# SIMD Column–Parallel Polygon Rendering

by

Matthew Willard Eldridge

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

Bachelor of Science

and

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995



Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
May 12, 1995

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Seth Teller
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Elizabeth C. Palmer
Director, Employment and Development, David Sarnoff Research Center
Company Supervisor

Accepted by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

## SIMD Column–Parallel Polygon Rendering

by

Matthew Willard Eldridge

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 1995, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science
and
Master of Science

## Abstract

This thesis describes the design and implementation of a polygonal rendering system for a large one dimensional single instruction multiple data (SIMD) array of processors. Polygonal rendering enjoys a traditionally high level of available parallelism, but efficient realization of this parallelism requires communication. On large systems, such as the 1024–processor Princeton Engine studied here, communication overhead limits performance.

Special attention is paid to study of the analytical and experimental scalability of the design alternatives. Sort–last communication is determined to be the only design to offer linear performance improvements in the number of processors. We demonstrate that sort–first communication incurs increasingly high redundancy of calculations with increasing numbers of processors, while sort–middle communication has performance linear in the number of primitives and *independent* of the number of processors. Performance is thus be communication–limited on large sort–first and sort–middle systems.

The system described achieves interactive rendering rates of up to 30 frames per second at NTSC resolution. Scalable solutions such as sort–last proved too expensive to achieve real–time performance. Thus, to maintain interactivity a sort–middle render was implemented. A large set of communication optimizations are analyzed and implemented under the processor–managed communication architecture.

Finally a novel combination of sort–middle and sort–last communication is proposed and analyzed. The algorithm efficiently combines the performance of sort–middle architectures for small numbers of processors and polygons with the scalability of sort–last architectures for large numbers of processors to create a system with speedup linear in the number of processors. The communication structure is substantially more efficient than traditional sort–last architectures both in time and in memory.

Thesis Supervisor: Seth Teller

Title: Assistant Professor of Electrical Engineering and Computer Science

Company Supervisor: Elizabeth C. Palmer

Title: Director, Employment and Development, David Sarnoff Research Center

**SIMD Column–Parallel Polygon Rendering**

by

Matthew Willard Eldridge

Submitted to the Department of Electrical Engineering and Computer Science

on May 12, 1995, in partial fulfillment of the

requirements for the degrees of

Bachelor of Science

and

Master of Science

## Abstract

This thesis describes the design and implementation of a polygonal rendering system for a large one dimensional single instruction multiple data (SIMD) array of processors. Polygonal rendering enjoys a traditionally high level of available parallelism, but efficient realization of this parallelism requires communication. On large systems, such as the 1024–processor Princeton Engine studied here, communication overhead limits performance.

Special attention is paid to study of the analytical and experimental scalability of the design alternatives. Sort–last communication is determined to be the only design to offer linear performance improvements in the number of processors. We demonstrate that sort–first communication incurs increasingly high redundancy of calculations with increasing numbers of processors, while sort–middle communication has performance linear in the number of primitives and *independent* of the number of processors. Performance is thus be communication–limited on large sort–first and sort–middle systems.

The system described achieves interactive rendering rates of up to 30 frames per second at NTSC resolution. Scalable solutions such as sort–last proved too expensive to achieve real–time performance. Thus, to maintain interactivity a sort–middle render was implemented. A large set of communication optimizations are analyzed and implemented under the processor–managed communication architecture.

Finally a novel combination of sort–middle and sort–last communication is proposed and analyzed. The algorithm efficiently combines the performance of sort–middle architectures for small numbers of processors and polygons with the scalability of sort–last architectures for large numbers of processors to create a system with speedup linear in the number of processors. The communication structure is substantially more efficient than traditional sort–last architectures both in time and in memory.

Thesis Supervisor: Seth Teller

Title: Assistant Professor of Electrical Engineering and Computer Science

Company Supervisor: Elizabeth C. Palmer

Title: Director, Employment and Development, David Sarnoff Research Center

## Acknowledgments

I would like to thank David Sarnoff Research Center and the Sarnoff Real Time Corporation for their support of this research. In particular, Herb Taylor for his expertise in assembly level programming of the Princeton Engine and many useful discussions; Jim Fredrickson for his willingness to modify the operating system and programming environment during development of my thesis; Joe Peters for his patience with my complaints; and Jim Kaba for his tutoring and frequent insights. I would like to thank Scott Whitman for many helpful discussions about parallel rendering and so willingly sharing the Princeton Engine with me. Thanks to all my friends at David Sarnoff Research Center and Sarnoff Real Time Corporation, you made my work and research very enjoyable.

I would like to thank my advisor, Seth Teller, for providing continual motivation to improve and extend my research. Seth has been a never ending source of suggestions and ideas, and an inspiration.

Most of all, a special thank you to my parents for supporting me here during my undergraduate years and convincing me to pursue graduate studies. Without their love and support I would have never completed this thesis.

This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

This thesis explores the implementation of an interactive polygonal renderer on the Princeton Engine, a 1024–processor single–instruction multiple–data (SIMD) parallel computer. Each processor is a simple arithmetic core able to communicate with its left and right neighbors.

This is an important architecture on which to study the implementation of polygon rendering, for many reasons:

- The interprocessor communication structure is simple and inexpensive, making the construction of larger systems practical, with cost linear in the number of processors.

- The use of SIMD processors allows very carefully managed communication to be implemented, and affords significant opportunities for optimizations.

- Current polygon rendering algorithms and architectures often exhibit poor scalability, and are designed with large and expensive crossbars to insure adequate interprocessor bandwidth. A reduction in the amount of communication required and the careful structuring of the communication algorithm, as mandated by the modest communication resources of the Princeton Engine, will have large effects on the feasibility of future larger machines.

Many parallel renderers have been implemented, but most were either executed on smaller machines or on machines with a higher dimension communication network than the Princeton Engine. The use of a large machine with narrow interconnect such as the Princeton Engine makes the polygon rendering problem significantly different.

Polygon rendering provides a viewport into a three–dimensional (3D) space on a two–dimensional (2D) display device. The space may be of any sort, real or imagined. There are two broad approaches taken to polygon rendering: realism and interactivity. Realism requires precise and generally expensive computation to accurately model a scene. Interactivity requires real–time computation of a scene, at a possible loss of realism.

Polygon rendering models a scene as composed of a set of polygons in 3D being observed by a viewer from any arbitrary location. Geometry computations transform the polygonal scene from a 3D model to a 2D model and then rasterization computations render the 2D polygons to the display device. Both the geometry and the rasterization computations are highly parallel, but their parallelization exposes a sorting problem. Polygon rendering generally relies on the notion of polygons being rendered in the "correct" order, so that polygons nearer to the observer's viewpoint obscure polygons farther away. If the polygon rendering process is distributed across processors this ordering constraint will require interprocessor communication to resolve.

The ordering constraint is generally resolved in one of three ways, referred to by where they place the communication in the rendering process. Sort–first and sort–middle algorithms place polygons that overlap once projected to the screen on the same processor for rasterization so that the occlusion can be resolved with local data. Sort–last rasterizes each polygon on the processor that performs the polygon's geometry calculations, and then uses pixel by pixel information after the fact to combine all of the processor's rasterization results. The advantages and disadvantages of these approaches are discussed, both in general, and with a specific focus on an efficient implementation on the Princeton Engine.

A major goal of this thesis is to find and extract scalability from the explored communication algorithms, so that as the number of processors increases the rendering performance will increase commensurately. We demonstrate that for some communication approaches, most notably sort–middle, the communication time is linear in the number of polygons and *independent* of the number of processors. Without some care communication can dominate execution time of polygon rendering on large machines. Sort–last algorithms make constant–time communication possible, independent of *both* the number of polygons and the number of processors, but unfortunately the constant cost is very large.

A novel combination of sort–middle and sort–last communication is proposed and analyzed to address the problem of finding scalable communication algorithms. The algorithm efficiently combines the performance of sort–middle architectures for small numbers of processors and polygons with the scalability of sort–last architectures to create a system with performance *linear* in the number of processors.

The system described in this thesis achieves interactive rendering rates of up to 30 frames per second. To maintain interactivity a sort–middle render was implemented, as scalable solutions such as sort–last proved too expensive to achieve real–time performance. A large set of optimizations are analyzed and implemented to address the scalability problems of sort–middle communication.

This thesis significantly differs from current work on parallel polygon rendering. Contemporary parallel rendering focuses very heavily on multiple–instruction multiple–data (MIMD) architectures with large, fast and expensive networks. We have instead pursued a SIMD architecture with a rather narrow network. Background is provided for software and hardware parallel renderers, both of which have significantly influenced this work.

## 1.1 Background

Polygon rendering systems vary widely in their capabilities beyond just painting polygons on the screen. The most common features found in most of the systems described below include:

**Depth Buffering** A pixel–by–pixel computation of the distance of the polygon from the users eye is stored in a "depth buffer." Polygon are only rendered at the pixels where they are closer to the observer than the previously rendered polygons.

**Smooth Shading** The color of the polygon is smoothly interpolated between the vertices to give the impression of a smoothly varying surface, rather than a surface made up of flat–shaded planar facets.

**Texture Mapping** The color of the polygon is replaced with an image to increase the detail of the scene. For example, an image can be texture mapped onto a billboard polygon.

More complex capabilities are present in many of the systems and, while technically interesting, are left unaddressed as they don't bear directly on the polygon rendering system described herein.

There have been numerous parallel polygon rendering systems built. I will discuss some of them here, and provide references to the literature. It is significant that most of the work deals either with large VLSI systems, or MIMD architectures with powerful individual nodes, and a rich interconnection network. This thesis bridges the gap between these approaches, merging the very high parallelism of hardware approaches with the flexibility of software approaches.

### 1.1.1 Hardware Approaches

High performance polygon rendering has traditionally been the domain of special purpose hardware solutions, as hardware can readily provide both the parallelism required and the communication bandwidth necessary to fully exploit the parallelism in polygon rendering. Turner Whitted provides a survey of the 1992 state of the art in hardware graphics architectures in [21].

I will briefly discuss three specific systems here: the PixelFlow architecture from the University of North Carolina, and two architectures from Silicon Graphics, Inc., the IRIS 4D/GTX and the Reality Engine. All of the architectures provide interactive rendering with various degrees of realism.

**PixelFlow**

PixelFlow, described in [12] and [13], is a large–scale parallel rendering system based on a sort–last compositing architecture. It provides substantial programmability of the hardware, and at a minimum performs depth buffering, smooth shading and texture mapping of polygons.

PixelFlow is composed of a pipeline of rendering boards. Each rendering board consists of a RISC processor to perform geometry calculation and a $128 \times 128$ SIMD array of pixel processors that rasterize an entire polygon simultaneously, providing parallelism in the extreme. The polygons database is distributed arbitrarily among the rendering boards to achieve an even load across the processors. After rendering all of its polygons each rendering board has a partial image of the screen, correct only for that subset of the polygons it rendered. The separate partial images of the screen are then composited to generate a single output image via a 33.8 gigabit/second interconnection bus.

PixelFlow implements shading and texture mapping in two special shading boards at the end of the pipeline of rendering boards. Each pixel is thus shaded at most once, eliminating needless shading of non–visible pixels, and the amount of texture memory required of the system is limited to the amount that will provide sufficient bandwidth to texture map full resolution images at update rates.

The sort–last architecture provides performance increases nearly linear in the number of rendering boards and readily admits load balancing. The PixelFlow architecture, when complete, is expected to scale to performance of at least 5 million polygons per second. By comparison the current Silicon Graphics Reality Engine renders slightly more than 1 million polygons per second.

### IRIS 4D/GTX

The Silicon Graphics IRIS 4D/GTX, described in [2], was first sold in 1989 and is the Princeton Engine's contemporary. The GTX performs smooth shading and depth–buffering of lit polygons and can render over 100,000 triangles a second. The GTX provides no texture mapping support.

Geometry computations are performed by a pipeline of 5 high performance processors which each perform a distinct phase of the geometry computations. The output of the geometry pipeline is fed into a polygon processor which dices polygons into trapezoids with parallel vertical edges. These trapezoids are consumed by an edge processor that generates a set of vertical spans to interpolate to render each polygon. A set of 5 span processors perform interpolation of the vertical spans. Each span processor is further supported by 4 image engines that perform the composition of the pixels with previously rendered polygons and manage the fully distributed framebuffer. The span processors are interleaved column by column of the screen and the image engines for each span processor are interleaved pixel by pixel. The fine grain interleaved interleaving of the span processors and image engines insure that most primitives will be rasterized in part by all of the image engines.

Communication is implemented by the sort–middle screen–space decomposition performed between the edge processor and the span processors. This choice, while limiting the scalability of the system, avoids the extremely high constant cost of sort–last approaches such as PixelFlow.

### Reality Engine

The Reality Engine, described in [1], represents the current high–end polygon rendering system from Silicon Graphics. It supports full texture mapping, shading, lighting and depth buffering of primitives at rates exceeding one million polygons a second.

Logically the Reality Engine is very similar to the architecture of the GTX. The 5–processor geometry pipeline has been replaced by 12 independent RISC processors each implementing a complete version of the pipeline, to which polygons are assigned in a round–robin fashion. A triangle bus provides the sort–middle interconnect between the geometry engines and a set of up to 20 interleaved fragment generators. Each fragment generator outputs shaded and textured fragments to a set of 16 image engines which perform multisample antialiasing of the image, and provide a fully distributed framebuffer. The most significant change from the GTX are the fragment generators, which are the combination of of the span processors and the edge processors.

The texture memory is fully duplicated at each of the 20 fragment generators to provide the high parallel bandwidth required to access textures at full throttle by all fragment generators, and as such the system operates with little performance penalty for texture mapping.

A fully configured Reality Engine consists of over 300 processors and nearly half a gigabyte of memory, and provides the closest analog to the Princeton Engine in raw computational power. Substantial differences in communication bandwidth and the speed of the processors make the comparison far from perfect however, and the Princeton Engine's performance is more comparable to its contemporary, the GTX.

### 1.1.2 Software Approaches

There have been a number of software parallel renderers implemented, both in the interests of realism and/or interactivity. The majority of these renderers have either been implemented on small machines (tens of processors) or on large machine with substantial interconnection. In either case the severity of the communication problem is minimized.

### Connection Machine

Frank Crow et al. implemented a polygonal rendering system on a Connection Machine 2 (CM–2) at Whitney/Demos Productions, as described in [6]. The CM–2 [18] is a massively parallel SIMD computer composed of up to 65,536 bit–serial processors and 2,048 floating–point coprocessors. The nodes are connected in a 2D mesh and also via a 12–dimensional hypercube with 16 processors at each vertex. Each node in the CM–2 has 8KB of memory, for a total main memory size of 512MB.

Crow distributes the framebuffer over the machine with a few pixels per processor. Rendering is performed with a polygon assigned to each processor. All processors render simultaneously and as pixels are generated they are transmitted via the hypercube interconnect to the processor with the appropriate piece of the framebuffer for depth buffering. This is therefore a large–scale implementation of sort–last communication. The issue of the bandwidth this requires from the network is left untouched. It is reasonable to assume that it is not inexpensive on such a large machine. Crow alludes to the problem of collisions between pixels destined for the same processor, but leaves it unresolved.

The use of a SIMD architecture forces all processors to always perform the same operations (on potentially different data). Thus if polygons vary in size all processors will have to wait for the processor with the biggest polygon to finish rasterizing before they proceed to the next polygon. This problem is exacerbated by the 2–D nature of the polygons, which generally breaks the rasterization into a loop over the horizontal "spans" of the polygons, with an inner loop over the pixels in each span. Without care this can result in all processors executing in time proportional to the worst case polygon width. Crow addresses this problem in part by allowing processors to independently advance to the next span of their current polygon every few pixels, but constrains all processors to synchronize at polygon boundaries. Our experience suggests that a substantial amount of time is wasted by this approach.

Crow provides a good overview of the difficulties of adapting the polygon rendering algorithm to a very large SIMD machine, but leaves some critical load balancing issues unresolved, and unfortunately provides no performance numbers.

### Princeton Engine

Contemporaneous with this thesis, Scott Whitman implemented a multiple–user polygon renderer on the Princeton Engine. The system described in [20] divides the Princeton Engine among a total of 8 simultaneous users, assigning a group of 128 processors to each user.

Sort–middle communication is used, with some discussion made of the cost of this approach. As in Crow [6], Whitman employs a periodic test during the rasterization of each scan–line to allow processors to independently advance to the next scan–line. He also synchronizes at the polygon level, constraining the algorithm to perform in time proportional to the largest polygon on any processor.

Whitman suggests that his algorithm would attain performance of approximately 200,000 triangles per second per 128 processors on the next generation hardware we discuss in §9.3. This assumes that the system remains devoted to multiple users, so a single user will not attain an additional 200,000 triangles per second for every group of 128 processors assigned to him.

Whitman also provides a good general discussion and background for parallel computer rendering methods in his book [19].

### Parallel RenderMan

Michael Cox addresses issues of finding scalable parallel algorithms and architectures for rendering, and specifically examines a parallel implementation of RenderMan [4].

Cox implements both a sort–first renderer, as an efficient decomposition of the rendering problem in image space, and a sort–last renderer, as an efficient decomposition in object space, to study the differences in efficiency between these two approaches. Both implementations were on a CM–5 from Thinking Machines, a 512 processor MIMD architecture with a powerful communication network.

The problems of each approach are discussed in detail. Sort–first (and sort–

middle) can suffer from load imbalances as the distribution of polygons varies in image space, and sort–last suffers from extreme communication requirements. Cox addresses these issues with a hybrid algorithm which uses sort–last communication as a back end to load balance to sort–first communication.

### Distributed Memory MIMD Rendering

Thomas Crocket and Tobias Orloff provide a discussion of parallel rendering on a distributed memory MIMD architecture in [5] with sort–middle communication. Particular attention is paid to load–imbalances due to irregular content distribution over the scene.

They use a row–parallel decomposition of the screen, assigning each processor a group of contiguous rows for which it performs rasterization. The geometry and rasterization calculations are performed in a loop, first performing some geometry calculations and transmitting the results to the other processors, then rasterizing some polygons received from other processors, then performing more geometry, etc. This distributes the communication over the entire time of the algorithm and can result in much better utilization of the communication network by not constraining all of the communication to happen in a burst between geometry and rasterization.

### Interactive Parallel Graphics

David Ellsworth describes an algorithm for interactive graphics on parallel computers in [7]. Ellsworth's algorithm divides the screen into a number of regions greater than the number of processors, and dynamically computes a mapping of regions to processors to achieve good load balancing. The algorithm was implemented on the Touchstone Delta prototype parallel computer with 512 Intel i860 compute nodes interconnected in a 2D mesh.

Polygons are communicated in sort–middle fashion with the caveat that processors are placed into groups. A processor communicates polygons after geometry calculations by bundling all the polygons to be rasterized by a particular group into a large message and sending them to a "forwarding" processor at that group that then redistributes them locally. This approach amortizes message overhead by creating larger message sizes.

This amortization of the communication network overhead is a powerful idea, and is the central theme of several optimizations explored in this thesis.

## 1.2 Overview

This thesis deals with a number of issues in parallel rendering. A great deal of attention is paid to efficient communication strategies on SIMD rings, and their specific implementation on the Princeton Engine.

Chapter 2 provides an overview of the Princeton Engine architecture which will drive our analysis of sorting methods, and our implementation.

Chapter 3 discusses the uniprocessor polygon rendering pipeline and then extends it to the multiprocessor pipeline. Approaches to the sorting problem are addressed, and issues in extracting efficiency from a parallel SIMD machine are raised.

Chapter 4 examines the scalability of polygon rendering algorithms, by looking at the scalability of the three primary components of rendering: geometry, communication, and rasterization. A general model for the performance of sort–first, sort–middle and sort–last communication on a 1D ring is proposed and used to make broad predictions about the scalability of the sorting algorithms in both the number of polygons and the number of processors. Attention is also paid to the load balancing problems that the choice of sorting algorithm can introduce and their effects on the overall scalability of the renderer. Chapter 4 concludes with justification for the choice of sort–middle communication for the Princeton Engine polygon rendering implementation, despite its lack of scalability.

Chapter 5 proposes and analyzes optimizations for sort–middle communication, both with an analytical model, and with the support of instruction–accurate simulation. The feasibility of very tight user control over the communication network is exploited to provide a set of precisely managed and highly efficient optimizations which more than triple the overall performance from the most naive communication implementation.

Chapter 6 proposes and analyzes a novel communication strategy that marries the efficiency of sort–middle communication for small numbers of polygons and processors with the constant time performance of sort–last communication to obtain a highly scalable algorithm. Cycle by cycle control of the communication channel is used to create a communication structure that while requiring the same amount of bandwidth as sort–last communication achieves nearly a factor of 8 speedup over a typical implementation by amortizing the communication overhead. The VLIW coding of the Princeton Engine allows a full deferred lighting, shading and texture mapping implementation which increases rasterizer efficiency by a factor of more than two.

Chapter 7 details our implementation of polygon rendering on the Princeton Engine. A complete discussion of the efficient handling of the communication channel is made, specifically discussing the queueing of polygons for communication to maintain saturation of the channel, and the use of carefully pipelined pointer indirection to achieve general all–to–all communication with neighbor–to–neighbor communication *without* any additional copy operations. Specific mechanisms and optimization for the implementation of fast and efficient parallel geometry and rasterization are also discussed.

Chapter 8 discusses the performance of our system, and provides an account of where all the instructions go in a single second of execution. Statistics from executions of the system are provided for three sample scenes of varying complexity, along with benchmark figures of the raw throughput of the system for individual geometry, communication, and rasterization operations. The results are compared with the GTX from Silicon Graphics, a contemporary of the Princeton Engine, and with the Magic–1, our next generation hardware.

Chapter 9 discusses our next generation hardware, the Magic–1, currently under test and development. The Magic provides significant increases in communication and computation support, in addition to a greatly increased clock rate, and is predicted to provide an immensely powerful polygon rendering system. Our estimations for an implementation of a sort–twice rendering system on this architecture are provided, and indicate a system with performance in the millions of polygons, with flexibility far beyond that of current high performance hardware systems.

Chapter 10 summarizes the results of this thesis, and concludes with a discussion of the contributions of this work.

Appendix A discusses the instruction–accurate simulator developed to explore the efficiency of different communication algorithms and the load–balancing issues raised on a highly parallel machine.

# Chapter 2

# Princeton Engine Architecture

The Princeton Engine [3], a super–computer class parallel machine, was designed and built at the David Sarnoff Research Center in 1988. It consists of between 512 and 2048 SIMD processors connected in a neighbor–to–neighbor ring. The Engine was originally designed purely as a video processor, and its only true input/output paths are video. It accepts two NTSC composite video inputs and produces up to four interlaced or progressive scan component video outputs, as shown in Figure 2-1.

The Princeton Engine is a SIMD architecture, so all processors execute the same instruction stream. Conditional execution of the instruction stream is achieved with a "sleep" state that processors may conditionally enter and remain in until awakened.

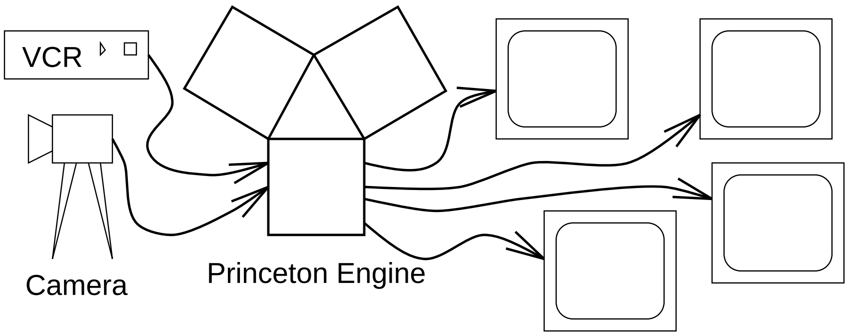



Figure 2-1: **Princeton Engine Video I/O:** the Princeton Engine supports 2 simultaneous video inputs and 4 simultaneous outputs.

The instruction stream implicitly executes

```
if wakeup then
    asleep = false
execute instruction
if !asleep then
    commit result
```

for every instruction. More complex `if ... then ... else ...` execution can be achieved by first putting some subset of the processors to sleep, executing the `then` body, toggling the `asleep` flag, and executing the `else` body.

The details of the processors, the communication interconnect and the video I/O capabilities are described below. The video I/O structure is particularly critical as it determines what algorithms can be practically implemented on the Princeton Engine.

## 2.1 Processor Organization

The Princeton Engine, shown schematically in Figure 2-2, is a SIMD linear array of 1024 processors connected in a ring. Each processor is a 16–bit load–store core, with a register file, ALU, a $16 \times 16$ multiplier and 640KB of memory.

The $16 \times 16$ multiplier computes a 32 bit result, and a product picker allows the selection of which 16 bits of the output are used, providing the operation $c = (a * b)/2^n$. This allows efficient coding of fixed–point operations. The Princeton Engine was designed to process video in real–time, and as such extended precision integer arithmetic (beyond 16 bits) and floating–point arithmetic delegated to software solutions and are relatively expensive operations. A 32–bit signed integer multiply requires 100 instructions, and a 32–bit signed integer division requires over 400 instructions.

The memory is composed of 2 types: 128KB of fast memory and 512KB of slow memory. Fast memory is accessible in a single clock cycle. Slow memory is divided into 4 banks of 128KB and is accessible in 2 cycles. The compiler places all user

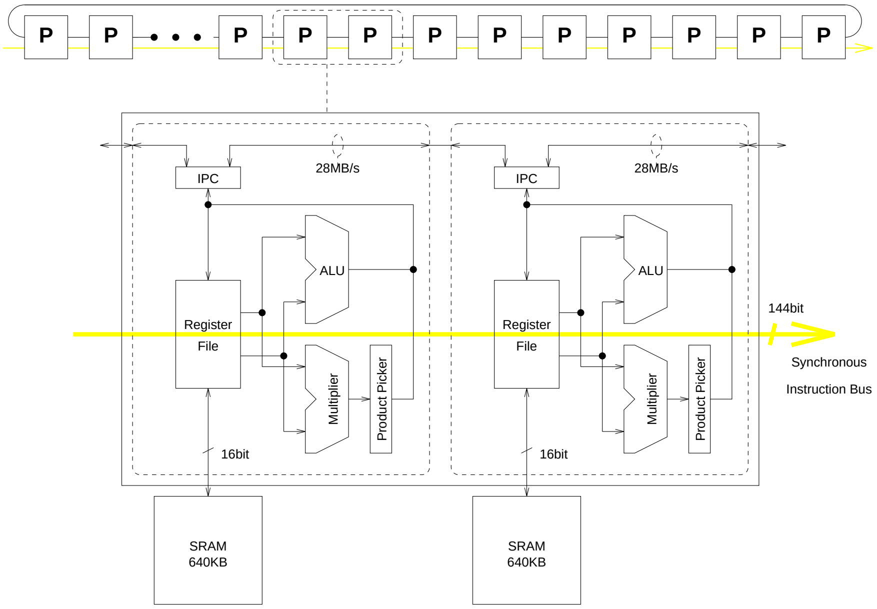



Figure 2-2: **Princeton Engine:** the processors are organized in a 1D ring. The instruction stream is distributed synchronously from the sequencer.

data and variables into fast memory. Any use of the slow memory is by explicit programmer access.

The instruction stream supplied by the sequencer is 144 bits wide. The VLIW (Very Long Instruction Word) coded instruction stream provides parallel control over all of the internal data paths of the processors. The average amount of instruction level parallelism obtained within the processors is between 2 and 3. Unlike typical uniprocessors, the Princeton Engine processors are not pipelined, as the instruction decode has already happened at the sequencer. The parallelism obtained by VLIW on the Princeton Engine is comparably to the parallelism obtained by pipelined execution in a RISC processor, and the 14MHz execution of the processing element is roughly equivalent to 14 RISC MIPS.

The processors are controlled by a central sequencer board which manages the program counter and conditional execution. Each of the 1024 processors operates at 14MHz for an aggregate performance of approximately 14,000 RISC MIPS or 14 billion instructions per second.

## 2.2 Interprocessor Communication

The processors are interconnected via a neighbor–to–neighbor "interprocessor communication" (IPC) bus. Each processor may selectively transmit or receive data on any given instruction, allowing multiple communications to occur simultaneously. There is no shared memory on the Princeton Engine, all communication between processors must via the IPC.

Interprocessor communication (IPC) occurs via a 16–bit data register usable on every cycle, for a bisection (neighbor to neighbor) bandwidth of 28MB/s. The ring can be configured as a shift register with 1024 entries. Each processor has control over one entry in the shift register. After a shift operation each processor's IPC register holds the value of its neighbor's (left or right) IPC register previous to the shift operation. During any given instruction the IPC is shifted left, right or unchanged. The IPC register may be read non–destructively, so once data is placed in the IPC registers it may be shifted and read arbitrarily many times, to distribute a set of values to multiple processors.

There are several modes of communication supported on the Princeton Engine. They are represented schematically in Figure 2-3:

**Neighbor–to–Neighbor** The most natural option. Each processor passes a 16–bit word to either its left or right neighbor. All processors pass simultaneously either to the left or to the right.

**Broadcast** One processor is the broadcaster, and all other processors are receivers.

**Neighbor–to–Neighbor–N** Each processor passes a single datum to a processor N processors away. All processors do this in parallel, so this is as efficient as neighbor–to–neighbor communication in terms of information density in the ring.

**Multi–Broadcast** Some subset of the processors are designated broadcasters. Each broadcaster transmits values to all processors, including itself, between itself

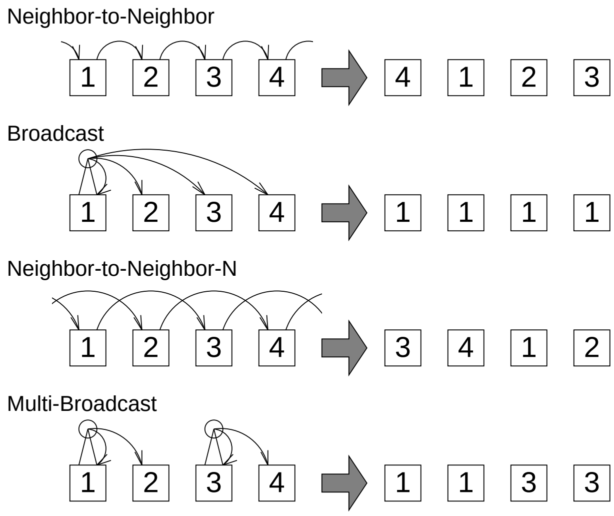



Figure 2-3: **Communication Options on the Princeton Engine:** the right hand side depicts the contents of the communication register on each processor after a single step of the communication method.

and the next broadcaster.

The discussion of communication approaches will ignore the possibility of using broadcast (in either mode) because it requires more instructions than neighbor–to–neighbor to transfer the same amount of data. Consider the transmission of $p$ data around a machine of size $n$, where each datum is the width of the communication channel. At the start of the communication each processor holds one datum, and at the end of the communication each processor has a copy of every datum. We can count the cycles required for each approach:

| approach | parallelism | inst/datum | total inst |
|---|---|---|---|
| neighbor–to–neighbor | n | $1 + 2 \cdot n$ | $p \cdot (1 + 2 \cdot n)/n \approx 2 \cdot p$ |
| broadcast | 1 | 7 | $7 \cdot p$ |

The neighbor–to–neighbor approach is three to four times faster than broadcasting because it never has to pay the expense of waiting for data to propagate to all of the processors (the propagation delay between neighbors is small). The instructions per datum is $1 + 2 \cdot n$ instead of $3 \cdot n$ because each processor loads their datum in parallel

(1 instruction) then performs repeated shifts and non–destructive reads of the communication register. Similar results apply to neighbor–to–neighbor–N communication versus multi–broadcast communication.

## 2.3 Video I/O & Line–Locked Programming

The video I/O structure of the Engine assigns a processor to each column of the output video. Each processor is responsible for supplying all of the output pixels for the column of the display it is assigned to. The addition of an "Output Timing Sequence" (OTS) allows processors to be assigned regions somewhat more arbitrary than a column of the output video.

