm, which is the key to the low power and high performance of the presented asynchronous logic comparator. I believe this paradigm could also be of benefit for other arithmetic circuits used in the graphics pipeline, such as variable-precision ALUs, and interpolators. Additionally, the counterflow pipeline implementation holds promise for hardware besides GPUs, and for reducing energy wasted in speculative execution. Furthermore, I would like to explore on-the-fly variable precision arithmetic to further reduce the energy consumption of mobile graphics cores.

BIBLIOGRAPHY

Arvo, J. (1995). Applications of irradiance tensors to the simulation of non-lambertian phenomena. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 335–342, New York, NY, USA. ACM Press.

Ashikhmin, M. and Ghosh, A. (2002). Simple blurry reflections with environment maps. *J. Graph. Tools*, 7(4):3–8.

ATI Technologies (2005). Radeon x1800 shader architecture technology white paper. `http://ati.amd.com/products/radeonx1k/whitepapers/X1800_Shader_Architecture_Whitepaper.pdf`.

Bartlett, V. and Grass, E. (1999). A low-power concurrent multiplier-accumulator using conditional evaluation. *ICECS 1999*, pages 629 – 633.

Benes, M., Nowick, S. M., and Wolfe, A. (1998). A fast asynchronous Huffman decoder for compressed-code embedded processors. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 43–56.

Berkel, C. H. K. v., Josephs, M. B., and Nowick, S. M. (1999). Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223–233.

Birtwistle, G. and Davis, A., editors (1995). *Asynchronous Digital Circuit Design*, Workshops in Computing. Springer-Verlag.

Blinn, J. F. and Newell, M. E. (1976). Texture and reflection in computer generated images. *Commun. ACM*, 19(10):542–547.

Blythe, D. (1999). Advanced graphics programming techniques using opengl. Technical report, SIGGRAPH course.

Blythe, D. (2006). The Direct3D 10 System. In *SIGGRAPH '06: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques.*

Catmull, E. (1974). *A Subdivision Algorithm for Computer Display of Curved Surfaces.* PhD thesis, University of Utah.

Christen, M. (2005). *Ray Tracing on GPU.* PhD thesis, University of Applied Sciences Basel, Switzerland.

Coombe, G., Harris, M. J., and Lastra, A. (2004). Radiosity on graphics hardware. In *GI '04: Proceedings of the 2004 conference on Graphics interface*, pages 161–168, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada. Canadian Human-Computer Communications Society.

Crow, F. C. (1984). Summed-area tables for texture mapping. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 207–212, New York, NY, USA. ACM Press.

Davis, A. and Nowick, S. M. (1995). Asynchronous circuit design: Motivation, background, and methods. In Birtwistle, G. and Davis, A., editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 1–49. Springer-Verlag.

Day, P. and Woods, J. V. (1995). Investigation into micropipeline latch design styles. *IEEE Trans. on VLSI Systems*, 3(2):264–272.

Debevec, P. (1998). Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Proceedings of SIGGRAPH 98*, pages 189–198.

Debevec, P. (2002). A tutorial on image-based lighting. In *IEEE Computer Graphics and Applications*.

Demers, J. (2004). *GPU Gems*, pages 375–390. Addison Wesley.

Diefenbach, P. (1996). *Multi-pass Pipeline Rendering: Interaction and Realism through Hardware Provisions*. PhD thesis, University of Pennsylvania, Philadelphia.

Doggett, M. (2005). Xenos: Xbox360 gpu. In *Eurographics (EG)*. ATI inc. `http://www.ati.com/developer/eg05-xenos-doggett-final.pdf`.

Dubois, P. and Rodrigue, G. (1977). An analysis of the recursive doubling algorithm. In *High Speed Computer and Algorithm Organization*, pages 299–305.

Efthymiou, A., Suntiamorntut, W., Garside, J., and Brackenbury, L. (2004a). An asynchronous, iterative implementation of the original booth multiplication algorithm. *ASYNC 2004*, pages 207 – 215.

Efthymiou, A., Suntiamorntut, W., Garside, J., and Brackenbury, L. (2004b). An asynchronous, iterative implementation of the original Booth multiplication algorithm. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 207–215. IEEE Computer Society Press.

Ekanayake, V. N., Clinton Kelly, I., and Manohar, R. (2005). Bitsnap: Dynamic significance compression for a low-energy sensor network asynchronous processor. In *ASYNC(2005)*, pages 144–154. IEEE Computer Society.

Epic Games (2004). Unreal tournament 2004.

Foley, T. and Sugerman, J. (2005). Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA. ACM Press.

Furber, S. B. and Liu, J. (1996). Dynamic logic in four-phase micropipelines. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press.

