DATA-PARALLEL RASTERIZATION OF MICROPOLYGONS

AN HONORS THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
OF STANFORD UNIVERSITY

Edward Luong
Principal Adviser: Patrick Hanrahan
January 2010

Abstract

We present two data-parallel algorithms for rasterizing micropolygons. Our approaches differ from current techniques by parallelizing across polygons. The first, LOCKSTEP, processes micropolygons in lockstep and runs with high utilization (65%) for a small number of SIMD units, despite variance in the amount of rasterization work for each micropolygon. The second, REPACK, compacts works and is able to maintain high utilization (85%) even for large number of SIMD units. While REPACK operates with high utilization, it incurs overhead from repacking data. We study the performance and utilization of our algorithms on simulated hardware with varying amounts of SIMD units. We make predictions about the overall cost of rasterizing micropolygons in a future real-time system.

Acknowledgements

This thesis represents a year of research in Pat Hanrahan's graphics group. Pat encouraged me to try my hand at research after I took his course on Image Synthesis. We quickly found a project that interested me, and since then, he has always been enthusiastic about my involvement with the project and extremely supportive of my work.

I was fortunate to have been able to inhabit Gates 381 all year. I am extremely lucky to have worked closely with Kayvon Fatahalian. Kayvon was instrumental in getting me up to speed with the research project. He was always available for questions and was rarely short on answers. His insight and guidance has been critical to the success of my research. In addition to Kayvon, I was lucky to have as officemates, Solomon Boulos and Matt Fisher. Solomon and Matt have both shared with me their deep knowledge of computer science, math, work, graduate school, and life in general. I am also glad to have worked with Kurt Akeley and Bill Mark. Their involvement in our research meetings have been invaluable.

I'd like to thank Mary McDevitt for all the feedback on all of my drafts. She was very flexible and worked around my extremely tight schedule.

Thanks to all my colleagues in the honors program, especially Lawson Wong and Sergey Levine. Also, I'd like to thank all my wonderful friends here at Stanford who have been so understanding of my busy schedule. In addition, thanks to all of my friends in the CS198 section leading program, where I learned so much about the value of clear communication.

I'd like to thank my parents, William Luong and Hue Luong. I would not be where I am without their encouragement and support. They have always stressed

the important of balance between life and work, which usually amounted to them reminding me to take more breaks. Also, my brother, Edison Luong, has been a continual inspiration in my life.

Finally, I'd like to thank Tiffany Nguyen for her understanding and loving support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Image synthesis is the process of transforming a scene, typically represented as mathematical models in 3D space ($\mathbb{R}^3$), into a flat, discrete representation (the image plane). Nearly all graphics systems break this process into three stages: geometry, shading, and rasterization. The geometry stage is usually the first stage and is responsible for processing incoming geometry for the rest of the pipeline. It includes per-polygon computations (such as tesselation) and per-vertex computations (such as displacement). The shading stage determines the color values on the surface. The color of the surface is usually computed using a combination of surface parameters, textures, and light sources. Finally, rasterization is the stage that discretizes scene data into an image. Broadly speaking, *rasterization* is the process of determining the color contribution of each polygon to the pixels in an image.

While rasterization is responsible for correctly interpolating color information, the main challenge in rasterization is determining visibility. Determining visibility consists of identifying the regions of surfaces in a scene that are visible from a virtual camera. Most visibility algorithms optimize for large polygons (many pixels in size). Many efficient algorithms exist, and nearly all modern GPUs have been able to exploit data-parallelism to attain high throughput for rasterization.

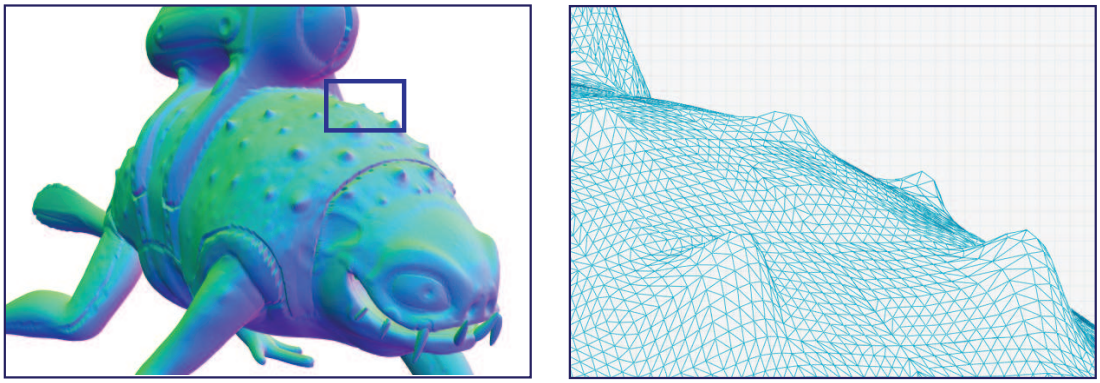


Figure 1.1: Complex surfaces such as this frog's skin are represented accurately using micropolygons.

There is increasing demand for rendering realistic images in real-time graphics systems, and in order to synthesize realistic images, we often need to handle complex geometry. Techniques for approximating complex geometry using a combination of coarse geometry and shading have been employed in current graphics systems. However, smooth surfaces with high curvature, or highly detailed surfaces–such as bumpy or displaced surfaces–cannot be faithfully reproduced with these techniques. As the bar for image quality rises, approximations that fall short of actual detailed geometry will no longer be suitable.

A simple way to capture high geometric detail is to tesselate geometry into *micropolygons*, polygons with less than a pixel area. Such a strategy is popular among offline renderers (Apodaca & Gritz, 2000; Hou, 2009); however, current real-time graphics systems are not equipped to handle micropolygon workloads. Several changes to the graphics pipeline are needed for a complete transition from large polygons to micropolygons. A study undertaken at Stanford University aims to design a micropolygon pipeline for future GPUs. Work that addresses changes to the tesselator appear in (Ano, 2009b).

The work discussed here focuses on rasterization. An efficient, high-throughput rasterizer is a critical component of a micropolygon rendering pipeline. Current rasterization algorithms are extremely inefficient for micropolygons due to the inability to quickly determine visibility when polygons become subpixel in size.

This thesis discusses data-parallel formulations of an algorithms for rasterizing micropolygons. It presents a detailed workload analysis of our implementaions and characterizes the micropolygon rasterization workload to guide future optimized hardware and software implementations.

Parts of this thesis appear in (Ano, 2009a).