When the Princeton Engine was originally designed it was conceived of only as a video supercomputer. It was expected that the processing would be virtually identical for every pixel, and thus programs shouldn't operate on frames of video, they should operate on pixels, and every new pixel would constitute a new execution of the program! The paradigm is enforced by resetting the program counter at the beginning of each scanline.

Before the end of the scanline all of the processors must have processed their pixel and placed the result in their video output register. Because the program counter is reset at the beginning of each scanline all programs must insure that their longest execution path is within the scanline "instruction budget" of 910 instructions at 14MHz. This necessarily leads to some obscure coding, as complex operations requiring more than 910 instructions must be decomposed into a sequence of smaller steps that can individually occur within the instruction budget.

## 2.4 Implementation

The processors are fabricated in a 1.5 micron CMOS process which allows two processors to be placed on the same die. The physical configuration of the machine is

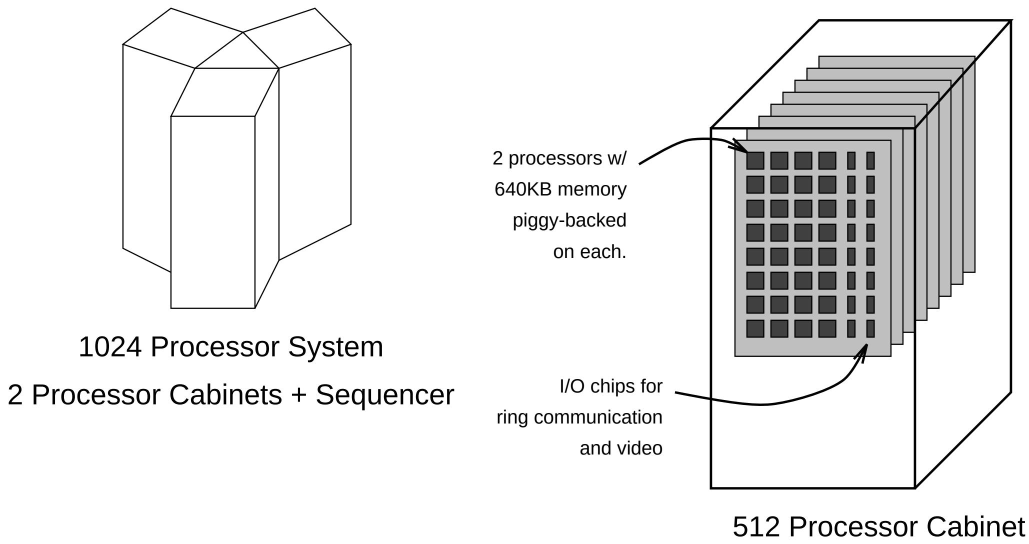



Figure 2-4: **Princeton Engine:** each of the cabinets shown is approximately 2 feet wide, 3 feet deep and 5 feet high. The processors are cooled by a forced air system.

shown in Figure 2-4. The machine consists of 1 to 4 processor cabinets and a sequencer cabinet. The processor cabinets hold 512 processors each, along with their associated memory. The sequencer cabinet contains the sequencer which provides instructions to the processors, the video input and output hardware, a HiPPi interface, and a dedicated controller card with an ethernet interface.

The sequencer provides the instruction stream to the processors. Serial control flow (procedure calls) is handled by the sequencer. Conditional control flow is performed by evaluating the conditional expression on all the processors in parallel. Processor 0 then transmits its result to the sequencer via the interprocessor communication bus. The sequencer branches conditionally based on the value it sees.

High–level program control is performed via an ethernet connection and a dedicated controller. The controller allows the starting and stopping of the sequencer, and the loading of programs into sequencer memory and data into processor memory. Facilities for non–video input and output are not supported in the video processing mode of operation, so they are not discussed here. Similarly, the HiPPi input/output channel is not compatible with the realtime video processing mode of operation. A piece of the HiPPi implementation, a status register used for flow control, can be used as a crude data-passing method between the controller and the processors. Data input through this channel is limited to relatively low rates (hundreds of values a second) as there is no handshaking provided.

The video input and output channel are distributed through a mechanism similar to the interprocessor communication register. Over the course of a video scanline the I/O chips sit in the background and shift in 2 composite video inputs and shift out 4 component video outputs. At the start of each scanline the user program can read the video input registers, and before the end of each line time the user program must write the video output registers.

# Chapter 3

# Graphics Pipeline

Polygonal rendering uses a synthetic scene composed of polygons to visualize a real scene or the results of computation or any number of other environments. Objects in the virtual space are defined as sets of 3D points and polygons defined with vertices at these points. In the most straightforward application of polygon rendering, these vertices are projected to the display plane (the monitor) via a transformation defined by a synthetic eyepoint and field of view, as shown in Figure 3-1. Subsequently the polygons defined by the set of vertices within the clip volume are rendered. Enhancements to this basic algorithm include occlusion, lighting, shading, and texture mapping.

Per–pixel occlusion insures that the polygons closest to the observer obscure poly-

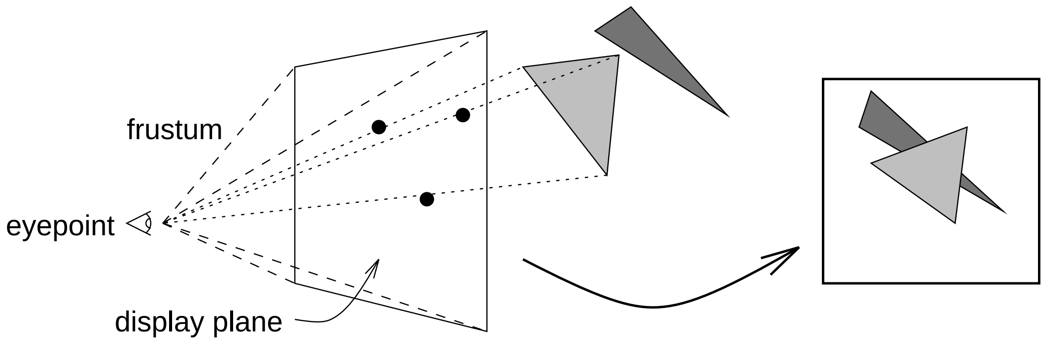



Figure 3-1: **Synthetic Eyepoint:** a synthetic eyepoint and frustum defines both the set of visible objects and a display plane.

gons further away. Occlusion has been implemented a number of ways, but is most commonly implemented with a depth–buffer. By computing the depth of each vertex of a polygon we can interpolate these depths across the interior pixels of the polygon to compute the depth of any pixel. When we write a pixel to the display device we actually perform a conditional write, only writing the pixel if the surface fragment it represents is "closer" to the eye than any surface already rendered at that pixel.

Lighting and shading greatly increase the realism of a scene by providing depth cues via changes in intensity across surfaces. Lighting assumes for any given polygon that it is the only polygon in the scene, so there are no shadows cast by the obstruction of light sources by other scene elements or reflection of light off of other surfaces. These effects are captured in other approaches to rendering such as ray–tracing, and are not generally implemented in polygonal renderers. Shading is performed by applying a lighting model, such as the Phong model [15], to a normal which represents the surface normal of the object being approximated at that point. The lighting model may either be applied at every pixel the polygon covers, by interpolating the vertex normals to determine interior normals, or it may be applied just to the vertex normals of the polygon, and the resulting colors interpolated across the polygon. The former approach (Phong shading) will often yield more accurate results and is free of some artifacts present in the latter approach (Gouraud interpolation) but it more computationally expensive. Figures 3-2, 3-3 and 3-4 are all examples of depth–buffered, Phong–lit, Gouraud–interpolated polygonal scenes.

Texture mapping is a further enhancement to polygonal rendering. Instead of just associating a color with each vertex of a polygon, we associate a $(u, v)$ coordinate in an image with each vertex. By careful assignment of these coordinates, we can create a mapping that places an image onto a set of polygons, for example: a picture onto a billboard, carpet onto a floor, even a face onto a head. Texture mapping, in a fashion analogous to shading and depth–buffering, linearly interpolates the texture coordinates at each vertex across the polygon. The color of a particular pixel within the polygon is determined by looking up the pixel in the texture map corresponding

Figure 3-2: **Beethoven:** a 5,030 polygon model with 4,021 non-culled polygons.

to the current values of $u$ and $v$. Figure 3-5 shows a sample texture mapped scene along with the corresponding texture map in Figure 3-6.

## 3.1 UniProcessor Pipeline

All of the above is combined into an implementation of a polygonal renderer in a "graphics pipeline". The sequence of operations is a pipeline because they are applied sequentially. The depth–buffer provides order invariance, so the results of one computation (polygon) have no effect on other computations.

The uniprocessor pipeline consists of two stages: geometry and rasterization. The geometry stage transforms a polygon from its 3D representation to screen coordinates, tests visibility, performs clipping and lighting at the vertices, and calculates interpolation coefficients for the parameters of the polygon. The rasterization stage takes the computed results of geometry for a polygon and renders the pixels on the screen.

A more complete description of the uniprocessor pipeline is now presented.

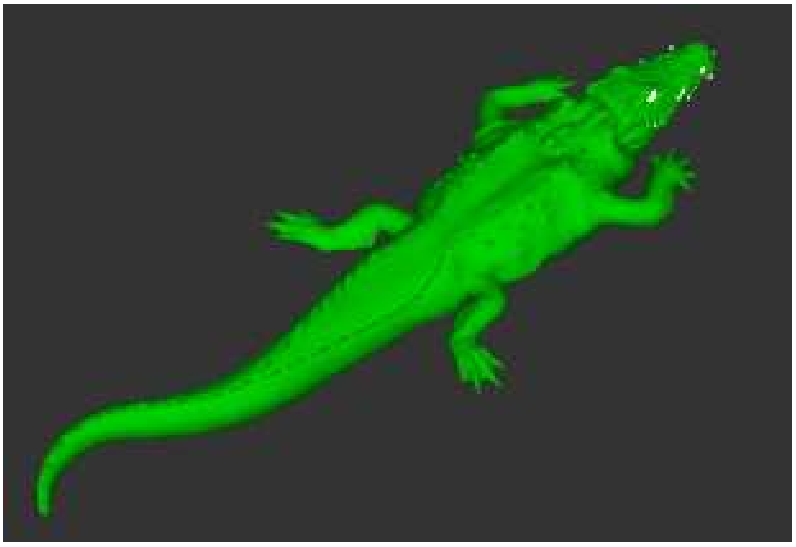


Figure 3-3: **Crocodile:** a 34,404 polygon model with 25,270 non–culled polygons.

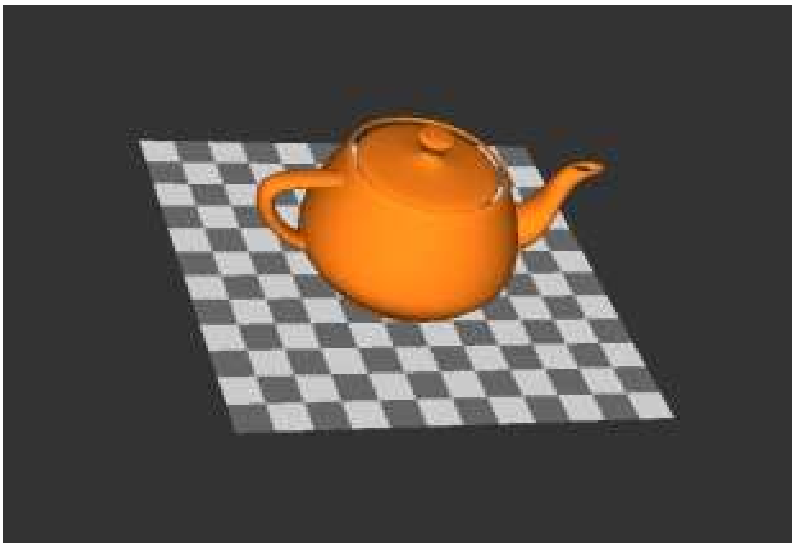


Figure 3-4: **Teapot:** a 9,408 polygon model with 6,294 non–culled polygons.

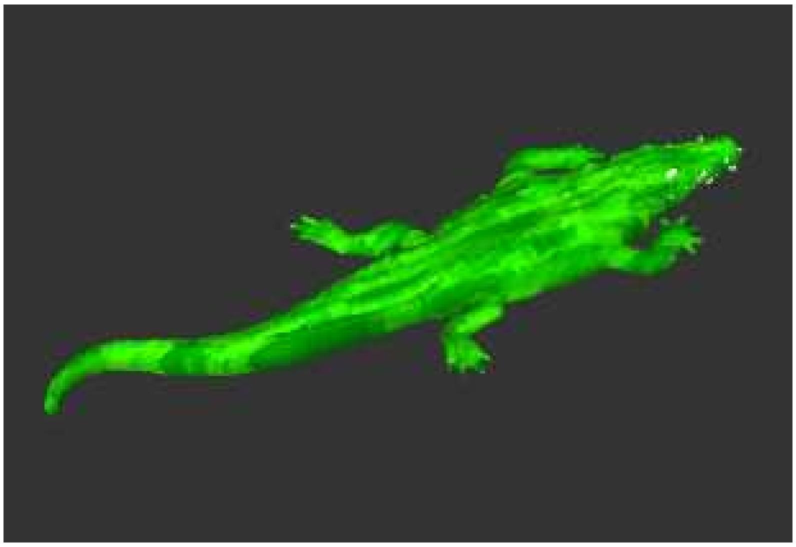


Figure 3-5: **Texture Mapped Crocodile:** a 34,404 polygon model with $\approx 10,200$ visible polygons.

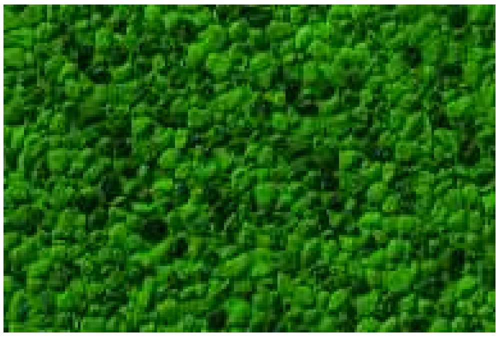


Figure 3-6: **Sample Texture Image:** the image of synthetic crocodile skin applied to crocodile.

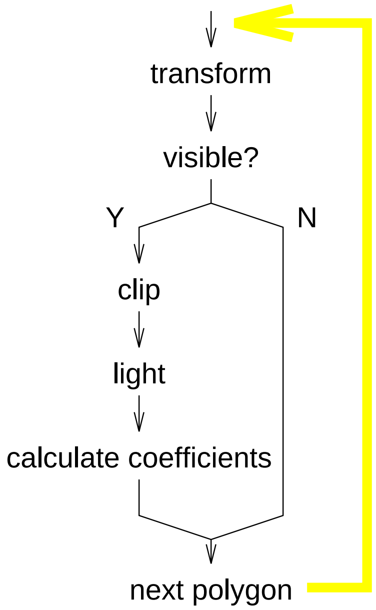



Figure 3-7: **Uniprocessor Geometry Pipeline**

### 3.1.1 Geometry

The geometry stage can be formed into a pipeline of operations to be performed, as shown in Figure 3-7 and explained below.

**Transform** The vertices are projected to the user's point–of–view. This involves translation, so the 3D origin of the points becomes the user's location, and rotation, so that the line of sight of the observer corresponds to looking down the z–axis (by convention) in 3D. This transformation is performed with a matrix multiplication of the form:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c & t_x \\ d & e & f & t_y \\ g & h & i & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3.1}$$

The use of a homogeneous coordinate system allows the translation to be included in the same matrix with the rotation.

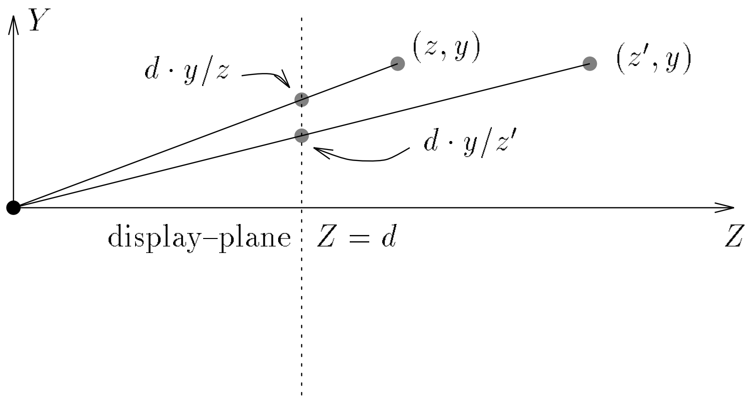



Figure 3-8: **2D Perspective Transformation:** the 2D case is analogous to the 3D case. Here the axes have been labeled to correspond to the standard 3D coordinate system.

The vertex eye–coordinates are now projected to the screen. This introduces the familiar perspective in computer (and natural) images, where distant objects appear smaller than closer objects. The screen coordinates of a vertex, $(s_x, s_y)$ are related to the transformed 3D coordinates $(x', y', z')$ by:

$$s_x = d\frac{x'}{z'} \tag{3.2}$$

$$s_y = d\frac{y'}{z'} \tag{3.3}$$

The 2D to 1D projection is shown in Figure 3-8. The 3D to 2D projection is identical, just repeated for each coordinate.

**Clipping & Rejection** Once the polygons are transformed to eye–coordinates they must be clipped against the viewable area. The rays cast from the eye–point of the observer through the four corners of the viewport define a viewing frustum. All polygons outside of this frustum are not visible and need not be rendered. Polygons that intersect the frustum are partially visible, and are intersected with the visible portion of the display plane, or "clipped." Clipping takes a polygon and converts it into a new polygon that is identical within the viewing frustum, and is bounded by the frustum.

Backface culling culls polygons based on the face of the polygon oriented towards the observer. Each polygon has two faces, and of course you can only see one of the two faces at any time. For example, if we constructed a box out of polygons we could divide the polygon faces into two sets, a set of faces potentially visible for any eyepoint inside the box, and a set of faces potential visible for any eyepoint outside of the box, and the union of these sets is all the faces of all of the polygons. If we know a priori that the viewpoint will never be interior to the box we *never* have to render any of the faces in the interior set. Polygons that are culled because their "interior" face is towards the viewer are said to be "backfacing" and are backface culled.

**Lighting** Lighting is now applied to the vertex normals. A typical lighting model includes contributions from diffuse (background or ambient) illumination, distant illumination (like the sun) and local point illumination (like a light bulb). Polygons are assigned properties including color (RGB), shininess and specularity which determine how the incident light is reflected. See Foley and van Dam [8] pg. 477 for a description of lighting models.

The normal associated with each vertex may also be transformed, depending on the viewing model. If the model of interaction assumes the viewer is moving around in a fixed space, then the lighting should remain constant, as should the vertex normals used to compute lighting. However, if the model assumes that the object is being manipulated by an observer who is sitting still then the normals but also be transformed. A normal is represented by a vector

$$\begin{bmatrix} n_x \\ n_y \\ n_z \\ 0 \end{bmatrix} \tag{3.4}$$

and is transformed by the inverse transpose of the transformation matrix.

**Linear Equation Setup** Rasterization is performed by iterating a number of linear equations of the form $Ax+By+C$. The coefficients of these equations represent the slopes of the values being iterated along the $x$ and $y$ axes in screen space. A typical system interpolates many values, including the color of the pixel, the depth of the pixel, and the texture map coordinates.

The computation of these linear equation coefficients, while straightforward, is time consuming. It requires both accuracy and dynamic range to encapsulate the full range of values of interest. Figure 3-9 shows the form of the computation for the depth. The same equations are evaluated for each component to be interpolated across the polygon.

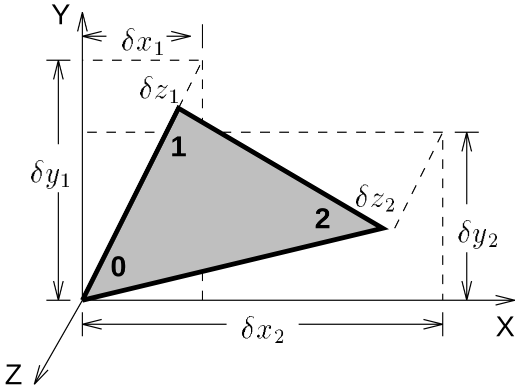
Due to the similarity of the work performed (lots of arithmetic) and in anticipation of the parallel discussion, I include linear equation setup with the geometry pipeline, but it could also be considered the start of the rasterization stage.

The rasterization stage described next renders the set of visible polygons by iterating the linear equations calculated during the geometry stage.

### 3.1.2 Rasterization

Rasterization must perform two tasks. It must determine the set of pixels that a polygon overlaps (is potentially visible at) and for each of those pixels it must evaluate the various interpolants associated with the polygon.

The typical uniprocessor software implementation of polygon rasterization analyzes the polygon to be rendered and determines for each horizontal scanline the left–most and right–most edge of the polygon. The algorithm can then evaluate the intersection of the polygon edges with any given scanline and rasterize the pixels within. This algorithm is efficient, as it examines only the pixels within the polygon and it is easy to determine the left and right edges.




Linear equation for an edge from vertex 0 to vertex 1:

$$E(x,y) = Ax + By + C \tag{3.5}$$
$$A = -\delta y_1 \tag{3.6}$$
$$B = \delta x_1 \tag{3.7}$$
$$C = x_0 \cdot y_1 - y_0 \cdot x_1 \tag{3.8}$$
$$\tag{3.9}$$

Linear equation for depth:

$$Z(x,y) = Ax + By + C \tag{3.10}$$
$$\Gamma = \delta x_1 \cdot \delta y_2 - \delta y_1 \cdot \delta x_2 \tag{3.11}$$
$$A = (\delta z_1 \cdot \delta y_2 - \delta y_1 \cdot \delta z_2)/\Gamma \tag{3.12}$$
$$B = (\delta x_1 \cdot \delta z_2 - \delta z_1 \cdot \delta x_2)/\Gamma \tag{3.13}$$
$$C = z_0 + A \cdot x_0 + B \cdot y_0 \tag{3.14}$$
$$\tag{3.15}$$

Figure 3-9: **Polygon Linear Equations:** The edge equations are unnormalized as only the sign of subsequent evaluations is important. Accurate magnitude is required for the other equations, resulting in more complex expressions for the coefficients.

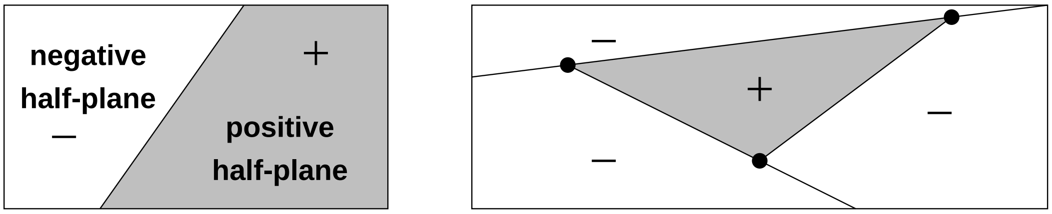



Figure 3-10: **Triangle Definition by Half–Plane Intersection:** a positive and negative half–plane are defined by a line in screen space. The interior of a polygon is all points in the positive half–plane of all the edges. Diagram is after [16].

Typical hardware implementations, for example [13], take a different approach. Each polygon can be defined as the intersection of the $n$ semi–finite plains bound by the $n$ edges of the polygon, as explained by Pineda in [16]. We can define the equation of a line in 2D as $Ax+By+C=0$ and thus a half–plane is given by $Ax+By+C \geq 0$. With the appropriate choice of coefficients we can insure that the equations of the $n$ semi–finite plains will all be positively (or negatively valued) within the polygon. Figure 3-10 depicts this approach.

This provides us with a very cheap test (the evaluation of one linear equation per vertex) to determine if any pixel is inside of a polygon. Furthermore, we can cheaply bound the set of pixels that are potentially interior to the polygon as the set of pixels interior to the bounding box of the polygon. This approach results in examining more pixels then will be rendered, as the bounding box is generally a conservative estimate of the set of pixels in a polygon. For triangles at least half of the pixels examined will be exterior to the polygon. Experimentally this has been determined to be a good tradeoff against the higher computational complexity of span algorithms because SIMD implementations benefit from a simple design with few exceptional cases.

The rasterization process is actually poorly described as a pipeline, because it consists of a series of early–aborts, at least in its serial form. Figure 3-11 shows a typical view of the rasterization process. The next pixel operation takes step horizontally until it reaches the right side of the bounding box, and then takes a vertical step and

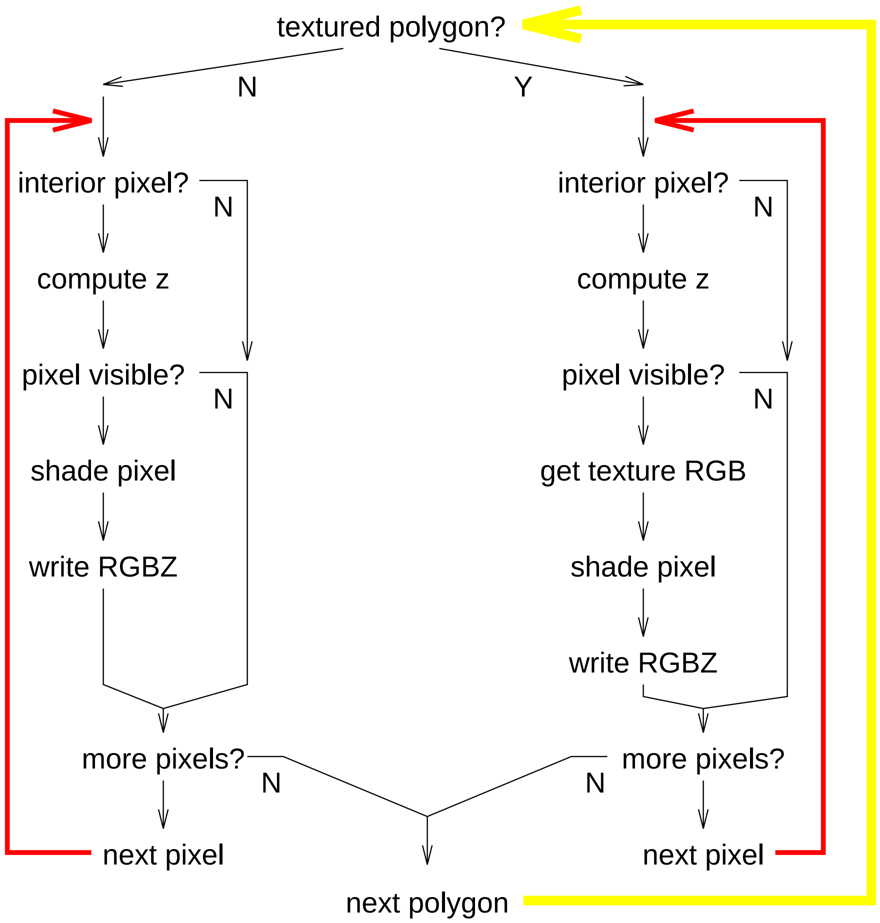



Figure 3-11: **Uniprocessor Rasterization**

returns to taking horizontal steps. The interior pixel test is the simple linear equation test shown in Figure 3-10. Compute Z evaluates the linear equation for depth and the pixel visible test determines if this pixel is closer to the observer than the current pixel in the frame buffer. If so it is then shaded and written to the frame buffer. The procedure for texture mapped polygons is identical, except the pixel that gets shaded is retrieved from a texture map, rather than just being the color of the polygon.

Considerable complexity has been swept under the rug in this view of rasterization. Most significant is the texture mapping operation, which is relatively complex in practice. The simplest way to perform texture mapping is "point sampling". Point sampling takes the color of a texture mapped pixel as the color of the closest texel (texture pixel) to the current interpolated texture coordinates. A more pleasing method uses a weighted combination of the 4 closest texels from the texture map. A yet more complex and pleasing approach is achieved with MIP–Mapping, described by Lance Williams in [22]. All of these methods attempt to correct the inaccuracy of point sampling a transformed image. A mathematically correct resampling of the input image would involve complex filtering and difficult computation. Approximation methods attempt to correct the sampling errors at lower cost, usually with acceptable visual results.

### 3.1.3 Details

There are a few auxiliary details which accompany the rendering process which have been ignored so far. Most importantly, there is generally a way for the user to interact with the system. A similarly important subject is double–buffering.

The particular input method provided by the system is not important, just some method must exist. These methods vary greatly, from being able to modify the polygon database and view matrix continually and interactively, to simply executing a script of predefined viewpoints.

Double–buffering is used to hide the machinery of polygon rendering. Users are (usually) only interested in the final rendered image, not all the work that goes into it. To this end, two frame buffers are used. One completed frame buffer is displayed, while the second frame buffer is rendered. When the rasterization of all polygons is complete, the frame buffers are exchanged, usually during the vertical retrace interval of the display, so the images don't flicker distractingly.

## 3.2 MultiProcessor Pipeline

The parallel version of the graphics pipeline differs significantly from the uniprocessor pipeline. There are now $n$ processors working on the rendering problem instead of a single processor, and optimally we would like a factor of $n$ speedup. How do we divide the work among the processors? Given that we have divided the work, how do we execute it in parallel? While these questions appear superficially obvious, they are not. By parallelizing the algorithm we have exposed that it is actually a sorting problem, resolved by the depth–buffer in the uniprocessor system, but presumably resolved in parallel in a multiprocessor implementation.

It is relatively obvious how to partition the geometry work. Instead of giving $p$ polygons to 1 processor, give $p/n$ polygons to each processor, and allow them to proceed in parallel. This should yield a factor of $n$ speedup, which is the most we can hope for. The rasterization stage is less obvious. Do we let each processor rasterize $1/n$ of the polygons? Do we let each processor rasterize $1/n$ of the total pixels? When a polygon has to be rasterized do all processors work on it at the same time?

We would also like to divide the work as evenly as possible among the processors. If we put more work on any one processor we have compromised the performance of our system. However, it is not always the case that the same number of polygons on each processor corresponds to the same amount of work, if, for example, the polygons vary in size.

Rasterization is actually a sorting algorithm based on screen location and depth. If we divide the work across multiple processors we must still perform this sorting operation. The addition of communication to the rendering pipeline enables us to resolve the sorting problem and render correct images, that is, the same images a uniprocessor renderer would generate.

Because the rasterization sorting issue has forced us to introduce communication, our first departure from the uniprocessor pipeline, it is natural to start by discussing its parallelization.

### 3.2.1 Communication

Given that we have to perform communication to produce the final image, we must choose how we parallelize the rasterization. This choice bears heavily on how the image is generated and will determine what communication options are available to us. There are two ways we could divide the rasterization among the processors:

1. give each processor a set of pixels to rasterize, or

2. give each processor a set of polygons to rasterize.

If we choose the first option then the sorting problem is solved by making the pixel sets disjoint, so a single processor has all the information necessary to correctly generate its subset of the pixels. This requires communication before rasterization, so that each processor may be told about all the polygons that intersects its pixel set. These approaches are referred to as "sort–first" and "sort-middle" because the communication occurs either first, before geometry and rasterization, or in the middle, between geometry and rasterization.

If we choose the second option then the polygons rasterized on any given processor will overlap some subset of the display pixels, and there may be polygons on some other processor or processors that overlap some of the same display pixels. So if we parallelize the rasterization by polygons rather than pixels then the sorting problem must be solved after rasterization. This approach is referred to as "sort–last" because the sorting occurs after both rasterization and geometry.