Gageldonk, H. v., Baumann, D., van Berkel, K., Gloor, D., Peeters, A., and Stegmann, G. (1998). An asynchronous low-power 80c51 microcontroller. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 96–107.

Govindaraju, N., Gray, J., Kumar, R., and Manocha, D. (2006). Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336, New York, NY, USA. ACM Press.

Gowan, M. K., Biro, L. L., and Jackson, D. B. (1998). Power considerations in the design of the alpha 21264 microprocessor. In *Proc. ACM/IEEE Design Automation Conf.*, pages 726–731.

GPUBench (2004). http://graphics.stanford.edu/projects/gpubench/.

Greene, S. (2003). Summed area tables using graphics hardware. Game Developers Conference.

Havran, V. (2001). *Heuristic Ray Shooting Algorithms.* PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague.

Heckbert, P. S. (1986). Filtering by repeated integration. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 315–321, New York, NY, USA. ACM Press.

Heidrich, W. and Seidel, H.-P. (1998). View-independent environment maps. In *HWWS '98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 39–ff., New York, NY, USA. ACM Press.

Hensley, J., Lastra, A., and Singh, M. (2004). An area- and energy-efficient asynchronous booth multiplier for mobile devices. In *Proc. Int. Conf. Computer Design (ICCD).*

Hensley, J., Scheuermann, T., Coombe, G., Singh, M., and Lastra, A. (2005). Fast summed-area table generation and its applications. In *Computer Graphics Forum*, volume 24, pages 547–555.

Hensley, J., Scheuermann, T., Singh, M., and Lastra, A. (2006). Fast hdr image-based lighting using summed-area tables. Technical Report TR06-017, University of North Carolina at Chapel Hill.

Hillesland, K. and Lastra, A. (2004). Gpu floating-point paranoia. In *Proceedings of GP2.*

Kameyama, M., Kato, Y., Fujimoto, H., Negishi, H., Kodama, Y., Inoue, Y., and Kawai, H. (2003). 3d graphics lsi core for mobile phone 'z3d'. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware*, pages 60–67.

Kapasi, U. J., Dally, W. J., Rixner, S., Mattson, P. R., Owens, J. D., and Khailany, B. (2000). Efficient conditional operations for data-parallel architectures. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 159–170, New York, NY, USA. ACM Press.

Karlsson, F. and Ljungstedt, C. J. (2004). Ray tracing fully implemented on programmable graphics hardware. Master's thesis, Chalmers University of Technology Göteborg, Sweden.

Kautz, J., Vazquez, P.-P., Heidrich, W., and Seidel, H.-P. (2000). A unified approach to prefiltered environment maps. In *Eurographics Workshop on Rendering.*

Kearney, D. and Bergmann, N. W. (1997). Bundled data asynchronous multipliers with data dependant computation times. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 186–197. IEEE Computer Society Press.

Killpack, K., Mercer, E., and Meyers, C. (2001). A standard-cell self-timed multiplier for energy and area critical synchronous systems. *ARVLSI 2001.*

Kim, H. (1999). Relative timing based verification of timed circuits and systems. In *Proc. Int. Workshop on Logic Synthesis.*

Knittel, G. and Schilling, A. (1995). Eliminating the z-buffer bottleneck. In *EDTC '95: Proceedings of the 1995 European conference on Design and Test*, page 12, Washington, DC, USA. IEEE Computer Society.

Lines, A. M. (June 1995, revised 1998). Pipelined asynchronous circuits. Master's thesis, California Institute of Technology.

Markoff, J. (2001). Computing pioneer challenges the clock. In *The New York Times (Technology Section).* `http://www.nytimes.com/2001/03/05/technology/05IVAN.html`.

Martin, A., Nystroem, M., and Penzes, P. (2001). *Power-Aware Computing*, chapter ET^2: A Metric For Time and Energy Efficiency of Computation. Kluwer Academic Publishers. `http://caltechcstr.library.caltech.edu/archive/00000308`.

Martin, A. J. (2001). Towards an energy complexity of computation. *Information Processing Letters*, 77:181–187.

Mesa3D (2006). Mesa3d graphics library. `http://mesa3d.org`.

Nowick, S. M. (1996). Design of a low-latency asynchronous adder using speculative completion. *IEE Proceedings, Computers and Digital Techniques*, 143(5):301–307.

Nowick, S. M., Yun, K. Y., and Beerel, P. A. (1997). Speculative completion for the design of high-performance asynchronous dynamic adders. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 210–223. IEEE Computer Society Press.

Pharr, M. (2004). Ambient occlusion. *Game Developers Conference.*

Pharr, M. and Humphreys, G. (2004). *Physically Based Rendering.* Morgan Kaufmann.

Ponomarev, D., Kucuk, G., Ergin, O., and Ghose, K. (2004). Energy efficient comparators for superscalar datapaths. *IEEE Transactions on Computers*, 53(7):892–904.