Outline

The next chapter provides the necessary background for the rest of the thesis. Chapter 3 presents our algorithm and data-parallel implementations. Chapter 4 provides performance results for a simulation of implementations. The last chapter summarizes the work and presents plans for future work.

Chapter 2

Background

2.1 REYES

The REYES rendering architecture is an architecture developed at Lucasfilm for quickly rendering high-quality images (Cook *et al.*, 1987). REYES was specifically designed to handle models with a large number of polygons. Models are input as high level primitives which are are split into sub-primitives (*Split*). This process repeats until each sub-primitive is determined to be small enough. The sub-primitives are then uniformly tesselated into pixel-sized micropolygons (*Dice*). Micropolygons are shaded (*Shade*) and then rasterized (*Visibility*). Since only a small set of fully-tesselated geometry ever needs to be in memory, the REYES architecture is extremely well-suited for scenes with high geometric complexity. The REYES architecture motivates the use of micropolygons to render high-quality images of scenes with high-geometric complexity.

2.2 Parallelism

In this thesis, we only consider data-level parallelism (data parallelism) as opposed to thread-level (multicore) parallelism. The most common form of data-level parallelism exploits the efficient execution of a single instruction on multiple data (SIMD). We refer to the "multiple data" as a vector, and the vector width is the amount of data-level parallelism an architecture supports per operation.

Thread-level parallelism, on the other hand, addresses parallelism from running separate threads in parallel. Different threads usually perform different tasks, and coordination between threads must occur if there are data dependencies. Thread-level parallelism usually does not scale as well as data-level parallelism. As the number of threads (cores) increases, either algorithms need to change to accommodate the extra threads or communication between threads becomes a performance barrier. Note that thread-level parallelism is more general than data parallelism. That is, multiple threads can be used to exploit data parallelism; however, the overhead needed for general thread-level parallelism is much higher than what is needed to support SIMD processing (Hennessy & Patterson, 2002).

2.3 Point Sampling

The frame buffer represents the image plane as a collection of (X, Y) samples. While the goal of rasterization is to determine the color values for each of these samples, fundamentally, rasterization is about computing visibility at each sample. More precisely, given a polygon, P, we determine the set of samples covered by P. A solution to this problem returns a list of polygon-sample pairs, (P, s), where each pair corresponds to P covering s.

The main geometric operation for computing visibility is *PointInPolygon*. For a polygon, P, and a sample, s, *PointInPolygon* tests if s is covered by P. A common implementation of *PointInPolygon* for convex polygons) uses edge functions: s is inside P if all edge functions evaluated at s are all positive.

An alternate of computing visibility frames the problem as point sampling visibility for a polygon in 2D (X, Y) space. This formulation makes extending rasterization to support motion blur and depth of field natural: rasterizing micropolygons under motion blur or camera defocus becomes the problem of point sampling in 3D (X, Y, T) space or 5D (X, Y, T, U, V) space, respectively. *PointInPolygon* generalizes into higher dimensions by using hyperplanes instead of edges. This type of sampling has been implemented (Ano, 2009a); however, it is out of the scope of this thesis.

A simple frame buffer contains a sample at the center of each pixel in the image; however, most modern systems use multiple samples per pixel for multisampling anti-aliasing (MSAA). MSAA helps remove aliasing artifacts ("jaggies") by increasing the sampling rate and, consequently, the Nyquist frequency. In addition to increasing the sampling rate, samples can be placed randomly on the frame buffer to improve quality. Randomization helps decrease structured aliasing by replacing it with noise. *Jittered sampling* is a common sampling pattern that approximates a Poisson disk sampling (Cook, 1986). With jittered sampling, if each pixel contains S samples, we divide the pixel into S uniform regions. We place a sample at the center of each subpixel region, then jitter it by some random amount, $(\triangle_x, \triangle_y)$. Jittered sampling provides similar benefits as Poisson disk sampling, and its structure is convenient for most output image formats (i.e., a grid of pixels).

2.4 Traditional Visibility Algorithms

Before reviewing previous techniques, we describe two metrics used to evaluate the efficiency of a rasterizer. *Sample test efficiency* (STE) is the ratio of the number of *PointInPolygon* hits to the number of *PointInPolygon* tests. Utilization is a measure of how efficiently vector processing units are utilized. High utilization is an indication of good parallelization. Typically, STE is determined by the algorithm, whereas utilization is determined by the actual implementation. However, some algorithms may lend themselves to better parallel implementations than

others.

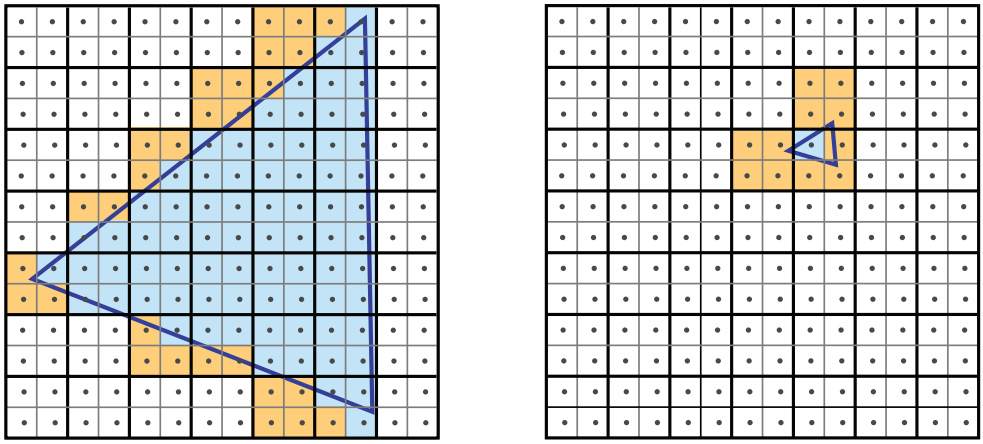
Most rasterization techniques can be split into four steps. First, the *Setup* step performs any necessary per-micropolygon setup computations. Second, in the *Bound* step, a set of candidate samples is identified. Third, the *Test* step computes *PointInPolygon* for each candidate sample. Last, *Process* updates the frame buffer for each tested sample that passes *PointInPolygon*. This thesis focuses on all steps up until *Process* because computing coverage is the biggest algorithmic challenge.