Figure 3-12 shows these communication options schematically. In all cases the communication is shown as an arbitrary interconnection network. Ideally we would

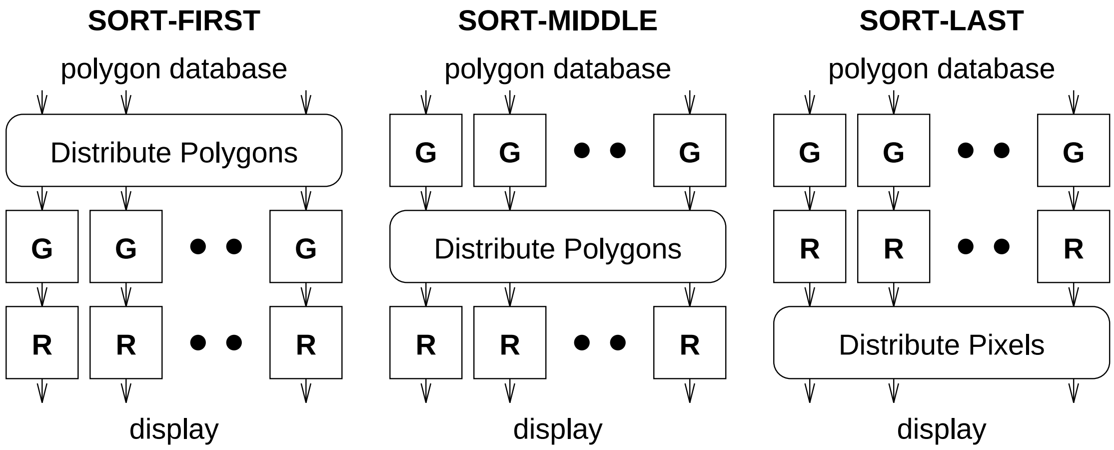



Figure 3-12: **Sort Order:** some possible ways to arrange the polygon rendering process. Figure is after [13].

like this network to be a crossbar with infinite bandwidth, so that the communication stage requires no time to execute. Of course in practice we will never obtain such a communication network.

### Sort–First

Sort–first performs communication before geometry and rasterization. The screen is diced up into a set of regions and each processor is assigned a region or regions, and rasterizes all of the polygons that intersect its regions. All of the regions are disjoint, so the final image is simply the union of all the regions from all the processors, and no communication in necessary to combine the results of rasterization. The communication stage must give each processor a copy of all the polygons that overlap its screen area. This information can only be determined after the world–space to screen–space transformation is made, so there is generally some non–trivial amount of computation done before the communication is performed.

### Sort–Middle

Sort–middle also assumes that each processor is rasterizing a unique area of the screen. However, the communication is now done after the geometry stage, so rather than communicating the polygons themselves, we can directly distribute the linear equation coefficients that describe the edges of the polygon on the screen along with the depth, shading, etc.

### Sort–Last

Sort–last assigns each processor a subset of the polygons to rasterize, rather than a subset of the pixels. As the polygons rasterized on different processors may overlap arbitrarily in the final image, the composition is performed after rasterization. Each processor can perform the geometry and rasterization stage with no communication whatsoever, only communicating once they have generated all of their pixels for the output image.

### Comparison

These three sorting choices all incur different tradeoffs, and the particular choice of algorithm will depend on both details of the machine (speed, number of processors, etc.) and the size of the data set to be rendered.

Sort–first communication can exploit temporal locality. Generally a viewer will navigate a scene in a smooth and continuous fashion, so the viewpoint changes slowly, and the location of each polygon on the screen changes slowly. If the location of polygons is changing slowly then the change in which polygons each processor will rasterize will be small, and sort–first will require little communication. This is most obvious if the viewer is standing still, in which case the polygons are already correctly sorted from rendering the previous frame to render the current frame. However, sort–first incurs the overhead of duplicated effort in the geometry stage. Every polygon has its geometry computation performed by *all* processors whose rasterization regions it overlaps. For large parallel systems, which will have correspondingly small rasterization regions, this cost could become relatively high.

Sort–middle communication calculates the geometry only once per polygon and thus will not have the duplication of effort incurred by sort–first. However, sort–middle must communicate the entire polygon database for each rendered frame, which is presumably substantially more expensive than a method which exploits temporal locality.

Sort–last communication traffics in pixels instead of polygons, and has the interesting property that the amount of communication is constant. Each processor will have to communicate no more than a screen's worth of pixels to composite the final image, so as the polygon database grows arbitrarily large sort–last will offer the cheapest communication. Although this constant bound on communication is useful, it is shown later to be a very large amount of communication compared to sort–first and sort–middle for our scenes of interest.

We have laid a framework for the discussion to follow, and, while there are interactions, the actual geometry and rasterization stages are to some extent independent of the communication topology. The communication topology will determine precisely what polygons are computed on what processor for each stage, but will have little effect on the nature of the computation itself. We will defer the actual choice of, and justification for, a communication topology until Chapter 4.

### 3.2.2 Geometry

The geometry calculations consists of five distinct steps:

- transform
- clip/reject
- project to screen coordinates
- light
- linear equation setup

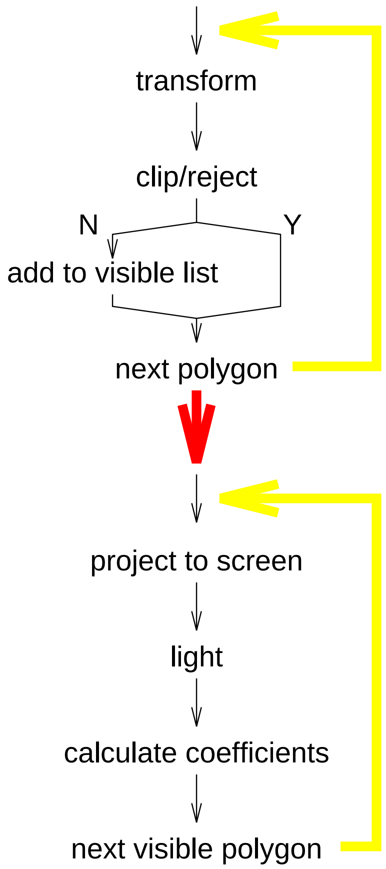



Figure 3-13: **SIMD Geometry Pipeline:** the pipeline is split between clipping and lighting.

Each of these stages has an upper–bound on the amount of work performed per polygon. The calculations to be performed for each polygon, while executing on different data and yielding different results, are identical algebraically. The execution of each processor over its subset of the polygons, whether obtained from the input polygonal database differently, or from the results of sort–first communication, will be identical, and the execution time will be determined by the processor with most polygons to process.

Often after the clip/reject stage a very large number of processors will have discarded their polygon and have nothing to do while the other processors compute. This will achieve poor utilization of the processors and a less than optimal speedup for the geometry stage. A revised geometry pipeline appropriate for parallel execution is shown in Figure 3-13. By breaking the geometry pipeline into two separate pieces and queueing the work between them we will first perform work proportional to the maximum number of polygons on any processor, then work proportional to the maximum number of *visible* polygons on any processor. The cost of the second stage of the pipeline per polygon will prove to be substantially higher than the first stage, so if the number of visible polygons is much smaller than the total number of polygons the savings could be substantial. Section 7.1 provides a discussion of this implementation issue.

### 3.2.3 Rasterization

The simple rasterization algorithm suggested in Figure 3-11 has the same implementation problems as the geometry stage. If the number of polygons varies significantly across processors the amount of work to be done on each processor will also vary significantly. This problem is compounded by the varying sizes of polygons. If we assume our algorithm can rasterize a pixel in some constant amount of time, we would like our time to rasterize a polygon to be proportional to the area of the polygon. However, the uniprocessor implementation has a forced serialization of the polygons, which introduces an expensive synchronization point in the algorithm. This synchronization forces all processors to rasterize their polygon in the *worst* case time. To achieve an efficient execution it will be necessary to decouple the rasterization process from the specific polygon being processed.

Figure 3-14 shows a suggested implementation for a more efficient parallel renderer. The double loop in the uniprocessor implementation, shown in Figure 3-11, has been replaced with a single loop, and the test case has been made more complex. This decouples the size differences in polygons between processors at the cost of making each iteration of the loop more expensive.

Differences in the size of polygons aside, there is reason to believe there will be substantial differences in the numbers of polygons on each processor. If we use a

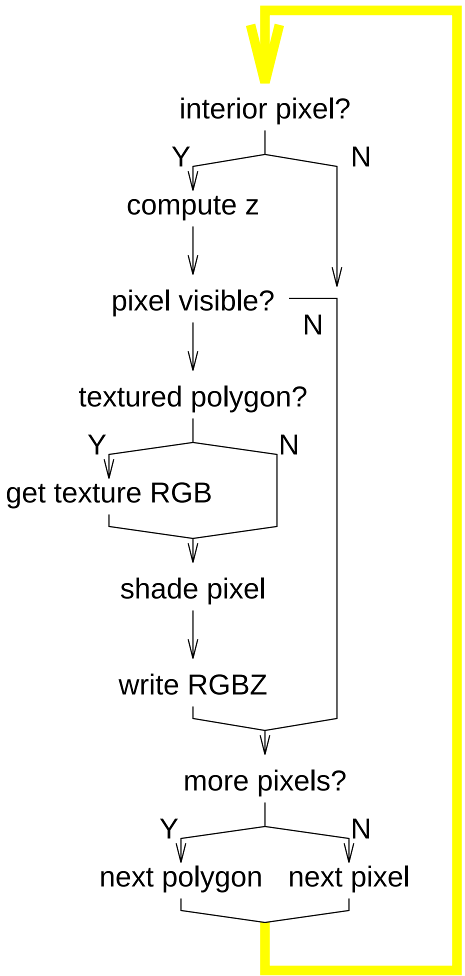



Figure 3-14: **SIMD Rasterization:** the rasterization process on a SIMD multi–processor. The double loop is replaced with a single loop to decouple the processor execution time from polygon size differences.

sort–first or sort–middle communication structure then any imbalances in the distribution of polygons across the screen (which there often are) will be reflected in imbalances in the number of polygons overlapping each processor's rasterization region, leading to further load imbalances. This suggests that an algorithm based on sort–last communication, which can assign polygons to processors arbitrarily, could perform significantly better than either sort–first or sort–middle communication, at least during the rasterization stage.

## 3.3 Summary

The major functionality of a polygon rendering pipeline has been described. Particular attention has been paid to the requirements for an efficient SIMD implementation. In general an attempt is made to queue up a sizable portion of work for each stage of computation on all processors to minimize the number of idle processors at any point in time. In effect an efficient SIMD graphics pipeline will move all polygons simultaneously through the pipeline, so that statistically each processor should be kept busy despite polygon by polygon variations in the amount of work to be done.

Unfortunately lack of uniformity in the polygon database, particularly with regards to asymmetric distribution of polygons over the screen, can result in difficult load balancing problems. The next chapter will discuss the scalability of the different communication schemes with an eye both to their inherent efficiency and their effects on the efficiency of the geometry and rasterization stages.

# Chapter 4

# Scalability

While we are interested specifically in a fast polygon rendering algorithm for the Princeton Engine, we are more generally interested in a fast algorithm for arbitrary parallel computers. In particular we would like a *scalable* algorithm so we can readily construct bigger and proportionally faster systems. Optimally we would like a linear speedup in the number of processors, so our algorithm must be $O(p/n)$. Two forces will prevent us from actually achieving this performance:

**Communication** Communication is necessary in a parallel renderer to combine the results of the parallel executions of the algorithm. Communication not only represents a cost non–existent in the uniprocessor case, it will also prove to be an $O(p)$ operation for our implementation, so communication will come to dominate performance for a large enough system. $O(1)$ solutions exist, but are extremely expensive, requiring time proportional to the number of pixels in the display.

**Imperfect load balancing** To obtain perfect linear speedup we must be able to perfectly divide the work among the processors. In general there are unavoidable differences in computational load between processors. Many rendering systems attack this problem by subdividing the problem to finer levels of parallelism which are more easily balanced, but for large systems these regions may become arbitrarily small, and the subdivision itself will start to incur substantial overhead.

While we desire linear performance improvements in the number of processors, we also desire the fastest possible solution for our particular architecture, the Princeton Engine. In particular, the goal of interactive frame rates will preclude the use of existing linearly scalable solutions. For example, sort–last communication requires constant time, independent of the number of processors. Unfortunately it will prove prohibitively expensive for our scenes of interest.

In this chapter we will discuss the expected scalability of the geometry, communication and rasterization stages. In addition we will provide a framework for picking the most efficient communication strategy in general, and specifically apply it to the Princeton Engine. The choice of sort order will play an important role, not only by determining the total amount of time spent performing communication and the scalability of the algorithm, but also by affecting the load balance and duplication of effort in the geometry and rasterization stages.

## 4.1 Geometry

The geometry calculations are *identical* for every polygon. Some polygons will be culled because they are out of the viewing frustum, or rejected because they are backfacing, but as discussed in §3.2.2 this can easily be countered by breaking the geometry pipeline into two pieces and queueing the visible polygons between these stages. The geometry calculations required for a polygon is constant, so we should be able to distribute polygons over processors uniformly, and obtain performance that is $O(p/n)$ for the geometry stage.

## 4.2 Rasterization

The driving factor of our analysis of parallelism has been the number of processors involved, particularly *large* numbers of processors. As the number of processors involved becomes larger the division of the work (assuming a constant amount of work to be done) requires a finer and finer granularity of the problem. Even for very large numbers of processors (thousands) there is generally enough parallelism available during the geometry stage to keep them all occupied. This is not as readily true for the rasterization stage.

If we assume a polygon parallel implementation of rasterization, we give each processor $p/n$ of the polygons to rasterize (and have thus assumed a sort–last architecture) and with a few weak assumptions about the distribution of polygons and their sizes, we obtain a system that exhibits excellent parallelism. Unfortunately we will soon see that sort–last algorithms are prohibitively expensive on the Princeton Engine. This leaves sort–first and sort–middle algorithms to be considered. Both of these assign a region of the screen to each processor for rasterization, rather than some number of primitives, and thus require image parallel rasterization.

Sort–first and sort–middle divide the screen into multiple regions and assign them to the processors in some fashion. As discussed in the following sections, sort–first and sort–middle are most efficient for small numbers of processors and small numbers of polygons. In particular the communication time for sort–middle proves to be linear in the number of polygons and independent of the number of processors. Sort–first introduces redundant calculation which increases with the number of regions the screen is tiled into, and thus with the number of processors. So both sort–first and sort–middle lack scalability for our systems of interest.

Sort–first and sort–middle also affect the efficiency of the rasterization stage. If we statically divide the screen into $n$ regions these regions will become quite small (tens of pixels) for large $n$. Unfortunately the polygon complexity of a scene is usually not distributed uniformly over the display, so some regions will have very few polygons, while other regions will have many. This can lead to substantial rasterization load imbalances between processors. However, if we implement sort–last communication this problem is avoided, as we will assign polygons rather than pixels to each processor.

Chapter 6 discusses a hybrid algorithm entitled "sort–twice" that combines the low cost of sort–middle for small numbers of polygons and processors with the excellent scalability of sort–last to obtain a system more efficient than either approach for scenes of tens of thousands of polygons.

## 4.3 Communication

The communication stage acts as a crossbar, communicating the results of one phase of computation to another, and may be placed in three distinct places in the rendering algorithm, as shown schematically in Figure 3-12. Sort–first and sort–middle both communicate polygons, or some partially computed representation of them, while sort–last traffics in pixels.

Neither the scalability nor the efficiency of any of the sort options appear intuitively obvious. As evidenced by the rasterization discussion the ordering of the communication will have effects on both the rasterization and geometry stages. A model is introduced to quantify a number of aspects of these systems and provide a basis for their comparison.

### 4.3.1 Model

The model abstracts quite far away from the actual rendering process. An attempt has been made to capture all of the first order effects in communication and as many of the second order effects as deemed practical. We will make a number of simplifying assumptions, and note when we are being overly optimistic or pessimistic in our assumptions. We have found that even with crude assumptions the model can provide a clearly preferable communication strategy. Because the communication structure has a significant impact on the geometry and rasterization stages they are

| variable | definition |
|---|---|
| $p$ | number of polygons |
| $n$ | number of processors |
| $G$ | time to perform geometry per polygon |
| $R$ | time to rasterize a pixel |
| $f$ | asymmetry factor. ratio of maximum number of polygons in a region of the screen to the average number of polygons per region. |
| $o$ | number of regions that a polygon overlaps |
| $Q$ | area of screen in pixels |
| $A$ | average area of a polygon in pixels |
| $C$ | time to communicate a datum from neighbor–to–neighbor. Different for each communication structure. |

Table 4.1: **Modeling Variables**

included in the model.

Table 4.1 presents our variables for modeling the communication problem. We will model the rendering process as occurring on a machine with $n$ processors, rendering a $p$ polygon scene.

We model the geometry stage as requiring time $G$ to execute per polygon, and the total time spent will be $G$ times the maximum number of polygons on any processor. Similarly, the rasterization stage is modeled as requiring time $R$ to rasterize a pixel, and will require a total time equal to the maximum number of pixels rasterized by any processor, times $R$.

For each communication scheme we will determine the amount of data $q$ that it communicates and the distance $d$ that each datum must travel. The communication topology is a 1-D ring so for efficiency reasons we will consider neighbor–to–neighbor communication. Passing a datum from processor $n_a$ to processor $n_b$ will therefore require $|n_a - n_b|$ passes, so time is proportional to distance.

All processors may pass a datum to their neighbor simultaneously, so at any point in time $n$ data can be in communication. So the total amount of time required to communicate the data between processors is $O(q \cdot d/n)$ – each datum takes time proportional $d$ to communicate, we can communicate $n$ of them simultaneously, and there are a total of $q$ data to distribute.

In general the distribution of polygons over the screen is not uniform. The deviation from uniformity, expressed as the maximum number of polygons in a screen region divided by the average, is $f$. In addition polygons can overlap more than a single region. The average number of regions a polygon overlaps is $o$. Note that the size of a region is decreasing in $n$, because the screen must be subdivided further and further to share the work among all processors as $n$ increases. As the size of the regions decreases both $f$ and $o$ will increase. $f$ will increase because the sampling of smaller and smaller regions of the screen will expose larger variations in the polygon distribution. Likewise, as the size of the regions decrease each polygon will overlap more regions, so $o$ will increase. The dependence of $f$ and $o$ on $n$ will have important effects on the efficiency of the different communication approaches.

We will now analyze the communication options using this framework to quantify their performance. Note that optimally we would like an algorithm which runs in time

$$t_{optimal} = \frac{p}{n}(G + AR) \tag{4.1}$$

so that as we increase the number of processors we obtain a perfect linear speedup. This assumes that we perform no communication, that we aren't penalized by the overlap factor, and there is no asymmetry in the distribution of load across the processors. The following analysis will show how each of the communication algorithms violates these assumptions in some way.

### 4.3.2 Sort–First

Sort–first places the communication stage as early as possible in the algorithm, actually preceding the geometry stage. Communication will place each polygon on the processor(s) responsible for rasterizing it.

If the set of processors responsible for rendering a polygon will change slowly, sort–

first can leverage this temporal locality to perform less communication. Modeling the exploitation of locality is difficult with our model, and we will make the optimistic assumption that sort–first algorithms operate with *no* communication. This assumption proves to be useful, as the sort–first approach still incurs substantial overhead over a uniprocessor pipeline which alone can be used to compare it to other approaches. In particular, by placing communication before geometry each processor that a polygon overlaps will have to perform the geometry computations for the polygon. The overlap factor $o$ expresses this duplication. Furthermore any asymmetries in the distribution of polygons over the screen regions, $f$, will be exposed to the geometry stage, so we will spend time $Gfo\frac{p}{n}$ performing geometry computations.

The rasterization stage will also be penalized by the asymmetry factor $f$, and the worst case processor will have to rasterize $f\frac{p}{n}$ polygons. However, the processor will only have to rasterize $A/o$ pixels of each polygon, for a total rasterization time of $AR\frac{f}{o}\frac{p}{n}$.

The total time for sort–first communication is then

$$t_{sf} = \frac{p}{n} f(oG + R\frac{A}{o}) \tag{4.2}$$

which is near optimal for $f$ and $o$ close to 1. We expect both $f$ and $o$ to be increasing in $n$ however, making the scalability of sort–first communication questionable. Our analysis in addition has completely ignored the communication overhead, which is presumably non–zero.

### 4.3.3 Sort–Middle

Sort–Middle performs the geometry calculations and then distributes computed results to the processors responsible for rasterization. No locality is exploited, so the processor which performs the geometry computations is uncorrelated with the rasterization processor and the expected distance a datum will travel is $n/2$. In reality many polygons will be rasterized by more than one processor. The overlap factor $o$

will determine how many processors each polygon will be rasterized by, and thus have to be communicated to. In general there is no reason to believe that the processors will be local to each other, so the actual distance may approach $n$ as $o$ increases. We will pessimistically assume $d = n$ here.

Sort–middle only computes the geometry once for each polygon, so it avoids the penalty of duplicated work incurred in sort–first. Like sort-first however, it pays the asymmetry penalty in screen polygon distribution during rasterization.

If we call the time to pass the computed results of a polygon from one processor to its neighbor $C_{sm}$ then communication will require time $\frac{p}{n} \cdot n \cdot C_{sm}$ Rasterization time is identical to the sort–first case, so we have a total time of

$$t_{sm} = \frac{p}{n}(+f\frac{R}{o}) + pC_{sm} \tag{4.3}$$

Note that the total communication time for sort–middle is $pC_{sm}$ which is *independent* of the number of processors, so as $n$ gets large sort–middle algorithms will spend all their time communicating.

### 4.3.4 Sort–Last

Sort–Last is a bit of a wild card. It doesn't communicate polygon data, but instead communicates pixels, the rasterization results. Each processor renders some arbitrary subset of the database, and sorting is performed after rasterization, as these subsets will overlap in some arbitrary way in the final image.

There are two ways to think about compositing this information. The rasterization results could either be considered as a framebuffer on each processor, or as a set of polygon fragments on each processor. The former view suggests that we combine the framebuffers from all processors to obtain a final framebuffer. The latter view suggests we combine the rendered pixels from each processor to generate a framebuffer. These two possibilities are referred to respectively as "dense", because it communicates an entire framebuffer from each processor, and "sparse", because it communicates only the pixels rasterized on each processor.

In both cases the total geometry and rasterization work is $p(G + AR)/n$. The asymmetry factor $f$ is not exposed because we can evenly divide the polygons over processors, and the overlap factor $o$ is not exposed because we allow processors to rasterize overlapping regions. Ignoring communication overhead, sort–last offers the linear scalability we have been seeking, and in fact provides the optimal parallel speedup.

Dense sort–last communicates a screens worth of pixels from each processor. However, each pixel only has to be sent to the processor's neighbor. Upon receipt of a pixel the processor either passes the pixel on to the next processor if it occludes the processor's pixel, or instead forwards its own pixel. Section 6.3 provides an example of this communication structure, and a more careful explanation of why the distance each pixel must travel is 1.

The screen is a total of $Q$ pixels, so dense sort–last must communicate a total of $nQ$ total pixels. $n$ pixels are communicated in parallel, and each pixel travels a distance of 1, so the total communication time is $QC_{sl_{dense}}$. A *constant* amount of time is spent in dense sort–last communication, independent of both the number of polygons and the number of processors.

Sparse sort–last lacks a complete framebuffer on each processor. If we assume the framebuffer for the final complete image is distributed over all the processors then each processor has to communicate each of its rasterized pixels to some other arbitrary processor. Thus the expected distance a pixel will travel is pessimistically $n$. Each polygon has an average area $A$, and there are a total of $p$ polygons, so there are $Ap$ pixels to communicate. We can communicate $n$ pixels in parallel, each over a distance $n$, for a total communication time of $ApC_{sl_{sparse}}$.

Dense sort–last requires a total amount of time

$$t_{sl_{dense}} = \frac{p}{n}(G + AR) + QC_{sl_{dense}} \tag{4.4}$$

to render a scene, while sparse sort–last requires time

$$t_{sl_{sparse}} = \frac{p}{n}(G + AR) + ApC_{sl_{sparse}} \tag{4.5}$$

to render a scene. Dense sort–last offers communication that is constant overhead, while sparse sort–last closely resembles sort–middle, but lacks the asymmetry penalties inflicted on rasterization in sort–middle structures.

## 4.4 Analysis

Thus far our model has only assumed that our execution times are bounded by the sum of the worst case times of each stage. This is true for any SIMD architecture. We will now quantify each of the variables in the model for the Princeton Engine. In some cases we will use actual figures obtained by examining an implementation, and in other cases representative numbers will be used.

Determination of the cost of communication of course requires a model of the actual communication mechanism. On the Princeton Engine each processor can pass a single 16–bit datum to its neighbor in 3 operations: write the communication register, performs the pass, and read the communication register. Passing a datum of size $a$ will require

$$i = 3 \cdot a + b \tag{4.6}$$

where $b$ is an overhead factor relating to setup and test of the received datum. The factor of 3 reflects that passing a single integer through the communication register requires writing the register, performing the pass, and reading the register. Due to resource constraints these operations can't be pipelined over each other.

There are 3 distinct communication approaches to analyze. Sort–first and sort–middle both deal in polygons (or computed results thereof) and sort–last deals in pixels. A good estimate of the size of the datum for each is necessary to determine realistic execution times. A polygon is described in our implementation by 24 integers,

| approach | | datum size | overhead | $C$ |
|---|---|---|---|---|
| sort–first | | 24 | 30 | 102 |
| sort–middle | | 42 | 30 | 156 |
| sort–last | | | | |
| | full | 3 | 15 | 29 |
| | sparse | 5 | 30 | 45 |

Table 4.2: **Communication Costs:** comparison of datum sizes for different communication approaches.

while a polygon descriptor (the sort–middle datum) is 42 integers. Dense sort–last requires the communication of three color components of 8–bits each and a 24–bit z value. A sparse sort–last approach also requires each pixel to be tagged with screen coordinates, for an additional two integers.

The overhead figures and datum size are based on an implementation for sort–middle. The sort–first datum size is the size of a polygon in our database, and the overhead figure is taken as equal to the sort–middle case because they are comparable operations. Sort–last has been approximated and is based on experience. The overhead figures reflect that during the dense approach the coordinates of the next pixel received are known a priori, so less computation needs to be performed on each received pixels. Sparse sort–last on the other hand requires a framebuffer address calculation for each received pixel, and the unpredictability of the pixels received makes overlapping operations difficult.

Our modeling made a number of assumptions. The simulator was instrumented to measure a number of the relevant quantities, and verified our assumptions. In particular, Figure 4-1 shows the distribution of polygons over processors for a sort–first algorithm, and in particular demonstrates the severe load balance problems resulting from the object detail being centered in the screen. Figure 4-2 verifies our assumption that the number of communication operations for sort–middle would depend linearly on the number of polygons, and independent of the number of processors.

Given the values of the parameters common to all the algorithms, specified in

| variable | value | |
|---|---|---|
| p | | left unspecified for a more general analysis |
| n | 1024 | the number of processors in our Princeton Engine |
| G | 5000 | obtained from our implementation |
| A | 100 | the common benchmark is 100–pixel polygons |
| R | 100 | obtained from our implementation |
| Q | 393216 | the screen is 768x512 pixels |
| f | 4.4 | taken from the sample teapot scene |
| o | 3.8 | taken from the sample teapot scene |

Table 4.3: **Modeling Values**

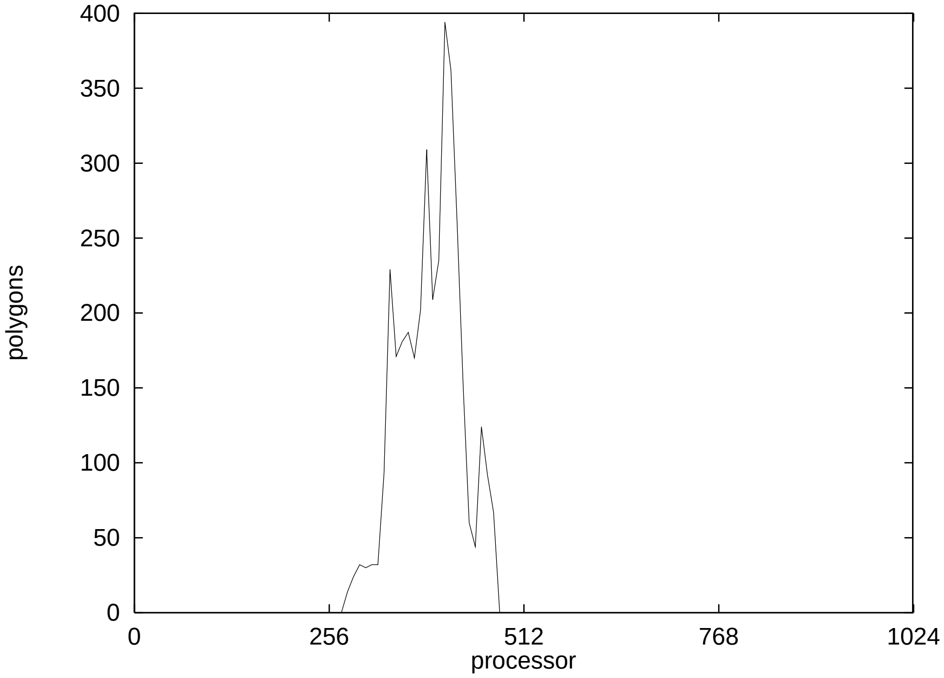



Figure 4-1: **Sort–First Load Imbalance:** the number of polygons per processor varies dramatically. The worst case processor has 394 polygons intersecting its region, while the average case is 23.3 polygons..

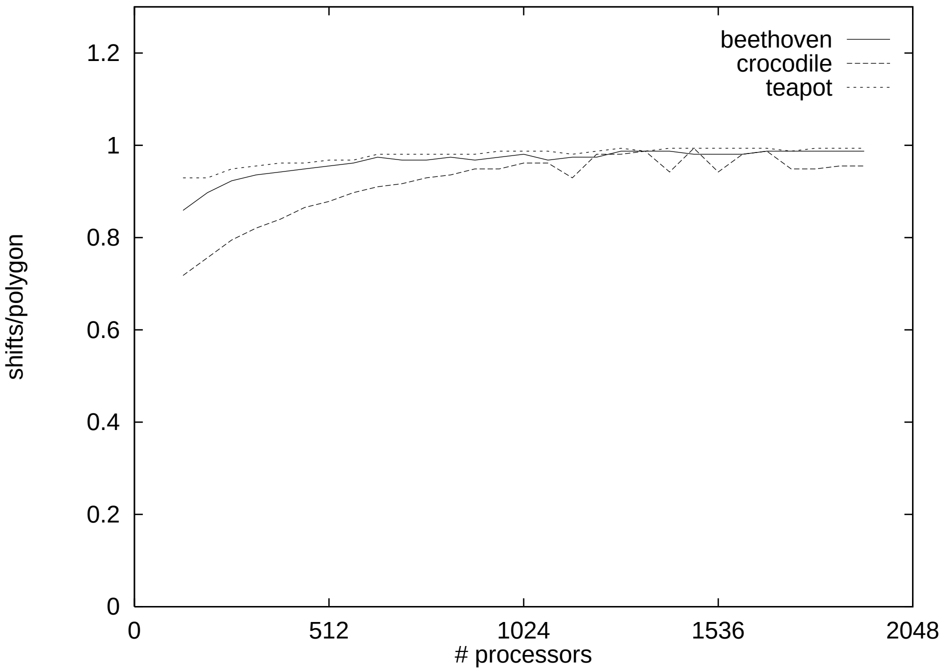



Figure 4-2: **Sort–Middle:** the number of shifts per polygon is the total number of shifts required to distribute the polygons, divided by the total number of polygons. The marginal cost of communicating a polygon is approximately constant at 1 shift per polygon.

Table 4.3, and the size of the datum each communication structure uses, given in Table 4.2, we now perform comparisons between these communication approaches to determine which is most practical on the Princeton Engine.