Popa, T. S. (2004). Compiling data dependent control flow on simd gpus. Master's thesis, University of Waterloo.

Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. (2002). Ray tracing on programmable graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 703–712, New York, NY, USA. ACM Press.

Rotem, S., Stevens, K., Ginosar, R., Beerel, P., Myers, C., Yun, K., Kol, R., Dike, C., Roncken, M., and Agapiev, B. (1999). RAPPID: An asynchronous instruction length decoder. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 60–70.

Schuster, S., Reohr, W., Cook, P., Heidel, D., Immediato, M., and Jenkins, K. (2000). Asynchronous interlocked pipelined CMOS circuits operating at 3.3-4.5 GHz. In *Int. Solid State Circuits Conf.*

Seitz, C. L. (1980). System timing. In Mead, C. A. and Conway, L. A., editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley.

Shin, M.-C., Kang, S.-H., and Park, I.-C. (2001). An area-efficient iterative modified-booth multiplier based on self-timed clocking. *ICCD 2001*, pages 511 – 512.

Singh, M. (2001). *The Design of High-Throughput Asynchronous Pipelines.* PhD thesis, Columbia University.

Singh, M. and Nowick, S. M. (2000a). Fine-grain pipelined asynchronous adders for high-speed DSP applications. In *Proceedings of the IEEE Computer Society Workshop on VLSI*, pages 111–118. IEEE Computer Society Press.

Singh, M. and Nowick, S. M. (2000b). High-throughput asynchronous pipelines for fine-grain dynamic datapaths. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 198–209. IEEE Computer Society Press.

Singh, M. and Nowick, S. M. (2001). MOUSETRAP: ultra-high-speed transition-signaling asynchronous pipelines. In *Proc. Int. Conf. Computer Design (ICCD)*, Austin, TX.

Singh, M., Tierno, J. A., Rylyakov, A., Rylov, S., and Nowick, S. M. (2002). An adaptively-pipelined mixed synchronous-asynchronous digital FIR filter chip operating at 1.3 GigaHertz. In *ASYNC*, Manchester, UK. IEEE Computer Society Press.

Sproull, R. F., Sutherland, I. E., and Molnar, C. E. (1994). The counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, 11(3):48–59.

Sutherland, I. and Fairbanks, S. (2001). GasP: A minimal FIFO control. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 46–53. IEEE Computer Society Press.

Sutherland, I. E. (1989). Micropipelines. *Communications of the ACM*, 32(6):720–738.

Tiwari, V., Singh, D., Rajgopal, S., Mehta, G., Patel, R., and Baez, F. (1998). Reducing power in high-performance microprocessors. In *Proc. ACM/IEEE Design Automation Conf.*, pages 732–737.

Tristram, C. (2001). It's time for clockless chips. *Technology Review Magazine*, 104(8):36–41. `http://www.technologyreview.com/magazine/oct01/tristram2.asp`.

Viola, P. and Jones, M. (2001). Robust real-time object detection. In *International Workshop in Statistical and Computation Theories of Vision-Modeling, Learning, Computing, and Sampling.*

Wald, I., Slusallek, P., Benthin, C., and Wagner, M. (2001). Interactive rendering with coherent ray tracing. In Chalmers, A. and Rhyne, T.-M., editors, *Eurographics (EG)*, volume 20(3), pages 153–164. Blackwell Publishing.

Wang, C.-C., Lee, P.-M., Wu, C.-F., and Wu, H.-L. (2003). High fan-in dynamic cmos comparators with low transistor count. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications.*

Wang, Y., Jiang, Y., and Sha, E. (2001). On area-efficient low power array multipliers. *ICECS 2001*, pages 1429 – 1432.

Weste, N. and Eshraghian, K. (1993). *Priniciples of CMOS VLSI Design, a Systems Perspective.* Addison-Wesley Publishing Co., second edition.

Williams, L. (1983). Pyramidal parametrics. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 1–11, ACM Press.

Williams, T. E. (1991). *Self-Timed Rings and their Application to Division.* PhD thesis, Stanford University.

Williams, T. E. (1992). Analyzing and improving the latency and throughput performance of self-timed pipelines and rings. In *Proc. Int. Symp. on Circuits and Systems.*

Williams, T. E. and Horowitz, M. A. (1991). A zero-overhead self-timed 160ns 54b CMOS divider. *IEEE Journal of Solid-State Circuits*, 26(11):1651–1661.

Yang, R. and Pollefeys, M. (2003). Multi-resolution real-time stereo on commodity graphics hardware.

Yun, K. Y., Beerel, P. A., Vakilotojar, V., Dooply, A. E., and Arceo, J. (1997). The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 140–153. IEEE Computer Society Press.