Load-balanced Rendering on a General-Purpose Tiled Architecture

by

Jiawen Chen

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2005 [June 2005]

Author .....
Department of Electrical Engineering and Computer Science
May 15, 2005

Certified by .....
Frédo Durand
Assistant Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Load-balanced Rendering on a General-Purpose Tiled Architecture

by

Jiawen Chen

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Commodity graphics hardware has become increasingly programmable over the last few years, but has been limited to a fixed resource allocation. These architectures handle some workloads well, others poorly; load-balancing to maximize graphics hardware performance has become a critical issue. I have designed a system that solves the load-balancing problem in real-time graphics by using compile-time resource allocation on general-purpose hardware. I implemented a flexible graphics pipeline on Raw, a tile-based multicore processor. The complete graphics pipeline is expressed using StreamIt, a high-level language based on the stream programming model. The StreamIt compiler automatically maps the stream computation onto the Raw architecture. The system is evaluated by comparing the performance of the flexible pipeline with a fixed allocation representative of commodity hardware on common rendering tasks. The benchmarks place workloads on different parts of the pipeline to determine the effectiveness of the load-balance. The flexible pipeline achieves up to twice the throughput of a static allocation.

Thesis Supervisor: Frédo Durand
Title: Assistant Professor

Acknowledgments

I would like to thank all my collaborators in this project, including Mike Gordon, Bill Thies, Matthias Zwicker, Kari Pulli, and of course, my advisor Professor Frédo Durand for all their hard work and support. Special thanks go to Mike Doggett for his unique insider perspective on graphics hardware. I can't forget Eric Chan, for his incredibly precise comments and crisp writing style. Finally, I would like to thank the entire Graphics Group at MIT for their support, and wonderful games of foosball.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Rendering realistic scenes in real-time has always been a challenge in computer graphics. Displaying large-scale environments with sophisticated lighting simulation, textured surfaces, and other natural phenomena requires an enormous amount of computational power as well as memory bandwidth. Graphics hardware has long relied on using specialized units and the inherent parallelism of the computation to achieve real-time performance. Traditionally, graphics hardware has been designed as a pipeline, where the scene description flows through a series of specialized fixed-function stages to produce the final image.

Recently, there has been a trend in adding programmability to the graphics pipeline. Modern graphics processors (GPUs) feature fully programmable vertex and fragment processors and reconfigurable texturing and blending stages. The latest features, such as floating-point per-pixel operations and dynamic flow control offer exciting possibilities for sophisticated shading as well as performing *general purpose* computation using GPUs.

Despite significant gains in performance and programmable features, current GPU architectures have a key limitation: a fixed resource allocation. For example, the NVIDIA NV40 processor has 6 vertex pipelines, 16 fragment pipelines, and a fixed set of other resources surrounding these programmable stages. The allocation is fixed at design time and remains the same for all software applications that run on this chip.

Although GPU resource allocations are optimized for common workloads, it is difficult for a fixed allocation to work well on all possible scenarios. This is due to a key difference between graphics processing and traditional digital signal processing (DSP) ap-

plications. DSP hardware is usually optimized for specific types of algorithms such as multiply-accumulate schemes for convolution and matrix-vector multiplication. Such algorithms operate with a *static data rate*, which allows the hardware to be extremely deeply pipelined and achieve excellent throughput. In contrast, graphics applications introduce a *variable data rate* into the pipeline. In graphics, a triangle can occupy a *variable* number of pixels on the screen. Hence, the hardware cannot be optimized for all inputs. For instance, consider a scene that contains detailed character models standing on a landscape. The detailed characters models are comprised of thousands of triangles, each of which cover only a few pixels on the screen, while the landscape contains only a few large triangles that cover most of the screen. In this case, it is impossible for a fixed resource allocation to balance the workload—the bottleneck is on either the vertex or pixel processing stage. Programmability only aggravates the load balancing problem. Many applications now perform additional rendering passes using programmable pixel shaders to do special effects, during which the vertex engines are completely idle. Conversely, vertex shaders are now being used to perform complex deformations on the geometry and the pixel engines are idle. And when an application spends much of its time rasterizing shadow volumes, almost the entire chip is idle. These are common scenarios that suffer from load imbalance due to a fixed resource allocation.

I propose a new approach to solving the load-balancing problem *statically* by using compile-time resource allocation. In this method, the programmer designs a set of *profiles* that specify the resource allocation at compile time. At runtime, the hardware switches between profiles on explicit "context switch" instructions, which can occur within a frame. The proposed method is somewhat extreme: the method requires hardware that has not only programmable units, but also a programmable network between units to control the data flow. While it is expected that GPUs will retain a level of specialization in the foreseeable future (in particular for rasterization), a fully programmable pipeline gives is a new point in the design space that permits efficient load-balancing.

The prototype implementation of the approach is made feasible by two unique technologies: a multicore processor with programmable communication networks and a programming language that allows the programmer to easily specify the topology of the graph-

ics pipeline using high-level language constructs. The pipeline is executed on the Raw processor [20], which is a highly scalable architecture with programmable communication networks. The programmable networks allow the programmer to realize pipelines with different topologies; hence, the user is free to allocate computation units to rendering tasks based on the demands of the application. The prototype is written completely using StreamIt [11], which is a high-level language based on a stream abstraction. The stream programming model of StreamIt facilitates the expression of parallelism with high-level language constructs. The StreamIt compiler generates the code to control Raw's networks and relieves the programmer of the burden of manually managing data routing between processor tiles. On the compiler side, the main challenge has been to extend StreamIt to handle the variable data rates present in 3D rendering due to the variable number of pixel outputs per triangle and shader lengths.

I must emphasize that the implementation if a proof of concept and cannot compete with commodity GPUs in terms of pure performance. Beyond the implementation of a load-balanced 3D graphics pipeline in this particular environment, the thesis is that load-balancing and increased programmability can be achieved through the following approach:

A multicore chip with exposed communication enables general-purpose computation and resource reallocation by rerouting data flow.

A stream-based programming model that facilitates the expression of arbitrary computation.

A compiler approach to static load-balancing facilitates the appropriate allocation of computing units for each application phase. This shifts load balancing away from the programmer who only needs to provide hints such as the expected number of pixels per triangle.

This thesis focuses on load balancing and the programming model at the cost of the following points, which I plan to address in future work. Emphasis is not placed on the memory system, although it plays a major role on performance, especially in texture caching. Additionally, the proposed method only explores *static* or compile-time load-balancing.

Dynamic load-balancing would be an exciting direction in future research. Finally, my method pushes full programmability quite far and does not use specialized units for rasterization, the stage of the 3D pipeline that seems the least-likely component to become programmable in the near future for performance reasons. Specializing triangle rasterizers to support other rendering primitives (e.g., point sprites) would also be interesting for future research.

Despite these limitations, and although the performance obtained by the simulation cannot compete with state-of-the-art graphics cards, I believe the proposed method is an important first step in addressing the load-imbalance problem in current graphics architectures. Solving this problem is important because doing so maximizes the use of available GPU resources, which in turn implies a more efficient and cost-effective rendering architecture. I hope that this work will inspire future designs to be more reconfigurable and yield better resource utilization through load-balancing.

A review of background, related work, the Raw Processor, and the StreamIt programming language are presented in Chapter 2. I describe the implementation in Chapter 3, including details of the algorithms used in each pipeline stage. Chapter 4 evaluates the implementation and proposes enhancements to the architecture. Finally, Chapter 5 will conclude the thesis and propose possible directions for future research.

Chapter 2

Background

This chapter provides some background on the evolution of graphics hardware from fixed function rasterizers to fully programmable stream processors. It also highlights other works relevant to this thesis, including parallel interfaces, load distribution, scalability, general purpose computation on GPUs (GPGPU), and the stream abstraction. Section 2.4 provides a brief overview of the Raw processor and the unique features that enable efficient mapping of streaming computation to the hardware. In Section 2.5, I give an overview of the StreamIt language, its programming model, and the mapping of StreamIt programs to Raw.

2.1 Fixed-Function Graphics Hardware

3D graphics hardware in the 1980s and 1990s were *fixed-function* pipelines. The typical graphics pipeline composed of the following stages, Figure B in Appendix B shows a pipeline block diagram [2]:

- **Command / Input Parser:** Receives geometry and state change information from the application.

- **Vertex Transformation and Lighting:** Transforms vertex attributes position, normal, texture coordinates) from 3D world space to 3D eye space and projected into 2D screen space. One of a fixed set of lighting models is applied to the vertices.

- **Primitive Assembly / Triangle Setup:** Vertices are assembled into primitives such as triangles, triangle-strips, and quadrilaterals. Primitive facing is determined and optionally discarded depending on user-selected culling modes (typically front or back).

- **Rasterization:** The primitive is rasterized onto the pixel grid and fragments are generated for texturing. Vertex colors and texture coordinates are interpolated across the primitive to generate fragment colors and texture coordinates.

- **Texturing:** One or two textures may be retrieved from memory using the fragment's texture coordinates and applied using a fixed set of texture operations (such as decal or color modulation).

- **Frame Buffer Operations:** The incoming fragment is optionally tested against a depth buffer, stencil buffer, and blended with the previous fragment in the frame buffer. The user can typically choose from one of several modes for each buffer operation.

- **Display:** The display device draws the contents of the frame buffer.

Components of the graphics pipeline were at most *configurable*. For example, the user may have the ability to select between one or two texture filtering modes, but is not able to customize the shading of a fragment. In these early architectures, exploiting the parallelism in the computation and using specialized hardware was key to achieving real-time performance.

Molnar et al. [12] have characterized parallel graphics architectures by their *sorting classification*. Due to the order dependent nature of graphics applications, parallel architectures require at least one synchronization point to enforce correct ordering semantics. Sorting can take place almost anywhere in the pipeline. A "sort-first" architecture distributes vertices to parallel pipelines during geometry processing before screen-space positions are known and takes advantage of object-level parallelism. A "sort-middle" architecture synchronizes between geometry processing and rasterization, redistributing screen-space

primitives to take advantage of image-space parallelism. A "sort-last" architecture redistributes fragments after rasterization to parallel frame buffers taking advantage of memory bandwidth.

The Silicon Graphics RealityEngine [1] and InfiniteReality [13] were highly scalable sort-middle architectures. These fixed-function pipelines were built using special-purpose hardware and distributed onto multiple boards, each of which contained dozens of chips. Both architectures used specialized hardware for geometry processing, rasterization, display generation (frame buffer). Despite off-chip and off-board communications latencies, these architectures achieved real-time framerates for large scenes by combining extremely fast specialized hardware with a highly parallel processing pipeline.

In 1996, Nishimura and Kunii [14] designed VC-1, a highly scalable sort-last architecture based on general purpose processors. While VC-1 relies on general purpose processors, it did not feature user programmability. VC-1 was able to achieve excellent performance by exploiting *virtual frame buffers*, which greatly simplified network communications and parallelized the computation.

In 1997, Eyles et al. designed PixelFlow, a highly-scalable architecture [5]. It was composed of multiple networked workstations each with a number of boards dedicated to parts of the rendering pipeline. While described as an "object-parallel" rendering system, PixelFlow had two synchronization points not unlike Pomegranate, a later "sort-everywhere" architecture [4]. It featured an 800 MB/s geometry sorting network and a 6.4 GB/s image composition network as well as API changes to permit programmer control over the parallel computation.

These designs all took advantage of both data and task parallelism. Since during computation, the data ordering is essentially independent (except for the one required synchronization point), multiple pipelines can be used for each stage. For example, in a sort middle architecture, after rasterization, the fragments are essentially independent. As long as two fragments that occupy the same position are drawn in the same order they arrived, they can be distributed onto an arbitrary number of pipelines. Computation tasks are also parallel: once a vertex has been transformed, it can go on to the next stage. The next vertex is independent of the current vertex. Hence, the pipeline can be made extremely deep.

The scalability of such parallel architectures have been studied from both an interface perspective [6] and from a network and load distribution perspective [4]. Both agree that in order to fully exploit parallel architectures, explicit programmer-controlled synchronization primitives are required. These synchronization primitives effectively change the programming model and forces the programmer to consider parallelism when rendering. Pomegranate [4] takes advantage of these synchronization primitives to design a "sort-everywhere" architecture, to fully parallelize every stage of the graphics pipeline.

2.2 Programmable Graphics Hardware

Increased transistor budgets provided by modern manufacturing processes has made it more viable to add a certain level of programmability to various functional units in the GPU. The first generation of truly programmable GPUs were the "DirectX 8 series", including NVIDIA's GeForce3 and ATI's Radeon 8500 [7]. Although these chips featured user-programmable vertex and pixel shaders, their programmability was limited. Programming was done in a low-level assembly-like language, with limited instruction counts, no branching, and a small instruction set. Despite these limitations, programmers were able to take advantage of these new capabilities. Multi-pass rendering, in particular, was and still is used extensively to create special special effects using programmable hardware. A typical multi-pass rendering scenario is rendering real-time shadows using the shadow map algorithm. In the first pass, the scene is rendered from the point of view of a light, writing only to the depth buffer. The depth buffer is read back as a texture, and the scene is rendered again from the point of view of the camera. The vertex shader is used to transform the object position into the coordinate system of the light, and the fragment shader compares the fragment's light-space depth value against the depth value stored in the texture. The result of this comparison is whether the fragment is lit or not, and can be used to control the resulting color.

Current-generation GPUs, such as NVIDIA's NV40 and ATI's R420 (DirectX 9 Series) are much more powerful than their DirectX 8 counterparts. They permit much longer shader lengths, and dynamic branching inside the programmable shaders. There are also

a number of high-level languages that target these platforms [8, 10, 18]. It is expected that future GPUs will feature "unified shaders", where both the programmable vertex and pixel shaders use the same physical processor. This design could increase overall throughput since in a typical application, the load is not balanced between the vertex and pixel stages. The ease of programmability, immense arithmetic capability and large memory bandwidth of GPUs have made them more and more attractive for general purpose, high-performance computing. The architecture of rendering pipelines are beginning to match that of traditional DSPs, albeit with much more arithmetic and memory capability. In particular, GPU architectures seem well suited for streaming computations. Buck et al. [3] have designed Brook, a streaming language that permits the implementation of fairly general streaming algorithms on the graphics processor, mapping these algorithms to vertex and pixel shaders.

2.3 Streaming Architectures

Owens et al. characterized polygon rendering as a stream operation and demonstrated an implementation of a fixed-function pipeline on Imagine, a stream processor [16]. In their implementation, the various stages of the graphics pipeline were expressed as stream *kernels* (a.k.a. *filters*), which were swapped in and out of the Imagine's processing units. Producer/consumer locality was exploited by storing intermediate values in a large set of stream registers and using relatively small kernels which minimizes the cost of context switches. Owens et al. have also compared Reyes, an alternative rendering pipeline organization, against the traditional OpenGL on a stream processor [17]. Reyes is very different pipeline from OpenGL, using tesselation and subdivision instead of triangle rasterization. It has several properties well-suited for stream computation: bounded-size primitives, only one programmable shading stage, and coherent texture memory access. The comparison shows that Reyes essentially has the same deficiencies as OpenGL: the variable data rate in the computation is now during surface subdivision instead of triangle rasterization.

My architecture builds on Brook and Imagine in that it also uses the stream abstraction, but *as the programming model.* Not only is it used to express vertex and fragment shader computation, but also to express the full graphics pipeline itself. The approach differs from

Imagine in that Imagine uses a *time-multiplexing* scheme: the data remains in memory and kernels are context switched to operate on data. My approach uses *space-multiplexing*, where kernels are mapped onto a set of processing units, and data flows through the kernels.

My stream programming method is also related to the shader algebra, [9] where shaders can be combined and code analysis leads to efficient compilation and dead-code elimination for complex vertex and pixel shader combinations. In this case, rather than eliminate dead-code from a limited number of programmable stages, depending on the application, unused pipeline stages can be dropped completely from the graphics pipeline.

Given the trend of increased programmability and unifying programmable units in GPUs, I consider rendering on an architecture that contains a single type of general-purpose processing element. The goal is to study the implications and challenges that this scenario imposes on the "driver" of such a processor. The driver will be of critical importance because it allocates resources depending on the rendering task and ensures that the processor is used efficiently.

2.4 The Raw Processor

The Raw processor [20,22] is a versatile architecture that achieves scalability by addressing the wire delay problem. Raw aims to perform as well as Application Specific Integrated Circuits (ASICs) on special-purpose kernels while still achieving reasonable performance on general-purpose programs. Raw's design philosophy is to address the wire-delay problem by exposing its rich on-chip resources, including logic, wires, and pins, through a new instruction set architecture (ISA) to the software. In contrast with other architectures, this allows Raw to more effectively exploit all forms of parallelism, including instruction, data, and thread level parallelism as well as pipeline parallelism.

Tile-Based Architecture. Raw is a parallel processor with a 2-D array of identical, programmable tiles. Each tile contains a *compute processor* as well as a *switch processor* that manages four networks to neighboring tiles. The compute processor is composed of an eight-stage in-order single-issue MIPS-style processor, a four-stage pipelined floating

point unit, a 32kB data cache, and a 32kB instruction cache. The current prototype is implemented in an IBM 180nm ASIC process running at 450MHz; on one chip, it contains 16 uniform tiles arranged in a square grid. The theoretical peak performance of this prototype is 6.8 GFLOPS. Raw's scalable design allows an arbitrary number of 4x4 chips to be combined into a "Raw fabric" with minimal changes. The performance evaluation for my architecture is done using *btl*, a cycle-accurate Raw simulator, modeling a 64-tile configuration. Though the prototype chip contains only 16 tiles, a 64-tile configuration is planned.