2.4.1 Tiling methods

Tiling methods typically tradeoff STE for wide data-parallelism. In its simplest form, a tiling method performs *PointInPolygon* tests on all samples in the frame buffer in parallel. Pixel-planes accelerated this computation by using specially designed memory units augmented with a tree of one-bit adders (Fuchs *et al.*, 1986). While this approach is massively parallel, much of the computation is wasted. If the average polygon covers A samples and there are R total samples, this algorithm has a fixed STE of $\frac{A}{R}$, which can be extremely low if used naïvely on a large framebuffer.

Tiling methods have been refined to improve STE. Rather than testing the whole frame buffer, tiling methods first divide the frame buffer into tiles (raster stamps). Polygons are sorted by which tiles they cover, and then the *PointInPolygon* tests for all samples in each tile are performed in parallel (Fuchs *et al.*, 1989; McCormack & McNamara, 2000). In summary, tiling methods are broken down as follows:

- *Setup* computes edge equations for polygon
- *Bound* determines tiles covered by polygon.
- for each covered tile, *Test* performs *PointInPolygon* tests in parallel using the precomputed edge equations.



Polygon: STE=73% **Micropolygon: STE=8%**

Figure 2.1: Rasterization of a polygon and a micropolygon using 2×2 tiles. A *PointInPolygon* test was performed for the sample in each colored box. Blue boxes yield hits and yellow boxes yield misses.

Many implementations using tile sizes ranging from 4×4 to 128×128 have been proposed (Pineda, 1988; Fuchs *et al.*, 1989; Seiler *et al.*, 2008). For large polygons, these schemes can attain both high utilization and high STE because overtesting only occurs on tiles that straddle the polygon border. However, for micropolygons, nearly every tile tested will contain a micropolygon edge. Figure 2.1 illustrates this effect with a small tile size. Samples within the polygon are shown in blue. Samples tested during rasterization, but not covered by the polygon, are shown in yellow. STE of the micropolygon is low because the area of the micropolygon is small compared to the area subtended by a tile. Consequently, STE becomes lower for larger tiles.

2.4.2 Hierarchical methods

Hierarchical methods are a variant of tiling methods that use a hierarchy of tiles of different sizes (Greene, 1996). Hierarchical methods identify hit samples by traversing the tile hierarchy and selectively refining tiles that lie on a polygon's edge (see figure 2.2). First, coverage for polygons against coarse tiles is computed. Each coarse tile that is covered is refined into smaller tiles unless the tile

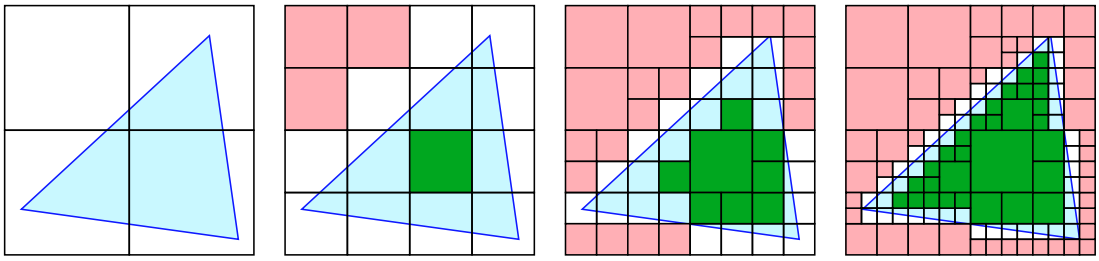


Figure 2.2: Traversal of the tile hierarchy for a polygon. Green tiles are determined to be completely inside a polygon and pink tiles are determined to be completely outside a polygon. Samples that lie within white tiles still need to be tested.

is determined to be completely inside a polygon or completely outside a polygon. This process generally ends when the number of samples in a tile equals the vector width of the architecture. This tile hierarchy traversal can be performed in a cache coherent manner using Hilbert curves (McCool *et al.*, 2001). The open source renderer, Aqsis, builds a kd-tree of samples and traverses this tree to find covered leaf nodes (Aqs, 2008). In general, hierarchical methods can be summarized as follows:

- *Setup* performs per-micropolygon computation for hierarchical traversal.
- *Bound* traverses the tile hierarchy. This step identifies tiles of various sizes that lie completely inside the polygon. It also identifies small tiles on the edge of the polygon.
- *Test* performs *PointInPolygon* tests for tiles on the edge of the polygon.

Hierarchical methods can attain high STE for large polygons because they can often identify large regions samples that yield hits without performing *PointInPolygon* tests for each sample. It is possible to generate more hits than *PointInPolygon* tests, attaining STE $> 100\%$. Although hierarchical methods may incur large setup costs (tile hierarchy traversal), there is often plenty of parallelism across samples within coarse tiles that are completely inside a polygon.

Unfortunately, with micropolygons, hierarchical methods fail to quickly identify coarse tiles that are completely covered for the same reasons tiling methods

have low STE. Micropolygons are almost all edges, or, alternatively stated, the region inside micropolygons are smaller than the most refined tile size. A hierarchical method will traverse the whole tile hierarchy for each micropolygon, only to identify tiles that still lie on a micropolygon's edge. The expensive cost of setup is not justified since STE will still be low and the amount of parallelism within the polygon will be limited.

Chapter 3

Implementation

In this chapter, we present an algorithm for computing polygon visibility and two data-parallel formulations of our algorithm. To be more precise, the visibility problem for micropolygons is framed as follows: we are given many, small polygons (micropolygons) and a structured set of points (frame buffer), and we wish to determine which points are covered by which polygons. We assume the polygons are either triangles or quads (however, we continue to use the word micropolygon in lieu of microtriangle or microquad).

3.1 Algorithm

Expensive setup and diminishing returns on parallelism within a polygon make previous algorithms inefficient for micropolygons. Moreover, due to the small size of miropolygons, the number of micropolygons generated to represent a scene will be much larger than the number of polygons generated to represent a scene.

We choose an algorithm specifically tailored for micropolygon workloads. Per-micropolygon setup is minimized to deal with an effect of Amdahl's Law (Hennessy & Patterson, 2002): per-micropolygon setup cost is a lower bound on total rasterization cost. In our algorithm, *Setup*/*Bound* computes a screen-space

RASTMP

```
Cull backfacing                          // Setup
BBOX = Compute MP bbox                   // Bound
for each sample in BBOX                  // Test
  hit = test MP against sample
  if hit
    Process hit                          // Process
```

Figure 3.1: Pseudocode for RASTMP.