### 4.4.1 Sort–First vs. Sort–Middle

If we compare sort–first and sort–middle we see that sort–middle is faster than sort–first when

$$
\begin{aligned}
t_{sm} &< t_{sf} && (4.7)\\
\frac{p}{n}(G + fR\frac{A}{o}) + pC_{sm} &< \frac{p}{n}f(oG + R\frac{A}{o}) && (4.8)\\
G + nC_{sm} &< foG && (4.9)\\
fo > 1 + \frac{nC_{sm}}{G} & && (4.10)
\end{aligned}
$$

For our numbers sort–middle is superior if $fo > 32.9$. This analysis suggests that sort–first will be about a factor 2 faster than sort–middle, as $fo = 16.7$ for our sample scene. Nonetheless sort–middle was implemented preferentially because it is believed that in reality sort–first will be significantly slower than assumed here, for two reasons. First, the analysis above assumed that sort–first would require no communication, which is clearly false. Second, it compared a worst–case behavior for sort–middle. In reality through optimizations (which do not apply to the sort–first case) we can implement sort–middle about twice as fast as assumed here. Overall we expect that sort–middle is at least as fast as sort–first on this architecture, and likely to be significantly faster.

### 4.4.2 Sort–Last: Dense vs. Sparse

Ignoring common factors (the rasterization and geometry time) we see that the time required for a sparse sort–last algorithm is proportional to $p$, while the dense sort–last time is constant. Thus we can calculate the number of polygons where dense sort–last communication is preferable to sparse to be

$$p > \frac{Q \cdot C_{sl_{dense}}}{A \cdot C_{sl_{sparse}}} \tag{4.11}$$

or $p > 2534$. A 2500 polygon scene is extremely simple. At 30fps a mere 75000 polygons per second would be rendered. So for all but the most trivial scenes dense sort–last proves superior to sparse sort–last.

### 4.4.3 Sort–Middle vs. Dense Sort–Last

Our two remaining algorithms are sort–middle and sort–last. Sort–middle has performance linear in the number of polygons, and dense sort–last has constant performance, so once again we can find the tradeoff point where dense sort–last will be the algorithm of choice.

$$p > \frac{Q \cdot C_{sl_{dense}}}{C_{sm} + \frac{(f/o-1)AR}{n}} \tag{4.12}$$

If $p > 60,123$ then sort–last will be preferable. The time required to perform the sort–last composite will be $Q \cdot C_{sl_{dense}} = 11.4$M instructions, which is nearly an entire second on the Princeton Engine. So sort–last, while offering constant time performance, and thus scalability, proves prohibitively expensive.

Figure 4-3 compares the performance of these algorithms. As would be expected, sort–first requires the fewest instructions, followed by sort–middle and finally the sort–last sparse implementation.

Sort–middle is a definite win on this architecture. A number of optimizations are discussed later in the paper which will apply to most of these communication topologies, however orders of magnitude separate the performance of these algorithms, and there are no orders of magnitude in performance to be found in any optimizations, so sort–middle remains the optimal choice of communication structure.

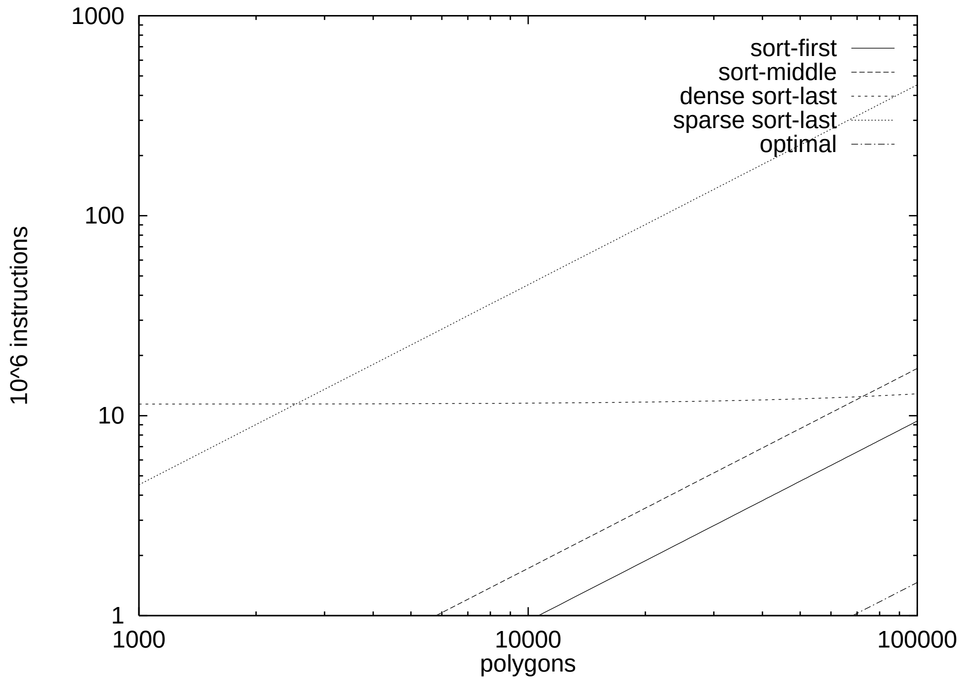



Figure 4-3: **Communication Costs:** comparison of instructions required to execute various communication structures versus number of polygons.

## 4.5 Summary

A framework has been provided to examine various communication alternatives for polygonal rendering. To provide a straightforward and analytical solution a number of assumptions have been made. Assumptions have been both optimistic (sort–first requires no communication) and pessimistic (sort–middle requires all-to-all communication) and have yielded insightful results.

The application of this analysis to the Princeton Engine reveals that algorithms, such as sort–last, which execute in constant time, may prove to be prohibitively expensive, while communication that executes in linear time, such as sort–first, may induce so much overhead that they prove impractical. Sort–middle, which runs in linear time in the number of polygons and *independent* of the number of processors would seem at first glance to be an infeasible solution, but actually proves to be the most efficient of the algorithms for our scenes of interest.

The observation that sort–last yields constant performance and sort–middle yields performance independent of the "clumping" of primitives that sort–first is subject to is significant. We will return to these two algorithms in Chapter 6 to develop a hybrid algorithm which extracts the constant time benefits of sort–last and the efficiency of sort–middle for small numbers of processors.

# Chapter 5

# Communication Optimizations

Chapter 4 motivated the implementation of a sort–middle communication scheme to maximize performance. However, if we look at the ratio of the time spent communicating to the total execution time

$$\frac{n \cdot C_{sm}}{G + n \cdot C_{sm} + fA\frac{R}{o}} \tag{5.1}$$

and evaluate for the Princeton Engine we see that communication will occupy approximately 75% of our total execution time. Even allowing for our pessimistic assumptions, we are already clearly facing the issue of scalability for sort–middle communication. The single largest obstacle to the implementation of a high–speed polygonal renderer on a SIMD ring is the communication bottleneck.

We have already determined that sort–middle communication will execute in $O(p)$ time, so we will focus considerable effort on minimizing the constant factors in the performance.

A number of possibilities could be exploited in the design of a more efficient sort–middle communication algorithm. They are largely orthogonal to each other, so when combined they yield a tree of implementations, show in Figure 5-1. Fortunately some of the branches prove ineffective, and they can be abandoned early in the analysis.

The options to be explored in the design of the communication algorithm have

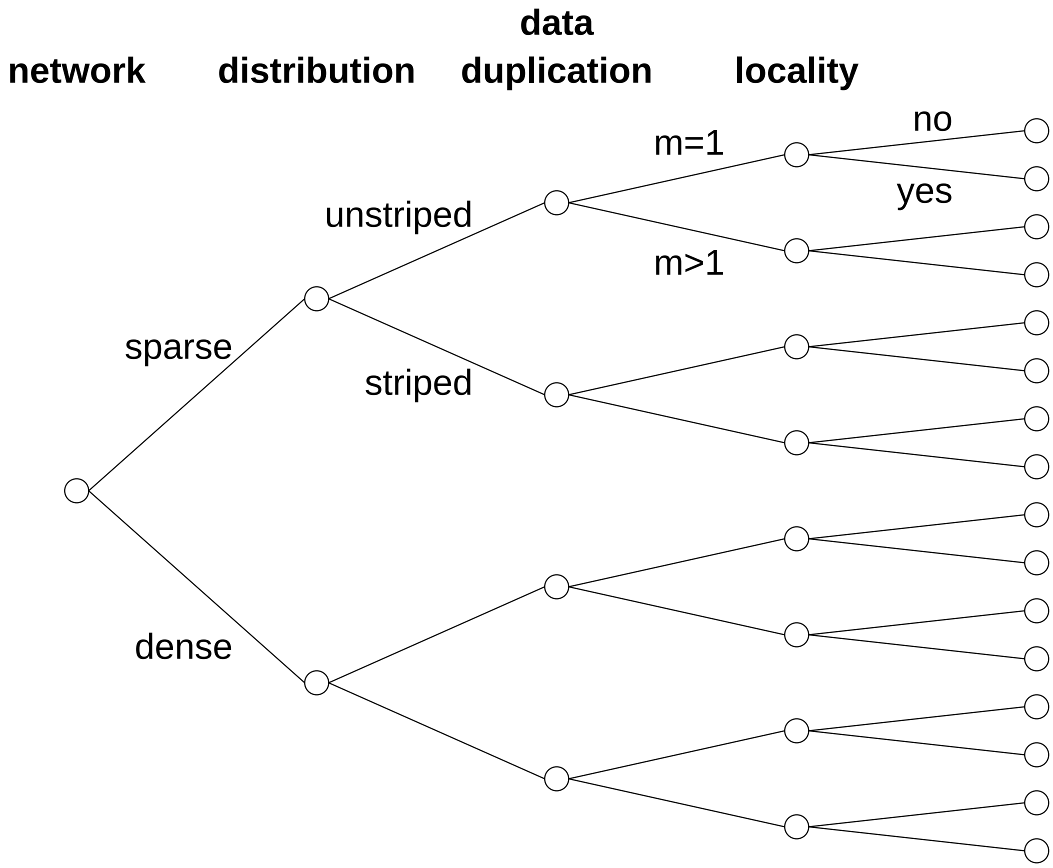



Figure 5-1: **Communication Optimizations:** the set of possible communication optimizations create a tree of possibilities to explore.

been ordered by both ease of implementation, and expected importance. The options are:

**Network** The communication network has a finite capacity for information. It may be desirable to attempt to saturate this channel, placing as much information into it as possible. There is a cost associated with saturating the channel because the setup and tests to achieve saturation require time, during which no communication is achieved. The optimization to keep the network as full as possible is referred to as a "dense" network (not to be confused with dense sort–last compositing).

**Distribution** The cost of passing a datum between arbitrary processors is decreasing in the size of the passes used. Using large passes (neighbor–to–neighbor–N) amortizes the cost of writing and reading the communication register, along with the overhead of the pass operation. A two–phase sort–middle algorithm which uses large initial passes to get data close to its destination, followed by neighbor–to–neighbor passes to precisely deliver the data could be more efficient than a naive sort–middle algorithm.

**Data Duplication** The polygon database could be duplicated on the processors, so that instead of there being a copy of a polygon on only 1 processor, there is a copy on $m$ processors. By duplicating the polygons periodically across the processors we have reduced the maximum distance any polygon must travel between geometry computations and rasterization, thus reducing communication time, at the possible expense of geometry time.

**Temporal Locality** Interactive polygon rendered scenes, such as animations, building walkthroughs, and molecular modeling, generally have high frame–to–frame coherence. A polygon rendered in one frame will be rendered in very nearly the same position in the next frame. Communication could account for this and allow polygons to "migrate" around the ring, staying reasonably close to the processors that have to rasterize them. As time progresses the polygons will move from processor to processor to track the changing view frustum, but these moves will be small, thus decreasing the amount of time spent in communication.

All of these options, while appealing, are difficult to analyze without actually implementing the algorithms. In general they deal with the subtle points of the communication, and while they may prove to be predictable, a general model is difficult to find.

Before discussing the optimization in detail a description of the implementation of sort–middle communication is given to clarify the issues addressed.

## 5.1 Sort–Middle Overview

The most straightforward implementation of sort–middle communication is shown in Figure 5-2. After the geometry stage each processor has a queue of computed polygon "descriptors", shown schematically as triangles in the figure. Communication begins with each processor sending the first polygon in its queue to its right neighbor, and simultaneously receiving its left neighbor's polygon. The processors then perform $n-2$ more passes, each time passing the descriptor they received from their left neighbor to their right neighbor. A copy of each received polygon is kept if it overlaps the rasterization region for the processor. Once the first batch of polygons has traveled all the way around the ring the processors then all dequeue the next polygon from their queues and repeat the process until all polygons have been communicated. When complete each processor has seen each of the first polygons from all the processors queues.

## 5.2 Dense Network

The communication of a polygon has been assumed to require $n$ passes, where $n$ is number of processors. This assumes that each processor is interested in each poly-

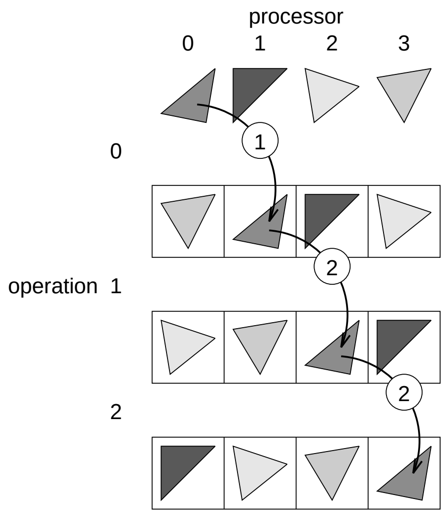



Figure 5-2: **Sort–Middle Communication:** the work queues, with a single polygon in each, are shown at the top. The first operation (1) passes the first polygon from each processor to its neighbor. The subsequent operations (2) pass these polygons around the ring until everyone has seen them.

gon. In reality only some small subset of the processors will rasterize any particular polygon, and once all processors in this set have seen the polygon it needn't be communicated any further. On any given pass operation up to $n$ polygons will be in communication. By ceasing the communication of polygons that have been fully distributed we can more fully utilize the communication slots available on each iteration of the algorithm.

These options can be referred to as sparse and dense communication, not to be confused with the sort–last approaches. The sparse approach naively communicates every polygon to every processor, and the dense approach stops communicating a polygon as soon as all of its rasterization processors have seen it.

Figure 5-3 compares the performance of sparse and dense communication under simulation. The data sets shown all have an average distance between the geometry processor and the rasterization processor for each polygon of approximately 512 processors (half the total number of processors). It has been assumed that the cost of the sparse and dense distribution schemes per pass operation are identical, which is

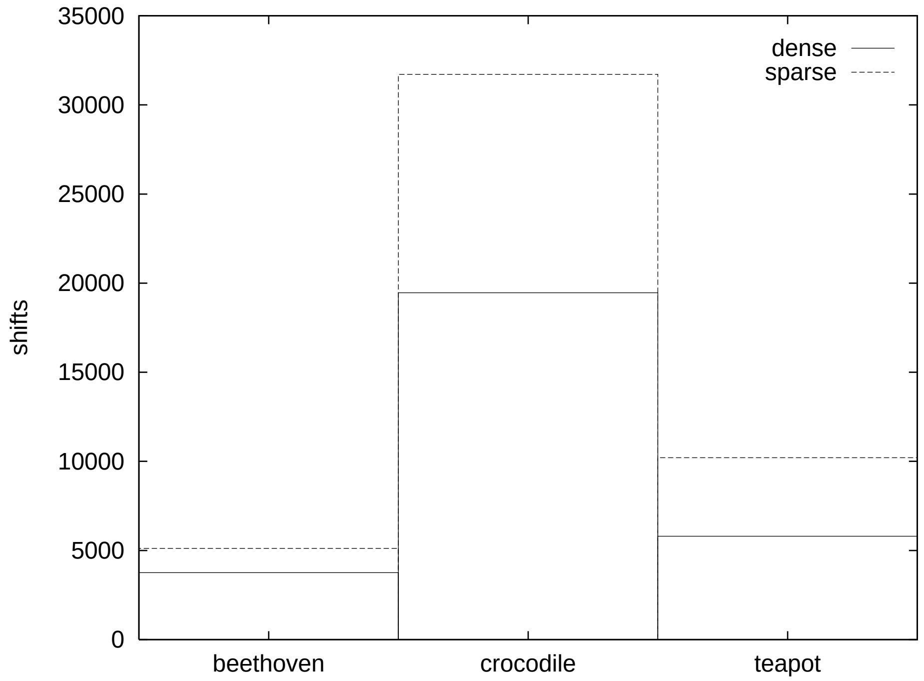



Figure 5-3: **Simulation of Sparse vs. Dense Networks:** the sparse network can require up to twice as long to distribute a given data set.

not strictly true. However, unless the tests necessary to obtain dense communication were very expensive it is clearly advantageous to use dense communication. The remainder of the optimization analyses are performed with the assumption that dense communication is used.

## 5.3 Distribution

The number of instructions required by a neighbor–to–neighbor pass is $i = 3a + b$ where $a$ is the size of the datum and $b$ is the overhead of the operation. The factor of 3 reflects the distinct operations on the Princeton Engine of writing the communication register, performing the pass operation, and reading the communication register. Architecture constraints prohibit pipelining these operations over each other.

If we perform a pass of some arbitrary distance $d$ (neighbor–to–neighbor–N) it will then require $i = (2+d)a+b$ instructions. A datum could be communicated a distance $d$ by performing a single pass of length $d$, or $d$ neighbor–to–neighbor passes. Clearly a single long pass is substantially cheaper, as you only pay the setup and overhead costs once, rather than $d$ times. Ellsworth discusses an analogous strategy in [7] which places processors into groups and bundles together polygon messages across group boundaries into a single message to amortize the cost of message overhead. Similar amortization isn't possible here, as our overhead is per polygon, not per message.

The cheapest way to get a polygon from one processor to another is to perform a single shift of the size necessary, rather than a number of small shifts. However, performing a number of special shifts of varying sizes will lead to poor utilization of the communication network. Instead we consider a two–stage algorithm in which polygons first take strides of size $d$ around the ring to get close to their destinations cheaply, and then are passed neighbor–to–neighbor to be precisely aligned. The instruction cost saving in this approach could prove to be substantial. Figure 5-4 shows the tradeoff between cost and distance. Note that we can obtain performance

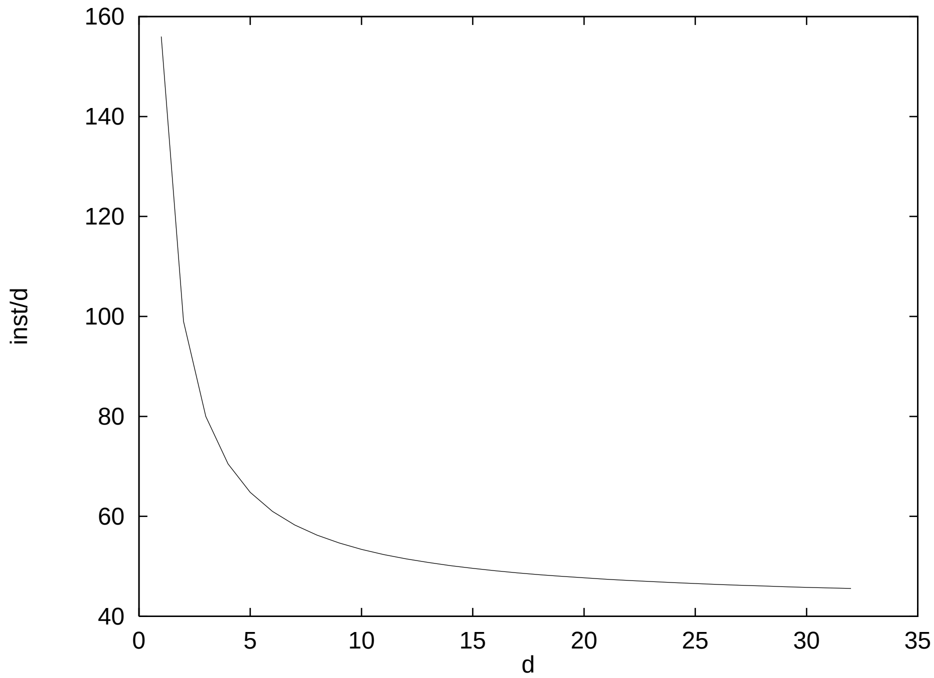



Figure 5-4: **Pass Cost:** the cost of communicating a datum per unit distance is inversely proportional to the distance of the pass.

no better than the case of a single pass which places the polygon precisely on its destination; so our performance improvement is limited to the ratio of the time for a single pass of length $d$ to the time for $d$ neighbor–to–neighbor passes:

$$\lim_{d \to \infty} \frac{a(2+d)+b}{d(3a+b)} = 3 + \frac{b}{a} \tag{5.2}$$

For our system $a = 42$ and $b = 30$, so we can expect best–case speedup of 3.7 in the time required to communicate a single datum.

We can solve for the optimal stride size $d$ under some assumptions about the distribution of data:

**Uniform Source Distribution** The polygons to be rasterized are uniformly distributed across the processors after the geometry stage. This is a good assumption, as the polygon database is distributed uniformly over the processors initially.

**Uniform Destination Distribution** The number of polygons to be rasterized by each processor is the same. This therefore assumes a uniform polygon distribution over the screen. It is likely some processors (those rasterizing the middle of the scene) will receive many more polygons than other processors. This introduces asymmetries in the communication pattern and will be a source of error between our analysis and simulation results.

**Single Destination** Each polygon is rasterized by a single processor, so the overlap factor is 1. This is a strong assumption. As we have already seen in Chapter 4 the average polygon is rasterized on about 4 different processors. However, column–parallel decompositions of the screen, in which each processor is assigned a column of the screen to rasterize, closely match this assumption because the processors that rasterize a polygon will be adjacent.

**Full Ring** At any step in the distribution all points of the ring are carrying data. This is not strictly true, because as the communication reaches completion there is no abrupt transition from the ring being full of information to being empty. For large data sets ($p \gg n$) and the dense distribution scheme this should be a reasonable approximation.

Given these assumptions, and their accompanying caveats, we can calculate an optimal distribution pattern, and its performance relative to a simpler neighbor–to–neighbor algorithm.

We have assumed that the polygon sources and distributions are uniformly distributed, so the expected distance a polygon will travel is $n/2$. A two-stage distribution scheme, were we first perform passes of some stride $d$ and then passes of stride 1 will require a total of $i^*$ instructions, composed first of instructions that perform passes of size $d$, and then instructions that perform the neighbor–to–neighbor passes. The size $d$ passes will require $i_d$ instructions:

$$i_d = \frac{p}{n}\frac{n}{2}\frac{1}{d}\{(2+d)a+b\} \tag{5.3}$$

Each pass requires $(2+d) \cdot a + b$ instructions, each polygon takes an average of $n/2 \cdot 1/d$ passes to communicate in the first stage, and there are a total of $p$ polygons being communicated, $n$ at a time.

After this first pass of communication each polygon will be an average of $d/2$ processors from where it needs to be, so, similar to equation 5.3, the second pass of distribution will require $i_1$ instructions:

$$i_1 = \frac{p}{n}\frac{d}{2}\{(2+1)a + b\} \tag{5.4}$$

and $i^* = i_d + i_1$:

$$i^* = \frac{p}{n}\left[\frac{n}{2}\frac{1}{d}[(2+d)a+b] + \frac{d}{2}[(2+1)a+b]\right] \tag{5.5}$$

Solving to find the minimum number of instructions yields

$$d = \sqrt{\frac{(2a+b)n}{3a+b}} \tag{5.6}$$

$$i^* = p\left[\sqrt{\frac{(3a+b)(2a+b)}{n}} + \frac{a}{2}\right] \tag{5.7}$$

The optimal stride is $d = 27.36$ for $n = 1024$, $a = 42$ and $b = 30$.

Figure 5-5 compares the predicted and simulated performance of the algorithm. The curve is very flat at the minimum, which is reasonable to expect, since the cost per unit distance traveled approaches 1 asymptotically with increasing $d$. As $d$ increases, data is distributed less closely to its optimal position in the first stage and the cost of the second stage grows.

The simulation shows a clear minimum number of instructions per polygon for $d = 8$ or $d = 16$. This is a little more than half of the analytically determined optimal $d$. The discrepancy arises because the "Dense Ring" assumption starts to break down earlier for large values of $d$, so our analysis underestimated $i_1$. The simulations show speedups of 1.96, 2.40 and 1.96, while our predicted speedup is 3.10, compared to the

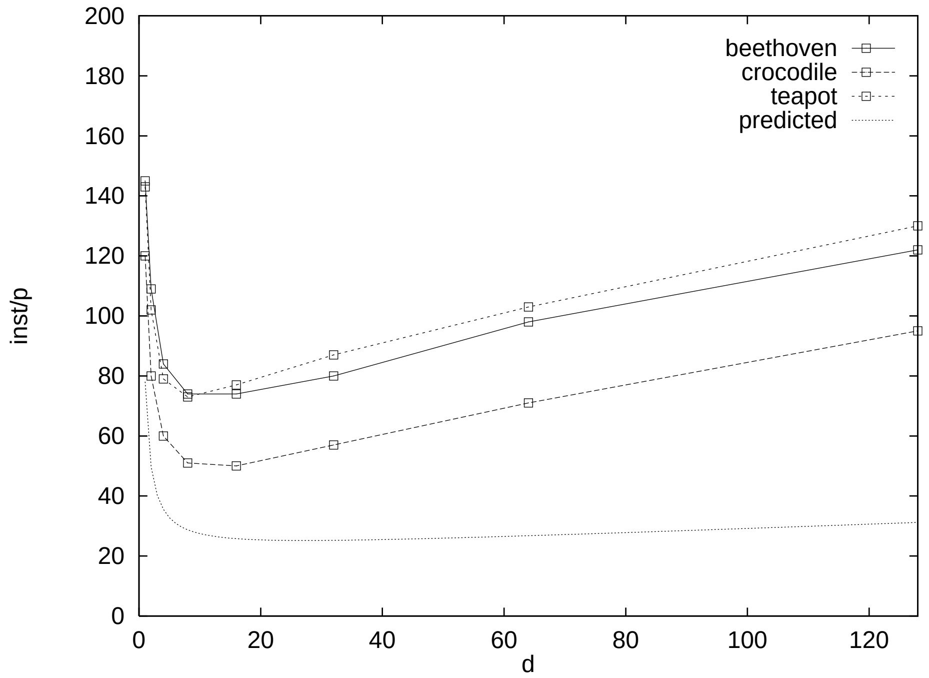



Figure 5-5: $i^*$ **vs.** $d$ **for Distribution Optimization:** data based on simulation of a 1024–processor machine. $d$ is constrained to be a factor of $n$ by the simulator.

maximum speedup of 3.7.

## 5.4 Data Duplication

Data duplication is an intriguing possibility for reducing the total communication requirements. By placing more than one copy of each polygon in the processor array we reduce the maximum possible distance any polygon must travel to be communicated to any given processor.

If we duplicate the polygon database $m$ times across the array then a polygon which lies on processor $k < n/m$ also lies on processor $k + n/m$, processor $k + 2n/m$, through processor $k + (m-1)n/m$. The maximum distance a polygon will have to travel from a source to destination is now $n/m$ instead of $n$. Furthermore, the distribution of the copies across the array is deterministic, so any processor can determine *without communication* which of $m$ processors should source each polygon for which it has a copy. Thus we have reduced the amount of communication we have to perform at the expense of doing extra work in the geometry stage. Each processor will now have to examine $pm/n$ polygons, and compute the geometry for the subset of them that it determines it is the best source for.

To analyze the potential performance improvements of data duplication we will use the same assumptions as used during the distribution analysis, namely: uniform source and destination distribution, single destination and dense ring.

To make a meaningful analysis we have to keep an eye on the cost of the geometry operations. As already discussed, the geometry stage may be broken into two pieces, first examining all of the polygons on a processor and determining which of them are visible, and then performing the complete geometry calculations only for the visible polygons. In the case of data duplication the visibility test will include deciding if this processor can communicate the polygon most cheaply of the $m$ processors with a copy of it. Let $G_i$ be the number of instructions required for a processor to transform a polygon to the screen and determine if it should source it. Let $G_f$ be the remaining instructions to complete the geometry stage per polygon. The number of instructions required to perform the geometry calculations and distribute the results for a duplication factor $m$ is $i_m$:

$$i_m = \frac{pm}{n} G_i + \frac{p}{2n} G_f + \frac{p}{2n} \frac{n}{2m} (3a + b) \tag{5.8}$$

Note that the duplication costs a factor $m$ on the number of polygons we have to examine ($G_i$), but costs *nothing* for the $G_f$ calculations, as these are only performed by one processor for each polygon. The final term in the sum is the communication time, which now takes distance $\frac{n}{2m}$ for each polygon, thanks to the duplication. We assume that half of all polygons are visible.

Implementation reveals that $G_i \approx 1000$ and $G_f \approx 5000$ instructions. Solving for the optimal choice of $m$ yields

$$m = \sqrt{\frac{n(3a+b)}{4G_i}} \tag{5.9}$$

$$i_m = p \left[ \sqrt{\frac{G_i(3a+b)}{n}} + \frac{G_f}{2n} \right] \tag{5.10}$$

and $m = 6.32$ for $n = 1024$, $a = 42$ and $b = 30$.

It is interesting to note that the optimum constraint does *not* depend on the cost of $G_f$, as we have assumed that the visible polygons are uniformly distributed across the processors for the $G_f$ stage of computation. If there are any non–uniformities in the destination distribution these will be reflected back in the choice of which processors source the polygons for them, creating a non–uniformity in the geometry stage.

Our analysis thus far has been largely ad hoc due to the difficulty of modeling asymmetries in the source distribution and destination distribution. The simulator can model these effects precisely, and yields interesting results. Figure 5-6 compares the analytical expression for the optimal choice of $m$ to the simulator results.

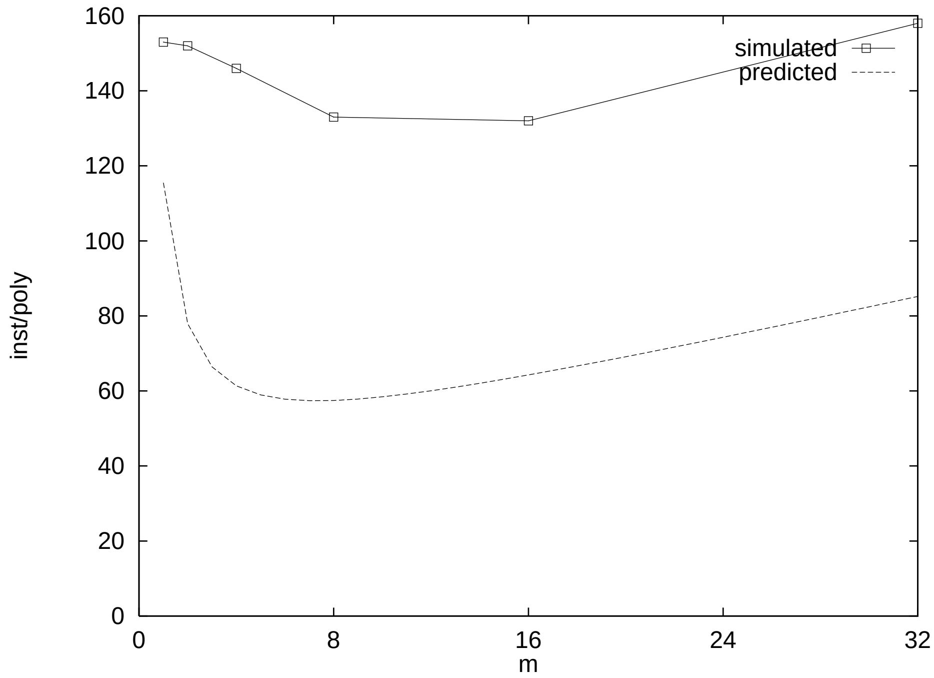



Figure 5-6: **Polygon Duplication:** instructions executed per polygon on a simulated 1024 processor machine vs. $m$.

Although the curves have the same general shape, there is a large discrepancy between the two results. Part of the discrepancy is due to the mistaken assumption that 50% of the polygons will be visible, while the actual number is closer to 67% in this scene. The simulator also expresses results in instructions per *visible* polygon, which is in some ways more useful than instructions per total polygon, because it reflects the effort that goes into the useful (visible) polygons, rather than all of the polygons. However, these two factors alone far from account from the difference in the results.