On-Chip Communication Networks. The switch processors control four 32-bit full-duplex on-chip networks. The networks are register-mapped, blocking, and flow-controlled, and they are integrated directly into the bypass paths of the processor pipeline. As a key innovative feature of Raw, these networks are exposed to the software through the Raw ISA.

There are four networks total: two *static* and two *dynamic*. The static networks are used for communication patterns that are known at compile time. To route a word from one tile to another over a static network, it is the responsibility of the compiler to insert a route instruction on every intermediate switch processor. The static networks are ideal for regular stream-based traffic and can also be used to exploit instruction level parallelism [21]. The dynamic networks support patterns of communication that vary at runtime. Items are transmitted in packets; a header encodes the destination tile and packet length. Routing is done dynamically by the hardware, rather than statically by the compiler. There are two dynamic networks: a *memory* network for trusted clients (data caches, I/O, etc.) and a *general* network for use by applications.

Memory System. On the boundaries of the chip, the network channels are multiplexed onto the pins to form flexible I/O ports. Words routed off the side of the chip emerge on the pins, and words put on the pins by external devices appear on the networks. Raw's memory system is built by connecting these ports to external DRAMs. For the 16 tile configuration, Raw supports a maximum number of 14 ports, which can be connected to

up to 14 full-duplex DRAM memory banks, leading to a memory bandwidth of 47GB per second. Tiles on the boundary can access memory directly over the static network. Tiles on the interior of the chip access memory over the memory dynamic network and incur a latency proportional to the distance to the boundary. Data-independent memory accesses can be pipelined, hence, overall throughput is unaffected.

Raw as Graphics Hardware. Raw is an interesting architecture for graphics hardware developers, because its design goals share a number of similarities with current GPUs. Raw is tailored to effectively execute a wide variety of computations, from special purpose computations that are often implemented using ASICs to conventional sequential programs. GPUs exploit data parallelism (by replicating rendering pipelines, using vector units), instruction level parallelism (in super-scalar fragment processors), and pipeline parallelism (by executing all stages of the pipeline simultaneously). Raw, too, is capable of exploiting these three forms of parallelism. In addition, Raw is scalable: it consists of uniform tiles with no centralized resources, no global buses, and no structures that get larger as the tile count increases. In contrast to GPUs, Raw's computational units and communication channels are fully programmable, which opens up almost unlimited flexibility in laying out a graphics pipeline and optimizing its efficiency on Raw.

On the other hand, the computational power of the current 16-tile, prototype Raw processor is more than an order of magnitude smaller than the power of current GPUs. GPUs can perform far more parallel operations in a single cycle than Raw: Raw's computation units do not perform vector computation. In addition, Raw is a research prototype implemented with a 0.18μm process; an industrial design with a modern 90nm process would achieve higher clock frequencies.

Hence, my prototype implementation is not intended to compete with current GPUs in terms of absolute performance, but demonstrates the benefits of a flexible and scalable architecture for efficient resource utilization.

2.5 The StreamIt Programming Language

StreamIt [11,23] is a high level stream language that aims to be portable across communication-exposed architectures such as Raw. The language exposes the parallelism and communication of streaming programs without depending on the topology or granularity of the underlying architecture. The StreamIt programming model is based on a *structured stream abstraction*: all stream graphs are built out of a hierarchical composition of filters, pipelines, split-joins, and feedbackloops.

As I will describe in more detail in Chapter 3, the structured stream graph abstraction provided by StreamIt lends itself to expressing data parallelism and pipeline parallelism that appear in graphics pipelines. In particular, I will show how to use StreamIt for high-level specification of rendering pipelines with different topologies. As previously published, StreamIt permits only static data rates. In order to implement a graphics system that allows different triangle sizes, variable data rates is a necessity. Section 3.2 will describe how variable data rates are implemented in the StreamIt language and compiler.

Language Constructs. The basic unit of computation in StreamIt is the *filter* (also called a *kernel* and used interchangeably). A filter is a single-input, single-output block with a user-defined procedure for translating input items to output items. Filters send and receive data to and from other filters through FIFO queues with compiler type-checked data types. StreamIt distinguishes between filters with *static* and *variable* data rates. A static data rate filter reads a fixed number of input items and writes a fixed number of output items in each cycle of execution, whereas a variable data rate filter has a varying number of input or output items in each cycle.

In addition to the filter, StreamIt provides three language constructs to compose *stream graphs*: pipeline, split-join, and feedback-loop. Each of these constructs, including a filter, is called a *stream*. In a pipeline, streams are connected in a linear chain so that the outputs of one stream are the inputs to the next stream. In a split-join configuration, the output from a stream is split onto multiple (not necessarily identical) streams that have the same input data type. The data can be either duplicated or placed in a weighted round-robin scheduling policy. The split data must be joined somewhere downstream unless it is the sink to the

stream. The split-join allows the programmer to specify data-parallelism between streams. The feedback loop enables a stream to receive input from downstream, for applications such as MPEG.

2.6 Compiling StreamIt to Raw

A compiler for mapping static data rate StreamIt to Raw has been described in previous work [11]. Extending the compiler to support *variable* data rates for graphics is described in Section 3.2. Compilation involves four stages: dividing the stream graph into load-balanced partitions, laying out the partitions on the chip, scheduling communication between the partitions, and generating code. The operation of these stages is summarized below.

Partitioning StreamIt hides the granularity of the target machine from the programmer. The programmer specifies as abstract filters, which are independent of the underlying hardware. It is the responsibility of the compiler to partition the high level stream graph into efficient units of execution for the particular architecture. Given a processor with N execution units, the partitioning stage transforms a stream graph into a set of no more than N filters that run on the execution units, while satisfying the logical dataflow of the graph as well as constraints imposed by the hardware. To achieve this, the StreamIt partitioner employs a set of *fusion*, *fission*, and *reordering* transformations to the stream graph. Workload estimations for filters are calculated by simulating their execution and appropriate transformations are chosen using a greedy strategy. However, simulation can only provide meaningful workload relationships for stream graphs with static data rates. Hence, in stream graphs that contain variable data-rate filters partitioning is performed separately for each static subgraph. Refer to Section 3.2 for the discussion on variable data rates.

Layout The layout stage assigns filters in the partitioned stream graph to computational units in the target architecture while minimizing the communication and synchronization present in the final layout. For Raw, this involves establishing a one-to-one mapping from filters in the partitioned stream graph to processor tiles, using Raw's networks for com-

munication between filters. Computing the optimal layout is NP-Hard, to make this problem tractable, a cost function is developed that measures the overhead of transmitting data between tiles for each layout [11]. Memory traffic, static network communication, and dynamic network communication are all components of the cost function. For example, off-chip memory accesses latency is shortest when the filter is allocated near the edge of the chip. The cost function measures the memory traffic and communication overhead for a given layout, as well as the synchronization imposed when independent communication channels are mapped to intersecting routes on the chip. A layout is found by minimizing the cost function with a simulated annealing algorithm. The automatic layout algorithm is a useful tool for approximating the optimal layout. The programmer is free to manually specify the layout for further optimization.

Communication Scheduling The communication scheduling stage maps the communication channels specified by the stream graph to the communication network of the target. While the stream graph represents communication channels as infinite FIFO abstractions, a target architecture will only provide limited buffering resources. Communication scheduling must avoid deadlock and starvation while trying to utilize the parallelism explicit in the stream graph. On Raw, StreamIt channels between static data rate filters are mapped to the static network. The static network communication schedule is computed by simulating the firing of filters in the stream graph and recording the communication pattern for each switch processor. Outputs of variable data rate filters are transmitted via the general dynamic network and scheduled differently. (Section 3.2. More details on scheduling can be found in the original publication about mapping StreamIt to Raw by Gordon et al. [11].

Code Generation Code generation for Raw involves the generation of *computation* code for the compute processors and *communication* code for the switch processors. Computation code is generated from the output of the partitioning and layout stages. The code is a mixture of C and assembly and is compiled using Raw's GCC 3.3 port. Assembly communication code for the switch processors is generated directly from the schedules obtained in the communication scheduling stage.

Chapter 3

Flexible Pipeline Design

This chapter describes the design and implementation of a flexible graphics pipeline on Raw using the StreamIt programming language. The primary goal is to build a real-time rendering architecture that can be reconfigured to adapt to the workload. The architecture should also be fully programmable: not just in the vertex and pixel processing stages, but throughout the pipeline and should not be restricted to one topology. Reconfigurable, in this case, means that the programmer is aware of the range of inputs in the application and has built profiles at compile-time. At runtime, an explicit "switch" instruction is given to reconfigure the pipeline. My design leverages the Raw processor's programmable tiles and routing network as well as the StreamIt language to realize this architecture. Section 3.1 discusses how to express the components of a graphics pipeline using higher order StreamIt constructs and how the resulting stream graph is mapped to the Raw architecture. In Section 3.2, I describe extending StreamIt with variable data rates for graphics computation. Finally, Section 3.3 describes how the flexible pipeline can be used for load balancing.

3.1 Pipeline Design

The StreamIt philosophy is to implement filters as interchangeable components. Following that philosophy, each stage in my flexible pipeline is implemented as a StreamIt filter and allocated to Raw tiles by the StreamIt compiler. The programmer is free to vary the pipeline topology by rearranging the filters and recompiling, with different arrangements reusing

the same filters. The flexible pipeline has several advantages over a fixed pipeline on the GPU. First, any filter (i.e. stage) in the pipeline can be changed. For example, in the first pass of shadow volume rendering, texture mapping is not used, and dead-code elimination can be performed. The entire pipeline is changed so that texture coordinates are neither interpolated nor part of the dataflow. Second, the topology does not even need to conform to any traditional pipeline configuration. In the image processing case study, (see Section 4.4) the current GPU method would render the scene to a texture, and use a complex pixel shader to perform image filtering. Raw can simply be reconfigured to act as an extremely parallel image processor.

In the case studies (Chapter 4) the performance of a flexible pipeline is compared against a fixed-allocation reference pipeline. The reference pipeline models the same design tradeoff as made in GPUs in fixing the ratio of fragment to vertex units. It is implemented using StreamIt and emulates most of the functionality of a programmable GPU. It is manually laid out on Raw (Figure 3.1). The pipeline stages include Input, Programmable Vertex Processing, Triangle Setup, Rasterization, Programmable Pixel Shading, and Reconfigurable Frame Buffer Operations that write to the frame buffer.

The reference pipeline is a sort-middle architecture [12], Figure 3.1 displays its stream graph. Input stage is connected to off-chip memory through an I/O port. Six tiles are assigned to programmable vertex processing, and they are synchronized through one synchronization tile. The synchronizer consumes output of the vertex shaders using a round-robin strategy and pushes the data to the triangle setup tile. The rasterizer uses the homogeneous rasterization algorithm to avoid clipping overhead [15]. Triangle setup computes the vertex matrix and its inverse, the screenspace bounding box, triangle facing, and the parameter vectors needed for interpolation. It distributes data to the 15 pixel pipelines. The pixel pipelines are screen locked and interleaved. Each pipeline is assigned to every 15th column. The pixel pipelines each consist of three tiles, a rasterizer that outputs the visible fragments of the triangle, a programmable pixel processor, and a raster operations tile, which communicates with off-chip memory through an I/O port to perform Z-buffering, blending and stencil buffer operations. Due to routing constraints, not all Raw tiles are used: in particular, no two variable data rate paths can cross. The code for triangle setup,

rasterization, and frame buffer operations is listed in Appendix A.

In contrast to the reference pipeline, my flexible pipeline builds on the same filters as the reference one, but it exploits the StreamIt compiler for automatic layout onto Raw. The pipeline is parameterized by the number of split-join branches at the vertex and pixel stages, and the programmer provides the desired number of branches for a given scenario. The programmer can also increase the pipeline depth manually by dividing the work into multiple stages. For example, a complex pixel shading operation may be divided into two stages, each of which fits onto one tile. In some cases, the programmer also omits some of the filters when they are not needed. Together, flexible resource allocation and dead-code elimination greatly improve performance.

While the automatic layout of filters to tiles by the compiler provides great flexibility, the layout is often not optimal. The compiler uses simulated annealing to solve the NP-hard constrained optimization problem. Hence, when using automatic layouts, in order to satisfy routing constraints, a number of tiles are left unallocated. Automatic layout can act as a good first approximation so the programmer can iterate on pipeline configurations without having to manually configure the tiles. It also has a reasonable runtime: only 5 minutes on a Pentium Xeon 2.2 GHz with 1 GB of RAM. In the case studies, only automatic layout is used for flexible pipeline configurations.

3.2 Variable Data Rates

Variable data rates are essential for graphics rendering. Because the number of pixels corresponding to a given triangle depends on the positions of the vertices for that triangle, the input/output ratio of a rasterizer filter cannot be fixed at compile time. This contrasts with traditional applications of stream-based programming such as digital signal processing that exhibit a fixed ratio of output to input and can be implemented using synchronous models. In particular, the original version of StreamIt relies on the fixed data rate assumption.

The StreamIt language and compiler are augmented to support variable data rates between filters. The language extension is simple, allowing the programmer to tag a data rate as variable. On the compiler side, each phase of the mapping of StreamIt to Raw is

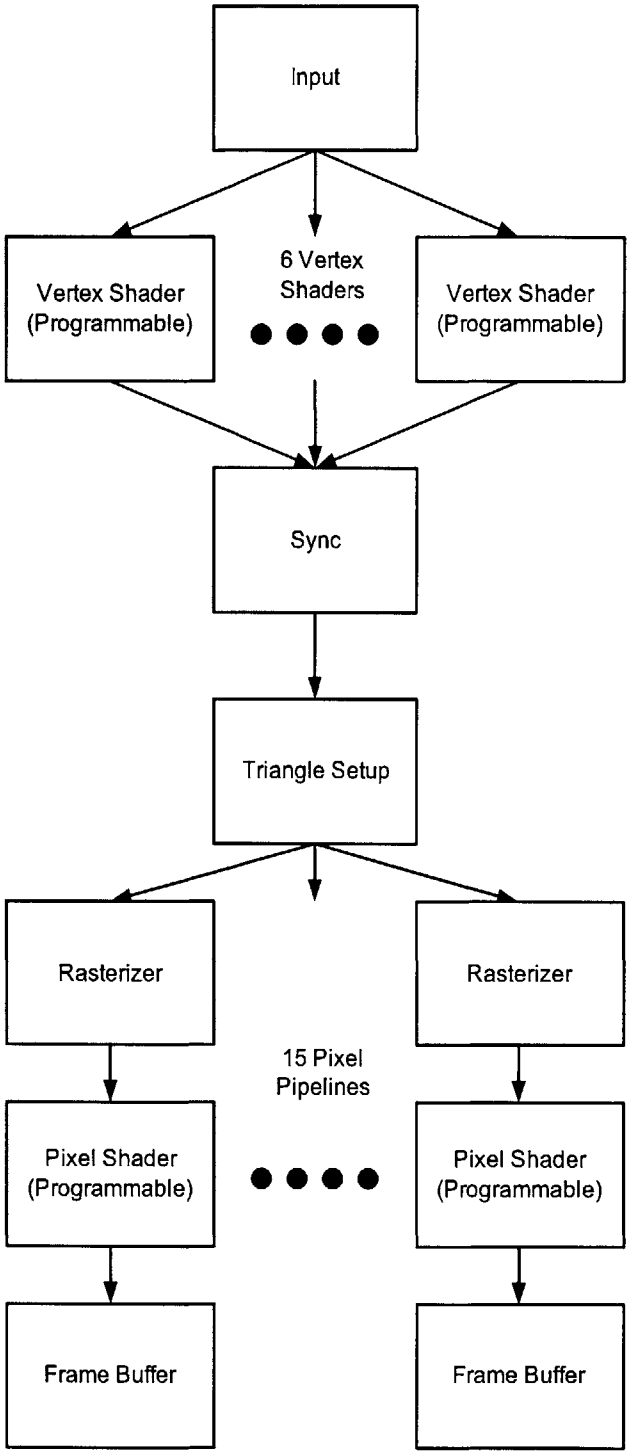
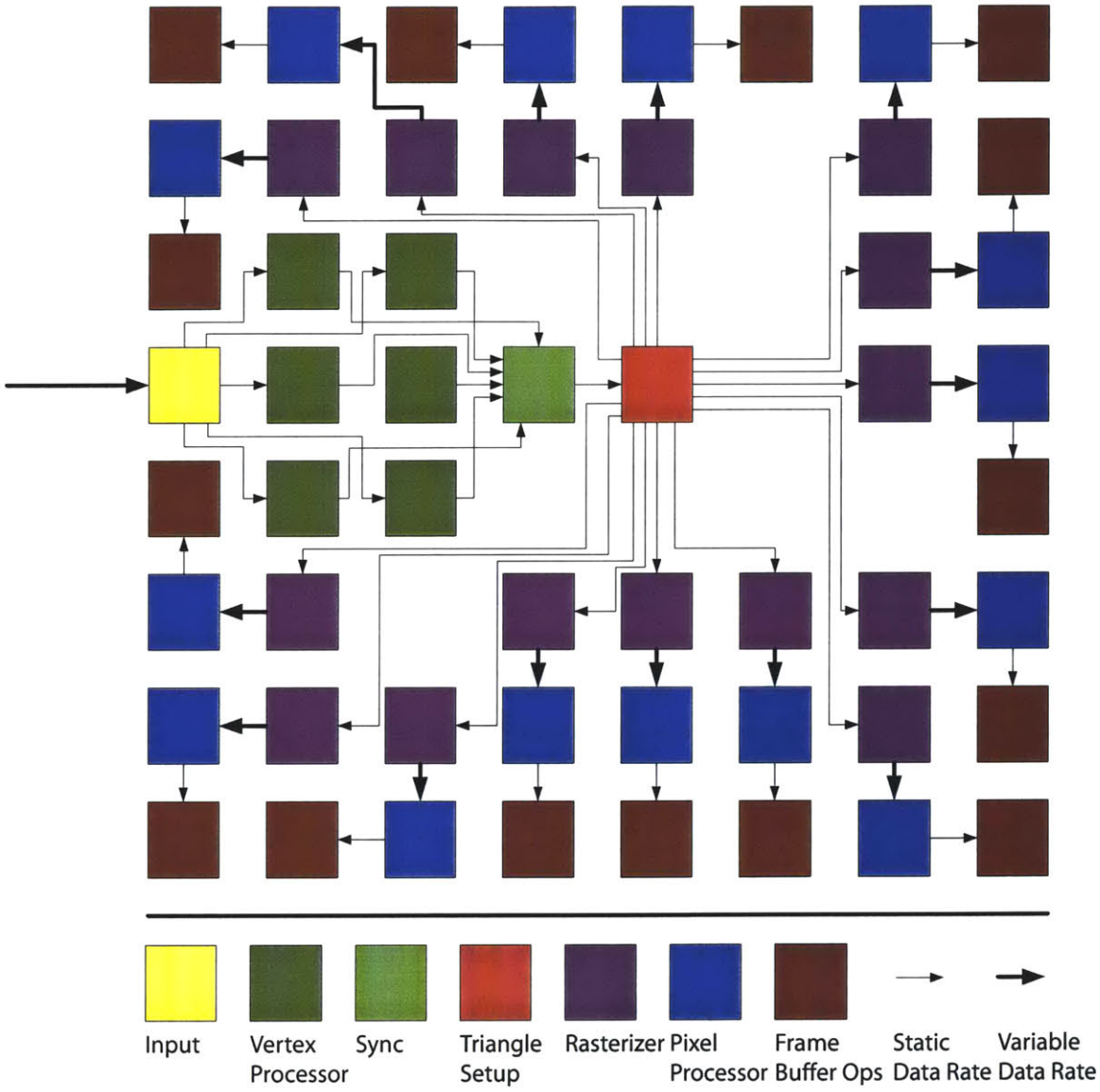