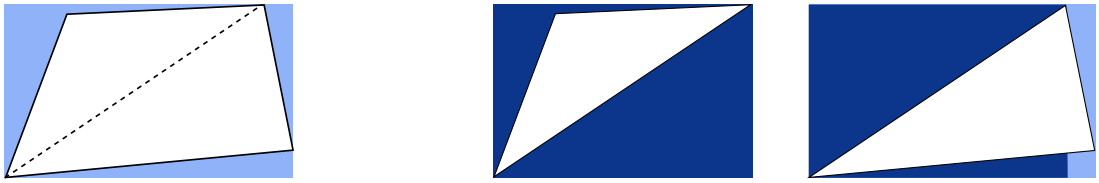


Figure 3.2: Differences in bounding boxes for quads versus two triangles. Light blue regions are tested once whereas dark blue regions are tested twice.

axis-aligned bounding box (AABB) of the polygon. The AABB identifies candidate samples to perform *PointInPolygon* tests. By using jittered sampling, we can quickly retrieve these candidate samples by snapping the AABB to subpixel boundaries. While other simple geometric structures, such as oriented bounding boxes, can be used to identify a tighter set of candidate samples, traversing them generally requires more work.

For each candidate sample, we perform a *PointInPolygon* test. *PointInPolygon* is implemented using edge equations. If a sample is covered by the polygon, we process the micropolygon-sample pair. For the visibility problem, this amounts to returning the micropolygon-sample pair. For rasterization, this amounts to updating the frame buffer with the appropriate color value.

The use of bounding boxes to determine candidate samples motivates supporting quads in our graphics pipeline. First, a bounding box can fit a quad quite well, and in the worst case, cannot be any worse than the union of the bounding boxes of two triangles. For example, traversing the candidate samples of two

neighboring triangles may cause samples to be tested twice. Figure 3.2 illustrates this effect. The light blue regions contain samples that are tested only once that result in misses. The dark blue regions are samples that are needlessly tested twice. Second, a pipeline that supports quad may create less work for the whole pipeline. A tesselator can generate roughly half as many quads for a scene than the number of triangles generated for the same scene.

In most systems, triangles are preferred over quads because triangles are simpler to handle. Triangles are always convex whereas the convexity does not always hold for quads. However, when quads are very small, problems that arise from assuming convexity can be tolerated without objectionable artifacts. (Alternatively, without too much complexity, the *PointInPolygon* test can also be written to correctly handle convex quads.) We continue to frame our solution in terms of triangles; however, our solution is easily extended for quads.

In summary, our algorithm can be broken down into four steps (pseudocode is also provided in figure 3.3):

- *Setup* performs per-micropolygon setup computation, including back-face culling.
- *Bound* computes an AABB for the micropolygon. (Note: this is still considered per-micropolygon setup.)
- *Test* performs *PointInPolygon* tests between samples in the AABB and the micropolygon.
- *Process* updates the frame buffer using color information from the micropolygon vertices.

LOCKSTEP: parallel formulation of (RASTMP)

```
Cull backfacing
Compute edge equations [optional]
BBOXes = ComputeBBoxes(MPs)
S_max = hmax(BBOXes.numSamples)
for i = 0 to S_max
  hits = PointInPolys(s_i, MPs)
  Process(hits, MPs)
```

Figure 3.3: Pseudocode for LOCKSTEP (RASTMP).

3.2 Parallelization

Although micropolygons only cover a few samples, the large number of micropolygons provides a rich source of data-parallelism. We implemented two data-parallel formulations of RASTMP that parallelizes across micropolygons. This approach to parallelization is a departure from common rasterizers, which exploit parallelism within micropolygons as opposed to across micropolygons.

Our implementations run on a simulated vector machine, which supports standard vector instructions for operating on vectors of floats and integers. We also require special instructions for manipulating bit masks and support for masked instructions. We explain these instructions in more detail when needed. We expect our implementations to translate naturally to a virtualized pipeline with vector instructions or to a hardware implementation with many data-parallel execution units.

3.2.1 LOCKSTEP

The first implementation, LOCKSTEP, keeps all of the data-parallel executions of the algorithm in lockstep. For each batch of polygons, *Setup* optionally pre-compute edge equations that are used later in *Test*. The benefits of pre-computating edge equations depend on the architecture. In addition, *Bound* (a portion of *Setup*) computes AABBs for micropolygons. *Test* iterates S_{max} times, where S_{max} is

the maximum number of candidate samples among all bounding boxes. This requires a `hmax` vector instruction that computes the maximum element of a vector. Each iteration, a *PointInPolygon* test is performed s_i where s_i is the i-th sample in the AABB. We mask out lanes when the iteration count exceeds the total number of candidate samples. (Recall that masked out lanes lead to lower utilization.) If an iteration of *Test* yields any hits, *Process* returns the vector of hits. As a small optimization, we can skip *Process* if all lanes fail.

For a rasterizer, *Process* must update the frame buffer with the correct color values from the micropolygon. To determine the color, we have to store additional shading information for each micropolygon vertex. In our system, micropolygons are already shaded; consequently, the rasterizer only needs to maintain the color of each vertex. Combining location data (needed for visibility) and color (needed only for rasterization), each micropolygon vertex requires 28 bytes. For a triangle, this amounts to 84 bytes per micropolygon. A more space-efficient algorithm can share micropolygon vertex data between micropolygons.

Updating the frame buffer in parallel may introduce a data hazard. If two micropolygons hit the same sample (collision), processing cannot proceed in parallel because they will both try to update the frame buffer location. We expect collisions to be rare because of the way a tesselator generates micropolygons. Most tesselators will tesselate patches of a surface at a time. Collisions can only occur when the patch lies on a silhouette edge (which may be created by displacement), or when two patches that overlap in screen-space are tesselated consecutively.

One solution for software implementations can provide two execution paths: one for batches of work without collisions proceeds in parallel and another for batches of work that might have a collision serializes (Owens, 2002). Assuming collision detection is cheap (a method using hashing is proposed) and collisions (and false positives) are rare, this approach is acceptable. A hardware solution to this problem also exists (McCormack *et al.*, 1998). It builds batches of non-colliding hits and processes the batch in parallel when filled or when a collision is about to be introduced. For simplicity, our current implementation serializes all accesses to the frame buffer and performs all other computations in parallel.