Figure 5-7 reveals how our analysis could have led us so far astray. The clumping of the object in the middle of the scene creates a clumping of the geometry work on approximately one fourth of the processors, so the geometry stage was 4 times more expensive then we expected. Furthermore the communication is more expensive because it has all become relatively local, and large parts of the ring go unutilized. However, the simulation still reveals a performance improvement on the order of 20% for the best case improvement at $m = 16$.

It appears that we have found another method for reducing the total amount of communication, at the expense of increased geometry computation. At the optimal point $m = 16$ we spend an average of 132 instructions per polygon versus 153 for the no duplication case, a savings of 20%. The logical extension to to this simulation is to incorporate the distribution optimization already discussed for a further performance improvement.

Figure 5-8 shows the instructions per visible polygon as a function of $d$, the data distribution step size, and $m$, the duplication factor. The minimum point is found at $d = 16$ and $m = 1$, which indicates that the duplication optimization, while effective in and of itself, does not combine well with other optimizations. Intuitively the duplication optimization is minimizing the distance any polygon has to be traveled, thus making the distribution optimization less effective.

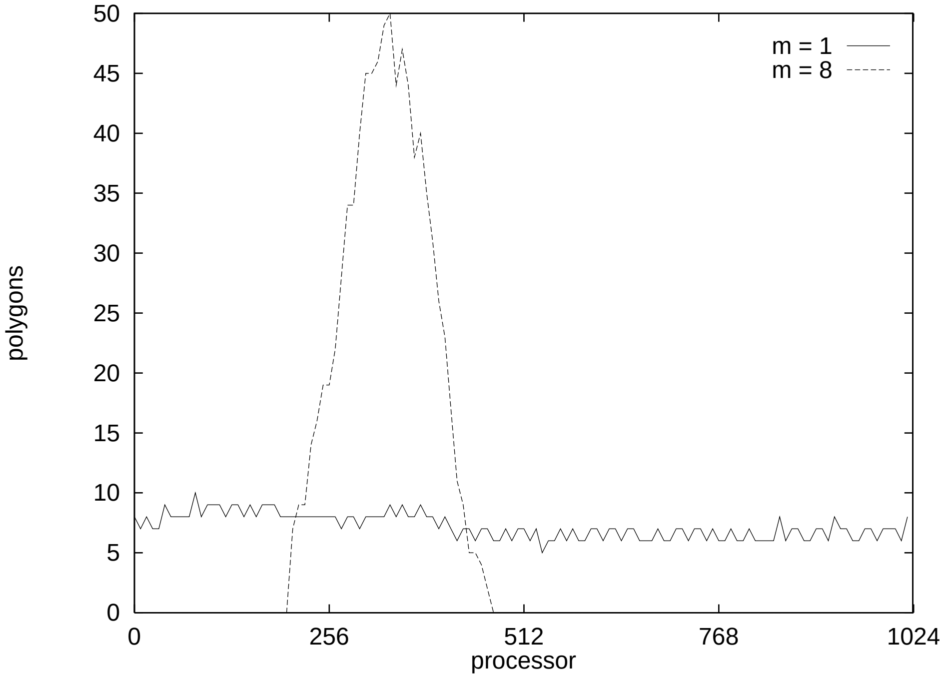



Figure 5-7: **Polygons Per Processor vs.** $m$: the asymmetric distribution of scene content is reflected in the distribution of polygons during geometry processing. Data is displayed as the maximum of 8 adjacent processors for readability and is taken from a simulation of the teapot data set.

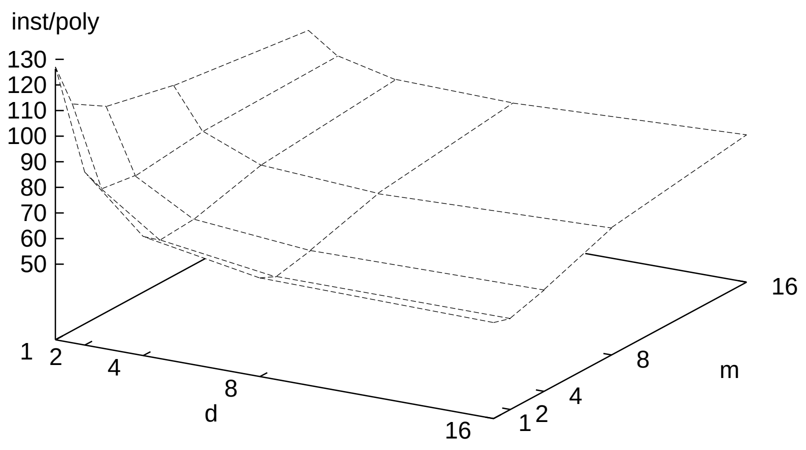



Figure 5-8: **Instructions Per Polygon as a function of $m$ and $d$:** the optimal choice of $d$ (16) and the optimal $m$ (8) do not together yield an optimal solution. Simulation is based on the crocodile data set.

## 5.5 Temporal Locality

Temporal locality attempts to exploit locality in a similar fashion to data duplication. The polygons are allowed to migrate around the ring for the geometry computations, rather than always being computed on the same source processor. A polygon remains as close as possible to the processors that will be rasterizing it, thus minimizing communication time. Unlike sort–first only a single copy of the polygon is computed during the geometry stage, thus removing the duplicated computation penalty, but still incurring any load imbalances due to non–uniform polygon distribution over the screen. As in the data duplication case, this is most meaningful for maps of processors to pixels that keeps adjacent screen areas on adjacent processors, such as a column–parallel decomposition.

Temporal locality can be thought of as data duplication where $m = n$, except we don't have to actually duplicate the entire database on each processor.

The communication time is greatly reduced because each polygon is rasterized on processors immediately adjacent to the processor which performed its geometry. The average distance a polygon must travel becomes the average width of a polygon. In this case another communication cost gets added into the overhead: we must now communicate the polygon (not the computed representation we will rasterize) between processors from frame to frame. These moves will be small in general, as we are exploiting locality. However, they are complex. For instance, a polygon could move slightly to the left of where it is on the screen, or slightly to the right, so we must now do passes in each direction if we don't want to have to pass polygons all the way to the right to move them slightly to the left. Furthermore we need to do something with the culled polygons. At some point they could be visible again and we need to keep them around on some processor.

Ignoring these issues we can first just examine the kind of load imbalances we are facing in this system. In general they will be much higher than those of data duplication because we are getting the polygons in an *optimal* position which will expose all of the asymmetries in the load distribution. Figure 5-9 shows the polygon distribution during the geometry stage for the teapot dataset. Here the worst case processor has 176 visible polygons to perform the geometry for, at a cost of $G_i + G_f$ instructions per polygon, ignoring the culled polygons which must also be handled.

The overhead of this approach over a balanced geometry stage is the excess number of polygons on this processor. The teapot data set has 9,408 polygons, or 10 polygons per processor, so temporal locality imposes a 166 polygon penalty on the effective polygon load per processor. We can compute the number of extra instructions this requires per visible polygon as $166 \cdot (G_i + G_f)/p_{vis} = 167.8$ which exceeds even a naive implementation of sort–middle communication which incurs a marginal cost of 156 instructions per polygon. Once we include the necessity to perform communication both to pass a polygon to its neighbors for rasterization and to maintain temporal locality we are likely to be far in excess of this figure.

Temporal locality remains an intriguing optimization due to its similarity with sort–first communication structures. However, its vulnerability to asymmetries in the polygon distribution over the screen make it unsuitable for use here.

## 5.6 Summary

Four communication optimizations were examined: network, distribution, duplication, and locality. Specific attention is paid to the application of the optimizations to the Princeton Engine, and the interaction of the optimizations.

Maximizing the density of information in the communication channel, the most obvious optimization, produces startlingly positive results. Distribution continues this trend and offers speedups of greater than 100% by amortizing the cost of loading and storing the communication register over passes larger than neighbor–to–neighbor.

Data duplication strives to improve the performance of the communication time by decreasing the maximum (and the mean) distance between a source processor and the destination processors. Non–uniform scene content distribution becomes reflected

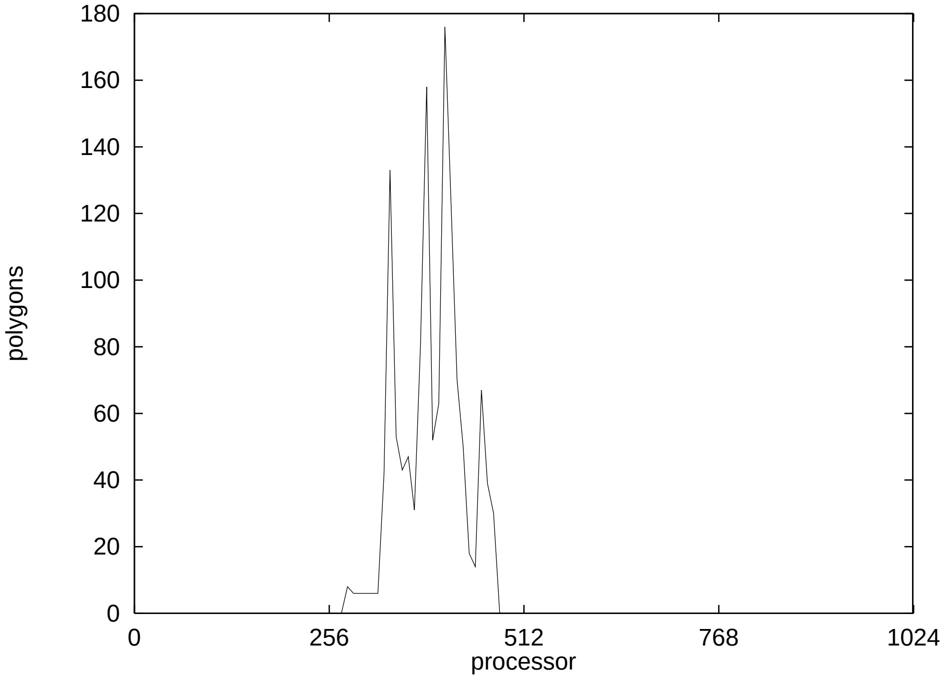



Figure 5-9: **Temporal Locality:** the number of polygons per processor for geometry computations directly reflects the underlying scene complexity for each column. The worst case processor has 176 polygons to perform geometry calculations for, almost 30 times the average number of polygons to compute. Data is displayed as the maximum of 8 adjacent processors for readability.

in the distribution of work during geometry computations, making this approach more similar to sort–first. While offering a significant speedup, it offers a smaller performance improvement than the distribution method, and does not offer any performance improvements when combined with distribution.

Temporal locality is conceptually the logical extreme of data duplication, where the duplication is achieved by sort–first style communication. It proves to be vulnerable to the same load imbalances that sort–first communication falls victim to. The extremely fine subdivision of the screen coupled with a highly variable distribution of scene content creates a few extremely heavily loaded processors which impose a huge performance penalty on the geometry calculations.

# Chapter 6

# Sort–Twice Communication

As part of this thesis a novel new approach to the sorting problem was developed, entitled "sort–twice." Sort–twice leverages the efficiency of sort–middle communication for low polygon count scenes to create an implementation of sort–last compositing that is practical with modest bandwidth processor interconnect. The implementation of a two–stage communication algorithm allows a novel overlapping of the rasterization pipeline with the second communication stage to create a highly efficient deferred shading and texturing system.

## 6.1 Why is Sort–Last so Expensive?

Sort–last has primarily been the domain of hardware solutions due to the very high bandwidth required. For example, PixelFlow [12] uses a dedicated 256–bit bus at 132MHz, for a bisection bandwidth of 4.2GB/s, to provide the necessary compositing interconnect. By comparison, the Princeton Engine, which employs the same style of linear interconnect, has a meager 28MB/s of interconnection bandwidth, more than two orders of magnitude less than PixelFlow.

Why is so much bandwidth required? The bandwidth for sort–last is purely driven by the number of samples to composite and the update rate of the display. A modest single–sample system of of 768 pixels by 512 pixels at 30 frames per second requires a staggering 71MB/s of communication, which, although attainable on our second generation machine, would leave little time for any other computation.

## 6.2 Leveraging Sort–Middle

Sort–twice communication leverages the low expense of sort–middle communication *over short distances* to make sort–last compositing practical on a general purpose machine such as the Princeton Engine. The use of sort–middle communication in a system supporting multiple users provides the foundation for this new approach.

Figure 6-1 shows a 15 processor machine supporting 5 simultaneous users. Each user is assigned to a group of 3 processors for geometry calculations, and an adjacent group of 3 processors for rasterization. The skew between the geometry and rasterization processors allows all communication shifts to be performed in the same direction, a performance boon on a SIMD machine. The communication stage only has to pass a polygon across a maximum of 5 processors, rather than 14 processors if the machine were dedicated to a single user. This system delivers performance to each user indistinguishable from a machine the same size as the group of processors assigned to the user.

If we increase the total number of processors while leaving the number of processors assigned to each user unchanged the aggregate polygon rendering performance will increase *linearly*. Whitman discusses the implementation of a parallel renderer on the Princeton Engine specifically to support multiple users in [20].

Note that the number of processors assigned to a user has no effect on the number of polygons each processor can transform per second, or the number of pixels it can rasterize. If we assign a single processor to each user we don't need to perform any communication and we can maximize the aggregate rendering performance of the system, of course at the expense of the per user polygons delivered each second.

If we wish to focus all of our computing resources on a single user we can still use the multiple user decomposition, except each group of processors while conceptually

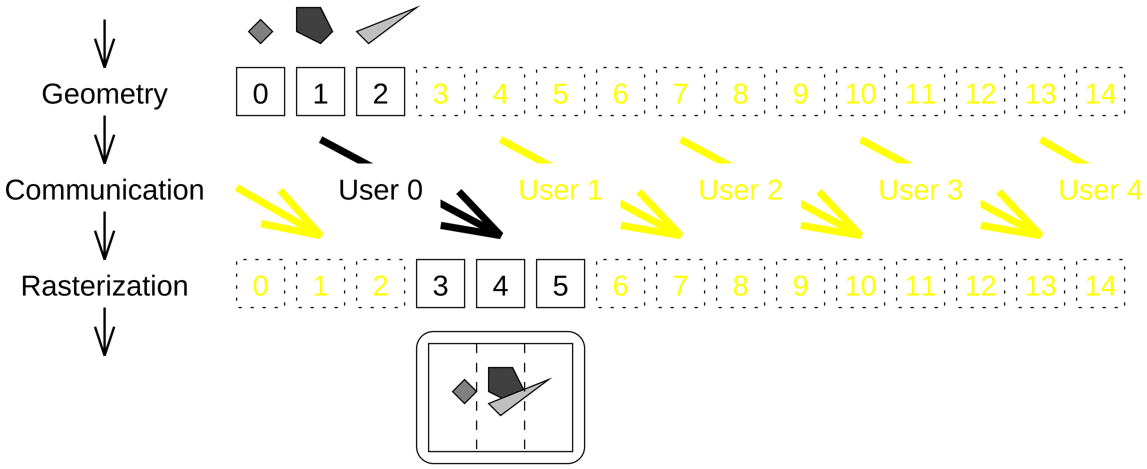



Figure 6-1: **Multiple Users on a Parallel Polygon Renderer:** each user is assigned to a small fraction of the processors.

working for different users will in reality be working for the same user. We will divide the machine into groups of $g$ processors, where $g$ is the group size ($g = 3$ in Figure 6-1). Similarly, we divide the screen into $g$ non–overlapping regions, numbered 0 through $g-1$. A polygon that overlaps a region $r$, $0 \leq r < g$ may be rasterized by any processor $m$, $m = r + k \cdot g$. If we distribute the polygon database over all processors, no matter what processor performs the geometry computations for a polygon the polygon descriptor will have to travel a maximum of $g-1$ processors to arrive at a processor that can rasterize it, as opposed to a maximum of $n-1$ processors in the typical sort–middle implementation. After all the processor groups complete rasterization, we can composite the images generated by the distinct sets of processors in a fashion analogous to dense sort–last compositing to generate the final output image.

Equation 4.3 gives the time for sort–middle rendering. If we change this equation to reflect the new maximum distance a polygon must travel we see

$$t'_{sm} = \frac{p}{n}(G + gC_{sm} + f\frac{R}{o}) \tag{6.1}$$

and our new implementation requires time proportional to the number of polygons and inverse in the number of processors, for a linearly scalable system! However, we now must pay the cost of performing sort–last composition to combine the results of all the individual rendering groups.

## 6.3 Sort–Last Compositing

Sort–last compositing deserves further explanation for this algorithm, as it departs significantly from the classical dense sort–last approach. Processors may no longer rasterize arbitrary pixels from the entire screen. Instead each processor may render only pixels in its *region* of the screen. The $n/g$ processors that have rendered pixels in the same region of the display must then composite their results to generate the final output image.

As Figure 6-2 shows, the region of the screen that a processor is responsible for repeats every $g$ processors. Thus the communication for the sort–last compositing stage of sort–twice is completely deterministic. Most importantly, this compositing can be performed in constant time, in a fashion analogous to normal sort–last compositing.

The final result of the composition operations is each pixel is known at *some* deterministic processor $m$. The correct pixel is not known at all the processors that could have rendered the pixel, as this requires all–to–all communication and would require time proportional to $n$. If we only have "the correct answer" for any particular pixel on one processor we can achieve the composition in constant time.

The composition of a single pixel is performed as follows: processor $m+g$ takes its copy of the pixel and passes it to processor $m+2g$, which composites the pixel with its own rendered pixel, and passes the result to processor $m+3g$. This processor is repeated until finally the pixel is passed to processor $m$, which composites the pixel into its framebuffer. Thus $n/g-1$ shifts are required to composite a single pixel. Following the path of pixel 4 in Figure 6-2, it starts on processor 0 which simply passes its copy of the pixel to processor 3. Processor 3 then composites the received pixel with its pixel 4 and passes the result to processor 6. This continues, until finally

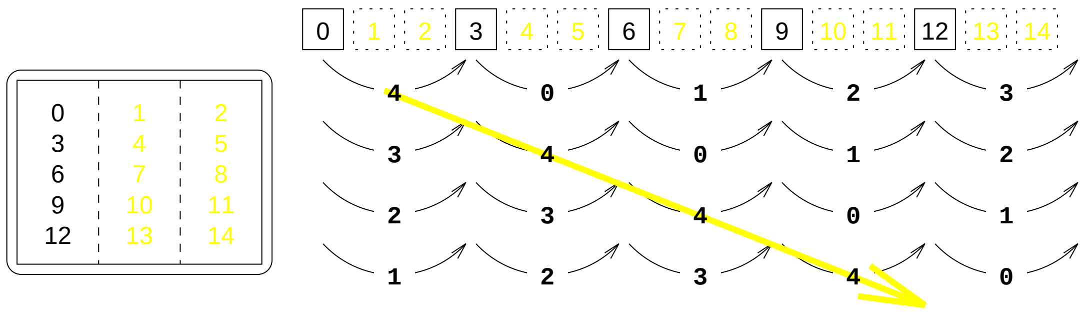



Figure 6-2: **Sort–Last Compositing:** a sample machine with 15 processors and a group size of 3. The assignment of processors to regions is shown on the display at left.

processor 12 receives the pixel and composites it into its framebuffer.

As Figure 6-2 shows, all $n/g$ processors responsible for the same region can simultaneously composite $n/g$ pixels in $n/g - 1$ shifts. There are $g$ such groups of processors compositing, so $n$ pixels are composited in $n/g - 1$ operations. The resulting composited image will be distributed across the processors. The processor that will hold the final composited version of each pixel is shown in Figure 6-3.

There are $P$ pixels in the output frame, so the frame will require

$$\frac{P}{n}(\frac{n}{g} - 1) = P(\frac{1}{g} - \frac{1}{n}) \tag{6.2}$$

shifts to composite. The number of shifts is bounded by $P/g$, a constant time solution for some choice of $g$, independent of the number of processors and the number of polygons. Note that if $g = 1$ then we have the dense sort–last composition, as discussed previously.

While this composition requires communication operations inversely proportional to the group size, each communication operation is across $g$ processors, with cost proportional to the group size, for overall constant performance.

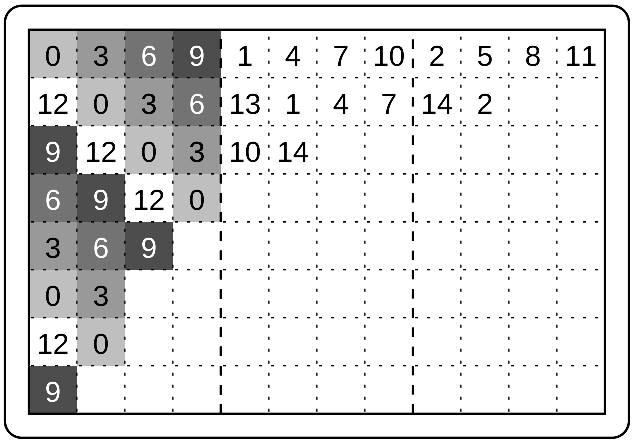



Figure 6-3: **Pixel to Processor Map:** the final mapping of fully composited pixels to processors results from the interleaving of the compositing operations. The processor that is responsible for each pixel labels the pixel.

## 6.4 Sort–Twice Cost

Compositing the final image will require $P(\frac{1}{g} - \frac{1}{n})$ communication operations. Each operation will be a pass of a pixel across $g$ processors.

The cost of communicating a pixel is of the same form as given for polygons, $i = (2 + d)a + b$. In this case $d = g$ (each pass is across an entire group), $a = 3$ (24 bits of color and 24 bits of z) and $b = 15$, as taken from table 4.2. So while the number of shifts is decreasing linearly in the group size, the cost of the shifts is increasing linearly in the group size, and the composition will require constant time.

Examining the total number of instructions required to perform the compositing we see

$$i = P(\frac{1}{g} - \frac{1}{n})\,[(2+g)a + b] \tag{6.3}$$

$$\frac{i}{P} \approx a + \frac{2a+b}{g} \text{ if } (n \gg g) \tag{6.4}$$

Equation 6.4 is enlightening. While the number of instructions executed per sample is independent of $n$ and $p$, it is inversely proportional to $g$, so we would like to make $g$ large to minimize the cost of each sample we composite. For our model we have $a = 3$ and $2a + b = 21$. As $g$ increases we pay less and less per sample, asymptotically approaching a mere 3 instructions per pixel, for a factor of 8 speedup over the classical sort–last approach ($g = 1$). As with the "distribution" optimization, we are amortizing the cost of the tests and write/read of the communication register over a larger pass (caused by a larger $g$). It is interesting that the classical dense sort–last compositing case is simultaneously the most expensive to composite, requiring 24 instructions per pixel, and the most expensive in memory, requiring an entire framebuffer on each processor.

## 6.5 Deferred Shading and Texture Mapping

As shown in Figure 6-4, the time required to perform sort–last composition is decreasing in the group size, so logically we would like to maximize the group size (with an eye on the cost of sort–middle communication) to maximize our overall performance. During the pure communication of the current pixel data (not loading or storing registers or performing tests on the data, but the actual time the data is in transit), the processors are essentially idle. It is known a priori that the processors next operation will be to composite the pixel it receives with its own pixel. While waiting to receive this pixel, shading and texture mapping for this processor's pixel may be performed. By overlapping shading and texture mapping operations until the sort–last communication stage we save time and effort in the rasterization stage, and leave our communication time unaffected.

Our expression for the cost of a pass is $(2+d)a+b$ instructions. $2a$ instructions are spent writing and reading the communication register, and $b$ instructions are spent testing the received datum and preparing to transmit the next. $da$ instructions are spent with data simply in transit, which means our VLIW processors are executing essentially just `passRight` for each instruction, and nothing more. For a pixel size of

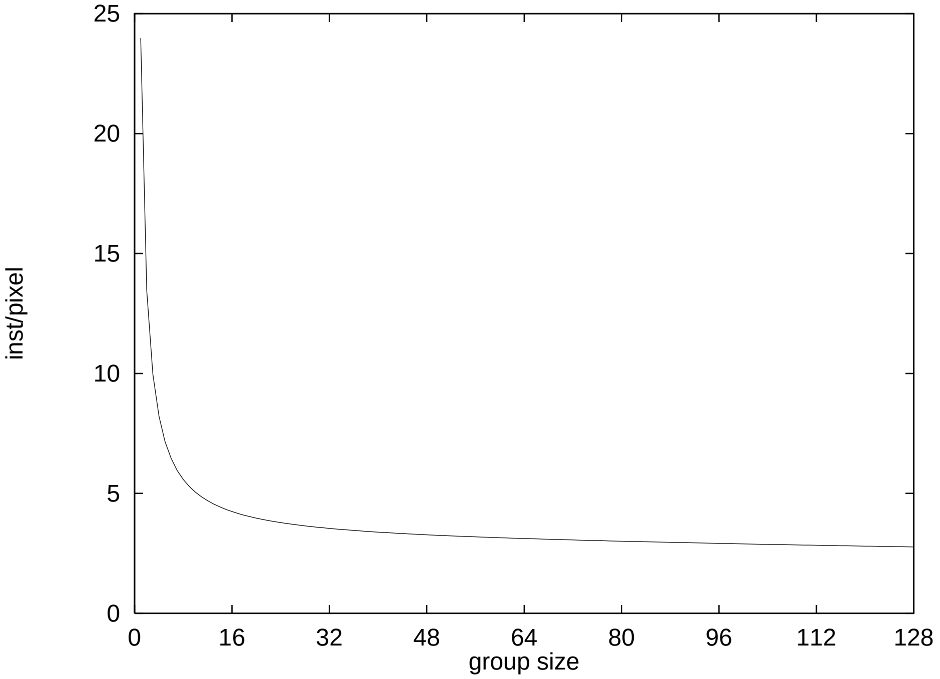



Figure 6-4: **Sort–Twice Compositing Time:** the payoff for increasing $g$ diminishes rapidly. Note that simultaneous with increasing $g$ and decreasing the composition time, the sort–middle time is increasing.

$a = 3$ and a modest group size of $g = 64$, this amounts to 192 instructions *per pixel* which go almost completely unused by the processors.

If we look ahead to the results in Chapter 8, our current implementation requires 202 instructions per bilinearly interpolated texture mapped pixel and only 80 instructions per shaded pixel, so texture mapping is a 122 instruction premium. Clearly with 192 instructions available we could defer both texture mapping and shading.

By deferring these operations until after rasterization, the rasterization stage is made much simpler. In particular, consider the case of texture mapping. Previously if any polygon was texture mapped, all processors had to operate in texture mapped mode and take the performance penalty. Now the performance penalty (except the extra parameters to interpolate) is completely deferred until sort-last composition, so the texture mapping overhead is entirely hidden.

A rapid prototype of this system shows that in unoptimized (compiler generated) code it will rasterize a single pixel with deferred texture mapping and shading in 98 instructions, a factor of 2 improvement over our current implementation.

## 6.6 Performance

The time to composite a frame is completely deterministic, and we have had good luck predicting sort–middle communication, so we can confidently predict the performance of sort–twice communication. Figure 6-4 shows the instructions required per pixel for various choices of $g$. If we consider the limiting case $n \gg g \gg 21$ we must have at least 3 instructions per pixel to composite, and we can see that this will require $1.17 \cdot 10^6$ instructions for a 768 by 512 pixel display. The Princeton Engine operates at 14MHz so the total execution time to composite an output image is 0.084 seconds. If we do nothing but composite frames we can operate at a maximum of 11.9 frames per second. Clearly this solution is infeasible for an interactive rendering environment on the Princeton Engine.

It is interesting to consider the attainable performance of this system in and of itself however. Part of the performance is still tied up in the sort–middle communication. As discussed earlier, a reasonable model for sort–middle communication assumes that each polygon will have to be passed over half the processors. Sort–twice reduces the number of effective processors to the group size.

The sort–middle stage will execute

$$i_{sm2} = \frac{p}{n}\frac{g}{2}((2+d)a+b) = \frac{p}{n}84g \tag{6.5}$$

instructions, where $p$ is the number of visible polygons. The sort–last stage, assuming $n/g \gg 1$, will execute

$$i_{sl2} = P(3+\frac{21}{g}) \tag{6.6}$$

instructions. Solving for the optimal group size (as determined by the minimum number of instructions) gives

$$g = \sqrt{\frac{21Pn}{84p}} \tag{6.7}$$

$$i_{s2} = i_{sm2} + i_{sl2} = 3P + \sqrt{\frac{441Pp}{n}} \tag{6.8}$$

so the group size increases with number of processors, amortizing the fixed costs of a pass operation, and decreases in the number of polygons, reducing the cost of the sort–middle operations.

By comparison, pure sort–middle communication with the distribution and dense network optimizations requires

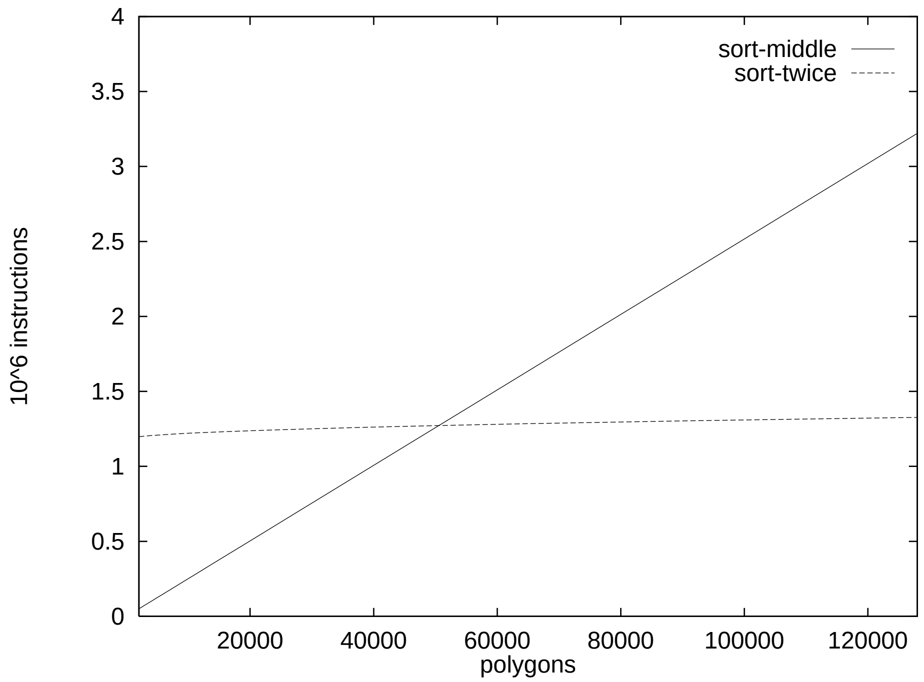



Figure 6-5: **Predicted Sort–Middle vs. Sort–Twice Communication Time:** at about 50,000 polygons per frame the performance of sort–twice will exceed the performance of sort–middle.

$$i_{sm} = p\left(\sqrt{\frac{(3a+b)(2a+b)}{n}} + \frac{a}{2}\right) = p\left(\frac{133.4}{\sqrt{n}} + 21\right) \tag{6.9}$$

instructions. The number of instructions required by sort–middle and sort–twice intersect at $p = 50503$. This is a reasonably high number of polygons per frame. Assuming we are rendering at 30 frames per second this implies a polygon rate of over 1.5M rendered polygons per second.

Figure 6-5 shows the predicted performance of the sort–twice communication algorithm versus the performance of sort–middle. In particular, note the extremely gentle increase in communication time as a function of the number of polygons for sort–twice.