Figure 3-1: Reference Pipeline Stream Graph.



Reference Pipeline Layout on an 8×8 Raw configuration. Color-coded squares represent Raw tiles (unused tiles have been omitted for clarity). Data arrives from an I/O port off the edge of the chip.

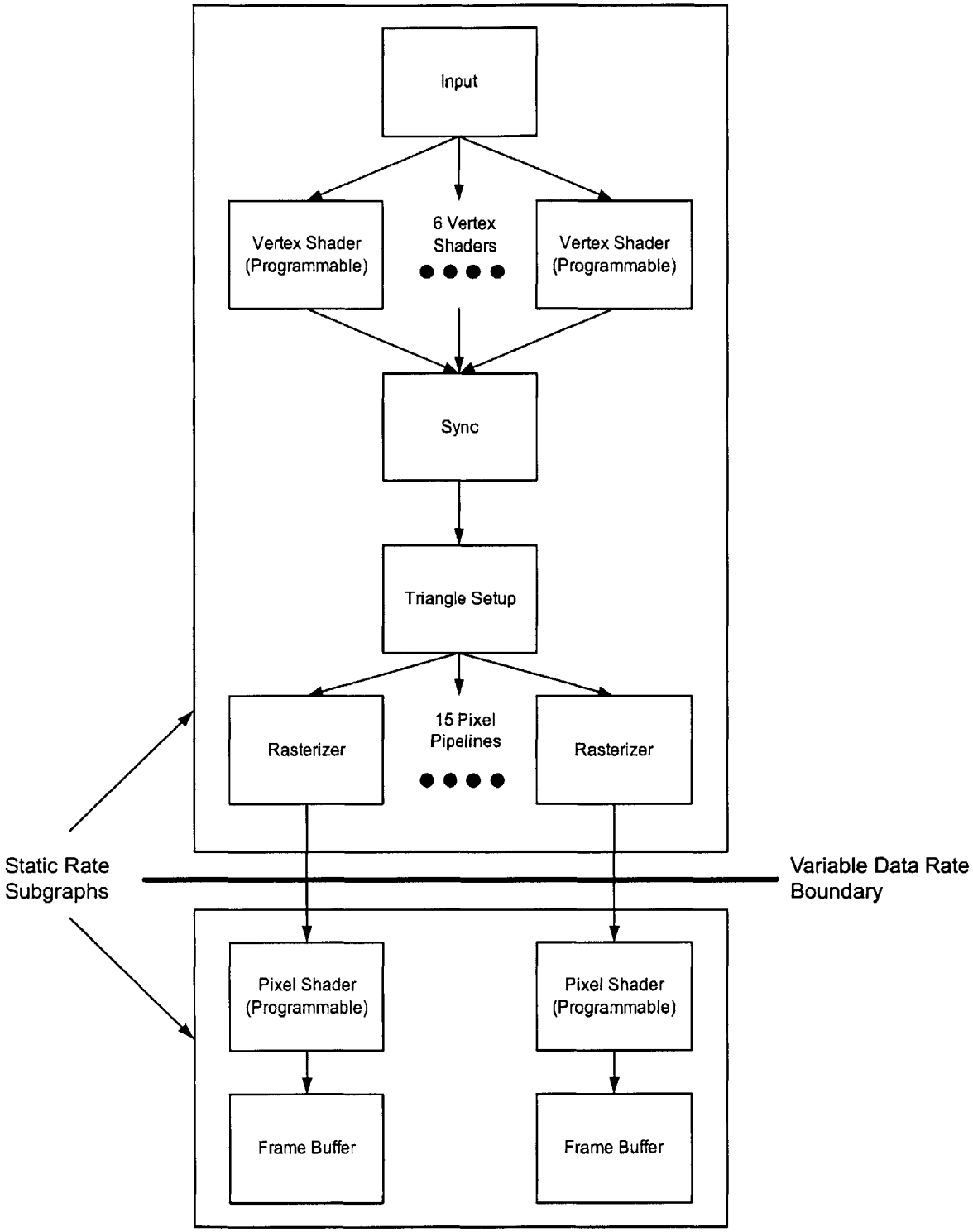
modified.

Partitioning with Variable Data Rates The partitioner supports variable rates by dividing the stream graph into *static-rate subgraphs*. Each subgraph represents a stream in which child filters have static data rates for internal communication. A variable data rate can appear only between subgraphs. Because each subgraph resembles a static-rate application, the existing partitioning algorithm can be used to adjust its granularity. However, as variable data rates prevent the compiler from judging the relative load of subgraphs, the compiler relies on a programmer hint as to how many tiles should be allocated to each subgraph. Figure 3.2 shows the reference pipeline's stream graph partitioned into static-rate subgraphs.

Recall that the flexible graphics pipeline is parameterized by split-joins that provide data-level parallelism for the vertex and pixel sides of the pipeline. The programmer specifies the number of branches for the split joins, thereby choosing the respective expected loads.

Layout with Variable Data Rates Variable data rates impose two new layout constraints. First, a switch processor must not interleave routing operations for distinct static subgraphs. Because the relative execution rates of subgraphs are unknown at compile time, it is impossible to generate a static schedule that interleaves operations from two subgraphs (without risking deadlock). Second, there is a constraint on the links between subgraphs: variable-rate communication channels that are running in parallel (e.g., in different paths of a split-join) must not cross on the chip. Even when such channels are mapped to the dynamic network, deadlock can result if parallel channels share a junction (as a high-traffic channel can block another). In my implementation, these constraints are incorporated into the cost function in the form of large penalties for illegal layouts.

Communication Scheduling with Variable Data Rates Communication scheduling requires a simple extension: channels with variable data rates are mapped to Raw's general dynamic network (rather than the static network, which requires a fixed communication pattern). Within each subgraph, the static network is still used. This implementation avoids



Reference Pipeline stream graph partitioned into static-rate subgraphs.

the cost of constructing the dynamic network header for every packet [11]; instead, the header is constructed at compile time. Even though the rate of communication is variable, the endpoints of each communication channel are static.

3.3 Load-Balancing Using the Flexible Pipeline

The flexible pipeline exploits both data-level parallelism and pipeline parallelism to gain additional throughput. Data-level parallelism is expressed using the StreamIt split-joins construct. Split-joins specify the number of vertex units and the number of parallel pixel pipelines after the rasterizer. Pipeline parallelism must be controlled manually by the programmer, who can find a more fine-grained graph partition than the compiler.

The static load-balancing scenario is as follows. Given an input scene, it is profiled on the reference pipeline to determine where the bottlenecks are (usually the vertex or pixel stage). If the bottleneck is in the vertex or pixel stage, the programmer iteratively adjusts the width of the split-joins to change the ratio between vertex and pixel stages and re-profiles the scene until the load is balanced across the chip. If the load-imbalance is elsewhere (for example, in the triangle setup stage due to the sort-middle architecture), a different graph topology may be used.

Chapter 4

Performance Evaluation

This chapter describes a series of case studies where I have profiled the performance of the reference and flexible pipelines on four common rendering tasks. These experiments serve to test if a flexible pipeline can outperform a pipeline with fixed resource allocation. Given a workload, there should be a load-imbalance between the units in the fixed pipeline that causes suboptimal throughput. Once the bottleneck is identified, the flexible pipeline should be able to reallocate its resources to balance the load and achieve better throughput.

My experiments include the cases of a complex pixel shader (Section 4.2), a multi-pass algorithm (Section 4.3), an image processing algorithm (Section 4.4), and a particle system (Section 4.5). This chapter presents the details of each experiment and demonstrates how the load imbalance in each case can be improved by reallocating resources appropriately, with over 100% increase in throughput in some cases. I conclude the chapter with a discussion on how the Raw hardware could be improved to attain even greater performance for rendering tasks.

4.1 Experimental Setup

Each experiment involves a real-world rendering task that induces a strongly imbalanced load on different pipeline stages and that do not necessarily use all functional units of today's GPUs at the same time. These scenarios demonstrate that the flexible pipeline avoids bottlenecks and idle units, a common problem in today's architectures.

Instead of comparing the graphics performance of Raw against a real GPU, I compare them against the reference pipeline on Raw instead. It is unrealistic to compare Raw against a modern GPU because today's GPUs have several orders of magnitude the computational power of Raw. GPUs not only have far more arithmetic power, but also memory bandwidth and number of registers. For a more realistic comparison, I simulate a GPU's fixed pipeline topology on Raw, and compare the performance and load-balance of the flexible pipeline against this reference.

Performance numbers are listed in terms of triangles per second and percent utilization. The screen resolution is fixed at 600x600 and 32-bit color. Pipeline stage utilization is computed as the number of instructions completed by all tiles assigned to that stage divided by the number of cycles elapsed. Note that this metric for processor utilization is unlikely to reach 100% in any scenario, even in highly parallel computations such as image filtering (4.4). While each tile is fully pipelined, it is unlikely to achieve 1 instruction per clock cycle. Floating point operations incur a 4 cycle latency, memory access costs 3 cycles even on a cache hit, and there are likely to be data hazards in the computation. Furthermore, Raw tiles do not feature predication and all conditionals must be expressed as software branches.

4.2 Case Study 1: Phong Shading

Consider the case of rendering a coarsely tessellated polyhedron composed of large triangles with per pixel Phong shading. In the vertex shader, the vertex's world space position and normal are bound as texture coordinates. The rasterizer interpolates the texture coordinates across each triangle and the pixel shader computes the lighting direction and the diffuse and specular contributions. Most of the load is expected to be on the fragment processor. Vertex and pixel shader code are in the appendix (Figures A and A).

Reference Pipeline As expected, the reference pipeline suffers from an extreme load imbalance. The fragment processor has a 68% utilization, the rasterizer at 17%, while the other units are virtually idle ($< 1\%$) (Figure 4.2). Overall chip utilization is only 25.5%.

	Vertex Units	Pixel Units	tris / sec	Throughput Increase
Reference Pipeline	6	15	4299.27	N/A
Automatic Layout	6	15	4353.08	1.25%
Automatic Layout	1	15	4342.67	1.01%
Automatic Layout	1	24 (12 pipelines, 2 units each)	6652.18	54.73%

Table 4.1: Case 1 Performance Comparison

Throughput is 4300 triangles per second.

Flexible Pipeline I tried several different allocations for this scenario. The first test compares the same pipeline configuation as the reference pipeline, but using automatic layout instead; it yielded a marginal improvement. In both cases, the pixel processor was the bottleneck: the pixel processor's utilization was 68%, the rasterizer at 17%, and the other units virtually idle. Since the fragment stage was the bottleneck, the next test left the number of pixel pipes the same but used only one vertex unit, the utilization and throughput was virtually unchanged, supporting the hypothesis. The final configuration was to allocate most of the tiles to pixel processing, with a pipelined 2-stage pixel shader. This configuration yielded the greatest gain in performance.

In this allocation, the first pixel processor is at 74% utilization, and the second at 60%. The rasterization stage's utilization increases to 31%. The load-balance has improved significantly, even though it is not balanced across the entire chip. Overall chip utilization is only 39.5%. This allocation achieves a throughput of 6652 triangles per second, a 55% increase over the fixed allocation.

4.3 Case Study 2: Multi-Pass Rendering—Shadow Volumes

To demonstrate the utility of a flexible pipeline, I benchmarked shadow volume rendering, a popular technique for generating real-time hard shadows, on Raw. In this algorithm, the load shifts significantly over the three passes. In the first pass, the depth buffer is

initialized with the depth values of the scene geometry. In the given scene, scene, the triangles are relatively large and the computation rasterization bound. In the second pass, the shadow volume itself is rendered. This incurs a even greater load on the rasterizer which has to rasterize large shadow volume polygons, and the frame buffer operations, which must perform a depth test and update the stencil buffer. The final pass is fragment processing bound, where the fragment shader is used to light and texture the final image. I analyze the first two passes in this case study.

Reference Pipeline On the first pass, as expected, the rasterization stage is the bottleneck at 69% utilization (see Figure 4.3). It takes approximately 55 floating point operations for the software rasterizer to output each fragment. The pixels are output in screen aligned order and memory access is very regular for the large triangles. The frame buffer updates only achieve a 7% utilization. The other units in the pipeline are virtually idle, with the exception of the pixel shader use at 6% simply forwarding rasterized fragments to the frame buffer operations unit. Throughput is 988 triangles / second and utilization is 25%. On the second pass, the results are virtually identical, with a slight increase in utilization at the frame buffer operations stage where the Z-Fail algorithm updates the stencil buffer based on triangle orientation. Throughput is 796 triangles / second and utilization is 23%.

Flexible Pipeline Noticing that the computation is rasterization limited for both the first and second pass, the allocation is changed so that only one tile is assigned to vertex processing. The final pass is fragment bound, so the allocation from Case Study #1 is used. Notice that in this multi-pass algorithm, a different allocation can be used for each pass. In fact, multiple allocations can be used *within* a pass. Since neither the first nor the second pass requires pixel shading, the pixel shading stage is removed completely, and the tiles are re-allocated to increase the number of pixel pipelines to 20 (see Figure 4.3). Since the input vertices do not contain any attributes other than position, interpolation and parameter vector calculations for those attributes can be safely removed from the rasterization and triangle setup stages. The flexible pipeline achieves more than 100% increase in throughput over the reference pipeline in both passes. In the first pass, throughput is 2232 triangles / sec

	Vertex Units	Pixel Units	tris / sec	Throughput Increase
Ref., Depth Pass	6	15	987.79	N/A
Ref., Shadow Pass	6	15	796.25	N/A
Auto., Depth Pass	1	20	2223.16	125.06%
Auto., Shadow Pass	1	20	1798.53	125.88%

Table 4.2: Case 2 Performance Comparison

with 27% overall utilization. In the second pass, throughput is 1800 triangles / sec with 27% overall utilization. It is interesting to note that although overall chip utilization has increased, the utilization in the rasterization stage has actually decreased from 71% down to 49% due to the elimination of dead code.

4.4 Case Study 3: Image Processing—Poisson Depth-of-Field

Image processing requires a quite different pipeline architecture than 3D rendering. Since Raw is a general purpose architecture, the computation does not need to be mapped onto a traditional graphics pipeline. Consider the Poisson-disc fake depth-of-field algorithm by ATI [19]. In a GPU implementation, the final pass of the algorithm would require submitting a large screen-aligned quadrilateral and performing the filtering in the pixel shader. The operation is extremely fragment bound since the scene contains only 2 triangles and the pixel shader must perform many texture accesses per output pixel.