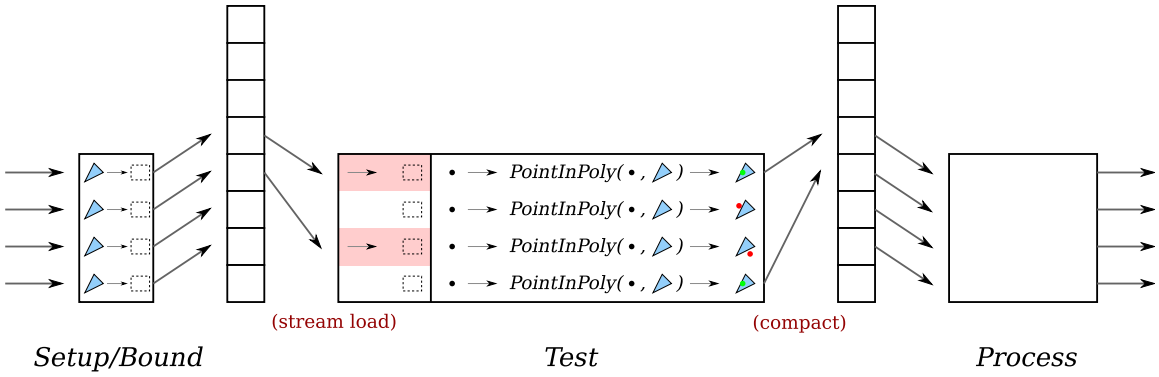


Figure 3.4: Diagram of REPACK. The stages are separated by buffers. *Test* (conditionally) stream loads its input and compacts its output. Buffers are twice as wide as the vector width.

3.2.2 REPACK

Divergent execution (or *divergence*) occurs when vector lanes need to perform different instructions due to branching. A typical solution to divergence masks out lanes of divergent execution. For example, an simple implementation of absolute value on a vector might multiply each negative element by -1. All lanes with a positive element will be masked out. Data-parallel implementations try to remove divergence because masking out lanes causes lower utilization.

There are two main sources of divergence in LOCKSTEP. First, variance in micropolygon spatial extent introduces dynamic loop bounds for *Test*. Although the tessellator upstream attempts to produce micropolygons of similar size and aspect ratio, we still expect both the shape and size of the bounding box to vary. Theoretically, a perfect tessellator could remove divergence by always producing perfect batches of micropolygons with AABBs that cover the same number of samples. In practice, however, keeping all execution units in lockstep will suffer from divergence in *Test* from bounding box variation. Second, a sample is only processed if it passes the *PointInPolygon* test. Divergence due to conditionally processing a hit is related to STE. An algorithm with 100% STE would not suffer any divergence in *Process*.

Our second implementation, REPACK, attempts to remove divergence in LOCKSTEP by compacting work between stages. We separate our work into three stages: *Setup* and *Bound*, *Test*, *Process*. We place buffers before each stage of execution. Figure 3.4 provides an overview of the stages and buffers. When a buffer accumulates enough work to sustain high utilization, the stage executes in parallel over the data. We push hit samples from *Test* into a buffer before processing them to attain high utilization in *Process*.

We address divergence in *Test* that stems from variation in bounding box sizes by *stream loading*. With stream loading, we treat *Test* as a stage that operates on a stream of incoming data (samples within bounding boxes). In each iteration of *Test*, all lanes will iterate through the next sample within the bounding box. If a lane finishes iterating, it is marked as available. At the beginning of each execution of *Test*, new bounding boxes are loaded dynamically into all available lanes. Stream loading allows us to stagger the scan-conversion of bounding boxes and still keep high utilization.

On our vector machine, compacting data to a buffer required a new operation, *PrefixSum*, and a masked *Scatter*. *PrefixSum* takes a mask as input and outputs a vector of `int`'s where the i-th element is the number of bits active in the mask up to and including index i. The *PrefixSum* determines indices into the buffer for the *Scatter*. A stream load is executed using *PrefixSum* and a masked *Gather*. The *PrefixSum* computes indices into the buffer and the masked *Gather* loads the corresponding data into the available lanes. (Stream loading can be viewed as the opposite of compaction.) Since stream loading operations only affect available lanes, it will not run at full utilization. An implementation may choose to only stream load if some fraction of the vector lanes are available. This will lead to less overhead from stream loading; however, *Test* will run at lower utilization overall.

Each buffer between stages is a buffer with a fixed-width that is equal to double the vector width. We have a simple scheduler that runs the latest stage that can operate at full utilization (i.e., its input buffer has at least a vector width amount of work). This scheduler is easy to implement and can (almost) always guarantee full utilization. Moreover, the size of the buffers guarantees that there

will be enough room in the stage's output buffer if all lanes produce outputs. The only time we cannot attain full utilization (excluding stream loading) is when we run out of micropolygons. In this situation, rasterization is almost complete and the only work remaining is flushing all stages of leftover work.

Chapter 4

Results

Since our implementations are architecture agnostic, we use operation counts as an approximation of execution time. Our code is instrumented to measure vector operations as well as utilization. These measurements provide insight into the overall performance of our implementations under various parameters.

For a real implementation, wall clock time and throughput for rasterizing a scene with micropolygons is the real metric of performance. When future hardware is designed, utilization will become a much more important metric than operation counts. An implementation with low utilization will not be cost-effective, power-efficient, nor scalable to wider data-parallelism. A good rasterizer would find a fair balance between STE efficiency and utilization.

Higher multisampling rates can yield high-quality renderings; however, we suspect real-time systems to tolerate the loss in quality for performance. The choice of no multisampling is still interesting for generating shadow maps. Our results are reported for $4\times$MSAA; however, when relevant, we describe how values scale with sampling rate. Intuitively speaking, increasing the sampling roughly corresponds to increasing the size of the micropolygon (where size measured in samples covered, instead of pixels).

We are interested in how our implementation performs with various vector widths, which we denote by W. We conducted our experiments on a library we created that supports data-parallel operations with different fixed vector widths.

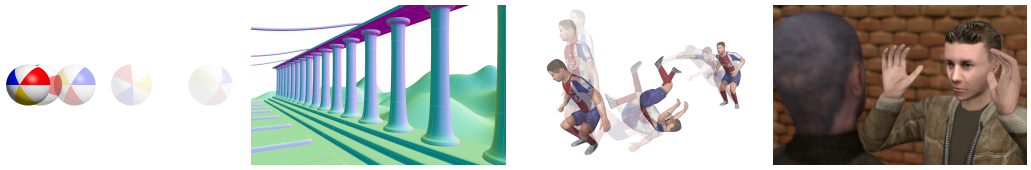


Figure 4.1: Our test scenes are select frames from animation sequences, from left to right: BALLROLL, COLUMNPIVOT, TIGERJUMP, and TALKING.