Figure 6-6 compares the simulated performance of sort–middle and sort–twice

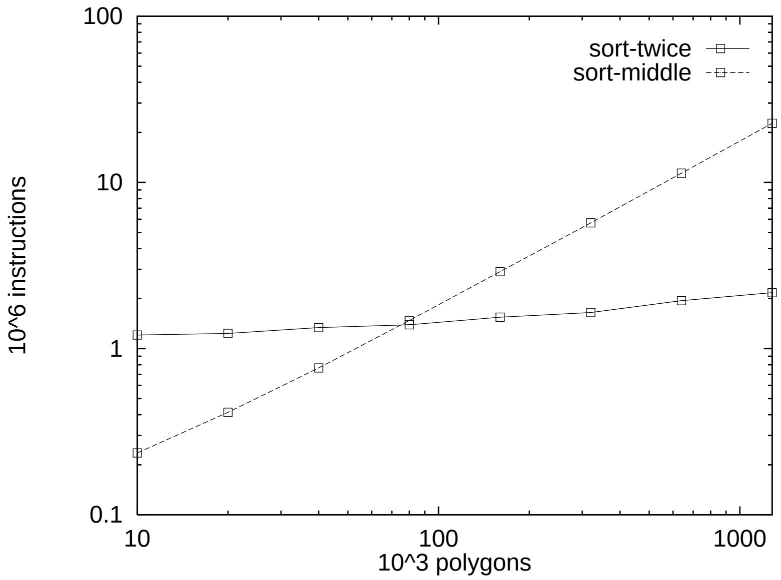



Figure 6-6: **Simulated Sort–Middle vs. Sort–Twice Communication Time:** the raw communication times of the algorithms closely match the predicted results. Note that the simulator restricts the choice of group size $g$ to be a factor of the number of processors. At each data point $g$ was chosen as the largest factor of $n$ smaller than the predicted optimal choice of $g$.

communication algorithms for scenes of up to 1.28M polygons. The scenes were generated as random polygons uniformly distributed over the screen with 50% of the polygons visible.

Figure 6-7 includes the costs of geometry and rasterization. For scenes beyond 64,000 polygons sort–twice consistently outperforms sort–middle.

## 6.7 Summary

Sort–twice offers an extremely efficient method to obtain linearly scalable performance in the number of processors, without paying the full cost of dense sort–last compositing.

The display is divided into $g$ regions, and every processor $r + kg$ is allowed to rasterize any polygon that intersects region $r$. By allowing a number of different

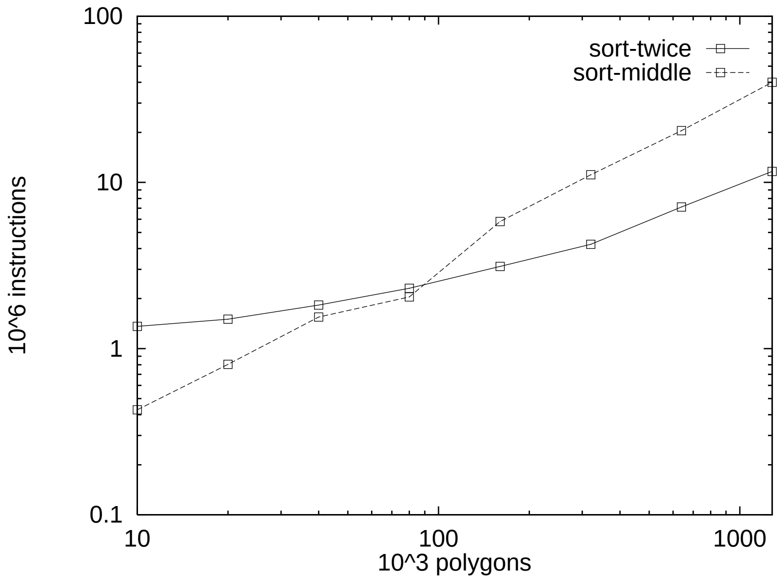



Figure 6-7: **Simulated Sort–Middle vs. Sort–Twice Execution Time:** instruction counts include geometry and rasterization stages for both approaches. All polygons are textured, and the sort–twice implementation uses deferred shading and texturing during the sort–last operation.

processors to rasterize the same region the amount of sort–middle communication required is minimized. The $n/g$ copies of each region of the screen are then composited after rasterization in sort–last fashion. The number of pixels to composite from each processor is $1/g$ of the total display pixels but they must be passed a distance $g$ during composition. The increased distance for each communication operation amortizes the overhead operations associated with communication, and substantially reduces the total time spent in sort–last communication.

The intelligent use of the communication instructions during sort–last composition allows the implementation of deferred shading and texture mapping at no cost. Rasterization can be made more than twice as fast as the non–deferred implementation provided with sort–middle schemes.

The sort–last composition still requires operations proportional to the number of pixels, and in the limit can execute in no fewer instructions than the total number of pixels in the display times the size of each pixel. In the limiting case a factor of 8 speedup is obtainable on the Princeton Engine over a normal sort–last implementation. Sort–twice communication creates an efficient sort–last implementation, not only in time and in memory, but also by admitting a more efficient rasterization scheme with fully deferred shading and texture mapping.

# Chapter 7

# Implementation

The previous chapters, while providing the background and analysis necessary to specify the system, have only hinted at the actual underlying implementation. This chapter will discuss the issues in the actual implementation of an interactive polygon rendering system on the Princeton Engine. I will primarily discuss efficiency issues and optimizations. The actual line–by–line details of the code are straightforward, and representative of the calculations performed in any number of polygon renderers.

The polygon renderer supports depth buffering, lighting, shading and texture mapping of triangular primitives. The rasterization stage has been implemented with a column–parallel decomposition of the screen, which readily admits the future inclusion of various optimizations of the communication structure, and in addition simplifies the rasterization process for each polygon. Figures 7-1, 7-2, 7-3, and 7-4 are representative output of the system, captured through a digital frame store attached to the Princeton Engine. The histogram across the bottom of the figures is linearly proportional to the number of polygons that intersect each column of the display.

There are a number of caveats to implementing polygon rendering on the Princeton Engine. First is the general issue of managing the implementation pipeline on a SIMD machine, discussed in §7.1, and second is the difficulty of dealing with the line–locked programming paradigm, discussed in §7.2. These constraints provide substantial motivation for our implementation, so we will discuss them first, and then

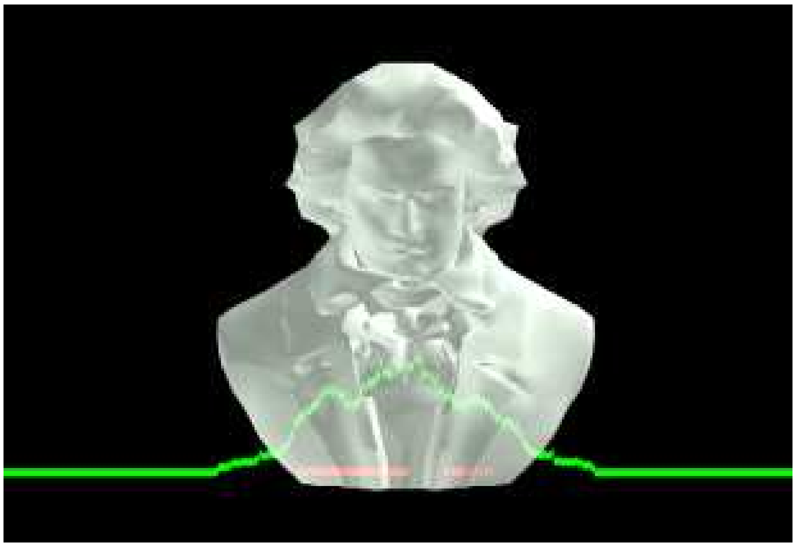


Figure 7-1: **Beethoven:** $\approx 3,700$ visible polygons in a 5,030 polygon model rendered at 10 frames per second. The histogram in green represents the number of polygon slices rasterized by each processor. The peak number of slices (at approximately the center of the figure) is 224. The short red bars note which processors had a time consuming rasterization load.

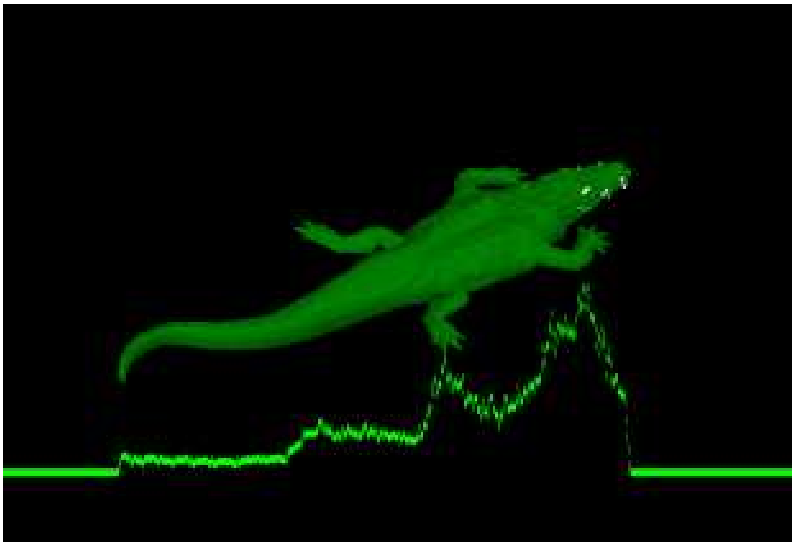


Figure 7-2: **Crocodile:** $\approx 10,600$ visible polygons in a 34,404 polygon model rendered at 6 frames per second. The peak rasterization load is 364 polygon slices.

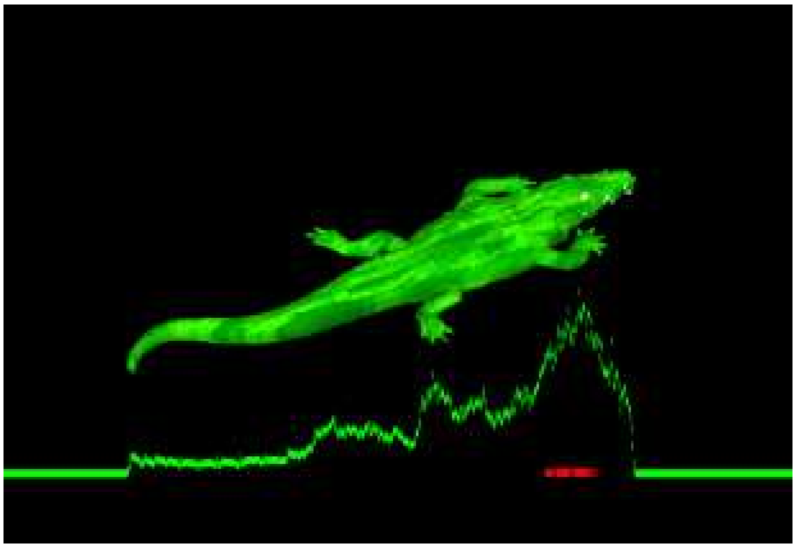


Figure 7-3: **Texture Mapped Crocodile:** $\approx 10,200$ visible polygons in a 34,404 polygon model rendered at 5 frames per second. The peak rasterization load is 370 polygon slices.

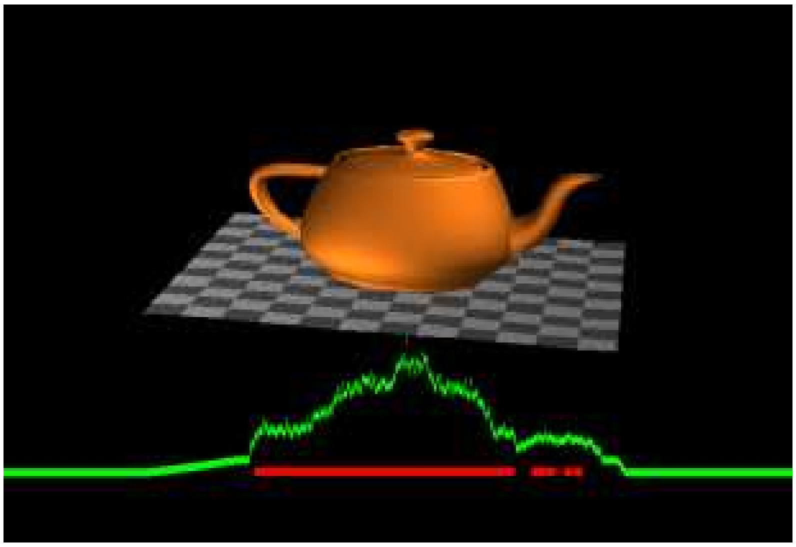


Figure 7-4: **Teapot:** $\approx 4,100$ visible polygons in a 9,408 polygon model rendered at 10 frames per second. The peak rasterization load is 260 polygon slices. Note the highlights from two light sources.

the actual implementation of the 3 pipeline stages: geometry, communication and rasterization.

## 7.1 Parallel Pipeline

The pipeline model of execution provides a great deal of parallelism, but lends itself to a direct implementation only where a true pipeline exists in hardware. The Princeton Engine, while highly parallel, is not a pipeline of processors, it is a vector of processors. To extract maximum efficiency from the system the pipeline algorithm must be carefully implemented.

A straightforward implementation would assign each of $n$ processors a polygon, perform all the geometry, communication and rasterization for those $n$ polygons, and then process the next $n$ polygons. Why is this not a good approach? The graphics pipeline *rejects* information as it proceeds. The first step in the geometry stage transforms and culls polygons. In general, some large fraction of the polygons we start with will almost immediately be thrown away, leaving large amounts of resources unused. Furthermore, different polygons will require differing amounts of time to rasterize, as a good implementation will make rasterization proceed in time proportional to the area of the primitive. If all processors rasterize at most a single polygon and then return to geometry computations the algorithm will execute in time proportional to the biggest polygon.

This motivates a rearrangement of the pipeline into a set of sequentially executed stages. An efficient implementation will first transform and test visibility for *all* polygons, then perform lighting, then clipping, etc. This serialization of the algorithm imposes no loss in performance, as clearly all of the same operations must be executed regardless of order. The only cost associated with this serialization is the storage necessary to buffer results between stages. By processing the entire polygon database at each stage before proceeding variations in load between processors exposed on a polygon by polygon basis should be minimized.

## 7.2 Line–Locked Programming

A polygon rendering implementation on the Princeton Engine faces a number of obstacles beyond extracting parallelism. Most challenging is the line–locked programming paradigm enforced by the hardware.

During each horizontal scanline the processors execute an instruction budget of 910 instructions, at the end of which the program counter is reset to the start of the program in preparation of the next line of video. After the overhead of clocking out video and other bookkeeping tasks the available program instruction budget is approximately 850 instructions per scanline.

The forced segmentation of the program into blocks of code 850 instructions long greatly increases the difficulty of extracting efficiency from the implementation. Code must not only be efficient in its utilization of processors, it must be efficient in its use of code blocks. Any code block less than 850 instructions in length wastes execution time needlessly.

A clever approach is taken to handling this constraint. The program is broken into a set of procedures, each of which executes in less than 850 instructions. At the beginning of each scanline the procedure addressed by a global function pointer is called. Each procedure is then responsible for setting this pointer to the address of the next procedure to execute. In large areas of straight–line code, such as during the geometry stage, each procedure just sets the pointer to the procedure which continues its execution, while in sections of code with a single tightly coded loop, such as rasterization, the procedure may leave the pointer unchanged until it detects completion by all processors, thus executing the same function for multiple scan–lines.

This method, while somewhat awkward, maps relatively easily to polygon rendering. For example, all of the state is already maintained in global variables to minimize the overhead of passing variables, so the function dispatch does not need to concern itself with parameters, and can be a very low overhead operation.

Of course, just performing the packing of instructions into instruction budget sized blocks is difficult, as every time the program is recompiled careful attention must be paid that the instruction budget has not been exceeded by any of the procedures. In some cases complete utilization of the instruction budget is impossible. For example, long division requires slightly more than 400 instructions to execute on the Princeton Engine, making it nearly impossible to execute more than a single long divide within the instruction budget, so sections of the geometry code which execute multiple long divides are often inefficient because they don't have enough extra work to pack into the left–over instructions. Unfortunately the compiler provides no direct support for these code packing operations, and the process is often an arduous trial and error effort. The Magic–1, our next generation hardware described in §9.3, eliminates the line–locked programming constraint.

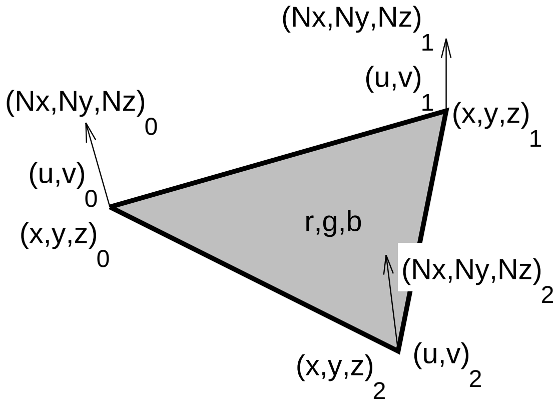
## 7.3 Geometry

The geometry stage provides the most obvious parallelism, as all visible (after rejection and clipping) polygons require exactly the same amount of work to compute. However, to extract efficiency we still have to break the pipeline into two pieces. First we transform all the polygons and test for visibility, generating a list of visible polygons on each processor, and then we complete the geometry processing on just the visible set of polygons. This two stage process allows us to efficiently handle a large set of polygons, hopefully many of which are not visible. Thus we can perform a cheap transform and test operation on all polygons, and defer the expensive work till later, when we will have to perform it on fewer polygons.

### 7.3.1 Representation

The polygon representation is shown in Figure 7-5. All polygons are triangles for simplicity. An all integer representation has been chosen for its greater efficiency than floating point.

Each vertex has associated with it a 3D location, $(x, y, z)$, a normal, $(n_x, n_y, n_z)$




**VERTEX:** (14 bytes)

| member | bytes |
|---|---|
| X | 2 |
| Y | 2 |
| Z | 2 |
| Nx | 2 |
| Ny | 2 |
| Nz | 2 |
| u | 1 |
| v | 1 |

**POLYGON:** (48 bytes)

| member | bytes |
|---|---|
| Vertex 0 | 14 |
| Vertex 1 | 14 |
| Vertex 2 | 14 |
| Red | 1 |
| Green | 1 |
| Blue | 1 |
| texture reference | 3 |

Figure 7-5: **Polygon Representation**

and a texture coordinate, $(u, v)$. The vertex coordinates are 16–bit integers. The vertex normals are 16–bit signed fixed–point with 14 fractional bits. The texture map coordinates are 8–bit unsigned integers, with 8 fractional bits, for the full range of $[0, 1)$ in texture coordinates. The decision was made not to support per vertex color to save space and simplify the lighting model. The polygon, in addition to 3 vertices, includes a color, $(r, g, b)$, and a reference to the texture to use, if applicable.

### 7.3.2 Transformation

The polygon data set is modeled as a single rigid object, so a single transformation matrix may be used for all polygons.

Transformation matrices are transmitted to the Princeton Engine by a control application running on a remote workstation. There is no direct method to input data in real time to the Princeton Engine, with the obvious exception of video information. A clever reuse of a HiPPi register allows transformation matrices to be sent to the engine at frame rates.

The transformation matrix is input as 12 integers, specifying the 9 elements of the rotation matrix with 14 fractional bits and the transformation entries as signed integers.

The use of 16–bit integers for both the vertex coordinates and the transformation matrix provides a significant advantage. The world to eyespace transformation may now be performed with 16–bit arithmetic which allows the fast (single clock cycle) single–precision data paths to be used throughout.

The transformation to screen–space requires 2 divides: $s_x = x/z$ and $s_y = y/z$. As already discussed, division is particularly expensive on this machine, so we make the optimization of performing division by table lookup. Division of a k–bit number by a k–bit number may always be performed by lookup in a $2^k$ entry k–bit table with no loss of precision. A $2^{15}$ entry lookup table of $1/z$ is provided on each processor. A full $2^{16}$ entries are not necessary as division is only done by positive $z$. While expensive in memory, this table will prove to be invaluable later for perspective correct texture mapping.

### 7.3.3 Visibility & Clipping

Visibility is performed as a set of screen–space tests. To be visible a polygon must lie in front of the display plane and the bounding box of the polygon must intersect the visible area of the display–plane. All other polygons are rejected as they must be non–visible.

All polygons are defined with vertices in counterclockwise order when facing the viewer, admitting a trivial backfacing test. The backfacing test is performed by evaluating one vertex of the triangle in the linear equation defined by the other two vertices. If the vertex lies in the negative half–plane the polygon is back–facing and is culled.

Clipping is not explicitly performed in this implementation, as our particular method for rasterization makes it unnecessary. The rasterizer never examines off–screen pixels, so the details of the polygon intersection with screen edges are unimportant.

The optimization discussed earlier of generating a visible list of polygons first and then completing the geometry processing only for the visible polygons is not implemented. For scenes of interest the geometry stage typically consumes less than 15% of the total instructions without this optimization.

### 7.3.4 Lighting

The implemented lighting model supports three light sources: an ambient, a directional, and a point light source. All three light sources are constrained to be white, and the polygon is constrained to be a single color, so that we can interpolate a single quantity, the white light intensity reflected from a face to the observer, rather than separate interpolations of the red, green and blue light reflected to the observer.

The ambient light controls the background level of illumination in the scene. Control is provided over the intensity of the light.

The directional light acts as a light source infinitely far away from the scene, so all light incident upon the scene has the same direction vector. Controls are provided to modify the direction and intensity of the light.

The point light source models a physical light placed somewhere in close proximity to the scene, so the direction of incident rays to a vertex depends on the position of the vertex relative to the light source. Controls are provided over the position and intensity of the light.

Separate controls are provided over the diffuse component of reflection and the specular component of reflection for the object.

The use of a lighting model with directional and point light sources also requires the ability to compute unit vectors from arbitrary vectors. A lookup table is used to perform the required inverse square root.

### 7.3.5 Coefficient Calculation

Calculation of coefficients is the most time consuming function of the geometry stage. The world to eye transformation, culling, clipping and lighting are all performed in less than 1500 instructions. The calculation of the coefficients of the linear equations to iterate require another 4500 instructions to execute.

Figure 3-9 shows the form of the equations to compute. The expense of these operations results from the necessity of performing two 32–bit divides for each parameter to interpolate. Note that the divisor $\Gamma$ is common to *all* the calculations (depth, intensity, texture coordinates), as it only depends on vertex screen coordinates, and not the parameter being interpolated.

One optimization that would provide a large performance improvement would be careful calculation of the inverse of the common denominator $\Gamma$ once into a 32–bit fixed point number. Subsequent divisions could be performed as a multiply and a shift. Unfortunately this requires a 32–bit multiply with a 64–bit result, an option not supported by the compiler, and deemed to troublesome to implement in light of the relatively small (less than 15%) fraction of time spent executing the geometry code.

The polygon descriptor generated as the final result of the geometry computations, shown in Figure 7-6, completely describes the polygon for rasterization. Communication, the next stage of operation, will transmit these descriptors between processors for rasterization.

## 7.4 Communication

The implementation uses sort–middle communication. All of the optimization analysis for sort–middle communication was based on the assumption of locality of destination for each polygon to be communicated. This directed our choice of screen decomposition for rasterization, as already suggested, to be column–parallel. By assigning each processor a single column of pixels on the display to rasterize we leave

**POLYGON DESCRIPTOR:**
(84 bytes)

| linear equation | bytes | support information | bytes |
|---|---|---|---|
| edge0 | 8 | texture reference | 3 |
| edge1 | 8 | r,g,b | 3 |
| edge2 | 8 | left | 2 |
| zdepth | 12 | right | 2 |
| intensity | 8 | top | 2 |
| texture u | 12 | bottom | 2 |
| texture v | 12 | # passes | 2 |

**LINEAR EQUATION:**

(8 bytes)

| coefficient | bytes |
|---|---|
| A | 2 |
| B | 2 |
| C | 4 |

-or-

(12 bytes)

| coefficient | bytes |
|---|---|
| A | 4 |
| B | 4 |
| C | 4 |

Figure 7-6: **Polygon Descriptor:** result of all of the geometry computations for a single polygon.

a quarter of the processors idle during rasterization (the display is only 768 columns wide) but we gain a great deal of locality in polygon destination. All processors rasterizing a polygon are guaranteed to be contiguous, so our previous analyses which relied on the assumption of a single destination processor for communication are approximately correct, and our optimizations will prove effective.

The use of the "distribution" scheme for decreasing communication time was unfortunately not implemented. The consideration and analysis of this possibility did not arise until late in the implementation, preventing its inclusion. However, analysis shows that it happens to map to the line–locked programming paradigm very efficiently. Passing a single polygon from neighbor to neighbor would require $(2+d)\cdot a+b$ operations, where $a = 42$ (a 42 integer polygon descriptor), $b = 30$ (the test overhead) and $d = 16$ optimally, for a total of 786 instructions. 786 instructions fit comfortably within the budget of 850 instructions, and allows time for a global–if to detect completion of the first pass of the distribution algorithm.

A description of the mechanics for performing neighbor–to–neighbor communication of the polygon descriptors is now presented. Neighbor–to–neighbor–N for the distribution optimization is an obvious extension, simply using large pass sizes.

### 7.4.1 Passing a Single Polygon Descriptor

The most essential communication operation is the neighbor–to–neighbor pass. Arbitrary communication can then be performed with repeated neighbor–to–neighbor passes. Passing a polygon descriptor (represented as 42 16–bit integers) from one processor to its neighbor is a loop over the descriptor, passing a 16–bit piece of it at a time:

```
void pass( int *sourceDescriptor, int *destDescriptor )
{
    int counter;

    for ( counter = 0; counter < 42; counter++ ) {
        communicationRegister = sourceDatum[counter];
        passRight(); /* or passLeft(); */
        destDatum[counter] = communicationRegister;
    }
}
```

The `passRight()` (or `passLeft()`) operation is a compiler primitive that shifts the datum in each processor's communication register one processor to the right (or left). Each processor is performing the same action, so each processor is simultaneously sourcing and receiving a polygon descriptor. The loop is completely unrolled and coded in assembly language for efficiency.

### 7.4.2 Passing Multiple Descriptors

In general each processor will have more than a single descriptor to distribute, and more than a single descriptor will be destined for this processor. It is the responsibility of each processor to store this data as it arrives.

Each processor manages a `source` array of data and a `receive` array of data. The `source` array contains all of the descriptors that this processor will communicate to the other processors. The `receive` array contains, at the end of the communication, all of the descriptors that the other processors communicated to this processor.

```
void distribute( int n, DESCRIPTOR source[], DESCRIPTOR receive[] )
{
    DESCRIPTOR *s, *r;

    s = source;
    r = receive;

    while( globalAnd( n > 0 ) ) {
        pass(s,r);
        s = r;
        if (  GOOD(r) ) r++;
        if ( !INTERESTING(s) && (n-- > 0))
            s = ++source;
    }
}
```

Figure 7-7: **Implementation for Data Distribution**

The variable `s` points to the next descriptor to pass, while `r` points to the location to store the next descriptor received. `s` ping–pongs back and forth between the source array and receive array. Initially each processors sources a datum from its `source` array, and receive a datum into its `receive` array. When a processor receives a descriptor it immediately points `s` at it, the assumption being that it will probably have to pass this descriptor further along, as it is unlikely this processor is the last processor that needs to see this polygon. This store and forward operation insures that each polygon descriptor travels around the ring far enough to be seen by all processors rasterizing it. Note that whatever descriptor `s` was pointing at it was just passed to this processors neighbor, which is now responsible for it. Thus the automatic `s=r` operation forgets which descriptor this processor just sourced, but it simultaneously becomes another processors responsibility.

If the descriptor received is `GOOD` (a descriptor for a polygon that this processor will be in part responsible for rasterizing) the processor increments the receive pointer `r` so that it doesn't overwrite it with the next polygon received. Otherwise `r` is left alone, and the next descriptor received will overwrite it. This can cause (and generally does) a situation where `s == r`, and the processor just continually forwards polygons for other processors.

If the descriptor received by a processor on any given cycle is not `INTERESTING` then it has already been communicated to all processors that need to see it. The processor will then source one of its remaining `source` polygons if it has any left. If the processor has no more descriptors to contribute it will simply pass off this "uninteresting" polygon to the next processor, which may or may not choose to replace it with a polygon of its own, etc.

The `while` loop terminates when no processor is still communicating data of interest. If no one is communicating data of interest then the communication must be complete, and the function terminates. This has to be done via a global test as it isn't known a priori how long it will take to communicate a given number of polygons[1].

This implementation avoids as many copy operations as possible, as a copy will require $a$ reads and $a$ writes ($a$ is the descriptor size), which is almost as expensive as the communication operation itself. This is particularly important for communication, as it is generally the case that on a given cycle only some small percentage of the processors will be handling data they need a copy of, and the rest will be just forwarding data intended for others. Any conditional operations (such as a conditional copy) would be particularly expensive, as only a small fraction of the machine would be utilized.

When communication has completed each processor has a copy of all the descriptors for the polygons it will be responsible for rasterizing, and the algorithm proceeds directly to rasterization.

[1]The actual implementation optimizes to perform this test periodically by making an optimistic guess of how many iterations of the loop the remaining descriptors will take to communicate, and after that many iterations a new estimate is formed, etc. The estimates will decrease monotonically (because they are based on the remaining number of polygons each processor has to communicate, which is decreasing if progress is being made) and the frequency of completeness checks will therefore increase until actual completion. With a little care the number of tests can be minimized without performing more communication than necessary.

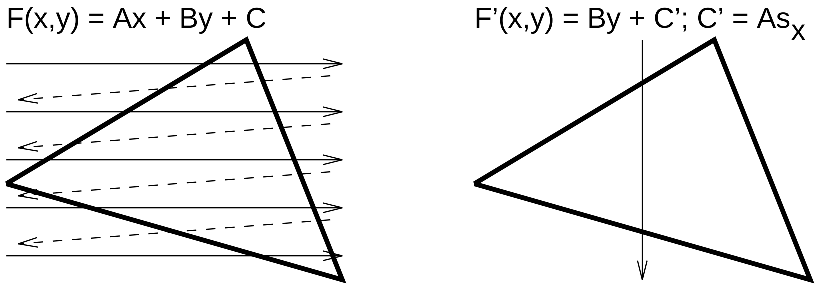



Figure 7-8: **Linear Equation Normalization:** linear equations are normalized to the horizontal position of the processor in screen space.

## 7.5 Rasterization

After communication each processor has a copy in the `receive` array of all polygon descriptors that intersect its rasterization region. A column–parallel decomposition of the screen has been implemented, so each processor is responsible for rasterizing a vertical slice of each of these polygons into its single column of the framebuffer.

Rasterization is broken into a 2–step process in the implementation:

- Normalize linear equations to processor column position
- Rasterize polygons

### 7.5.1 Normalize Linear Equations

The equations that reside on every processor as part of the polygon descriptors are of the form $F(x, y) = Ax + By + C$. We have implemented a column–parallel decomposition of the screen, so each processor is responsible for rasterizing a single column of pixels; thus each processor only needs to iterate the linear equations vertically. The normalization $C' = C + As_x$ is made for each equation, where $s_x$ is the horizontal position of this processor's column on the display. These normalizations require approximately 350 instructions per polygon and are applied to the `receive` polygon array on each processor immediately after the completion of the communication stage.

Now simpler equations of the form $F'(x, y) = By + C'$ can be evaluated on each processor, as shown in Figure 7-8. Of course the equations are evaluated iteratively, so the actual evaluation process is $F(x, y) = F(x, y - 1) + B$ for each vertical step.

Rasterization begins once all the processors have normalized their linear equations for all received polygons.

### 7.5.2 Rasterize Polygons

Polygon rasterization, discussed in section 3.2.3, is implemented by iterating a set of linear equations over the bounding box of a polygon. In this case we only iterate the equations over a vertical span of the polygon due to the column–parallel decomposition of the screen. To rasterize a single pixel we compare the currently interpolated depth for the polygon to the corresponding pixel in the depth–buffer. If the current pixel occludes the pixel in the depth–buffer we then shade and texture map the pixel and place it in the framebuffer and its depth value in the depth–buffer. Thus we have an occlusion model that is accurate on a pixel–by–pixel basis.