In the flexible pipeline, each tile is allocated as an image filtering unit. The tile configuration is expressed as a 62-way StreamIt split-join. Currently, the StreamIt compiler requires that split-joins have a source; hence, two tiles are "wasted" for data routing routing; otherwise, the full 64 tiles could be used. The color and depth buffers are split into 62 blocks. At 600x600 resolution, the blocks fit in the data cache of a tile. The configuration gets only 38% utilization of the chip and throughput of 130 frames per second (see Figure 4.4). Due to the memory-intensive nature of the operation, 100% utilization is not reached—a cache hit still incurs a 3 cycle latency so the result is near the expected 33% utilization.

4.5 Case Study 4: Particle System

For the fourth experiment, consider automatic tessellation and procedural deformation of geometric primitives. In this test, the vertex shaders are modified to receive a complete triangle as input and output 4 complete triangles. Each vertex is given a small random perturbation. The input triangles comprised of a particle system, since these primitives occupied little screen area and required no shading, this scene is expected to be vertex-bound in performance on the reference pipeline.

Reference Pipeline It turns out, however, that the bottleneck lies in the triangle setup stage. Triangle setup has a 49% utilization, the rasterizer is at 22%, and the other units are stalled ($< 4\%$) (see Figure 4.5). In retrospect, this is unsurprising; the sort-middle architecture required output vertices to be synchronized and contained only one triangle setup stage. Since the triangles are small, setup takes a proportionally large amount of computation relative to rasterization. Overall chip utilization was only 7.8%.

Flexible Pipeline The flexible pipeline has the immediate advantage of removing unnecessary work such as texture coordinate interpolation in the rasterizer and parameter vector computation in triangle-setup. The automatically laid out version of the reference pipeline performed slightly better, with utilization up to 9.1% and throughput up by 21%. Assuming that the computation was vertex limited, I reallocated some of the pixel pipelines to vertex units. With 10 pixel pipelines and 9 vertex shaders, performance was increased by 71% over the reference pipeline, with utilization up to 20.6%. To test the hypothesis that the computation was vertex limited, I increased the number of pixel pipelines up to 12, and there was virtually no change. Finally, noticing that triangle setup was a bottleneck, I pipelined the stage by dividing the work onto two tiles and forwarding the necessary data. The pipelined version obtained a performance increase of over 157% over the reference pipeline and utilization up to 24.6%. Pipelining the triangle setup yielded the most dramatic increase in throughput and demonstrated the communication limited nature of the architecture. Even though I originally misjudged where the bottleneck would be, this experiment still illustrates the benefit of a flexible architecture: it can achieve a substantial

	Vertex	Pixel Units	tris / sec	Throughput Increase
Ref. Pipeline	6	15	62300.98	N/A
Automatic	6	15	75465.37	21%
Automatic	8	12	106812.25	71%
Automatic	10	10	107604.02	73%
Automatic (2-stage tri. setup)	8	12	159857.91	157%

Table 4.3: Case 4 Performance Comparison

performance gain by transferring a tile originally assigned to an idle stage to a busy one.

4.6 Discussion

In the above experiments, I compared processor utilization and throughput achieved by a fixed versus flexible resource allocation under several rendering scenarios and configurations of the flexible pipeline. They showed that a flexible resource allocation increases utilization and throughput up to 157% and I believe that these results are indicative for the speed-ups that could be obtained by designing more flexible GPU architectures. In these scenarios, the flexible pipeline was able to adapt to the workload and partially alleviate the bottleneck in the computation. The image processing experiment also showed that a flexible resource allocation leads to a more natural representation of the computation. Instead of relying on the fixed topology and forcing a inherently 2D computation into 3D by drawing a rectangle and using a pixel shader, the flexible pipeline can easily parallelize the operation.

One fact that has not been discussed yet is the *switching cost*, or the time it takes to reconfigure the chip for a different resource allocation. The simplest scheme for reallocating resources would be to first flush the pipeline, then all the tiles would branch to another section of code to begin their new task. The overall cost would be one pipeline flush, a branch, and possibly an instruction cache miss. The latter two would most likely be masked by the latency of the flush. For applications that use the single-pass and multi-pass rendering algorithms considered in the experiments above, switching cost would not be an issue. In the single-pass algorithms considered, the application would be in the process of switching

to a new shader, and would most likely need to flush the pipeline. Similarly, for in the case of multi-pass algorithms, a pipeline flush is required to guarantee correctness. However, it is possible that the programmer may want to switch configurations within a pass without changing a shader. For example, consider a scene that is rendered in two parts. First, the background composed of large triangles is rendered. Then detailed characters are rendered over the background. The first part is fragment-bound while the second part is vertex-bound. In this case, the programmer would want to switch from devoting tiles to fragment processing to devoting resources to vertex processing. The switch would require a flush between drawing the background and drawing the character whereas on a GPU, such a flush is not necessary. This additional overhead has not been well studied and may become future work.

Although the flexible pipeline uses resources much more efficiently over a wider range of applications, the absolute performance obtained by Raw is orders of magnitude lower than current GPUs. The triangle throughput, even in the best case, is several orders of magnitude less than that of an NVIDIA NV40. If case 1 is considered without the expensive pixel shader, the load becomes severely imbalanced in the rasterizer. A serious weakness of the Raw architecture is the use of general-purpose computation for rasterization. In a GPU pipeline, the rasterizer performs a tremendous amount of computation every clock, however, all of it is very simple floating point arithmetic. A DirectX 9 class GPU's rasterizer can output 16 fragments every clock and push the data to the fragment units. In contrast, a Raw tile must fetch and execute all the instructions to rasterize each fragment. My current implementation of homogeneous rasterization requires 55 floating point instructions and one branch per fragment in steady state. Clearly, the specialized rasterizer is a huge advantage for the GPU and integrating one into a Raw architecture would greatly improve performance. It would also be an interesting research question how such an augmentation would be done.

Another advantage of the GPU is the presence of floating-point vector ALUs and vectorized busses. Almost all ALUs on a modern GPU are equipped with 128-bit vector ALUs that can perform a multiplication and addition in one cycle. They can also forward data across wide busses and store all values in vector registers. In contrast, Raw's ALUs, busses,

and register only operate on scalar values. Simply equipping Raw with vector ALUs would yield at least a 4x speedup. If the entire chip was vectorized, even greater speedups can be expected due to lower latency in transmitting data between tiles.

Figure 4-1: Case Study 1 Output image. Resolution: 600×600.

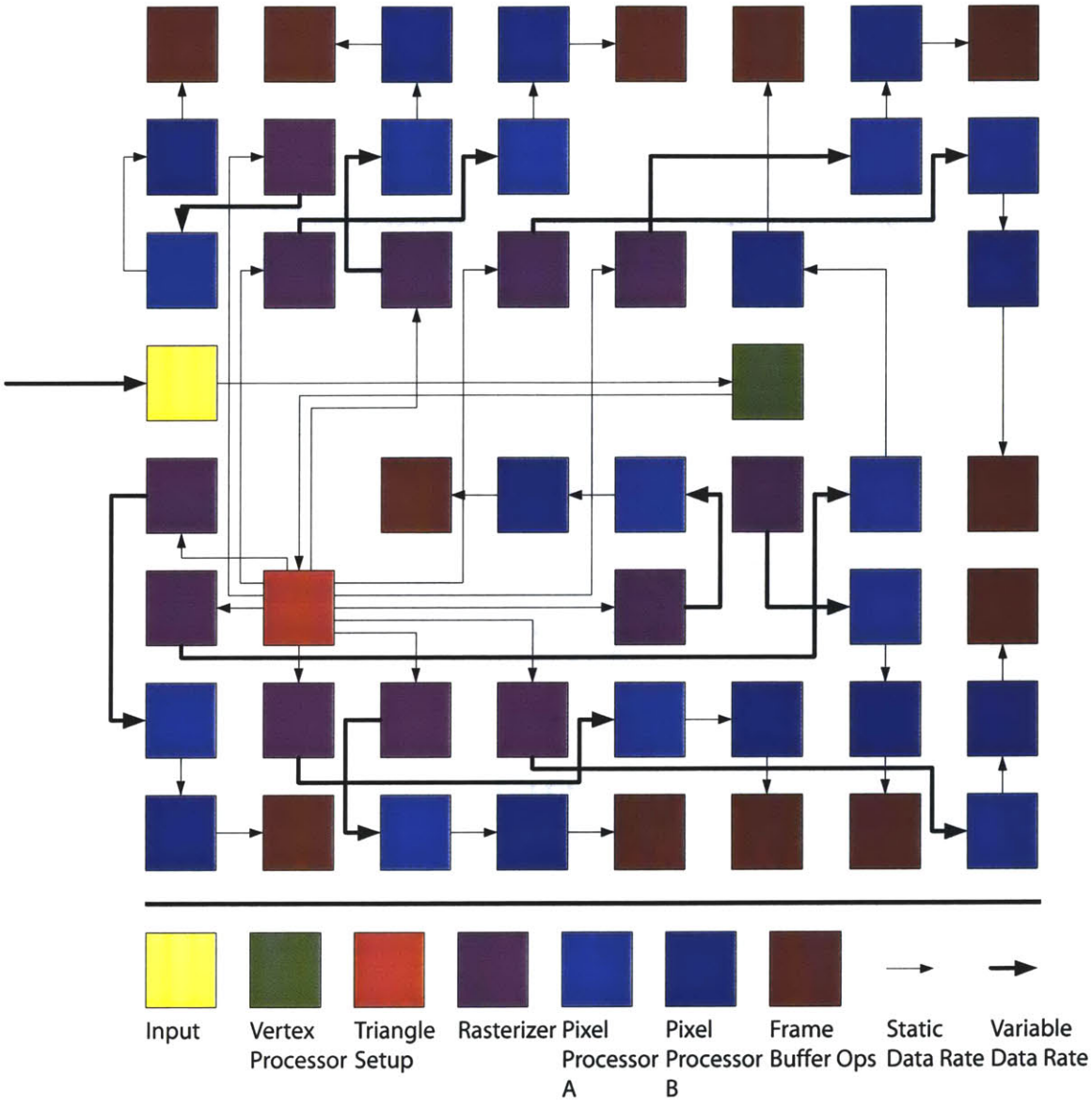


Figure 4-2: Compiler generated allocation for case study #1, last case. 1 vertex processor, 12 pixel pipelines, 2 fragment processors for each pixel pipeline. Unallocated tiles have been removed to clarify the routing.

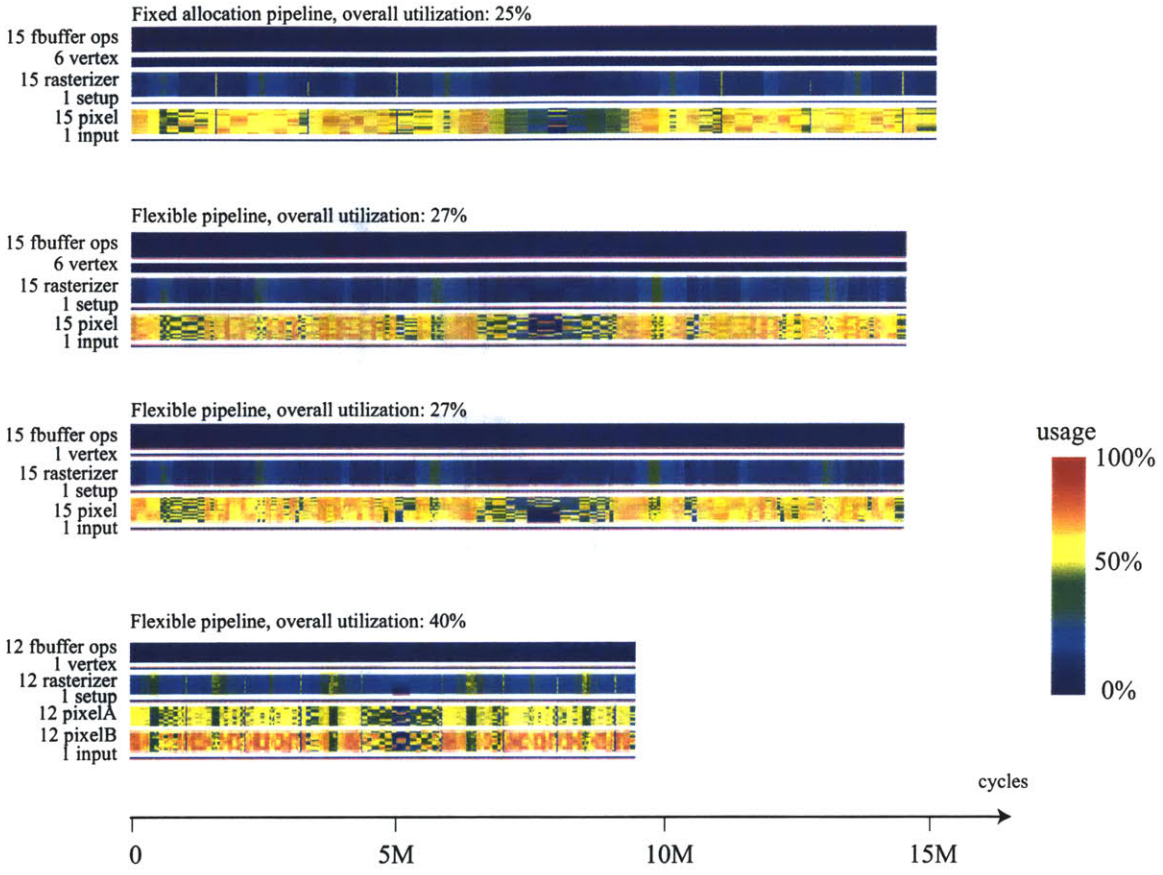


Figure 4-3: Case 1 utilization graphs.

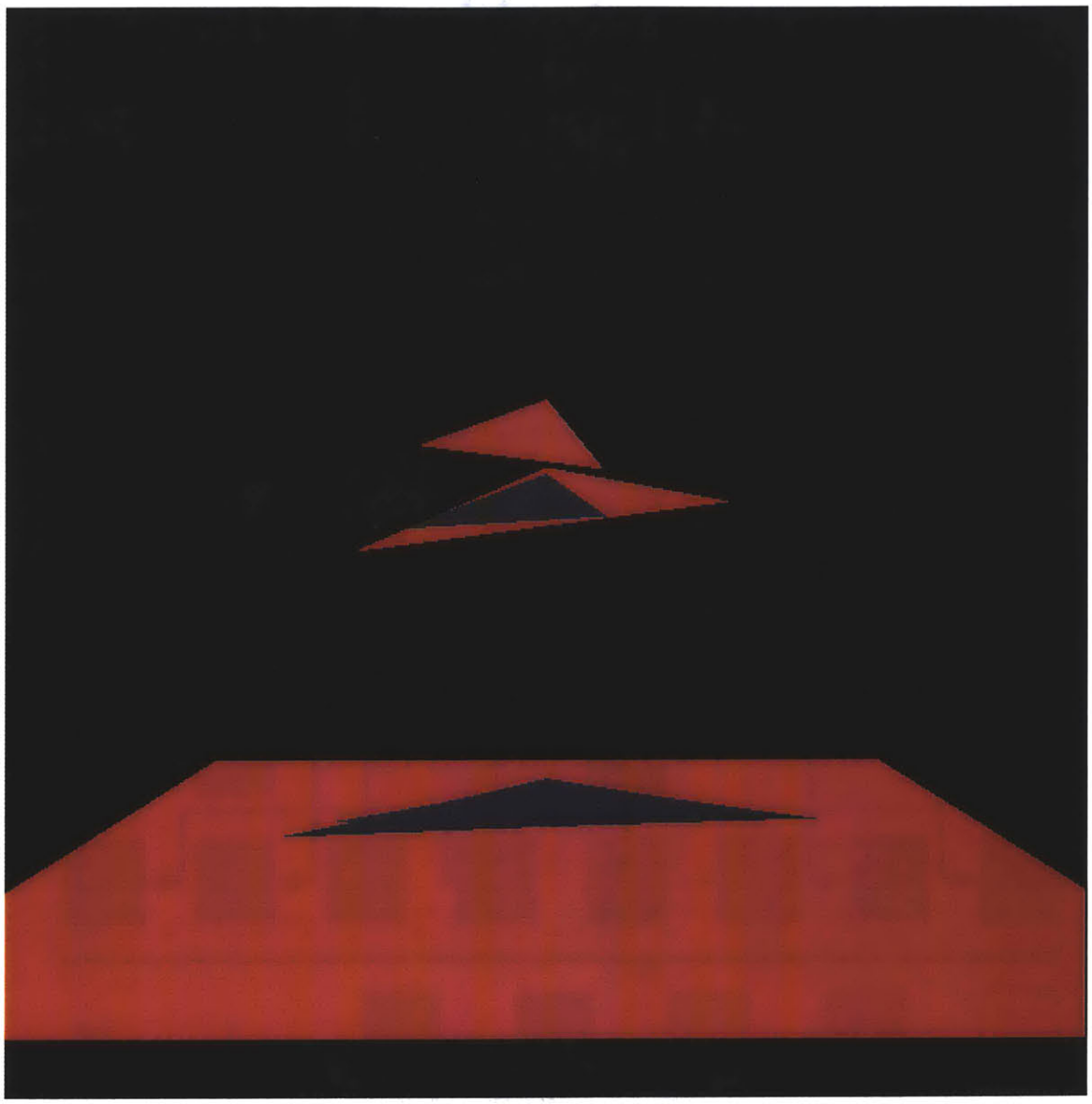


Figure 4-4: Case Study 2 Output image. The relatively small shadow caster still creates a very large shadow volume. Resolution: 600×600.