We counted the total number of operations performed in addition to the utilization of each operation (determined by vector masks). To show how performance scales with vector width, we report results for $W = 8, 16, 32,$ and 64. To put this in perspective, most CPU vector units are 4 wide ($W = 4$), and most GPUs have SIMD units that range from 8 wide to 16 wide (Fatahalian, 2008). Larrabee's instruction set uses vector width 16 (Seiler *et al.*, 2008). Our results report parallel operations (i.e., to obtain the number of mathematical operations, scale the operation counts by W.)

Our results are averaged across test scenes rendered at 1728×1080 resolution. Our scenes are taken from animation sequences scene in figure 4.1. The tesselator targets generating triangles with half-pixel area and quads with pixel area. Unfortunately, we cannot use real data sets and decouple our results from the tesselator output at the same time. However, we do know that our tesselator will produce the exact same vertices when generating either triangles or quads. Unless specified otherwise, we use results using a tesselator that generates triangles. All of our results are reported with backface culling enabled.

4.1 Comparison of Implementations

4.1.1 LOCKSTEP

Table 4.1 provides a breakdown of utilization in each stage of the LOCKSTEP. While utilization is high for *Bound*, it starts to drop for *Test* and more dramatically for

	Vector Width (W)			
	8	16	32	64
Bound	98% (0.10)	98% (0.09)	97% (0.08)	96% (0.08)
Test	83% (0.68)	78% (0.65)	74% (0.64)	70% (0.63)
Process	25% (0.23)	20% (0.26)	17% (0.28)	15% (0.29)
Overall Util	71%	65%	60%	56%
Overall Op/MP	61.94	34.11	18.44	9.87

Table 4.1: Breakdown of utilization per stage and fraction of time spent in each stage (in parenthesis) for the LOCKSTEP. Overall utilization falls off quickly for larger vector widths. The decrease in total operations per micropolygon shows benefit of parallelism. *Setup* is omitted because it constitutes less than 0.01 of the runtime.

Process. These are the main areas of divergence in our algorithm. *Test* dominates the runtime, constituting about two-thirds of the total runtime, and, consequently, its utilization is the most important data point. Note that utilization in *Bound* is independent of multisampling rate. Both *Test* and *Process* have higher utilization with higher mutlisampling rates.

Execution in *Test* diverges because of variation in bounding boxes. As we process more micropolygons in parallel, divergence from irregular bounding box sizes becomes more severe. For example, consider a micropolygon with an unusually large bounding box. When this bounding box is the only lane active in *Test*, utilization will be $\frac{1}{W}$. Hence, lower utilization in *Test* due to bounding box variation is more pronounced for larger W.

Process runs at low utilization because each micropolygon only covers a few samples. Each iteration of *Test* will only yield a few hits (roughly STE). As a small optimization, if an iteration of *Test* does not output any hits, we can completely skip *Process*. An even better optimization comes from observing that since *Process* follows *Test*, its utilization is modulated by *Test*'s utilization. We can decouple the utilization of *Test* and *Process* by aggregating hits from *Test* into a buffer. When enough work accumulates, *Process* can process all these hits in parallel.

	Vector Width (W)			
	8	16	32	64
Bound	98% (0.07)	98% (0.07)	97% (0.07)	96% (0.06)
Test Stream-load	20% (0.08)	15% (0.11)	13% (0.12)	12% (0.13)
Test	99% (0.77)	99% (0.75)	99% (0.73)	97% (0.73)
Process	99% (0.08)	99% (0.08)	99% (0.08)	98% (0.08)
Overall Util	93%	90%	88%	86%
Overall Op/MP	71.31	36.77	18.83	9.58

Table 4.2: Breakdown of utilization per stage and fraction of time spent in each stage (in parenthesis) for the REPACK. We break out the stream loading step, *Test Stream-load*.

	Overhead (%)
Bound	21%
Test Stream-load	100%
Test	26%
Process	35%

Table 4.3: Overhead for each stage of the pipeline. Note that all instructions in *Test Stream-load* are considered overhead.

4.1.2 REPACK

The REPACK is able to maintain high utilization without incurring too much overhead. Table 4.2 gives utilization results for the REPACK. We report stream-loading in *Test* as a separate data point, *Test Stream-load*. The low utilization of *Test Stream-load* actually lowers the overall utilization of *Test*. (If we were to combine these results, *Test* utilization is closer to 75%, as opposed to 99%.) We can trade off utilization between *Test Stream-load* and *Test* by changing how often new stream-loading occurs. For example, we can force *Test Stream-load* to execute only when it will run with at least 50% utilization. Consequently, *Test*'s utilization will decrease because not all iterations will be at full utilization. However, these optimizations are architecture-specific and are therefore not discussed in this thesis.

Table 4.3 shows the amount of overhead incurred in each stage in our particular implementation. We define overhead as any extra operations the REPACK executes that the LOCKSTEP did not have to. Note that all operations in *Test Streamload* should be considered overhead in *Test*. The amount of overhead incurred for any particular architecture varies. Our results suggest that the overhead incurred eventually pays back by yielding higher utilization in *Process*. The actual crossover point is architecture dependent.

As a consequence of deferring computation, micropolygon data must be retained for each micropolygon in flight in the pipeline. In our system, micropolygons are already shaded–no data for shading computations is passed along. Each micropolygon vertex contains position and color information (28 bytes per vertex). Although each micropolygon has a smaller memory footprint than a polygon in a modern rasterizer, it is not reasonable to replicate this data at each stage of the pipeline.

The average working set is about $3.5W$. Since each stage is effectively at full utilization, queues before each stage are filled with at least W work. The two buffers add roughly $2W$ micropolygons to the active working set. The *Test* stage also keeps W micropolygons active when iterating through samples. The remaining micropolygons in the working set are from new micropolygons that have been loaded in. Note that by design, the maximum working set size is $4W$. This result suggests that memory accesses and cache coherency may be a limiting factor for software implementations as W increases. Hardware implementations will have the problem of managing the set of active micropolygons.

Since our implementation serializes access to the frame buffer, conflicts in *Process* do not occur. A real implementation needs to handle conflicts when they do occur. Conflicts are extremely rare: we measure that conflicts occur less than 1% of the times, even with $W = 64$. However, depending on the detection scheme, false positives may be common. We do not evaluate any collision detection schemes and but note that it may be an important factor for actual implementations.