There are a number of issues and optimizations which were tackled in the implementation. The difficult implementation issues will be discussed first, and the optimizations will be discussed at the end of the section.

#### Decoupling Rasterization and Polygon Size

The most important issue (although arguably an optimization) in the rasterizer is allowing processors to *independently* advance to their next polygon after completing rasterization of their current polygon. This decouples the rasterization process from size differences in the particular polygons being rasterized on each processor at any point in time. Without this decoupling each processor would rasterize its first polygon, wait until all other processors were done, and then all processors would simultaneously proceed to their second polygon, etc. forcing all processors to rasterize their polygon in time proportional to the size of the worst case polygon.

The decoupling is achieved by performing a periodic test during pixel rasterization to allow processors to move on to their next polygon. Every $k$–pixels all processors test if they have finished rasterizing their current polygon, and those that have update the equations they are iterating with the coefficients of their next polygon to rasterize. This test and conditional advancement is expensive because it requires copying indirectly addressed data into registers. Furthermore, the line–locked programming paradigm makes some choices of how often to perform the test (the value of $k$) more efficient than other, in terms of total instructions executed every during each scanline. The shaded polygon rasterizer performs one test and eight pixel rasterizations ($k = 8$) per scanline, and the full texture mapping rasterizer performs one test and four pixel rasterizations per scanline.

This optimization is similar to those implemented by Crow [6] and Whitman [20]. Crow and Whitman both use a similar test to decouple the scanline of the polygon each processor is working on, but still force synchronization at the polygon level. This suggests they only gain efficiency where polygon are similar in area, but have varying aspect ratios. Our approach is insensitive to polygon aspect ratios (because each processor only rasterizes a single column of any polygon) *and* to polygon area, which would seem to offer a more substantial advantage.

### Perspective–Correct Texture Mapping

Our model of textured polygons associates a texture coordinate $(u, v)$ with each of the vertices of the polygon, and the texture coordinate of any point within the polygon may be found as the linear combination of these vertex values. However, this is linear interpolation in *object* space, but not in *screen space.* The general form of the interpolation should be

$$u = Ax + By + C \tag{7.1}$$

$$v = Dx + Ey + F \tag{7.2}$$

in object space, but we interpolate all our parameters in screen space.

Naive screen space interpolation of quantities nonlinear in screen space leads to distracting artifacts, perhaps most noticeable in texture mapping. The errors made destroy the foreshortening effect associated with objects twisted away from the observer. If we carefully perform only interpolations of quantities that are linear in screen space we can create a *perspective correct* renderer.

Examining our perspective transformation, $s_x = x/z$ and $s_y = y/z$, we see we can express these equations as

$$u = z(As_x + Bs_y + C) \tag{7.3}$$

$$v = z(Ds_x + Es_y + F) \tag{7.4}$$

so linear interpolation of $u/z$ and $v/z$ can be correctly performed in screen space, and the value of $u$ and $v$ recovered at each pixel by multiplication with $z$. However, multiplication by $z$ reintroduces the problem of screen space linearity.

If we examine the equation of a plane in 3D, and thus the equation of depth, we see that like texture coordinates, depth is not linear in screen space, the *inverse* depth is linear in screen space. However, we can easily interpolate inverse $z$ instead of $z$, and still use depth–buffering. Our compositing test for $1/z$ values is simply the complement of the test for $z$.

Now we can trivially interpolate $u/z$, $v/z$ and $1/z$ for each polygon, and texture mapping will require the calculation

$$u = \frac{u/z}{1/z} \tag{7.5}$$

$$v = \frac{v/z}{1/z} \tag{7.6}$$

at each pixel. We reuse the inverse $z$ table used to perform the perspective division, avoiding the very high cost of performing two divides per iteration of the rasterization loop.

Heckbert and Moreton provide a comprehensive treatment of the issue of object–linear vs. screen–linear quantities in [9].

#### Bilinear Interpolation

Texture mapping is muddied further beyond perspective correction by the necessity of performing sampling. A simple texture mapping algorithm performs point–sampling of the texture image, taking the color of the polygon pixel as the texel closest to the current interpolated $(u, v)$ coordinate. Unfortunately small motions of objects can induce jittering in the sampling of the texels and create unpleasant artifacts under point sampling. Bilinear interpolation helps to reduce these effects by computing a pixel color as the weighted average of the 4 texels nearest to the texture coordinate.

On many architectures this provides a substantial performance penalty, as it requires performing 4 texel fetches for every texture mapped pixel. However the Princeton Engine has enormous aggregate memory bandwidth, and in fact there is a copy of all of the textures on every processor, so the additional texel fetches are a trivial expense.

### 7.5.3 Rasterization Optimizations

#### Distinct Texture–Mapped and Non–Texture–Mapped Algorithms

The inclusion of texture mapping adds significantly to the cost of rasterization. A texture mapped pixel requires approximately twice as long to rasterize as a pixel without texture mapping. Many scenes of interest have no texture mapping at all, so a control is provided in the interface to disable texture mapping which results in the use of an optimized shading–only rasterizer.

#### Early Abort

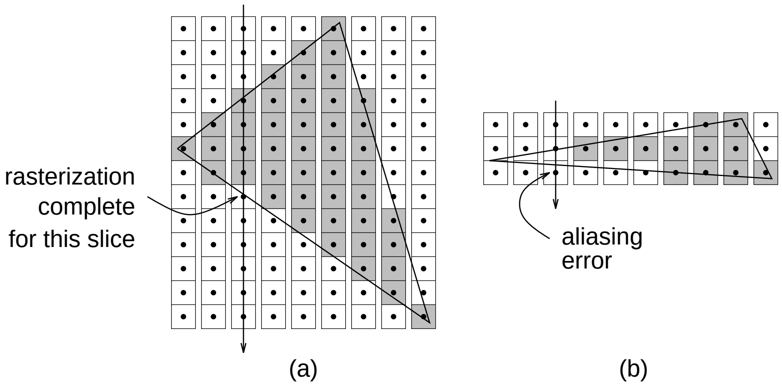



Figure 7-9: **Rasterizing:** triangles occupy less than half of the pixel covered by their bounding boxes. In both cases significant advantage can be made of noting when rasterization exits the polygon. Case (b) adds the complexity that a careless rasterizer will step over the polygon due to integer pixel coordinates and rasterize forever, waiting to enter the polygon.

Observe that when rasterizing polygons, shown in Figure 7-9, that *at least* one half of the area in the bounding box of a triangle is *not* within the triangle. It would be advantageous to avoid examining all of these pixels needlessly. We include a simple test which on average avoids rasterizing 50% of these "exterior" pixels. As rasterization proceeds, a flag is set when the polygon is entered. Because triangles are necessarily convex, as soon as one of the edge equations test negative *and* the flag is set, the rasterizer knows that it has completely rendered its piece of this polygon and rasterization of the next polygon can begin.

### 7.5.4 Summary

This section has described the major implementation issues and optimizations in the rasterizer. Particular attention has been paid to the decoupling of the polygon being processed from rasterization. In addition perspective–correct texture mapping, requiring an expensive two divides per pixel, has been efficiently implemented with a table lookup.

The use of separate rasterization algorithms with and without texture mapping support provides a substantial performance gain for scenes without texturing. An early abort mechanism allows rasterization to detect completion of a polygon before the entire vertical extent of the bounding box has been examined, and provides a significant reduction in the total pixels examined to rasterize a given polygon.

Thus far the geometry, communication and rasterization stages have been described. The next section will discuss the final issue: the actual control of the renderer.

## 7.6 Control

A virtual trackball is used to interact with rendering software. A workstation is used to display a "control cube", shown in Figure 7-10, which represents the orientation and translation of the object rendered by the Princeton Engine. The cube may be grabbed with the mouse and rotated and translated arbitrarily. The user may also set the cube spinning about an arbitrary axis of rotation. Quaternions, described in [17], are used to interpolate between successive positions of the rotating control cube. The GL modelview matrix [14] corresponding to the orientation of the cube is transmitted to the Princeton Engine continually via an auxiliary data channel. The renderer may be operating at less than the matrix update rate, in which case some of the updates are ignored, so while the update rate of the engine display may not match the virtual trackball, the rate of rotation and translation of the rendered object does.

Clever reuse of a communication register used to support a HiPPi[2] interface allows a single integer to be input to all processors per scanline. We can input $525 \cdot 30 = 15750$ integers a second to the engine through this channel. A low data rate, but more than adequate for simple updates, such as the transformation matrix. The bandwidth into the engine through this "auxiliary" channel is obviously substantially larger than that required for simple matrix updates, and allows the future possibility of sending extra information through this channel, such as the positions and properties of light sources

[2]Unfortunately use of the actual HiPPi channel requires reprogramming the IO circuitry of the engine and can't be used concurrently with video output.

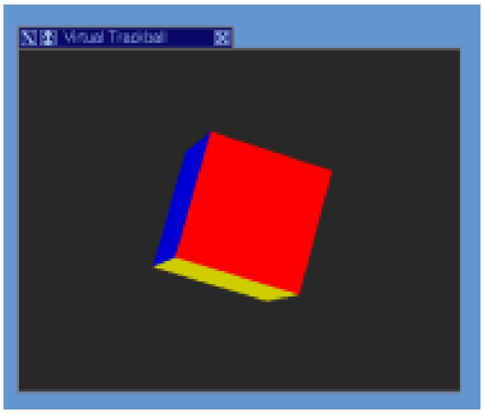



Figure 7-10: **Virtual Trackball Interface**

within the world, or simulation data.

A full handshaking protocol is used between the workstation and the engine, which insures perfect synchronization. The channel is also robust and readily handles dropped, scrambled and delayed data, so that the occasional error is gracefully handled. A serial interface between the workstation running the virtual trackball software and the Princeton Engine controller is used to avoid control difficulties due to variations in ethernet traffic.

## 7.7 Summary

This chapter has discussed the implementation of a column–parallel polygon renderer on the Princeton Engine. The polygon renderer supports depth buffering, lighting, shading and texture mapping of triangular primitives.

An explanation of the geometry stage has been provided, including the details of the representation of the polygon database and the form of the calculations performed. The inclusion of a per processor inverse $z$ table supports efficient perspective divisions, and is reused later to support efficient perspective–correct texture mapping.

The communication stage has been carefully described, as it dominates the execution time of the algorithm, and its efficiency bears heavily on the overall efficiency of the program. The details of queueing polygon descriptors for transmission and receipt is explained, and a careful explanation of the actual programmed managed communication algorithm is given.

Significant issues and optimization of the rasterization stage, second only to communication in total execution time, have been examined. Decoupling of the polygon size dependence in rasterization in conjunction with an early abort mechanism affords an efficient implementation. The use of an inverse $z$ lookup table provides visually pleasing perspective correct texture mapping cheaply.

An intuitive virtual trackball interface provides real–time user interaction and control of the rendered scene.

Chapter 8 will provide performance results of this implementation and a comparison with the performance of a Princeton Engine contemporary, the Silicon Graphics GTX.

# Chapter 8

# Results

The polygon renderer has achieved peak performance of over 60,000 visible, shaded and bilinearly interpolated texture mapped polygons per second. There are a number of per polygon and per pixel performance figures that combine to yield the final aggregate performance of the system. This chapter will first provide the raw performance of each operation, and then an accounting of how these instructions are spent per visible polygon rendered. Finally a comparison will be made between the Princeton Engine performance and that of its contemporary, the Silicon Graphics GTX.

## 8.1 Raw Performance

The raw performance of each of the individual stages of the implementation is readily quantified. Unfortunately the line–locked programming paradigm of the the Princeton Engine often obscures the true performance of the algorithm. Where feasible figures are provided both for the actual implementation and for a hypothetical Princeton Engine lacking the line–locked constraint.

The line–locked programming constraint makes it most natural to express operations as the number of operations achieved per line time, where a line time is 910 instructions, and includes the framebuffer management code and any unused cycles.

NTSC video operates at 30 frames per second[1] and 525 lines per frame, for a total of 15750 lines per second.

### 8.1.1 Geometry

Transforming to screen coordinates, lighting, and computing linear equation coefficients is very fast:

$$1024 \text{ PROCS} * \frac{1}{7}\frac{\text{POLYGON}}{\text{PROC} \cdot \text{LINES}} * 15750\frac{\text{LINES}}{\text{SECOND}} = \mathbf{2,304,000}\frac{\text{SHADED POLYGONS}}{\text{SECOND}} \tag{8.1}$$

with texture mapping (which requires the calculation of linear equations for the additional parameters $u$ and $v$):

$$1024 \text{ PROCS} * \frac{1}{11}\frac{\text{POLYGON}}{\text{PROC} \cdot \text{LINES}} * 15750\frac{\text{LINES}}{\text{SECOND}} = \mathbf{1,466,182}\frac{\text{TEXTURED POLYGONS}}{\text{SECOND}} \tag{8.2}$$

### 8.1.2 Communication

Characterizing the communication performance is difficult, as the distribution of polygons and average distance they have to travel will affect the efficiency of the shift ring. A useful benchmark is the number of polygon passes (proc $n \rightarrow$ proc $n+1$) per second:

$$1024 \text{ PROCS} * 5\frac{\text{PASSES}}{\text{PROC} \cdot \text{LINE}} * 15750\frac{\text{LINES}}{\text{SECOND}} = \mathbf{80,640,000}\frac{\text{POLYGON PASSES}}{\text{SECOND}} \tag{8.3}$$

If each polygon requires approximately 512 passes (travels halfway around the ring to reach its destination) then the polygons communicated per second is

[1] NTSC video, as defined by the SMPTE draft standard actually operates at 59.94005994 fields per second, or approximately 29.97 frames per second.

$$80,640,000\frac{\text{PASSES}}{\text{SECOND}} * \frac{1}{512}\frac{\text{POLYGON}}{\text{PASSES}} = \mathbf{157,500}\frac{\text{POLYGONS}}{\text{SECOND}} \tag{8.4}$$

### 8.1.3 Rasterization

Performance remains largely communication limited, which is independent of polygon size[2]. This makes the usual "100–pixel polygon" a less useful benchmark, as it stresses the rasterizing performance excessively over the communication performance. More relevant is pixels per second, as this yields insight to the average depth complexity that can be handled. An aggregate rate of almost 50 million bilinearly interpolated, Gouraud shaded texture mapped pixels is attained. Without texture mapping, almost 100 million pixels can be rendered per second. A second of NTSC video is $768 \times 525 \times 30 = 12,096,000$ pixels, or less than one eighth of the peak pixel fill rate of this implementation.

If we consider texture–mapped (bilinearly interpolated), Gouraud–shaded pixels per second, we can characterize the resulting performance as[3]:

$$768 \text{ PROCS} * 4\frac{\text{PIXELS}}{\text{LINE} \cdot \text{PROC}} * 15750\frac{\text{LINES}}{\text{SECOND}} = \mathbf{48,384,000}\frac{\text{PIXELS}}{\text{SECOND}} \tag{8.5}$$

Without the bilinear interpolation necessary for texture mapping, the pixel fill rate doubles:

$$768 \text{ PROCS} * 8\frac{\text{PIXELS}}{\text{LINE} \cdot \text{PROC}} * 15750\frac{\text{LINES}}{\text{SECOND}} = \mathbf{96,768,000}\frac{\text{PIXELS}}{\text{SECOND}} \tag{8.6}$$

Without the line–locked programming paradigm, and ignoring the overhead to

[2]This is true to first order. Of course larger polygons will have to stay in the ring longer before everyone has seen them, but generally the size of a polygon is small compared to the distance it must travel.

[3]768 processors are specified, as only the rendering of on–screen processors is used in this algorithm.

change active polygons, the rasterizer requires 202 instructions per shaded texture mapped pixel, for an aggregate fill rate with a fully distributed framebuffer (all 1024 processors) of:

$$1024 \text{ PROCS} * 14 \cdot 10^6 \frac{\text{INST}}{\text{SEC} \cdot \text{PROC}} * \frac{1}{202} \frac{\text{PIXEL}}{\text{INST}} = \mathbf{70,970,297} \frac{\text{PIXELS}}{\text{SECOND}} \tag{8.7}$$

Shaded pixels only require 80 instructions per pixel, for an aggregate fill rate of:

$$1024 \text{ PROCS} * 14 \cdot 10^6 \frac{\text{INST}}{\text{SEC} \cdot \text{PROC}} * \frac{1}{80} \frac{\text{PIXEL}}{\text{INST}} = \mathbf{179,200,000} \frac{\text{PIXELS}}{\text{SECOND}} \tag{8.8}$$

The full pixel rendering rate is *never* realized because we perform a bounding box scan of the triangular primitives, which guarantees we will test (and not render) some non–primitive pixels. We also perform a periodic test (not after every pixel) to conditionally advance to the next polygon, which while improving overall performance, also increases the number of pixels we will examine for each primitive necessarily. If we approximate the loss in efficiency due to the pixels examined outside of the polygon and the granularity in the rasterization algorithm, the rasterizer achieves 50% parallel efficiency. Assuming uniform load balancing, the renderer can rasterize:

$$48,384,000 \frac{\text{PIXELS}}{\text{SECOND}} * 0.50 * \frac{1}{100} \frac{\text{POLYGON}}{\text{PIXELS}} = \mathbf{241,920} \frac{\text{100–PIXEL POLYGONS}}{\text{SECOND}} \tag{8.9}$$

without texture mapping:

$$96,768,000 \frac{\text{PIXELS}}{\text{SECOND}} * 0.50 * \frac{1}{100} \frac{\text{POLYGON}}{\text{PIXELS}} = \mathbf{483,840} \frac{\text{100–PIXEL POLYGONS}}{\text{SECOND}} \tag{8.10}$$

### 8.1.4 Aggregate Performance

| | | | percent of time in each phase | | |
|---|---|---|---|---|---|
| object | frames·s$^{-1}$ | polys·s$^{-1}$ | geometry | communicate | rasterize |
| beethoven | 10.0 | 36,000 | 9 | 46 | 45 |
| | 10.0 | 32,000 | 8 | 42 | 50 |
| | 10.0 | 32,000 | 9 | 44 | 47 |
| crocodile | 6.0 | 56,400 | 13 | 61 | 26 |
| | 6.0 | 49,800 | 13 | 57 | 30 |
| | 6.0 | 62,400 | 13 | 69 | 18 |
| teapot | 10.0 | 44,000 | 12 | 52 | 36 |
| | 10.0 | 43,000 | 10 | 54 | 36 |
| | 10.0 | 44,000 | 10 | 55 | 35 |

Table 8.1: **Rendering Performance for Typical Scenes:** Rendered scenes always count an integral number of frame times to compute, as framebuffers can only be swapped at frame boundaries. Any "excess" time in the frame spent idling is counted as render time, causing an unfairly high estimate of render time versus computation and distribution.

The actual performance of the renderer is measured by rendering the scenes shown in figures 7-1, 7-2 and 7-4. Typical results are given in Table 8.1. Multiple entries for the same object are at different orientations, as communication and rendering cost will depend on orientation. Figure 8-1 shows the breakdown of rendering time between geometry, communication and rasterization.

## 8.2 Accounting

The Princeton Engine provides an aggregate 14BIPs of computing performance. At 60,000 polygons a second this implies approximately 233,000 instructions per rendered polygon. This cost seems excessive, but we can account for where all the instructions have gone. Examining the scene with our peak performance, the texture mapped crocodile, we tally the instruction usage.

Our scene has 34,404 polygons, 10,400 of which are visible at the given orientation. The scene is rendered at 6fps, for an aggregate performance of 62,400 polygons per second. We will express times for operation in terms of scanlines, because all functionality is implemented in scanline sized procedure blocks.

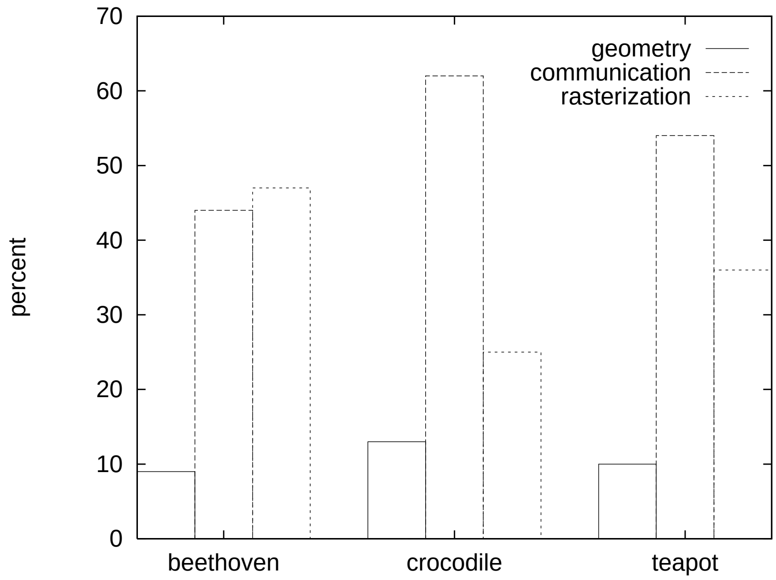



Figure 8-1: **Execution Profile:** the communication time for most scenes is the dominant performance factor. As the number of polygons increase (and their size decreases) communication time becomes even more dominant.

The geometry stage is not optimized to perform a two–stage pipeline, so all polygons have to be fully processed. A texture–mapped polygon requires 11 scanlines to compute, for a total of

$$910 \frac{\text{INST}}{\text{LINE}} \cdot 11 \frac{\text{LINES}}{\text{POLYGON}} = 10,010 \frac{\text{INST}}{\text{POLYGON}} \tag{8.11}$$

there are a total of 34,404 polygons, for

$$34,404 \frac{\text{POLYGONS}}{\text{FRAME}} \cdot 10,010 \frac{\text{INST}}{\text{POLYGON}} \cdot 6 \frac{\text{FRAMES}}{\text{SECOND}} = 2.07 \cdot 10^9 \frac{\text{INST}}{\text{SECOND}} \tag{8.12}$$

We'll count the communication stage a bit differently. A single pass consumes MIPs across all of the processors and we can perform 5 passes in a scanline, for

$$910 \frac{\text{INST}}{\text{LINE} \cdot \text{PROC}} \cdot \frac{1}{5} \frac{\text{LINES}}{\text{PASS}} \cdot 1024 \text{PROCS} = 186,368 \frac{\text{INST}}{\text{PASS}} \tag{8.13}$$

Each polygon has a marginal pass cost (total number of passes divided by the number of polygons) of 0.6, for a total of

$$186,368 \frac{\text{INST}}{\text{PASS}} \cdot 0.6 \frac{\text{PASS}}{\text{POLYGON}} = 111,820 \frac{\text{INST}}{\text{POLYGON}} \tag{8.14}$$

The communication stage thus requires

$$10,400 \frac{\text{POLYGONS}}{\text{FRAME}} \cdot 111,820 \frac{\text{INST}}{\text{POLYGON}} \cdot 6 \frac{\text{FRAMES}}{\text{SECOND}} = 6.98 \cdot 10^9 \frac{\text{INST}}{\text{SECOND}} \tag{8.15}$$

By the time we reach the rasterization stage we have hit a large load imbalance, due to the asymmetric distribution of scene content. If we examine Figure 7-2 and note the histogram we see that the peak processor has 360 polygons intersecting its column. The average polygon height is 3.8 pixels (obtained from the simulator) and our texture mapping code forces polygons to rasterize in 4 pixel chunks, so each polygon will have an effective average height of 4 pixels. 2 polygons can be normalized in a scanline, and 4 pixels can be rasterized in a scanline, so the instructions required to process a single polygon are

$$910 \frac{\text{INST}}{\text{LINE}} \cdot \left(1 \frac{\text{NORMALIZE}}{\text{POLYGON}} \cdot \frac{1}{2} \frac{\text{LINE}}{\text{NORMALIZE}} + 4 \frac{\text{PIXELS}}{\text{POLYGON}} \cdot \frac{1}{4} \frac{\text{LINE}}{\text{PIXEL}}\right) = 1,365 \frac{\text{INST}}{\text{POLYGON}} \tag{8.16}$$

and our worst–case processor has 460 polygon slices to rasterize, for a total of

$$360 \frac{\text{POLYGONS}}{\text{PROC}} \cdot 1365 \frac{\text{INST}}{\text{POLYGON}} \cdot 1024\text{PROCS} \cdot 6 \frac{\text{FRAMES}}{\text{SECOND}} = 3.02 \cdot 10^9 \frac{\text{INST}}{\text{SECOND}} \tag{8.17}$$

Our total instructions for rendering this scene are thus $12.07 \cdot 10^9$ instructions per second, which very closely matches the 14BIP capability of the Princeton Engine. The extra 2BIPs (a non–trivial number of instructions) is accounted for in a number of places. There are a number of procedures that are called a single time, for example, to place the histogram on the image, or to initialize the database for rendering. In addition any cycles between the end rasterization and the start of the next frame (when the framebuffers can be swapped) are missing from this analysis. They have been counted by the renderer as rasterization instructions in Figure 8-1.

## 8.3 Performance Comparison

The performance of Princeton Engine polygon renderer is compared to that of that of its contemporary, the Silicon Graphics GTX, described in [2].

The comparison is a bit stilted of course. The GTX represents the state of the art in special purpose polygon rendering hardware for 1988. Likewise, the Princeton Engine, although never sold in volume, is a million dollar machine. Nonetheless they are comparable on a number of levels. Both machines exhibit a high level of parallelism, and are implemented in the technology era, so clock speeds may be expected to be similar, etc.

The geometry stage of the GTX is composed of 5 high performance "geometry engines" connected in a pipeline, with each processor handling a specific aspect of the geometry calculations. Together they provide an aggregate performance of 100 million floating point operations per second (Mflops). By comparison the Princeton Engine can perform a floating point multiply and divide in a single line time on each processor, for an aggregate performance of approximately 32Mflops. The Princeton Engine was not designed to perform floating point operations, and consequently pays a premium for their use.

By direct comparison both of these systems are implementing a geometry pipeline. The Princeton Engine implementation sacrifices dynamic range and uses an all integer representation for its primitives and can consequently compute approximately 2 million triangular primitives per second. The GTX performs (with substantially fewer processors of course) 137,000 triangular primitives per second.

It is hard to quantify the analog of the Princeton Engine communication stage on the GTX. The GTX has a dedicated high speed bus which transports primitives between the geometry pipeline and their final destination at the image engines. The bus was specifically designed to accept primitives at the full rate the geometry stage generates them, and will thus never be a performance limiting factor.

The rasterization stage requires some guessing to determine the performance of the individual image engines of the GTX. They are cited as having an aggregate performance of 40 million depth–buffered pixels per second, which is less than half of our pixel fill rate. Assuming each image engine must perform on the order of 20 integer operations per depth–buffered pixel the aggregate performance of the image engines is 800MIPs, or 0.8BIPs, slightly less than 5% of the Princeton Engine performance. Of course the Princeton Engine spends approximately 115 instructions per pixel, so they aren't directly comparable.

Curiously, the GTX has realtime video input and output capability, making it a particularly apt comparison to the Princeton Engine. Although not part of the polygon rendering problem, the performance of the Princeton Engine doubtless exceeds the GTX for video related operation.

Some features have no comparison. Most notably, the Princeton Engine renderer performs full bilinear–interpolated texture mapping, which the GTX offers no support for. In addition the Princeton Engine is a software solution, and will admit many extensions that are simply impossible in a hardware solution.

The final and most telling comparison is of course the aggregate performance. The GTX can render 137,000 connected (shared vertex) triangles per second, compared to the Princeton Engine implementation, which achieves a peak of 62,000 texture mapped triangles per second. Given the luxury of costless communication, as provided in the GTX, the Princeton Engine would immediately double its performance and become directly comparable to the GTX.

In 1988, the time of the Princeton Engine's first operation, it would have provided polygon rendering directly comparable to high end graphics workstations of the era. More significantly it offers an unparalleled flexibility of operation through an all software implementation.

# Chapter 9

# Future Directions

Due to time and resource constraints a couple of ideas in this thesis have remained unimplemented. Future work will address these issues when possible.

First and foremost is the implementation of the "distribution" optimization, expected to double the sort–middle communication optimization of the current implementation. The other issue to be addressed is an implementation of sort–twice communication. Both of these issues must also be examined in light of the next generation Princeton Engine, just becoming operational at the time of this writing, which will offer greatly improved capabilities.

The implementation of the "distribution" algorithm is expected to double the performance of the algorithm, obtaining at least 100,000 polygons per second on the Princeton Engine. Its implementation should consist chiefly of the addition of another procedure to the renderer and some careful management of the communication buffers.

## 9.1 Distribution Optimization

The distribution optimization for sort–middle communication promises a factor of 2 improvement in communication time. The distribution optimization performs communication in two stages, first sending polygons close to their final destination with a number of large steps, and then placing polygons on their final destination processors with neighbor–to–neighbor passes. Distribution gains performance over a regular sort–middle implementation because the overhead for communicating a polygon is per pass, so a large pass is comparatively cheaper than a neighbor–to–neighbor pass.

The distribution optimization yields a factor of 2 speedup in communication in simulation. Examining our peak performance scene, the crocodile at 62,400 polygons a second, we see that 69% of the time is spent in communication. If that time could be halved an immediate 50% improvement in performance would be seen, bringing the performance of the system to 95,000 polygons per second.

As discussed in §7.4, the distribution optimization maps very neatly to the line–locked programming paradigm of the Princeton Engine and implementation should be reasonably straightforward.

## 9.2 Sort–Twice

Sort–twice offers the most promising potential performance improvement seen so far. Our initial analysis suggests that sort–twice implemented on our second generation hardware, discussed in §9.3 would operate more than 4 times faster than sort–twice on the Princeton Engine, thanks to a tripling of the clock rate, and a wider communication buffer.

An implementation of sort–twice rendering would provide a further exploration of this design space and interesting feedback on the specific costs of the sort–last compositing step. Hopefully it could influence the design of future machines to include low cost support hardware to further streamline its operation.

## 9.3 Next Generation Hardware

The Princeton Engine is an eight year old architecture at this point. Currently a spin–out company from the David Sarnoff Research Center, the Sarnoff Real Time Corporation, is developing a commercial version of the engine called the "Magic–

1." The Magic, although based on the original Princeton Engine, has significant enhancements, and represents an intermediate step between the Princeton Engine and the "Sarnoff Engine," as described in [10].

The improvements include a tripled clock rate, more memory, disk array capability, 32–bit interprocessor communication, elimination of "line–locked" programming model, hardware support for bilinear and trilinear interpolation, support for high bandwidth video and non video data input, and a computed output timing sequence. The faster clock rate will directly yield a $3\times$ performance improvement per processor. The increased memory provides the possibility of very large polygonal data sets, and large texture maps. A computed "output timing sequence" enables each processor to independently determine when its pixel is output, this combined with a unique feedback capability that feeds the output of the engine into the input allows exploration of efficient sort–last compositing algorithms.

Beyond the basic clock rate improvements, the doubling of the width of the communication register will provide substantial performance gains, as pass operations will pass twice as much data. The overhead will be the same to write and read the register (still only a 16–bit core) and the test operations will presumably be the same, but distant passes will require fewer cycles to transmit the same amount of data. This will have a substantial impact on the performance of a sort–middle implementation with the "distribution" optimization.

More significantly, it will greatly cheapen the cost of a sort–twice approach. If we take the write/read and test overhead as negligible (due to the size of the passes employed) then compositing has become twice as cheap! Coupled with the clock–rate improvements, composition will only require one sixth of the time as the current architecture, a sizable improvement, which brings the attainable performance into the 30Hz update rate regime.

The new support for non video input data during operation will provide 20MB/s of input bandwidth which may be directed to any arbitrary subset of the processors simultaneously. This will allow the render to operate in a more convenient immediate mode style of rendering and to support databases of polygons and textures too large to fit in main memory.

Magic is fully operational at this point, and porting the polygon rendering to it will be a matter of obtaining adequate time on the machine given other conflicting demands.

# Chapter 10

# Conclusions

This thesis has addressed two parts of the broad topic of parallel polygon rendering.