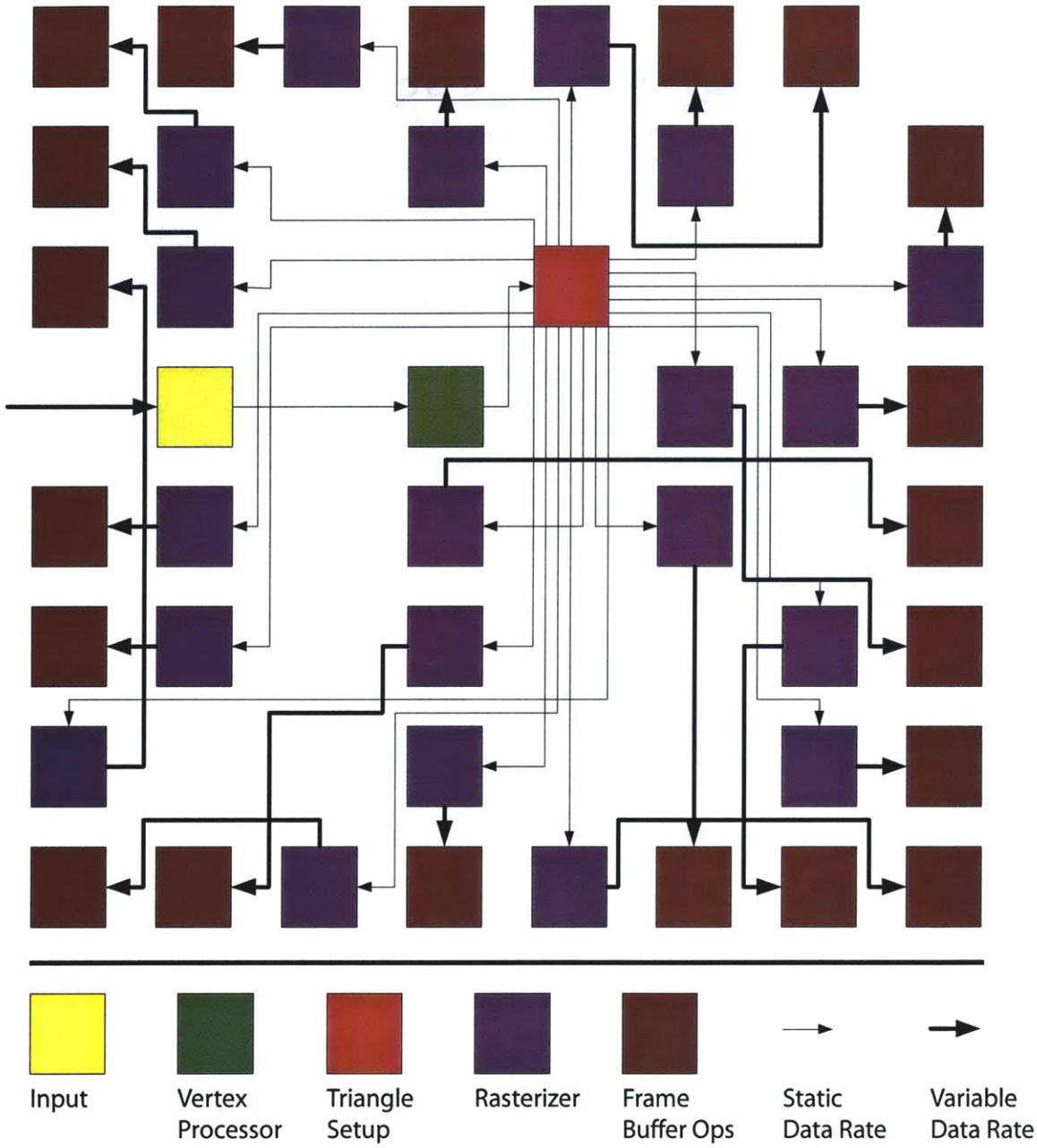


Figure 4-5: Compiler generated layout for case study #2. 1 Vertex Processor and 20 pixel pipelines. Unallocated tiles have been omitted to clarify routing.

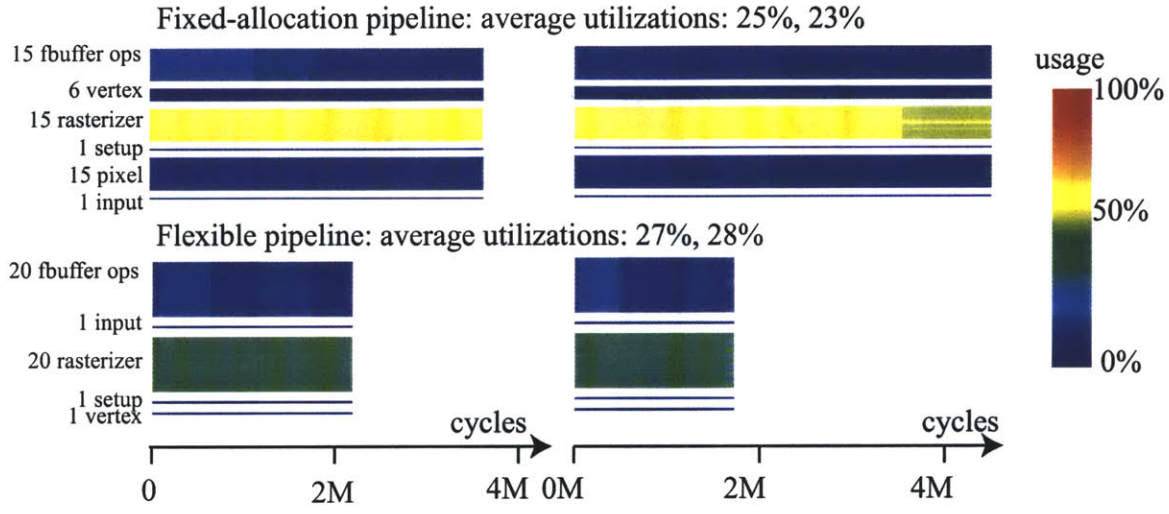


Figure 4-6: Case 2 utilization graph. Depth buffer pass on the left, shadow volume pass on the right.

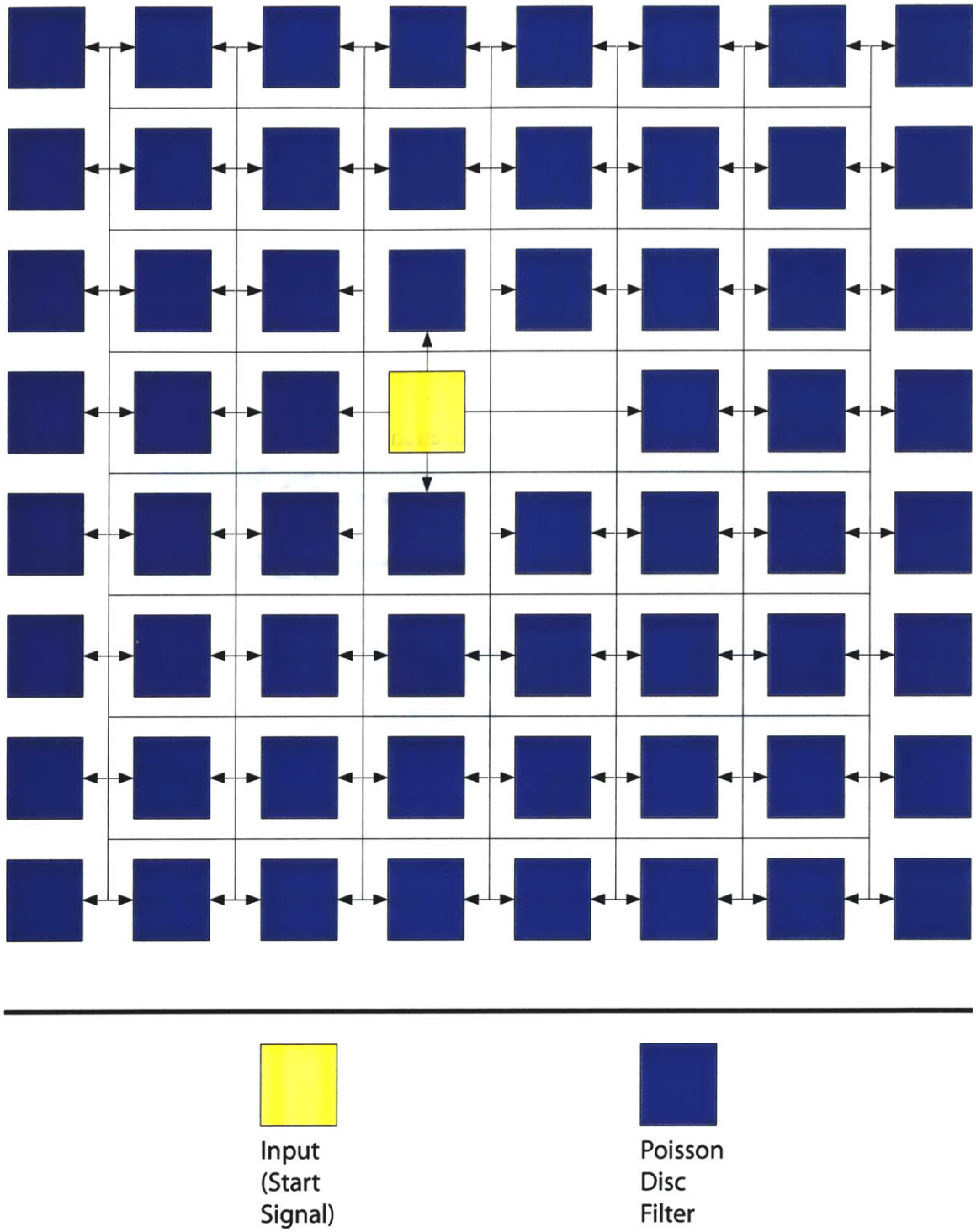


Figure 4-7: Compiler generated layout for case study #3. 1 tile wasted for the start signal and one was unused. 62 tiles perform the filtering. Unallocated tiles have been omitted to clarify routing.

Figure 4-8: Case Study 3 Output image. Resolution: 600×600.

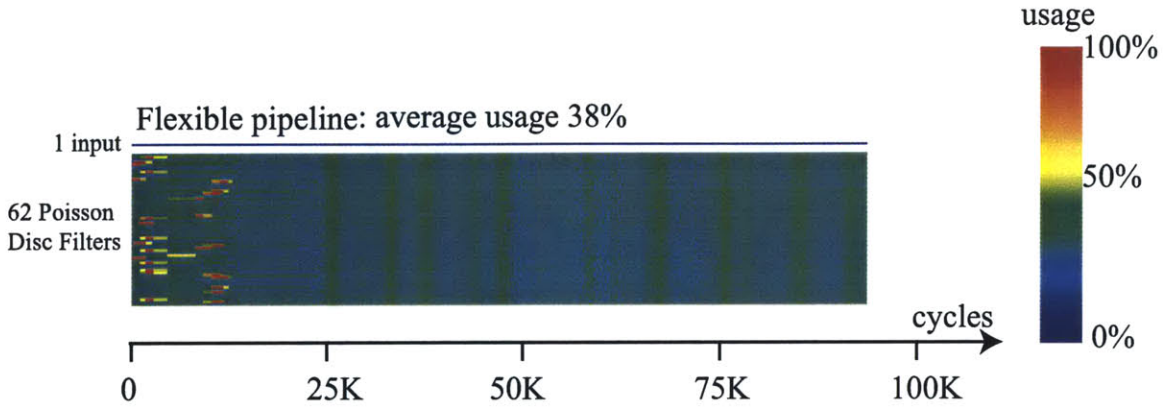


Figure 4-9: Case 3 utilization graph.

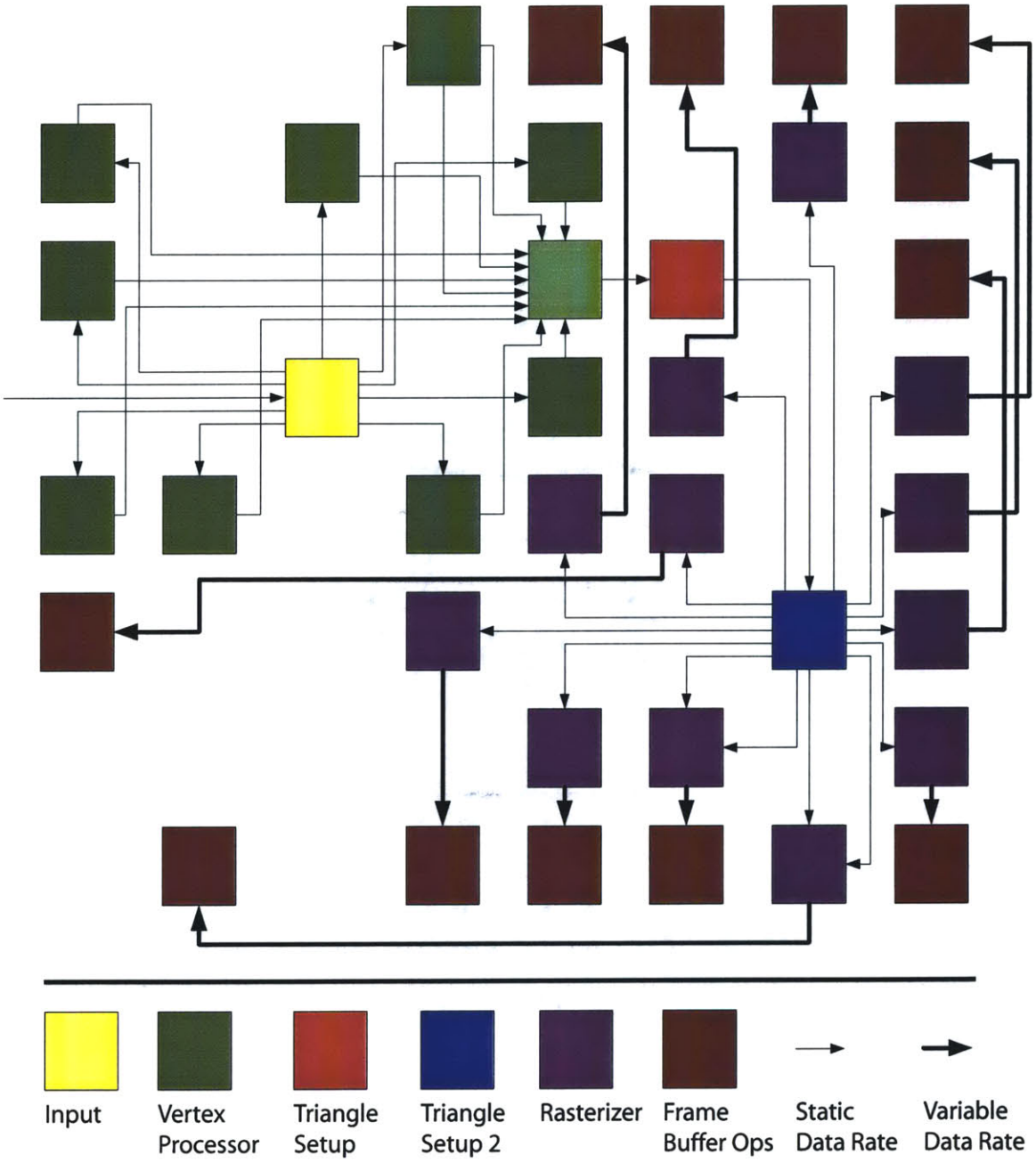


Figure 4-10: Compiler generated layout for case study #4, last case. 2 stage pipelined Triangle Setup. Unallocated tiles have been omitted to clarify routing.