	Vector Width (W)			
	8	16	32	64
Bound	98% (0.07)	97% (0.06)	96% (0.06)	95% (0.06)
Test	79% (0.68)	74% (0.66)	70% (0.65)	65% (0.65)
Process	36% (0.26)	31% (0.28)	27% (0.29)	25% (0.30)
Overall Util	69%	63%	59%	55%
Overall Op/MP	102.25	55.60	29.89	16.16

Table 4.4: Breakdown of utilization per stage and fraction of time spent in each stage (in parenthesis) for the LOCKSTEP when using quads instead of triangles.

	Vector Width (W)			
	8	16	32	64
Bound	98% (0.05)	97% (0.05)	96% (0.05)	95% (0.05)
Test Stream-load	17% (0.06)	12% (0.09)	10% (0.11)	9% (0.11)
Test	99% (0.77)	99% (0.75)	98% (0.73)	97% (0.73)
Process	99% (0.12)	99% (0.12)	99% (0.11)	98% (0.11)
Overall Util	94%	91%	89%	87%
Overall Op/MP	106.73	54.98	28.16	14.35

Table 4.5: Breakdown of utilization per stage and fraction of time spent in each stage (in parenthesis) for the REPACK when using quads instead of triangles.

4.2 Support for Rasterizing Quads

We repeated our experiment with our tesselator outputting quads instead of triangles. Table 4.4 and table 4.5 show that, in terms of utilization, quads behave similarly in both LOCKSTEP and REPACK. Overall utilization is higher primarily because of increased utilization in *Process*. Recall that utilization in *Process* is correlated with STE. We measured quads as having roughly $1.5\times$ better STE than do triangles. In addition, a tesselator should generate about half as many quads. For a scalar algorithm, these two effects would suggest nearly a $3\times$ improvement.

Unfortunately, we found that supporting quads only gave a 20% speedup (in terms of total operations). Figure 4.2 shows the total operations for various scenes. Two main effects limit the benefits of supporting quads. First, we end up with the

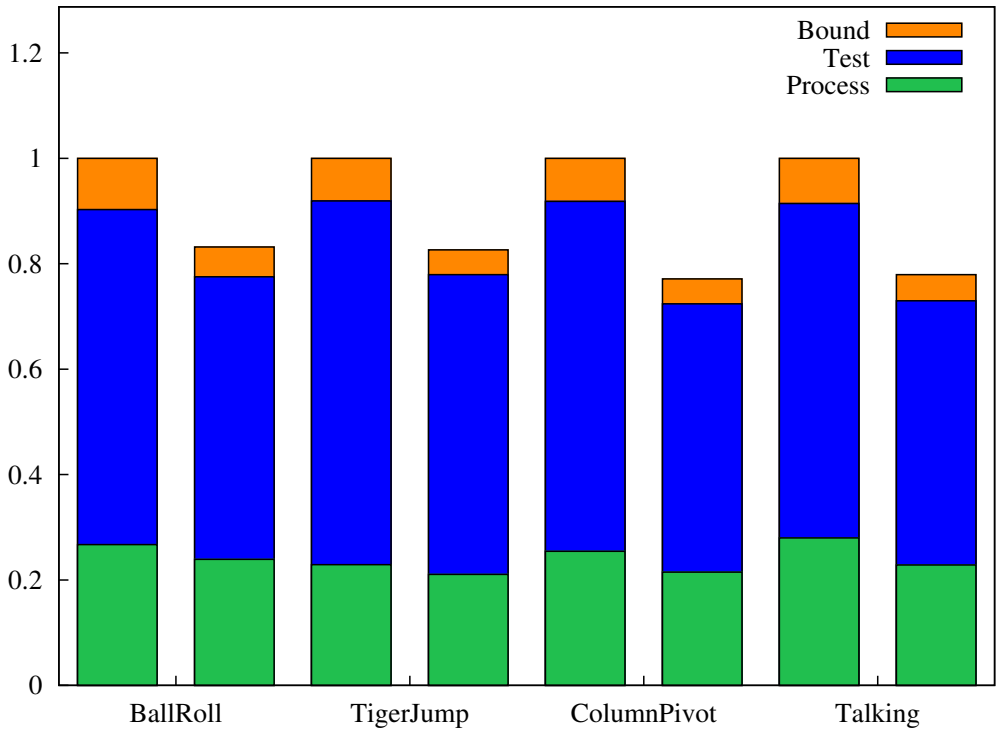


Figure 4.2: Comparison of total operations between triangles and quads for different scenes. For each scene, the column on the left is for triangles and the right column is for quads. The quad column is scaled relative to the triangle column. Note that the amount of work in *Process* is roughly equal. Time spent in *Bound* is halved since there are half as many micropolygons.

	Vector Width (W)			
	8	16	32	64
TRIANGLES (19% STE, 5.5M micropolygons)				
Bound	5.9	3.0	1.5	0.8
Test	42.0	22.3	11.8	6.3
Process	14.0	8.8	5.1	2.8
Total	61.9	34.1	18.4	9.9
QUADS (30% STE, 2.75M micropolygons)				
Bound	6.9	3.5	1.7	0.9
Test	69.1	36.7	19.5	10.5
Process	26.2	15.4	8.6	4.8
Total	102.2	55.6	29.9	16.2

Table 4.6: Difference in operations per micropolygon when using triangles versus quads.

same number of hits, regardless of which primitive we rasterize. Therefore, the amount of work in *Process* is basically fixed. *Process* can only be improved by increased utilization, which is only marginal. Amdahl's law limits the speedup of halving the number of micropolygons. Second, quads require more operations to rasterize. Table 4.6 shows a breakdown for operations per micropolygon for both triangles and quads. Overall, quads require about $1.6\times$ as many operations to rasterize.

4.3 Discussion

The algorithm we presented picked AABBs to identify candidate samples. While STE is only 20% for triangles, AABBs require very few operations to compute. Due to the low number of samples covered, other methods of identify candidate samples are not likely to be able to make up for setup costs in *Test* and *Process*.

Our results show that rasterizing micropolygons is very expensive. For example, with $W = 16$ and $4\times$MSAA, each micropolygon on average requires about 38

operations. (At $W = 16$, LOCKSTEP and REPACK have almost the same operations per micropolygon.) To render 5 million micropolygons (half-pixel area triangles, 1728×1080 resolution, scene depth complexity 2.5, half backface culled), at 60 frames per second requires 22 billion operations per second. This is equivalent to 20 Larrabee units (Seiler *et al.*, 2008), making rasterization a strong candidate for a fixed-function hardware implementation.