A careful analysis of the sort–middle communication problem has demonstrated its lack of scalability. However, particular attention to optimizations has demonstrated the effectiveness of a carefully tuned and controlled communication algorithm for obtaining substantial performance improvements over typical communication structures. The use of a two–pass communication algorithm reduces the high cost of performing many neighbor–to–neighbor communication operations and achieves network utilization much closer to maximum throughput.

The search for scalable and cost–effect communication strategy led to the development of a novel new communication strategy entitled sort–twice communication. Sort–twice communication marries the efficiency of sort–middle for small numbers of both polygons and processors with the constant time performance to obtain an algorithm with the scalability of dense sort–last communication while amortizing the high cost of neighbor–to–neighbor communication over groups of processors. Cycle by cycle control of the communication channel creates a very efficient pipeline of communication operations. The VLIW architecture of the Princeton Engine is leveraged to support a fully deferred shading and texture mapping model, with substantial increases in the performance of the rasterizer.

An implementation of a 3D rendering engine on a distributed framebuffer SIMD architecture that achieves fill rates of 100 million pixels a second and over 60,000 texture–mapped, lit and shaded polygons a second was presented. The renderer implementation addressed several issues:

- Communication of polygon descriptors in a ring topology
- Rasterization in a processor per column organization
- Flexible interaction with a real–time rendering environment on a slave machine

True real–time rendering performance has been demonstrated on a SIMD processor array, with results that scale well with increasing numbers of processors. Polygons are of relatively arbitrary size, and benchmarked as the classic "100–pixel polygon". The algorithms are readily modified and adapted to experimentation.

The Princeton Engine provides a unique flexible architecture for real–time interactive rendering. The next generation of the Princeton Engine, the Magic–1, is expected to provide rendering performance of over 1 million polygons per second in a sort–twice implementation.

# Appendix A

# Simulator

A detailed and accurate simulator was developed to experiment with algorithm changes and optimizations without having to rewrite the implementation. The simulator provides the flexibility and ease needed to rapidly try many options and combinations of features with little effort. The simulator also provides the ability to simulate machines unavailable for actual use, such as machines with more processors, or our next generation hardware.

This thesis has presented a large number of communication algorithms and options to be analyzed. It is impractical to rewrite the implementation to analyze each optimization, for a number of reasons:

- Writing code for the Princeton Engine is intricate and time consuming.
- The line–locked programming constraint often obscures the true performance of an algorithm by making it difficult to use all available cycles for computation.
- It is difficult to establish correctness of the implementation. The lack of debugging tools make data distribution errors difficult to detect and analyze.

Fortunately the Princeton Engine lends itself very well to simulation. The Princeton Engine is a SIMD architecture, so by observing the worst case execution path of a program we observe the total execution time required. Interprocessor communication is completely controlled by the user program, allowing any communication algorithm to be studied at the cycle level with complete accuracy.

Within the simulator attention is paid almost exclusively to the details of the communication stage, but only because it has proven the most difficult stage to analytically model, particularly when the interactions of more than a single optimization must be considered. The geometry and rasterization stages, while requiring significant amounts of computation in the implementation, are trivially simulated by observing that the worst case processor (most polygons to compute, most pixels to render) will determine the performance of each stage.

## A.1 Model

The simulator model is significantly abstracted from the Princeton Engine. It assumes some arbitrary number of processors connected in a neighbor–to–neighbor ring. The structure of the simulated algorithm is based largely on implementation experience, and is organized to support the maximum amount of flexibility in the communication stage.

### A.1.1 Input Files

The input to the simulator consists of two pieces, a "bounding–box" file that describes the polygon database as a set of polygon bounding boxes in screen space, and a "screen map" that specifies the decomposition of the screen for rasterization.

The bounding–box file is generated by preprocessing a polygon database as observed from a specific viewpoint. *Every* polygon in the original database is specified in the bounding–box file, including polygons that are culled as out of the viewing frustum or backfacing. Each bounding–box is tagged with a `1` or a `0` indicating visible/culled. The inclusion of culled polygons in the bounding–box file helps model the geometry stage, during which some fraction of the polygons computed will be backfacing and/or out of the viewing frustum. It is important to model these culled

```
NumPolygons=7
0 (406,163) (407,164)
0 (407,167) (408,167)
1 (406,182) (415,185)
1 (413,166) (418,171)
0 (411,183) (413,186)
1 (364,219) (370,223)
1 (381,204) (388,212)
```

Figure A-1: **Bounding–box File:** Each polygon is specified as a visible/nonvisible tag and a rectangular bounding–box.

polygons because:

- Culled polygons have a computational cost associated with rejecting them.
- It may be cheaper to compute and reject a polygon than a visible polygon, which will affect the total instructions executed in the geometry stage.
- Polygon culling yields asymmetries in the distribution of polygons across processors, which may affect the execution time of both the geometry and communication stages.

The use of a bounding–box file allows the details of the actual geometry calculations to be removed from consideration, insuring that we maintain a stable input data set for different simulations. It also removes the burden of insuring that the geometry pipeline is correctly implemented in the simulator in addition to the renderer. Figure A-1 shows a sample bounding–box input file.

The screen map allows completely arbitrary maps of pixels to processors for rasterization to be specified. A default column–parallel decomposition of the screen is assumed, but general rectangular regions may be specified, including non–contiguous regions, allowing the analysis of static load balancing techniques such as those suggested in [20]. The only constraints on the screen map is that every pixel is rasterized by precisely one processor[1]. A sample screen map is show in Figure A-2.

[1]The sort–twice case, for which multiple processors rasterize the same region of the screen, is

```
0 (  0,   0) (383, 255)
1 (  0, 255) (383, 511)
2 (384,   0) (767, 255)
3 (384, 256) (767, 511)
```

Figure A-2: **Sample Screen Map:** a 4 processor system, with each processor assigned a quadrant of a 768 by 512 pixel screen. The processor is given in the first column, followed by the rectangular region it rasterizes. Processors may be listed multiple times in the file for non–contiguous rasterization regions.

Given the polygon bounding boxes in the bounding–box file and the pixel to processor map in the screen map it can be determined precisely what processors will rasterize each polygon. Any given polygon is rasterized by the union of the processors whose pixels it overlaps.

### A.1.2 Parameters

The simulator is fully parameterized. All of the parameters of interest may be specified via command line options, and more drastic changes (such as in the width of the interprocessor communication bus, etc.) may be trivially made through recompilation. The parameters of interest are shown in Table A.1.

Most of the parameters are fairly obvious. The particular parameters of interest for specifying the communication structure are $m$, $t$, $d[]$ and $g$, which enable the simulation of the duplication, dense, distribution and sort–twice optimizations.

**Data Duplication** The data duplication pattern is specified by $m$, which is the number of times the polygon database is duplicated across the processors. It is specified as a geometry parameter because it is fully resolved by the geometry stage of the simulator, and not exposed to the communication stage.

**Dense Communication** The period between tests, $t$, specifies how often the simulated processors will test the polygon they just received to see if it is completely

---

discussed later.

| parameter | default | meaning |
|---|---|---|
| $n$ | 1024 | number of processors |
| $polyfile$ | –none– | bounding–box file |
| $mapfile$ | column parallel | pixel to processor map |
| geometry | | |
| $G_i$ | 1000 | instructions to transform and keep/reject polygon |
| $G_f$ | 5000 | instructions to light and compute linear equations for a polygon |
| $m$ | 1 | duplication factor |
| communication | | |
| $t$ | 1 | period between tests |
| $d[]$ | [1] | array specifying sequence of pass sizes to be used |
| $g$ | $n$ | number of processors per group |
| rasterization | | |
| $R$ | 100 | instructions to rasterize a pixel |

Table A.1: **Simulator Parameters**

communicated and can be replaced with their next polygon to communicate. Dense communication, $t = 1$, has been determined to always be a useful optimization and is used by default. For simulation of the sparse communication case, discussed in §5.2, $t = n$.

**Distribution** The distribution pattern is specified in $d[]$ as a sequence of pass sizes to be used during communication. The optimal distribution strategy, as determined experimentally, would be represented as $d = [16, 1]$.

**Sort–Twice** The group size $g$ is used for sort–twice simulations and specifies the number of processors responsible for rasterization of one complete copy of the screen.

All of these parameters interact as would be expected. Thus a duplication factor $m \neq 1$ and a distribution pattern can be combined in the same simulation, for example.

Notably absent is support for temporal locality optimizations. Temporal locality requires a substantial extension of the simulator. A common benchmark for temporal locality, used in [11] for example, is to render a scene, slightly change the viewpoint and render the scene again. The small change in the viewpoint forces some communication to be performed so that the efficiency of temporal locality in communication can be modeled. However, the use of a bounding–box file to collapse the polygon information destroys the very information necessary to specify arbitrary viewpoints. Simulation of temporal locality load imbalances in §5.5 exposed severe enough inefficiencies that the approach was abandoned at that point, eliminating the need for simulator support.

In addition to these parameters, there are a number of instrumentation knobs that may be turned on for any given simulation to collect extra data. Of particular use is the option to generate a histogram of the number of polygons on each processor before and after each phase of communication. This exposes both geometry load imbalances, as may be created by $m \neq 1$, and rasterization load imbalances as may be created by a non–uniform distribution of polygons over the display.

## A.2 Operation

The high–level diagram in Figure A-3 schematically represents the execution of the simulator. Almost all of the operation is wrapped up in the details of the as yet unspecified communication simulation. In addition there is some preprocessing to load the polygon database into the processors, and some post processing to count the expected instructions for rasterization. The following sections describe the operation of the stages of the simulator in detail.

### A.2.1 Data Initialization & Geometry

Data initialization determines the mapping of polygons to processors for geometry computations, and the mapping of pixels to processors for rasterization. The former is specified by simply placing polygons on processors, while the latter is determined

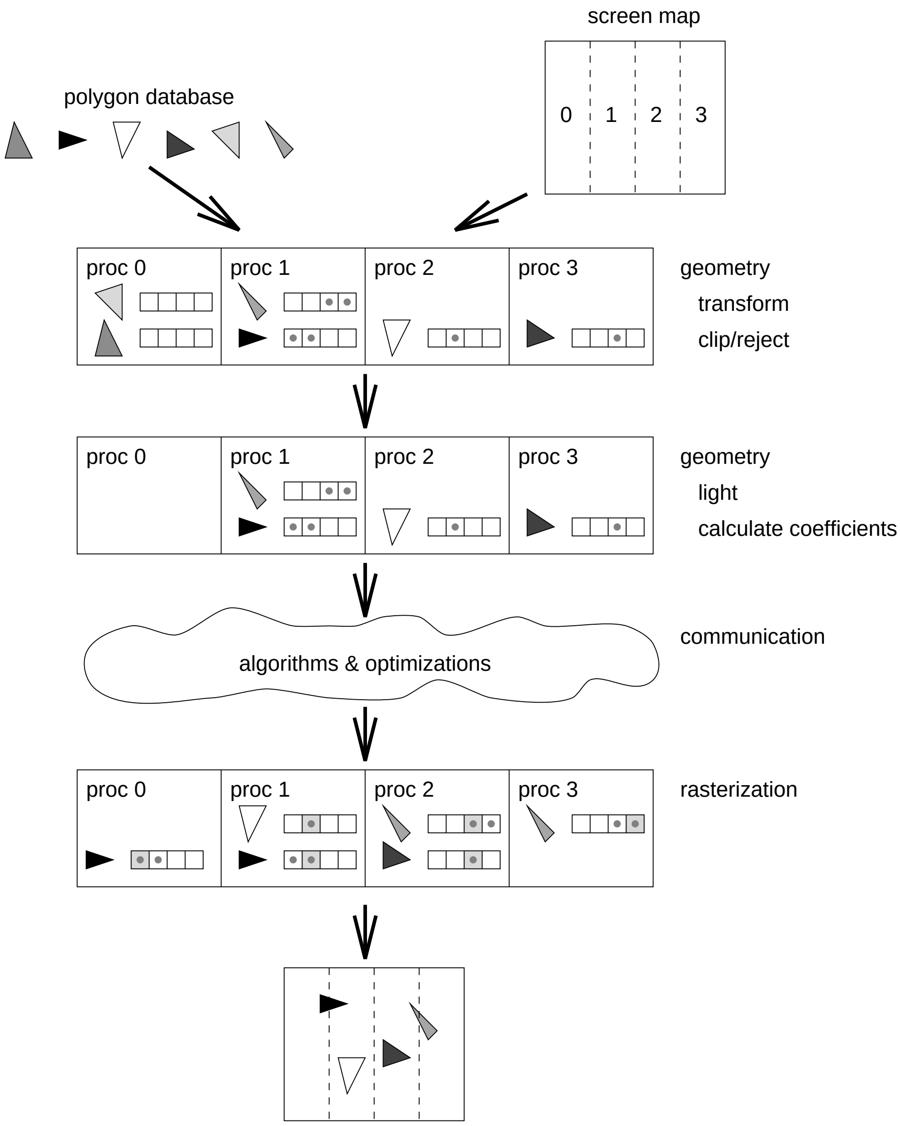



Figure A-3: **Simulator Model:** a 4 processor polygon renderer with a 6 polygon database.

by calculating a "destination vector" for each polygon which tags each processor responsible for rasterizing a part of the polygon.

The bounding–box file only defines the total set of polygons in the simulation, and leaves unspecified the mapping between polygons and processors for the geometry calculations. The simulator allows two distribution patterns:

**Normal** The polygons are assigned to the processors in round robin fashion. Numbering the polygons sequentially as they are read from the bounding–box file, polygon $b$ will lie on processor $i = mod(b, n)$ on an $n$ processor system.

**Duplicated** A duplication factor $m \geq 1$ specifies that each polygon should be duplicated $m$ times across the processor array. A polygon $b$ is instantiated on all processors $i = mod(b, n/m) + km$.

The normal distribution is just a degenerate case ($m = 1$) of the duplicated distribution pattern.

The possibility of a random assignment of polygons to processors was not included, as this can be simulated by shuffling the bounding–box input file before it is read by the simulator, and experimental results show that any correlation in the data set is already broken up by the round robin assignment of polygons to the processors.

The combination of the polygon bounding boxes from the bounding–box file and the pixel to processor map from the screen map provides all of the necessary information for performing communication. Each polygon is described in part by an $n$–bit destination vector, where a bit $i$ is set if and only if the polygon bounding box intersects the set of pixels rasterized by processor $i$. Figure A-3 shows the destination vectors associated with each polygon as a short bit vector next to the polygon. Note that multiple bits are set if more than a single processor will be responsible for rasterization of the polygon.

The use of a bounding–box file which specifies the rectangular extent of the polygons guarantees that we will never incorrectly assume a polygon doesn't overlap a

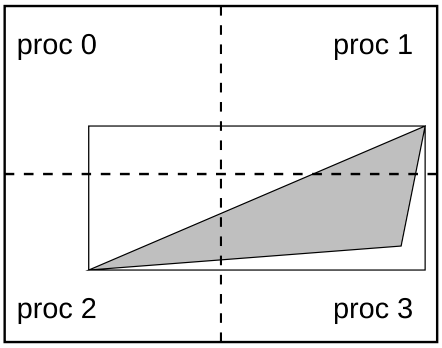



Figure A-4: **Destination False Positive:** the polygon is incorrectly assumed to intersect the rasterization region of processor 0.

processor's pixels, however, in some cases we may incorrectly assume coverage. Consider the polygon and screen map shown in Figure A-4: processor 0 will attempt to rasterize a piece of this polygon, even though it doesn't actually overlap processor 0's region. It is assumed that these occasional false positives incur a lower cost than a precise test. Given the cost of sort–middle communication it may make sense to perform a test to eliminate false positives, however, it is unnecessary for our simulations of interest. For both our implementation, and most of our simulations (barring sort–twice approaches), a column–parallel decomposition of the screen is used, in which case the bounding box of the polygon will only overlap a processor's column of the screen if there will be pixels to rasterize, so there is no excess communication done due to false positives.

The first stage of geometry computation will process a worst case of $\lceil p \cdot m/n \rceil$ polygons on a processor, and requires $G_i$ instructions per polygon to perform. In general each processor will be responsible for communicating, and thus performing the second stage of geometry computation, all visible polygons assigned to it. The data duplication optimization, discussed in §5.4, introduces a slight twist, as the same polygon will exist on $m$ processors; however, only one processor should communicate the polygon, in particular the processor which can perform the communication most cheaply. Each processor will source a polygon only if it is the rightmost processor left of the first processor rasterizing this polygon[2].

After the polygon database is loaded onto the processors the lighting model is applied to the visible polygons on each processor, and linear equation coefficients are calculated. This is a "pseudo-step" in the simulator, which only deals with polygon bounding boxes, and is included to model the actual algorithm. It requires $\max_i(p_{iv}) \cdot G_f$ instructions to execute, where $p_{iv}$ is the number of visible polygons on processor $i$. Note that at this point we only have a single copy of each polygon ready to distribute, regardless of the choice of $m$. The duplication factor $m$ is hidden from the remainder of the simulator in effect[3].

### A.2.2 Communication

Communication is the most challenging part of the polygon rendering algorithms to simulate. All other aspects of the simulation may be performed at a gross level by simply counting polygons (pixels) and multiplying by the number of instructions required per polygon (pixel). Communication however requires a detailed understanding of what happens in the interprocessor communication channel on a pass by pass level.

Communication is modeled as occurring on a 1D ring of processors. Each processor is assigned a "slot" in the ring which it can place a polygon descriptor into, and read a polygon descriptor from. The ring is represented by an array of pointers to polygon descriptors. Each polygon descriptor represents a minimal set of information, a reference number for the polygon (for later verifying the correctness of the communication) and a destination vector which specifies the processors that must receive a copy of this polygon for this stage of communication. Figure A-5 shows the ring.

[2]This is what we mean by "closest" - the processor that will have to make the fewest passes before the polygon has at least started to arrive at its destination. This proves to be a useful cost metric for column–parallel decompositions of the screen, where all destination processors are contiguous. A more sophisticated communication cost measure must be used for disjoint decompositions of the screen.

[3]Of course duplication may introduce large asymmetries in the values of $p_{iv}$, but that will affect only the simulated execution cost.

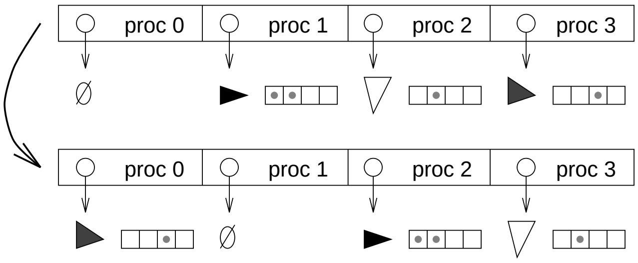



Figure A-5: **Simulator Communication:** the interprocessor communication is abstracted as passing an entire polygon descriptor neighbor–to–neighbor at once.

Processor 0 points to a "null" which indicates that there is no polygon descriptor currently associated with that location in the ring.

Each processor has an array of polygon descriptors to source and an array of polygons received from communication, analogous to the `source` and `receive` arrays in the implementation. Initially the ring is empty and the processors all have their source arrays full and their receive arrays empty, exactly like the actual implementation. Similarly, at the end of communication each processor will have an empty source array and a receive array with all polygons it will have to rasterize.

Communication operations are performed by "rotating" the ring. All of the pointers are shifted one place to the right in the array, with the pointer that falls off the end shifted back to the middle. Larger shifts (for the distribution optimization) are implemented by performing a larger rotation of the ring. The communication isn't actually broken down to the finest granularity of simulating each individual communication operation necessary to pass a polygon descriptor from processor to processor. The cost model for a pass is precisely the $i = (2+d)a+b$ model used in the analytical analyses, with $a = 42$ and $b = 30$.

After each pass operation each processor tests the destination vector of the polygon descriptor in its slot of the ring, and if the polygon is destined for it notes the reference number in the receive array, and clears its bit in the vector. A polygon has been communication to all interested processors when no bit in destination vector is set.

The dense, distribution, and sort–twice communication optimizations are all exposed within the communication simulation, and are discussed in turn below.

### Dense Communication

The sparse vs. dense communication optimizations are implemented via the $t$ variable which specifies how often the processors test the polygon descriptor in their "slot" to see if everyone has seen it. For simplicity's sake each processor will automatically replace a null in its slot of the ring with one of the polygons from its source array. The periodicity $t$ of the test determines how often the destination vector of each descriptor in the ring is tested and turned into a null if all zero.

### Distribution

The distribution optimization is relatively tricky to implement. It is performed by executing the communication algorithm a number of times, and between each execution the receive array for each processor is used to rebuild the source array. During all stages of communication except the last the shifts that are performed are of size greater than 1, so the notion of the destination vector becomes a bit muddied. Polygons can no longer be arbitrarily routed, but can only be sent to processors $i = s + kd$, where $s$ is the processor this polygon is sourced from and $d$ is the size of the pass. To achieve this distribution pattern the destination vectors are built for the actual polygon destinations and then all of the destinations are adjusted to align with the first valid destination before the desired destination processor, as shown in Figure A-6a.

In our simple example $d = [4, 1]$ and processor 1 is sourcing a polygon destined for processors 8, 9 and 10. The first stage of distribution results in both processor 5 and processor 9 having a copy of the polygon. The second pass of communication will be of size 1 and will route the polygon to its final destinations, processors 8, 9 and 10. If we aren't careful both processor 5 and 9 will try to route the polygon to all of these processors, which in the worst case results in these processors rasterizing the polygon twice and also wastes communication bandwidth. The observation is made that after performing a communication stage with passes of size $d$ each processor can

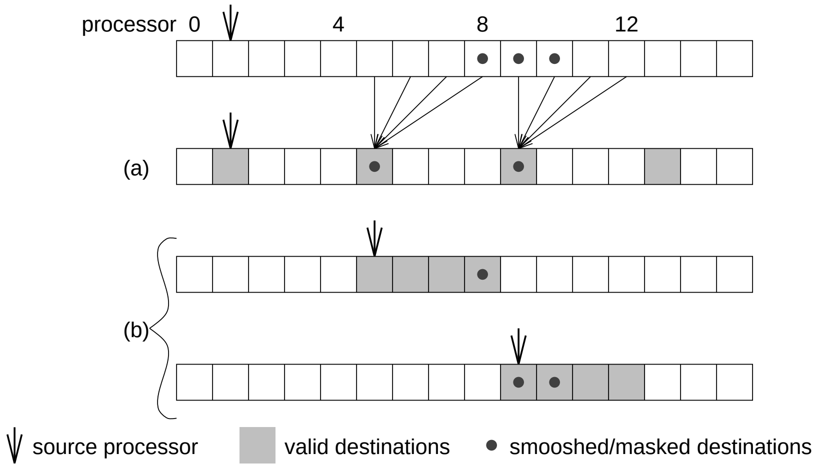



Figure A-6: **Destination Vector Alignment:** a 16 processor system using a distribution pattern of `4,1`. The destination vectors are for a polygon that is distributed from processor 1 to final destinations of processors 8, 9 and 10. (a) is the destination vector used for the first pass of communication and (b) are the destination vectors for processors 5 and 9 to perform the second and final pass of communication.

only pass any polygons it has to source over $d-1$ processors to avoid duplication. Thus processor 5 will only pass polygons as far as processor 8, etc. In fact we can always think of communication as starting with a pass of size $n$, the number of processors, after which each polygon may only be passed over at most $n-1$ processors before it will be seen twice by some processor. Figure A-6b shows the destination vectors used by processors 5 and 9 for the second stage of communication.

### Sort–Twice

The final optimization to deal with is sort–twice communication. The placement of processors into groups of size $g$ introduces a subtle complexity into the communication structure by making processors equivalent at some level. If a polygon overlaps region $r$ it can be rasterized by any processor $r + kg$, so the communication pattern is no longer deterministic. In reality of course it is deterministic because we want each polygon to travel the *minimum* amount of distance possible, which means its only destination is the first processor $r + kg$ after the processor it starts on. However,

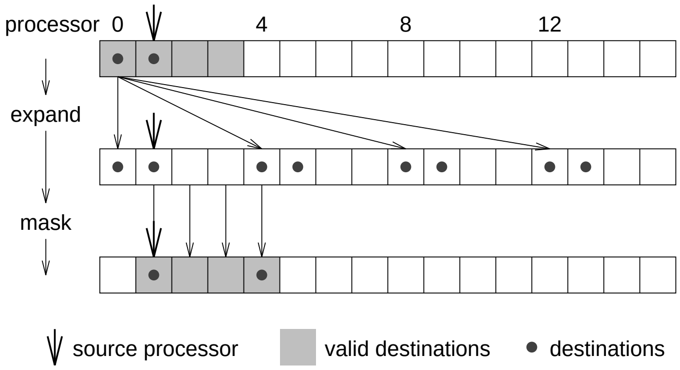



Figure A-7: **Destination Vector for $g \neq 1$:** the destination processor set is inferred from the first $g$ processors and then masked so that the polygon is received at most one of the processors which can rasterize the region(s) it overlaps.

the pixel to processor map will only map pixels to the first $g$ processors, because the screen is divided into $g$ regions, and only a single processor may be responsible for any pixel. Thus when the destination vector is built it only specifies destinations in the first $g$ processors. Figure A-7 shows a sample destination vector for sort–twice communication. The destination vector is expanded to set or clear all of the bits, based on the first $g$ processor bits, to reflect all the valid rasterization processors. The vector is then masked so that the polygon is shifted a maximum of $g - 1$ times, to avoid unnecessary communication. After this more complex destination vector construction, the communication may proceed as it normally would, although of course the extra instructions required to perform the sort–last compositing must be accounted for later. The sort–last composition is a data independent operation and the number of instructions required is just calculated based on the group size and the number of processors. The contribution is 0 instructions for $g = n$ (vanilla sort–middle communication) as would be expected.

This section has described all of the major features of the communication simulation and some of the care that is necessary in their implementation. It is worth noting that complexity in the simulation will likely also be reflected in any implementation, and any additional insight the simulator provides in implementation can be quite valuable.

### A.2.3 Rasterization

The rasterization stage is simulated in a fashion similar to the geometry stage. It is possible with reasonably high accuracy to guess how long it will take to rasterize a set of polygons. For each processor $i$, $r_i$ is computed, which is the sum of the number of pixels (rasterized by this processor) that all polygon bounding boxes overlap. The total rasterization time is then just $\max_i(r_i) \cdot R$, where $R$ is the time to rasterize a single pixel. This ignores details of the implementation such as the cost of advancing to the next polygon, and the effects of performing the test for advancement to the next polygon periodically rather than after every single pixel. These effects have been included in an ad hoc fashion by increasing the cost of rasterizing a single pixel. While not strictly correct, this approximation yields good results. The goal of the simulator has been to measure the effectiveness of various communication optimizations, the effects of which are entirely wrapped up in the geometry and communication stages, making the accuracy of the rasterization stage simulation less important.

## A.3 Correctness

The results of the simulator are to a large extent verifiable. A simple (and therefore probably correct) check is done at the end of the communication stage that checks that:

- Each polygon was received on all processors whose pixels it overlapped.
- No processor received a polygon it didn't need.
- No processor received more than one copy of a polygon.

The first check is of course most critical. If any processor doesn't receive a polygon that it needs to rasterize it could result in a hole in the rendered scene. The second check insures we aren't introducing extra work by giving processors polygons to rasterize that don't overlap their screen areas.

The necessity of the third check is not immediately obvious. This detects subtle errors which can be introduced with the distribution optimization. In this case a bug in the algorithm could (and did) accidentally distribute polygons in duplicate (and even triplicate) in some boundary cases.

Of course, there are some errors that the simulator can't detect, and they will necessary all relate to performance, rather than the correctness of the result. Such errors would include distributing a polygon further than necessary (past the last processor interested in it), or miscounting the instruction cycles for some operation. These areas of the code have all been carefully checked, and the results given by the simulator reflect our expectations, so it is believed they are correct.

## A.4 Summary

The analysis of various optimizations has demonstrated that while analytical models may accurately model the form of the correct answer, simulation may yield a substantially different, and presumably more accurate, solution. Largely these differences in accuracy are caused by the assumptions used to make the analytical problem tractable. Assumptions such as a "dense ring" are made to compensate for the lack of any good model for the actual distribution of information in the ring. Variations in the distribution of sources and destinations for polygons from uniformity exacerbates things further, making simulation invaluable to obtain accurate answers for real polygon databases.

The very factors that make the communication so hard to analytically model make it equally hard to simplify the simulation. Fortunately for our data sets it is not prohibitively expensive to perform a full simulation of the communication structure.

# Bibliography


[1] Akeley, K. Reality Engine Graphics. In *Proceedings Siggraph 93* (August 1993), pp. 109–116.

[2] Akeley, K., and Jermoluk, T. High–Performance Polygon Rendering. In *Proceedings Siggraph 88* (August 1988), pp. 239–246.

[3] Chin, D., Passe, J., Bernard, F., Taylor, H., and Knight, S. The Princeton Engine: A Real–TIme Video System Simulator. *IEEE Transactions on Consumer Electronics 34* (May 1988).

[4] Cox, M. B. *Algorithms for Parallel Rendering.* PhD dissertation, Princeton University, Department of Computer Science, May 1995.

[5] Crockett, T. W., and Orloff, T. A Parallel Rendering Algorithm for MIMD Architectures. In *Proceedings Parallel Rendering Symposium* (New York, 1993), ACM Press, pp. 35–42.

[6] Crow, F. C., Demos, G., Hardy, J., McLaughlin, J., and Sims, K. 3D Image Synthesis on the Connection Machine. In *Parallel Processing for Computer Vision and Display.* Addison Wesley, 1989.

[7] Ellsworth, D. A New Algorithm for Interactive Graphics on Multicomputers. *IEEE Computer Graphics and Applications 14*, 4 (July 1994), 33–40.

[8] Foley, J. D., van Dam, A., Feiner, S. K., Hughes, J. F., and Phillips, R. L. *Introduction to Computer Graphics.* Addison-Wesley, 1994.

[9] Heckbert, P., and Moreton, H. Interpolation for Polygon Texture Mapping and Shading. In *State of the Art in Computer Graphics: Visualization and Modeling*. Springer–Verlag, 1991.

[10] Knight, S., Chin, D., Taylor, H., and Peters, J. The Sarnoff Engine: A Massively Parallel Computer for High Definition System Simulation. *Journal of VLSI Signal Processing*, 8 (1994), 183–199.

[11] Molnar, S., Cox, M., Ellsworth, D., and Fuchs, H. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications* (July 1994), 23–32.

[12] Molnar, S., Eyles, J., and Poulton, J. PixelFlow: High–Speed Rendering Using Image Composition. In *Proceedings Siggraph 92* (July 1992), pp. 231–240.

[13] Molnar, S. E. *Image-Composition Architectures for Real-Time Image Generation.* PhD dissertation, The University of North Carolina at Chapel Hill, Department of Computer Science, October 1991.

[14] Neider, J., Davis, T., and Woo, M. *OpenGL Programming Guide*. Addison-Wesley, 1993.

[15] Phong, B. T. Illumination for Computer Generated Pictures. *Communication of the ACM 18*, 6 (1975), 311–317.

[16] Pineda, J. A Parallel Algorithm for Polygon Rasterization. In *Proceedings of SIGGRAPH '88* (1988), pp. 17–20.

[17] Shoemake, K. Animating Rotation with Quaternion Curves. In *Proceedings Siggraph 85* (July 1985), pp. 245–254.

[18] Connection Machine Model CM-2 Technical Summary. Tech. Rep. HA87-4, Thinking Machines, April 1987.

[19] Whitman, S. *Multiprocessor Methods for Computer Graphics Rendering.* AK Peters, Wellesley, MA, 1992.

[20] Whitman, S. A Load Balanced SIMD Polygon Render. *unpublished article* (August 1994).

[21] Whitted, T. A Taxonomy of Image Composition Architectures. In *International Summer Institute on State of the Start in Computer Graphics* (1992).

[22] Williams, L. Pyramidal Parametrics. *Computer Graphics 17*, 3 (July 1983), 1–11.