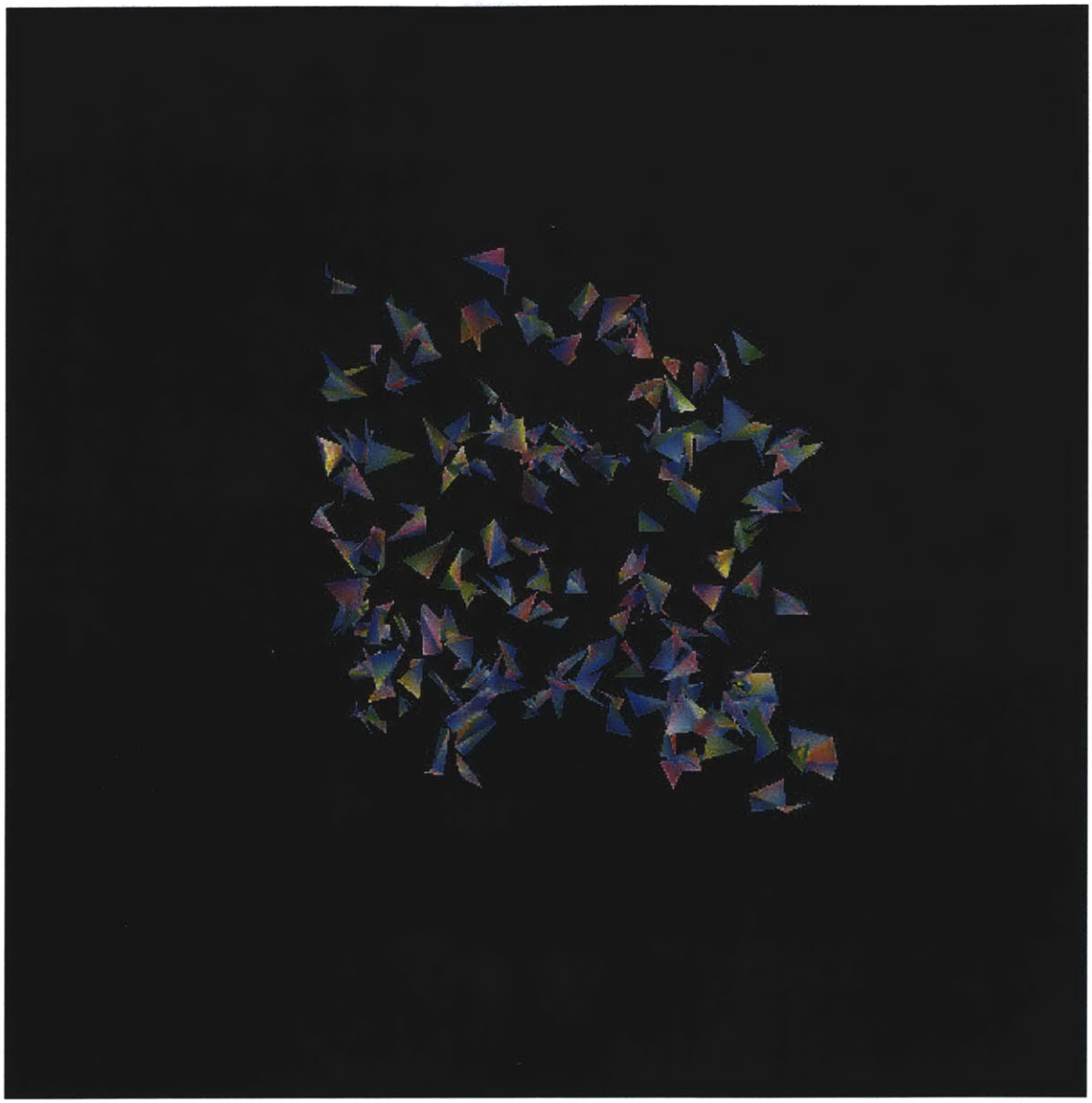


Figure 4-11: Case Study 4 Output image. Resolution: 600 × 600.

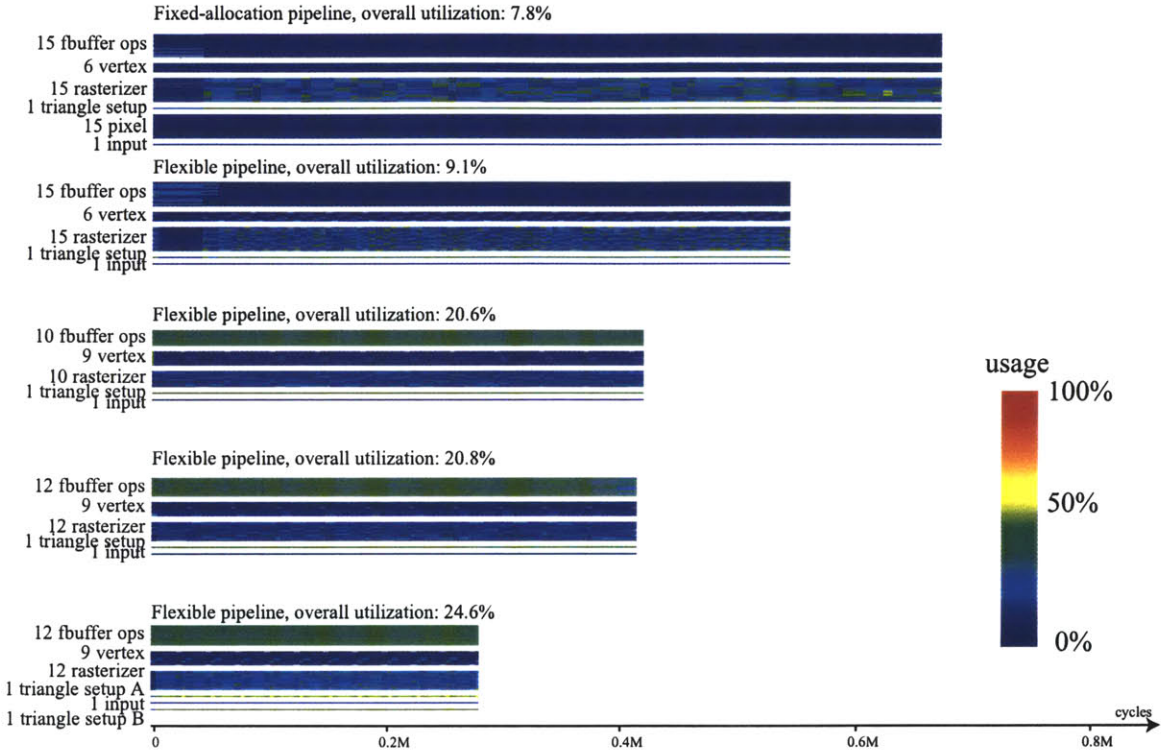


Figure 4-12: Case 4 utilization graphs.

Chapter 5

Conclusion

I have presented a graphics hardware architecture based on a multicore processor, where load balancing is achieved at compile-time using automatic resource allocation. Both the 3D rendering pipeline and shaders are expressed in the same stream-based language, allowing for full programmability and load balancing. Although the prototype cannot compete with state-of-the-art GPUs, I believe it is an important first step in addressing the load-balancing challenge in graphics architecture.

As discussed in Chapter 4, there are several limitations to the current design. One promising option for improving performance would be to replace certain Raw tiles with specialized rasterizers since this is the stage of the graphics pipeline that benefits most from specialization. Another direction for future research is studying the memory hierarchy for optimal graphics performance, in particular the pre-fetching of textures. Texture mapping is an important part of modern rendering and was not considered in this thesis. Dynamic load balancing is the most exciting avenue of future work. A first intermediate step might exploit the statistics from the previous frame to refine resource allocation or switch between different pre-compiled versions of the pipeline. In the future, I hope that graphics hardware will be able to introspect itself and switch resource allocation within a frame or rendering pass depending on the relative load of computation units and on the occupancy of its buffers. Achieving the proper granularity for such changes and the appropriate state maintenance are the biggest challenges.

Appendix A

Code

```
Vertex->TriangleSetupInfo filter TriangleSetup( int screenWidth, int screenHeight )
{
    Vertex v0;
    Vertex v1;
    Vertex v2;

    TriangleSetupInfo tsi;

    Vector3f[3] ndcSpace;
    Vector3f[3] screenSpace;

    Matrix3f vertexMatrix;
    Matrix3f vertexMatrixInverse;

    void normalizeW()
        {
            ndcSpace[0].x = v0.position.x / v0.position.w;
            ndcSpace[0].y = v0.position.y / v0.position.w;
            ndcSpace[0].z = v0.position.z / v0.position.w;

            ndcSpace[1].x = v1.position.x / v1.position.w;
            ndcSpace[1].y = v1.position.y / v1.position.w;
            ndcSpace[1].z = v1.position.z / v1.position.w;

            ndcSpace[2].x = v2.position.x / v2.position.w;
            ndcSpace[2].y = v2.position.y / v2.position.w;
            ndcSpace[2].z = v2.position.z / v2.position.w;
        }

    void viewport()
        {
            screenSpace[0].x = screenWidth * ( ndcSpace[0].x + 1.0 ) / 2.0;
            screenSpace[0].y = screenHeight * ( ndcSpace[0].y + 1.0 ) / 2.0;
            // shift z range from [-1..1] to [0..1]
            screenSpace[0].z = ( ndcSpace[0].z + 1.0 ) / 2.0;

            screenSpace[1].x = screenWidth * ( ndcSpace[1].x + 1.0 ) / 2.0;
            screenSpace[1].y = screenHeight * ( ndcSpace[1].y + 1.0 ) / 2.0;
            // shift z range from [-1..1] to [0..1]
            screenSpace[1].z = ( ndcSpace[1].z + 1.0 ) / 2.0;

            screenSpace[2].x = screenWidth * ( ndcSpace[2].x + 1.0 ) / 2.0;
            screenSpace[2].y = screenHeight * ( ndcSpace[2].y + 1.0 ) / 2.0;
            // shift z range from [-1..1] to [0..1]
            screenSpace[2].z = ( ndcSpace[2].z + 1.0 ) / 2.0;
        }

    void computeScreenBoundingBox()
        {
            int v0x, v0y, v1x, v1y, v2x, v2y;
            int temp;

            v0x = ( int )( screenSpace[0].x );
            v0y = ( int )( screenSpace[0].y );
            v1x = ( int )( screenSpace[1].x );
            v1y = ( int )( screenSpace[1].y );
            v2x = ( int )( screenSpace[2].x );
            v2y = ( int )( screenSpace[2].y );

            // x max
            if( v0x > v1x )
                {
                    temp = v0x;
                }
            else
                {
                    temp = v1x;
                }

            if( v2x > temp )
                {
                    temp = v2x;
                }

            tsi.maxX = temp + 1;

            if( v0y > v1y )
                {
                    temp = v0y;
                }
            else
                {
                    temp = v1y;
                }

            if( v2y > temp )
                {
                    temp = v2y;
                }

            tsi.maxY = temp + 1;

            // x min
            if( v0x < v1x )
                {
                    temp = v0x;
                }
            else
                {
                    temp = v1x;
                }

            if( v2x < temp )
                {
                    temp = v2x;
                }

            tsi.minX = temp;

            // y min
            if( v0y < v1y )
                {
                    temp = v0y;
                }
            else
                {
                    temp = v1y;
                }

            if( v2y < temp )
                {
                    temp = v2y;
                }

            tsi.minY = temp;

            if( tsi.minX < 0 )
                {
                    tsi.minX = 0;
                }
            if( tsi.maxX > ( screenWidth - 1 ) )
                {
                    tsi.maxX = screenWidth - 1;
                }

            if( tsi.minY < 0 )
                {
                    tsi.minY = 0;
                }
            if( tsi.maxY > ( screenHeight - 1 ) )
```

Figure A-1: Common Triangle Setup Code, Page 1 of 3.

```
            {
                tsi.maxY = screenHeight - 1;
            }
        }

    void computeVertexMatrix()
        {
            vertexMatrix.m[0] = v0.position.x;
            vertexMatrix.m[3] = v0.position.y;
            vertexMatrix.m[6] = v0.position.w;

            vertexMatrix.m[1] = v1.position.x;
            vertexMatrix.m[4] = v1.position.y;
            vertexMatrix.m[7] = v1.position.w;

            vertexMatrix.m[2] = v2.position.x;
            vertexMatrix.m[5] = v2.position.y;
            vertexMatrix.m[8] = v2.position.w;
        }

    void computeVertexMatrixInverse()
        {
            float d;

            d = ( vertexMatrix.m[0] * vertexMatrix.m[4] * vertexMatrix.m[8] - vertexMa
trix.m[0] * vertexMatrix.m[5] * vertexMatrix.m[7] - vertexMatrix.m[3] * vertexMatrix.m
[1] * vertexMatrix.m[8] + vertexMatrix.m[3] * vertexMatrix.m[2] * vertexMatrix.m[7] +
vertexMatrix.m[6] * vertexMatrix.m[1] * vertexMatrix.m[5] - vertexMatrix.m[6] * Vertex
Matrix.m[2] * vertexMatrix.m[4] );

            vertexMatrixInverse.m[0] = ( vertexMatrix.m[4] * vertexMatrix.m[8] - verte
xMatrix.m[5] * vertexMatrix.m[7] ) / d;
            vertexMatrixInverse.m[3] = -( vertexMatrix.m[3] * vertexMatrix.m[8] - Vert
exMatrix.m[5] * vertexMatrix.m[6] ) / d;
            vertexMatrixInverse.m[6] = -(-vertexMatrix.m[3] * vertexMatrix.m[7] + Vert
exMatrix.m[4] * vertexMatrix.m[6] ) / d;
            vertexMatrixInverse.m[1] = -( vertexMatrix.m[1] * vertexMatrix.m[8] - Vert
exMatrix.m[2] * vertexMatrix.m[7] ) / d;
            vertexMatrixInverse.m[4] = ( vertexMatrix.m[0] * vertexMatrix.m[8] - verte
xMatrix.m[2] * vertexMatrix.m[6] ) / d;
            vertexMatrixInverse.m[7] = -( vertexMatrix.m[0] * vertexMatrix.m[7] - vert
exMatrix.m[1] * vertexMatrix.m[6] ) / d;
            vertexMatrixInverse.m[2] = -(-vertexMatrix.m[1] * vertexMatrix.m[5] + Vert
exMatrix.m[2] * vertexMatrix.m[4] ) / d;
            vertexMatrixInverse.m[5] = -( vertexMatrix.m[0] * vertexMatrix.m[5] - vert
exMatrix.m[2] * vertexMatrix.m[3] ) / d;
            vertexMatrixInverse.m[8] = ( vertexMatrix.m[0] * vertexMatrix.m[4] - verte
xMatrix.m[1] * vertexMatrix.m[3] ) / d;
        }

    void computeEdgeEquations()
        {
            // edge01 = vertexMatrixInverse * [0 0 1]^T
            tsi.edge01.x = vertexMatrixInverse.m[6];
            tsi.edge01.y = vertexMatrixInverse.m[7];
            tsi.edge01.z = vertexMatrixInverse.m[8];

            // edge12 = vertexMatrixInverse * [1 0 0]^T
            tsi.edge12.x = vertexMatrixInverse.m[0];
            tsi.edge12.y = vertexMatrixInverse.m[1];
            tsi.edge12.z = vertexMatrixInverse.m[2];

            // edge20 = vertexMatrixInverse * [0 1 0]^T
            tsi.edge20.x = vertexMatrixInverse.m[3];
            tsi.edge20.y = vertexMatrixInverse.m[4];
            tsi.edge20.z = vertexMatrixInverse.m[5];
        }

    void computeWInterp()
        {
            // w coefficients = vertexMatrixInverse * [ 1 1 1 ]^T
            tsi.wInterp.x = vertexMatrixInverse.m[0] + vertexMatrixInverse.m[3] + vert
exMatrixInverse.m[6];
            tsi.wInterp.y = vertexMatrixInverse.m[1] + vertexMatrixInverse.m[4] + vert
exMatrixInverse.m[7];
            tsi.wInterp.z = vertexMatrixInverse.m[2] + vertexMatrixInverse.m[5] + vert
exMatrixInverse.m[8];
        }


    void computeZInterp()
        {
            tsi.zInterp.x = vertexMatrixInverse.m[0] * screenSpace[0].z + vertexMatrix
Inverse.m[3] * screenSpace[1].z + vertexMatrixInverse.m[6]  * screenSpace[2].z;
            tsi.zInterp.y = vertexMatrixInverse.m[1] * screenSpace[0].z + vertexMatrix
Inverse.m[4] * screenSpace[1].z + vertexMatrixInverse.m[7]  * screenSpace[2].z;
            tsi.zInterp.z = vertexMatrixInverse.m[2] * screenSpace[0].z + vertexMatrix
Inverse.m[5] * screenSpace[1].z + vertexMatrixInverse.m[8] * screenSpace[2].z;
        }

    void determineFacing()
        {
            float e01x = screenSpace[1].x - screenSpace[0].x;
            float e01y = screenSpace[1].y - screenSpace[0].y;

            float e12x = screenSpace[2].x - screenSpace[1].x;
            float e12y = screenSpace[2].y - screenSpace[1].y;

            float z = e01x * e12y - e01y * e12x;
            if( z > 0 )
                {
                    tsi.isFrontFacing = 1;
                }
            else
                {
                    tsi.isFrontFacing = 0;
                }
        }

    float computeInterpolantX( float u0, float u1, float u2 )
        {
            return vertexMatrixInverse.m[0] * u0 + vertexMatrixInverse.m[3] * u1 + ver
texMatrixInverse.m[6] * u2;
        }

    float computeInterpolantY( float u0, float u1, float u2 )
        {
            return vertexMatrixInverse.m[1] * u0 + vertexMatrixInverse.m[4] * u1 + ver
texMatrixInverse.m[7] * u2;
        }

    float computeInterpolantZ( float u0, float u1, float u2 )
        {
            return vertexMatrixInverse.m[2] * u0 + vertexMatrixInverse.m[5] * u1 + ver
texMatrixInverse.m[8] * u2;
        }

    void computeNormalInterp()
        {
            // nx
            tsi.nxInterp.x = computeInterpolantX( v0.normal.x, v1.normal.x, v2.normal.
x );
            tsi.nxInterp.y = computeInterpolantY( v0.normal.x, v1.normal.x, v2.normal.
x );
            tsi.nxInterp.z = computeInterpolantZ( v0.normal.x, v1.normal.x, v2.normal.
x );

            // ny
            tsi.nyInterp.x = computeInterpolantX( v0.normal.y, v1.normal.y, v2.normal.
y );
```