For the same scene using pixel-area quads brings this cost down to 16 Larrabee units. Although quads require more operations per micropolygon, our results show that support for quads translate into a significant net benefit in terms of the total number of operations needed in rasterization. Since not all high-level primitives can be easily tesselated into quads, we provide support for both triangles and quads.

Although we present two parallel formulations of our algorithm, the best formulation depends on the architecture. A hybrid approach of parallelizing across micropolygons as well as within micropolygons (larger raster stamp for testing) may yield the best results. However, these decisions depend heavily on the target hardware.

Chapter 5

Conclusion

5.1 Contributions

Determining visibility is the most difficult step for rasterizers. This thesis focuses on an algorithm, RASTMP, for point sampling a micropolygon for visibility. RASTMP is optimized for micropolygons by minimizing setup cost which must be paid per-micropolygon. Due to the small size of micropolygons, computing an axis-aligned bounding box is as efficient as previous alogithms for finding candidate samples and is much cheaper to compute. RASTMP works for both triangles and quads and we show that supporting quads leads to a direct 20% gain in rasterizer performance.

This thesis also presents two different parallel formulations of RASTMP. A detailed study of our formulations' simulated performance shows that both formulations attain high utilization. LOCKSTEP, which keeps all vector lanes in lockstep, works well for small vector widths. REPACK, which compacts work between stages, can maintain high utilization for large vector widths; however, REPACK incurs a fixed fraction of overhead. The actual cost of the overhead is architecture dependent.

5.2 Future Work

Support for cinematic effects such as motion blur and camera defocus is highly desired future graphics pipelines. Rasterizing a micropolygon with motion blur is equivalent to point sampling a prism in 3D (X, Y, T) space. Adding depth of field makes this problem equivalent to point sampling in 5D (X, Y, T, U, V). We have extended LOCKSTEP to support both 3D and 5D rasterization of micropolygons. The same extension for REPACK is partially implemented. A similarly detailed treatment of runtime characteristics of both formulations is an obvious next step.

Our current implementation of REPACK does not address how to maintain micropolygon data. One of the benefits of the REYES system is being able to keep a low memory footprint by streaming over data. REPACK defers computation which requires keeping micropolygon data around. Currently, we determine when a micropolygon will no longer be used by reference counting. This approach will be prohibitively expensive for any high-performance implementation.

This thesis does not address thread-level parallelism. Future work may examine how each stage of REPACK can be moved to different cores. This would require a further study into how to properly balance work between the stages.

Quads helped increase STE by preventing a sample that hit one triangle from being tested by a neighboring triangle. An interesting extension to this rasterizes patches of neighboring micropolygons at a time. A patch of micropolygons is larger, and thus, previous techniques for large polygons may apply for the whole patch. Once a candidate set of samples for the whole patch is determined, it may be possible to efficiently match up micropolygons to covered samples.

References

2008. *Aqsis: Sampling Wiki.* `http://wiki.aqsis.org/doku.php?id=dev:sampling&rev=1215038504`.

2009a. Data-Parallel Rasterization of Micropolygons With Defocus and Motion Blur. *In: High Performance Graphics 2009, submitted paper by authors.*

2009b. DiagSplit: Parallel, Crack-Free, Adaptive Tessellation for Micropolygon Rendering. *In: SIGGRAPH Asia 2009, submitted paper by authors.*

2009. *Introduction to rendering in Houdini.* `http://www.sidefx.com/docs/houdini8/content/base/render_about.xml`.

Apodaca, Anthony A., & Gritz, Larry. 2000. *Advanced RenderMan: Creating CGI for Motion Pictures.* Morgan Kaufmann.

Cook, Robert L. 1986. Stochastic sampling in computer graphics. *ACM Trans. Graph.*, **5**(1), 51–72.

Cook, Robert L., Carpenter, Loren, & Catmull, Edwin. 1987. The Reyes image rendering architecture. *SIGGRAPH Comput. Graph.*, **21**(4), 95–102.

Fatahalian, Kayvon. 2008. *From Shader Code to a Teraflop: How Shader Cores Work.* SIGGRAPH 2008 Class Notes: Beyond Programmable Shading: Fundamentals. `http://s08.idav.ucdavis.edu/fatahalian-gpu-architecture.pdf`.

Fuchs, Henry, Goldfeather, J., Hultquist, J. P., Spach, S., Austin, John, Frederick P. Brooks, Jr., Eyles, John, & Poulton, John. 1986. Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes. *Pages 169–187 of: Advances in Computer Graphics I (Tutorials from Eurographics'84 and Eurographics'85 Conf.)*. London, UK: Springer-Verlag.

Fuchs, Henry, Poulton, John, Eyles, John, Greer, Trey, Goldfeather, Jack, Ellsworth, David, Molnar, Steve, Turk, Greg, Tebbs, Brice, & Israel, Laura. 1989. Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. *SIGGRAPH Comput. Graph.*, **23**(3), 79–88.

Greene, Ned. 1996. Hierarchical polygon tiling with coverage masks. *Pages 65–74 of: SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM.

Hennessy, John L., & Patterson, David A. 2002. *Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann.

McCool, Michael D., Wales, Chris, & Moule, Kevin. 2001. Incremental and hierarchical Hilbert order edge equation polygon rasterizatione. *Pages 65–72 of: HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*. New York, NY, USA: ACM.

McCormack, Joel, & McNamara, Robert. 2000. Tiled polygon traversal using half-plane edge functions. *Pages 15–21 of: HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*. New York, NY, USA: ACM.

McCormack, Joel, McNamara, Robert, Gianos, Christopher, Seiler, Larry, Jouppi, Norman P., & Correll, Ken. 1998. Neon: a single-chip 3D workstation graphics accelerator. *Pages 123–132 of: HWWS '98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*. New York, NY, USA: ACM.

Owens, John D. 2002 (Nov.). *Computer Graphics on a Stream Architecture*. Ph.D. thesis, Stanford University.

Pineda, Juan. 1988. A parallel algorithm for polygon rasterization. *SIGGRAPH Comput. Graph.*, **22**(4), 17–20.

Seiler, Larry, Carmean, Doug, Sprangle, Eric, Forsyth, Tom, Abrash, Michael, Dubey, Pradeep, Junkins, Stephen, Lake, Adam, Sugerman, Jeremy, Cavin, Robert, Espasa, Roger, Grochowski, Ed, Juan, Toni, & Hanrahan, Pat. 2008. Larrabee: a many-core x86 architecture for visual computing. *Pages 1–15 of: SIGGRAPH '08: ACM SIGGRAPH 2008 papers*. New York, NY, USA: ACM.