Figure A-2: Common Triangle Setup Code, Page 2 of 3.

```
            tsi.nyInterp.y = computeInterpolantY( v0.normal.y, v1.normal.y, v2.normal.
y );
            tsi.nyInterp.z = computeInterpolantZ( v0.normal.y, v1.normal.y, v2.normal.
y );

            // nz
            tsi.nzInterp.x = computeInterpolantX( v0.normal.z, v1.normal.z, v2.normal.
z );
            tsi.nzInterp.y = computeInterpolantY( v0.normal.z, v1.normal.z, v2.normal.
z );
            tsi.nzInterp.z = computeInterpolantZ( v0.normal.z, v1.normal.z, v2.normal.
z );
        }

    void computeColorInterp()
        {
            // red
            tsi.rInterp.x = computeInterpolantX( v0.color.r, v1.color.r, v2.color.r );
            tsi.rInterp.y = computeInterpolantY( v0.color.r, v1.color.r, v2.color.r );
            tsi.rInterp.z = computeInterpolantZ( v0.color.r, v1.color.r, v2.color.r );

            // green
            tsi.gInterp.x = computeInterpolantX( v0.color.g, v1.color.g, v2.color.g );
            tsi.gInterp.y = computeInterpolantY( v0.color.g, v1.color.g, v2.color.g );
            tsi.gInterp.z = computeInterpolantZ( v0.color.g, v1.color.g, v2.color.g );

            // blue
            tsi.bInterp.x = computeInterpolantX( v0.color.b, v1.color.b, v2.color.b );
            tsi.bInterp.y = computeInterpolantY( v0.color.b, v1.color.b, v2.color.b );
            tsi.bInterp.z = computeInterpolantZ( v0.color.b, v1.color.b, v2.color.b );
        }

    void computeTextureInterp()
        {
            // t0.s
            tsi.t0sInterp.x = computeInterpolantX( v0.texCoord0.x, v1.texCoord0.x, v2.
texCoord0.x );
            tsi.t0sInterp.y = computeInterpolantY( v0.texCoord0.x, v1.texCoord0.x, v2.
texCoord0.x );
            tsi.t0sInterp.z = computeInterpolantZ( v0.texCoord0.x, v1.texCoord0.x, v2.
texCoord0.x );

            // t0.t
            tsi.t0tInterp.x = computeInterpolantX( v0.texCoord0.y, v1.texCoord0.y, v2.
texCoord0.y );
            tsi.t0tInterp.y = computeInterpolantY( v0.texCoord0.y, v1.texCoord0.y, v2.
texCoord0.y );
            tsi.t0tInterp.z = computeInterpolantZ( v0.texCoord0.y, v1.texCoord0.y, v2.
texCoord0.y );

            // t0.p
            tsi.t0pInterp.x = computeInterpolantX( v0.texCoord0.z, v1.texCoord0.z, v2.
texCoord0.z );
            tsi.t0pInterp.y = computeInterpolantY( v0.texCoord0.z, v1.texCoord0.z, v2.
texCoord0.z );
            tsi.t0pInterp.z = computeInterpolantZ( v0.texCoord0.z, v1.texCoord0.z, v2.
texCoord0.z );

            // t0.q
            tsi.t0qInterp.x = computeInterpolantX( v0.texCoord0.w, v1.texCoord0.w, v2.
texCoord0.w );
            tsi.t0qInterp.y = computeInterpolantY( v0.texCoord0.w, v1.texCoord0.w, v2.
texCoord0.w );
            tsi.t0qInterp.z = computeInterpolantZ( v0.texCoord0.w, v1.texCoord0.w, v2.
texCoord0.w );
        }

    work pop 3 push 1
        {
            v0 = pop();
            v1 = pop();
            v2 = pop();

            computeVertexMatrix();
            computeVertexMatrixInverse();

            normalizeW(); // clip space --> ndc space
            viewport(); // ndc space --> screen space

            computeScreenBoundingBox();
            computeEdgeEquations();

            // special interpolants
            computeWInterp();
            computeZInterp();

            // other interpolants
            computeNormalInterp();
            computeColorInterp();
            computeTextureInterp();

            // determine backfacing
            determineFacing();

            // push out
            push( tsi );
        }
    }
```

Figure A-3: Common Triangle Setup Code, Page 3 of 3.

```
// offset = rasterizer number (0, 1, 2, ... numUnits - 1)
TriangleSetupInfo->Fragment filter Rasterizer( int offset, int numUnits, int screenWid
th, int screenHeight )
{
    int numColumns;
    TriangleSetupInfo tsi;

    init
        {
            numColumns = screenWidth / numUnits;
        }

    float interpolate( float interpX, float interpY, float interpZ,
                       float ndcX, float ndcY, float w )
        {
            return( ( interpX * ndcX + interpY * ndcY + interpZ ) * w );
        }

    work pop 1 push *
        {
            tsi = pop();

            // given an x coordinate:
            // x / numUnits = group number
            // x % numUnits = offset within group
            // group number * numUnits = start of group

            int groupNumber = tsi.minX / numUnits;
            int startOfGroup = groupNumber * numUnits;
            int xStart = startOfGroup + offset;
            if( xStart < tsi.minX )
                {
                    xStart = xStart + numUnits;
                }

            for( int y = tsi.minY; y <= tsi.maxY; ++y )
                {
                    // for( int x = offset; x < tsi.maxX; x = x + numUnits )
                    for( int x = xStart; x < tsi.maxX; x = x + numUnits )
                        {
                            // compute NDC coordinates for current pixel position
                            float ndcX = ( float )( x ) * 2.0 / ( float )screenWidth -
 1.0;
                            float ndcY = ( float )( y ) * 2.0 / ( float )screenHeight
- 1.0;

                            // interpolate w
                            float w = 1.0 / ( tsi.wInterp.x * ndcX + tsi.wInterp.y * n
dcY + tsi.wInterp.z );

                            float inside01 = interpolate( tsi.edge01.x, tsi.edge01.y,
tsi.edge01.z, ndcX, ndcY, w );
                            float inside12 = interpolate( tsi.edge12.x, tsi.edge12.y,
tsi.edge12.z, ndcX, ndcY, w );
                            float inside20 = interpolate( tsi.edge20.x, tsi.edge20.y,
tsi.edge20.z, ndcX, ndcY, w );
                            if( inside01 >= 0 && inside12 >= 0 && inside20 >= 0 )
                                {
                                    // interpolate z
                                    float z = interpolate( tsi.zInterp.x, tsi.zInterp.
y, tsi.zInterp.z, ndcX, ndcY, w );
                                    if( z >= 0 )
                                        {
                                            Fragment f;
                                            f.x = x;
                                            f.y = y;
                                            f.z = z;
                                            f.isFrontFacing = tsi.isFrontFacing;

                                            f.nx = interpolate( tsi.nxInterp.x, tsi.nx
Interp.y, tsi.nxInterp.z, ndcX, ndcY, w );
                                            f.ny = interpolate( tsi.nyInterp.x, tsi.ny
Interp.y, tsi.nyInterp.z, ndcX, ndcY, w );
                                            f.nz = interpolate( tsi.nzInterp.x, tsi.nz
Interp.y, tsi.nzInterp.z, ndcX, ndcY, w );

                                            f.r = interpolate( tsi.rInterp.x, tsi.rInt
erp.y, tsi.rInterp.z, ndcX, ndcY, w ),
                                            f.g = interpolate( tsi.gInterp.x, tsi.gInt
erp.y, tsi.gInterp.z, ndcX, ndcY, w );
                                            f.b = interpolate( tsi.bInterp.x, tsi.bInt
erp.y, tsi.bInterp.z, ndcX, ndcY, w );

                                            f.texCoord0.x = interpolate( tsi.t0sInterp
.x, tsi.t0sInterp.y, tsi.t0sInterp.z, ndcX, ndcY, w );
                                            f.texCoord0.y = interpolate( tsi.t0tInterp
.x, tsi.t0tInterp.y, tsi.t0tInterp.z, ndcX, ndcY, w );
                                            f.texCoord0.z = interpolate( tsi.t0pInterp
.x, tsi.t0pInterp.y, tsi.t0pInterp.z, ndcX, ndcY, w );
                                            f.texCoord0.w = interpolate( tsi.t0qInterp
.x, tsi.t0qInterp.y, tsi.t0qInterp.z, ndcX, ndcY, w );

                                            // push fragment
                                            push( f );
                                        }
                                }
                        }
                }
        }
}
```

Figure A-4: Common Rasterizer code, using the homogeneous rasterization algorithm.

```
Raster->void filter FrameBufferOps( int offset, int numUnits, int screenWidth, int scr
eenHeight )
{
    int[ ( screenWidth / numUnits ) * screenHeight ] rgb;
    float[ ( screenWidth / numUnits ) * screenHeight ] zBuffer;

    int width;

    init
        {
            width = screenWidth / numUnits;

            for( int i = 0; i < width * screenHeight; ++i )
                {
                    zBuffer[i] = 1.1;
                }
        }

    work pop 1
        {
            Raster r = pop();

            r.x = r.x / numUnits;

            int index = r.x * screenHeight + r.y;
            if( r.z < zBuffer[ index ] )
                {
                    rgb[ index ] = ( ( int )( r.r * 255.0 ) << 16 ) | ( ( int )( r.g *
 255.0 ) << 8 ) | ( ( int )( r.b * 255.0 ) );
                    zBuffer[ index ] = r.z;
                }
        }
}
```

Figure A-5: Common Frame Buffer Operations Code.

```
Vertex->Vertex filter VertexShader( int id )
{
    Matrix4f modelView;
    Matrix4f projection;

    float worldX;
    float worldY;
    float worldZ;
    float worldW;

    float worldNX;
    float worldNY;
    float worldNZ;

    float eyeX;
    float eyeY;
    float eyeZ;
    float eyeW;

    float clipX;
    float clipY;
    float clipZ;
    float clipW;

    float inR;
    float inG;
    float inB;

    init
    {
        modelView.m[0] = 1;
        modelView.m[1] = 0;
        modelView.m[2] = 0;
        modelView.m[3] = 0;

        modelView.m[4] = 0;
        modelView.m[5] = 1;
        modelView.m[6] = 0;
        modelView.m[7] = 0;

        modelView.m[8] = 0;
        modelView.m[9] = 0;
        modelView.m[10] = 1;
        modelView.m[11] = 0;

        modelView.m[12] = 0;
        modelView.m[13] = 0;
        modelView.m[14] = -5;
        modelView.m[15] = 1;

        // nominal projection matrix
        // fov = 50 degrees, 1:1 aspect ratio, near = 1, far = 10
        projection.m[0] = 2.144507;
        projection.m[1] = 0;
        projection.m[2] = 0;
        projection.m[3] = 0;

        projection.m[4] = 0;
        projection.m[5] = 2.144507;
        projection.m[6] = 0;
        projection.m[7] = 0;

        projection.m[8] = 0;
        projection.m[9] = 0;
        projection.m[10] = -1.022222;
        projection.m[11] = -1;

        projection.m[12] = 0;
        projection.m[13] = 0;
        projection.m[14] = -2.022222;
        projection.m[15] = 0;
    }

    void computeEyeSpace()
    {
        eyeX = modelView.m[0] * worldX + modelView.m[4] * worldY + modelView.m[8]  * w
orldZ + modelView.m[12] * worldW;
        eyeY = modelView.m[1] * worldX + modelView.m[5] * worldY + modelView.m[9]  * w
orldZ + modelView.m[13] * worldW;
        eyeZ = modelView.m[2] * worldX + modelView.m[6] * worldY + modelView.m[10] * w
orldZ + modelView.m[14] * worldW;
        eyeW = modelView.m[3] * worldX + modelView.m[7] * worldY + modelView.m[11] * w
orldZ + modelView.m[15] * worldW;
    }

    void computeClipSpace()
    {
        clipX = projection.m[0] * eyeX + projection.m[4] * eyeY + projection.m[8]  * e
yeZ + projection.m[12] * eyeW;
        clipY = projection.m[1] * eyeX + projection.m[5] * eyeY + projection.m[9]  * e
yeZ + projection.m[13] * eyeW;
        clipZ = projection.m[2] * eyeX + projection.m[6] * eyeY + projection.m[10] * e
yeZ + projection.m[14] * eyeW;
        clipW = projection.m[3] * eyeX + projection.m[7] * eyeY + projection.m[11] * e
yeZ + projection.m[15] * eyeW;
    }

    work pop 1 push 1
    {
        Vertex vIn = pop();

        worldX = vIn.position.x;
        worldY = vIn.position.y;
        worldZ = vIn.position.z;
        worldW = vIn.position.w;

        // "ftransform()"
        computeEyeSpace();
        computeClipSpace();

        Vertex vOut;

        vOut.position.x = clipX;
        vOut.position.y = clipY;
        vOut.position.z = clipZ;
        vOut.position.w = clipW;

        vOut.normal.x = vIn.normal.x;
        vOut.normal.y = vIn.normal.y;
        vOut.normal.z = vIn.normal.z;

        vOut.color.r = vIn.color.r;
        vOut.color.g = vIn.color.g;
        vOut.color.b = vIn.color.b;

        // copy position as texture coordinate
        vOut.texCoord0.x = vIn.position.x;
        vOut.texCoord0.y = vIn.position.y;
        vOut.texCoord0.z = vIn.position.z;

        push( vOut );
    }
}
```

Figure A-6: Case Study #1 Vertex Shader Code.

```
Fragment->Raster filter PixelShader( int id )
{
    Vector3f lightPosition;
    Vector3f eyePosition;
    float shininess;

    init
        {
            lightPosition.x = -0.75;
            lightPosition.y = 0;
            lightPosition.z = 1.0;

            eyePosition.x = 0;
            eyePosition.y = 0;
            eyePosition.z = 5;

            shininess = 20.0;
        }

    float max( float x, float y )
        {
            if( x > y )
                {
                    return x;
                }
            else
                {
                    return y;
                }
        }

    work pop 1 push 1
        {
            Fragment f = pop();

            // compute light vector
            Vector3f lightVector;
            lightVector.x = lightPosition.x - f.texCoord0.x;
            lightVector.y = lightPosition.y - f.texCoord0.y;
            lightVector.z = lightPosition.z - f.texCoord0.z;

            // normalize light vector
            float lvNorm = sqrt( lightVector.x * lightVector.x + lightVector.y * light
Vector.y + lightVector.z * lightVector.z );
            lightVector.x /= lvNorm;
            lightVector.y /= lvNorm;
            lightVector.z /= lvNorm;

            // compute view vector
            Vector3f viewVector;
            viewVector.x = eyePosition.x - f.texCoord0.x;
            viewVector.y = eyePosition.y - f.texCoord0.y;
            viewVector.z = eyePosition.z - f.texCoord0.z;

            // normalize view vector
            float vvNorm = sqrt( viewVector.x * viewVector.x + viewVector.y * viewVect
or.y + viewVector.z * viewVector.z );
            viewVector.x /= vvNorm;
            viewVector.y /= vvNorm;
            viewVector.z /= vvNorm;

            // normalize normal
            float normalNorm = sqrt( f.nx * f.nx + f.ny * f.ny + f.nz * f.nz );

            f.nx /= normalNorm;
            f.ny /= normalNorm;
            f.nz /= normalNorm;

            // diffuse contribution
            float lDotN = f.nx * lightVector.x + f.ny * lightVector.y + f.nz * lightVe
ctor.z;
            lDotN = max( 0.0, lDotN );

            // specular contribution

            // compute reflection vector
            Vector3f reflectVector;
            reflectVector.x = 2.0 * lDotN * f.nx - lightVector.x;
            reflectVector.y = 2.0 * lDotN * f.ny - lightVector.y;
            reflectVector.z = 2.0 * lDotN * f.nz - lightVector.z;

            // normalize reflection vector
            float rvNorm = sqrt( reflectVector.x * reflectVector.x + reflectVector.y *
 reflectVector.y + reflectVector.z * reflectVector.z );
            reflectVector.x /= rvNorm;
            reflectVector.y /= rvNorm;
            reflectVector.z /= rvNorm;

            float rDotV = viewVector.x * reflectVector.x + viewVector.y * reflectVecto
r.y + viewVector.z * reflectVector.z;
            float iSpecular = pow( rDotV, shininess );
            iSpecular = max( 0.0, iSpecular );

            Raster r;
            r.x = f.x;
            r.y = f.y;
            r.z = f.z;

            r.r = f.r * lDotN + iSpecular;
            r.g = f.g * lDotN + iSpecular;
            r.b = f.b * lDotN + iSpecular;

            if( r.r > 1.0 )
                {
                    r.r = 1.0;
                }
            if( r.g > 1.0 )
                {
                    r.g = 1.0;
                }
            if( r.b > 1.0 )
                {
                    r.b = 1.0;
                }

            push( r );
        }
}
```

Figure A-7: Case Study #1 Pixel Shader Code.

```
Raster->void filter FrameBufferOps( int offset, int numUnits, int screenWidth, int scr
eenHeight )
{
    float[ ( screenWidth / numUnits ) * screenHeight  ] zBuffer = init_array_1D_float(
 "Pass0.zBuffer.xy." + offset + ".arr", ( screenWidth / numUnits ) * screenHeight );
    int[ ( screenWidth / numUnits ) * screenHeight ] stencilBuffer;

    int width;

    init
        {
            width = screenWidth / numUnits;

            for( int i = 0; i < width * screenHeight; ++i )
                {
                    zBuffer[i] = 1.1;
                }
        }

    work pop 1
        {
            Raster r = pop();

            r.x = r.x / numUnits;
            int index = r.x * screenHeight + r.y;

            // zFail algorithm
            if( r.z >= zBuffer[ index ] )
                {
                    if( r.isFrontFacing == 1 )
                        {
                            stencilBuffer[ index ] = stencilBuffer[ index ] - 1;
                        }
                    else
                        {
                            stencilBuffer[ index ] = stencilBuffer[ index ] + 1;
                        }
                }
        }
}
```

Figure A-8: Case Study #2: Shadow Volumes Z-Fail Pass Frame Buffer Operations Code.

```
int->void filter PoissonDOF( int width, int height, int lowWidth, int lowHeight )
{
    int[width * height] inputPackedArray = init_array_1D_int( "input_high.arr", width
* height );
    int[lowWidth * lowHeight] inputPackedArrayLow = init_array_1D_int( "input_low.arr"
, lowWidth * lowHeight );
    int[width * height] outputPackedArray;

    float maxCoCRadius;
    float radiusScale;

    float[8] poissonX = { -0.45, -0.9, -0.85, -0.2, 0.4, 0.55, 0.33, 0.8 };
    float[8] poissonY = { 0.04, 0.4, -0.3, -0.6, 0.34, -0.2, -0.58, 0.3 };

    float tmpR;
    float tmpG;
    float tmpB;

    float tmpRLow;
    float tmpGLow;
    float tmpBLow;

    int iTmpR;
    int iTmpG;
    int iTmpB;

    init
        {
            maxCoCRadius = 5.0;
            radiusScale = 0.25;
        }

    void getHigh( float x, float y )
        {
            int x0 = ( int )x;
            int x1 = x0 + 1;
            int y0 = ( int )y;
            int y1 = y0 + 1;

            int val00 = inputPackedArray[ y0 * width + x0 ];
            int val01 = inputPackedArray[ y1 * width + x0 ];
            int val10 = inputPackedArray[ y0 * width + x1 ];
            int val11 = inputPackedArray[ y1 * width + x1 ];

            float fracX = x - x0;
            float fracY = y - y0;

            float blue00 = ( val00 & 0xff ) / 255.0;
            float green00 = ( ( val00 >> 8 ) & 0xff ) / 255.0;
            float red00 = ( ( val00 >> 16 ) & 0xff ) / 255.0;

            float blue01 = ( val01 & 0xff ) / 255.0;
            float green01 = ( ( val01 >> 8 ) & 0xff ) / 255.0;
            float red01 = ( ( val01 >> 16 ) & 0xff ) / 255.0;

            float blue10 = ( val10 & 0xff ) / 255.0;
            float green10 = ( ( val10 >> 8 ) & 0xff ) / 255.0;
            float red10 = ( ( val10 >> 16 ) & 0xff ) / 255.0;

            float blue11 = ( val11 & 0xff ) / 255.0;
            float green11 = ( ( val11 >> 8 ) & 0xff ) / 255.0;
            float red11 = ( ( val11 >> 16 ) & 0xff ) / 255.0;

            float redTop = red00 + fracX * ( red10 - red00 );
            float redBot = red01 + fracX * ( red11 - red01 );

            float greenTop = green00 + fracX * ( green10 - green00 );
            float greenBot = green01 + fracX * ( green11 - green01 );

            float blueTop = blue00 + fracX * ( blue10 - blue00 );

            float blueBot = blue01 + fracX * ( blue11 - blue01 );

            tmpR = redTop + fracY * ( redBot - redTop );
            tmpG = greenTop + fracY * ( greenBot - greenTop );
            tmpB = blueTop + fracY * ( blueBot - blueTop );
        }

    work pop 1
        {
            pop();

            for( int y = 5; y < height - 5; ++y )
                {
                    print( 1 );
                    for( int x = 5; x < width - 5; ++x )
                        {
                            // fetch center tap
                            float blurriness = ( ( inputPackedArray[ y * width + x ] >
> 24 ) & 0x7F ) / 127.0;

                            float radiusHigh = blurriness * maxCoCRadius;
                            // float radiusLow = blurriness * radiusScale;

                            float redAccum = 0;
                            float greenAccum = 0;
                            float blueAccum = 0;

                            float xx = x / 5.0;
                            float yy = y / 5.0;

                            float coordLowX;
                            float coordLowY;
                            float coordHighX;
                            float coordHighY;

                            float redTapLow;
                            float greenTapLow;
                            float blueTapLow;

                            float redTapHigh;
                            float greenTapHigh;
                            float blueTapHigh;

                            float redTapBlurred;
                            float greenTapBlurred;
                            float blueTapBlurred;

                            int tmpX;
                            int tmpY;
                            int val;

                            for( int k = 0; k < 8; ++k )
                                {
                                    coordHighX = x + ( poissonX[k] * radiusHigh );
                                    coordHighY = y + ( poissonY[k] * radiusHigh );

                                    coordLowX = coordHighX / 5.0;
                                    coordLowY = coordHighY / 5.0;

                                    tmpX = ( int )( x + 0.5 );
                                    tmpY = ( int )( y + 0.5 );

                                    val = inputPackedArrayLow[ tmpY * lowWidth + tmpX
];

                                    tmpRLow = ( ( val >> 16 ) & 0xff ) / 255.0;
                                    tmpGLow = ( ( val >> 8 ) & 0xff ) / 255.0;
                                    tmpBLow = ( val & 0xff ) / 255.0;

                                    val = getHigh( coordHighX, coordHighY );
```

Figure A-9: Case Study #3 Poisson Disc Filter Code, Page 1 of 2.

```
                                tmpR = ( ( val >> 16 ) & 0xff ) / 255.0;
                                tmpG = ( ( val >> 8 ) & 0xff ) / 255.0;
                                tmpB = ( val & 0xff ) / 255.0;

                                redTapBlurred = tmpR + blurriness * ( tmpRLow - tm
pR );
                                greenTapBlurred = tmpG + blurriness * ( tmpGLow -
tmpG );
                                blueTapBlurred = tmpB + blurriness * ( tmpBLow - t
mpB );

                                redAccum = redAccum + redTapBlurred;
                                greenAccum = greenAccum + greenTapBlurred;
                                blueAccum = blueAccum + blueTapBlurred;
                            }

                        tmpR = redAccum * 0.125;
                        tmpG = greenAccum * 0.125;
                        tmpB = blueAccum * 0.125;

                        iTmpR = ( int )( 255.0 * tmpR );
                        iTmpG = ( int )( 255.0 * tmpG );
                        iTmpB = ( int )( 255.0 * tmpB );

                        outputPackedArray[ y * width + x ] = ( iTmpB | ( iTmpG <<
8 ) | ( iTmpB << 16 ) );
                    }
                }
            }
        }
```

Figure A-10: Case Study #3 Poisson Disc Filter Code, Page 2 of 2.

```
Vertex->Vertex filter VertexShader( int id )
{
    Vertex[6] vertices;

    Matrix4f modelView;
    Matrix4f projection;

    float worldX;
    float worldY;
    float worldZ;
    float worldW;

    float eyeX;
    float eyeY;
    float eyeZ;
    float eyeW;

    float clipX;
    float clipY;
    float clipZ;
    float clipW;

    init
        {
            // modelview matrix, identity for now
            modelView.m[0] = 1;
            modelView.m[1] = 0;
            modelView.m[2] = 0;
            modelView.m[3] = 0;

            modelView.m[4] = 0;
            modelView.m[5] = 1;
            modelView.m[6] = 0;
            modelView.m[7] = 0;

            modelView.m[8] = 0;
            modelView.m[9] = 0;
            modelView.m[10] = 1;
            modelView.m[11] = 0;

            modelView.m[12] = 0;
            modelView.m[13] = 0;
            modelView.m[14] = -5;
            modelView.m[15] = 1;

            // nominal projection matrix
            // fov = 50 degrees, 1:1 aspect ratio, near = 1, far = 10
            projection.m[0] = 2.144507;
            projection.m[1] = 0;
            projection.m[2] = 0;
            projection.m[3] = 0;

            projection.m[4] = 0;
            projection.m[5] = 2.144507;
            projection.m[6] = 0;
            projection.m[7] = 0;

            projection.m[8] = 0;
            projection.m[9] = 0;
            projection.m[10] = -1.022222;
            projection.m[11] = -1;

            projection.m[12] = 0;
            projection.m[13] = 0;
            projection.m[14] = -2.022222;
            projection.m[15] = 0;
        }

    void tesselate()
        {
            vertices[3].position.x = ( vertices[0].position.x + vertices[1].position.x
) / 2.0;
            vertices[3].position.y = ( vertices[0].position.y + vertices[1].position.y
) / 2.0;
            vertices[3].position.z = ( vertices[0].position.z + vertices[1].position.z
) / 2.0;
            vertices[3].position.w = 1;

            vertices[3].color.r = ( vertices[0].color.r + vertices[1].color.r ) / 2.0;
            vertices[3].color.g = ( vertices[0].color.g + vertices[1].color.g ) / 2.0;
            vertices[3].color.b = ( vertices[0].color.b + vertices[1].color.b ) / 2.0;

            vertices[4].position.x = ( vertices[1].position.x + vertices[2].position.x
) / 2.0;
            vertices[4].position.y = ( vertices[1].position.y + vertices[2].position.y
) / 2.0;
            vertices[4].position.z = ( vertices[1].position.z + vertices[2].position.z
) / 2.0;
            vertices[4].position.w = 1;

            vertices[4].color.r = ( vertices[1].color.r + vertices[2].color.r ) / 2.0;
            vertices[4].color.g = ( vertices[1].color.g + vertices[2].color.g ) / 2.0;
            vertices[4].color.b = ( vertices[1].color.b + vertices[2].color.b ) / 2.0;

            vertices[5].position.x = ( vertices[0].position.x + vertices[2].position.x
) / 2.0;
            vertices[5].position.y = ( vertices[0].position.y + vertices[2].position.y
) / 2.0;
            vertices[5].position.z = ( vertices[0].position.z + vertices[2].position.z
) / 2.0;
            vertices[5].position.w = 1;

            vertices[5].color.r = ( vertices[0].color.r + vertices[2].color.r ) / 2.0;
            vertices[5].color.g = ( vertices[0].color.g + vertices[2].color.g ) / 2.0;
            vertices[5].color.b = ( vertices[0].color.b + vertices[2].color.b ) / 2.0;
        }

    void transform()
        {
            for( int i = 0; i < 6; ++i )
                {
                    worldX = vertices[i].position.x;
                    worldY = vertices[i].position.y;
                    worldZ = vertices[i].position.z;
                    worldW = vertices[i].position.w;

                    computeEyeSpace();
                    computeClipSpace();

                    vertices[i].position.x = clipX;
                    vertices[i].position.y = clipY;
                    vertices[i].position.z = clipZ;
                    vertices[i].position.w = clipW;
                }
        }

    void computeEyeSpace()
        {
            eyeX = modelView.m[0] * worldX + modelView.m[4] * worldY + modelView.m[8]
* worldZ + modelView.m[12] * worldW;
            eyeY = modelView.m[1] * worldX + modelView.m[5] * worldY + modelView.m[9]
* worldZ + modelView.m[13] * worldW;
            eyeZ = modelView.m[2] * worldX + modelView.m[6] * worldY + modelView.m[10]
* worldZ + modelView.m[14] * worldW;
            eyeW = modelView.m[3] * worldX + modelView.m[7] * worldY + modelView.m[11]
* worldZ + modelView.m[15] * worldW;
        }

    void computeClipSpace()
        {
```

Figure A-11: Case Study #4 Vertex Shader Code, Page 1 of 2.

```
            clipX = projection.m[0] * eyeX + projection.m[4] * eyeY + projection.m[8]
 * eyeZ + projection.m[12] * eyeW;
            clipY = projection.m[1] * eyeX + projection.m[5] * eyeY + projection.m[9]
 * eyeZ + projection.m[13] * eyeW;
            clipZ = projection.m[2] * eyeX + projection.m[6] * eyeY + projection.m[10]
 * eyeZ + projection.m[14] * eyeW;
            clipW = projection.m[3] * eyeX + projection.m[7] * eyeY + projection.m[11]
 * eyeZ + projection.m[15] * eyeW;
        }

    void randomize()
        {
            for( int i = 0; i < 6; ++i )
                {
                    // make a "random" vector
                    float rx = rand( -1, 1 );
                    float ry = rand( -1, 1 );
                    float rz = rand( -1, 1 );
                    float norm = sqrt( rx * rx + ry * ry + rz * rz );
                    rx = 0.05 * rx / norm;
                    ry = 0.05 * ry / norm;
                    rz = 0.05 * rz / norm;

                    vertices[i].x += rx;
                    vertices[i].y += ry;
                    vertices[i].z += rz;
                }
        }

    // uniform tesselation
    // one triangle in, 4 triangles out
    work pop 3 push 12
        {
            vertices[0] = pop();
            vertices[1] = pop();
            vertices[2] = pop();

            tesselate();
            transform();
            randomize();

            push( vertices[0] );
            push( vertices[3] );
            push( vertices[4] );

            push( vertices[3] );
            push( vertices[1] );
            push( vertices[4] );

            push( vertices[0] );
            push( vertices[4] );
            push( vertices[5] );

            push( vertices[5] );
            push( vertices[4] );
            push( vertices[2] );
        }
}
```

Figure A-12: Case Study #4 Vertex Shader Code, Page 2 of 2.

Appendix B

Figures

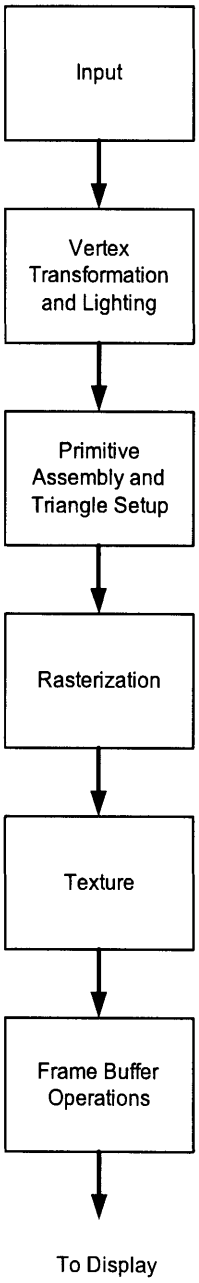


Figure B-1: Fixed function pipeline block diagram.

Bibliography

[1] Kurt Akeley. Reality Engine Graphics. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 109–116, 1993.

[2] OpenGL Architecture Review Board, Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide.* Addison-Wesley Publishing Company, Reading, Massachusetts, fourth edition, 14 November 2003.

[3] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004.

[4] Matthew Eldridge, Homan Igehy, and Pat Hanrahan. Pomegranate: A Fully Scalable Graphics Architecture. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 443–454, 2000.

[5] John Eyles, Steven Molnar, John Poulton, Trey Greer, Anselmo Lastra, Nick England, and Lee Westover. PixelFlow: The Realization. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 57–68, 1997.

[6] Homan Igehy, Gordon Stoll, and Pat Hanrahan. The Design of a Parallel Graphics Interface. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, pages 141–150, 1998.

[7] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A User-Programmable Vertex Engine. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 149–158, 2001.

[8] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Transactions on Graphics*, 22(3):896–907, 2003.

[9] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader Algebra. *ACM Transactions on Graphics*, 23(3):787–795, August 2004.

[10] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader Metaprogramming. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 57–68, 2002.

[11] Michael Gordon and William Thies and Michal Karczmarek and Jasper Lin and Ali S. Meli and Christopher Leger and Andrew A. Lamb and Jeremy Wong and Henry Hoffman and David Z. Maze and Saman Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[12] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.

[13] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. InfiniteReality: A Real-Time Graphics System. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 293–302, 1997.

[14] Satoshi Nishimura and Tosiyasu L. Kunii. VC-1: A Scalable Graphics Computer with Virtual Local Frame Buffers. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 365–372, 1996.

[15] Marc Olano and Trey Greer. Triangle Scan Conversion Using 2D Homogeneous Coordinates. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 89–95, 1997.

[16] John D. Owens, William J. Dally, Ujval J. Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery. Polygon Rendering on a Stream Architecture. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 23–32, 2000.

[17] John D. Owens, Brucek Khailany, Brian Towles, and William J. Dally. Comparing Reyes and OpenGL on a Stream Architecture. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 47–56, 2002.

[18] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A Real-time Procedural Shading System for Programmable Graphics Hardware. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 159–170, 2001.

[19] Thorsten Scheuermann. Advanced Depth of Field. *Game Developers Conference 2004*, 2004.

[20] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro*, pages 25–35, Mar/Apr 2002.

[21] Michael Bedford Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures. In *HPCA '03: Proceedings of The Ninth International Symposium on High-Performance Computer Architecture (HPCA'03)*, page 341, 2003.

[22] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, 2004.

[23] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction*, April 2002.