Graphics Hardware (2002)
Thomas Ertl, Wolfgang Heidrich, and Michael Doggett (Editors)

Comparing Reyes and OpenGL on a Stream Architecture

John D. Owens, Brucek Khailany, Brian Towles, and William J. Dally[†]

Stanford University Computer Systems Laboratory

Abstract

The OpenGL and Reyes rendering pipelines each render complex scenes from similar scene descriptions but differ in their internal pipeline organizations. While the OpenGL organization has dominated hardware architectures over the past twenty years, a Reyes organization differs in several important ways from OpenGL, including a shader coordinate system that supports coherent texture accesses, a single shader in the vertex stage, and tessellation and sampling instead of triangle rasterization.

Hardware for the OpenGL pipeline has been well-studied, but the lack of a hardware Reyes implementation has prevented a comparison between the two pipelines. We analyze and compare implementations of an OpenGL and a Reyes pipeline on the Imagine stream processor, a high performance programmable processor for media applications. This comparison both demonstrates the applicability of Reyes for hardware implementation and exposes many issues that architects will face in implementing Reyes in hardware, in particular the need for efficient subdivision algorithms and implementations.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors; C.1.2 [Processor Architectures]: Multiple Data Stream Architectures—Single-instruction-stream, multiple-data-stream processors (SIMD).

1. Introduction

Graphics hardware continues to make remarkable gains in performance, but the fundamental organization of today's consumer graphics hardware has changed little from the original IRIS pipeline first developed by Silicon Graphics in the early 1980's. The OpenGL graphics system[16], a widely used standard for interactive graphics applications, is a direct descendant of this early hardware. However, the efficiency and flexibility of these simple, triangle-based pipelines are taxed under the current trends of decreasing triangle size and the demand for complex, programmable shaders. This naturally leads to the question of how alternative pipelines lend themselves to high-performance implementations. At one extreme of the performance-realism spectrum is the Reyes rendering pipeline[4]. Reyes was designed at Lucasfilm to render extremely complex scenes with total emphasis on the photorealistic, high-fidelity imagery targeted by today's real-time graphics hardware.

In this paper, we examine the issues associated in implementing the Reyes rendering pipeline with the goal of real-time frame rates. We also identify several key characteristics of a pipeline that contribute to an efficient implementation: arithmetic intensity, data reference locality, predictable memory access patterns, and instruction- and data-level parallelism.

The Reyes pipeline, as well as other graphics pipelines, already contains abundant parallelism as well as high arithmetic intensity (the number of operations per fragment). In addition, the uniform size of the rasterization primitives, called *micropolygons*, produces a predictable number of fragments, which simplifies memory allocation and streamlines the rasterization and fragment-processing steps. We detail our rasterization algorithm, which takes advantage of these uniform primitives.

Another key aspect of the Reyes pipeline is support of high-level primitives, such as subdivision surfaces. While subdivision surfaces significantly reduce the amount of

[†] Gates Computer Science Building 4A, Stanford, CA 94305 USA; {jowens, khailany, btowles, billd}@cva.stanford.edu

memory bandwidth consumed by loading models, they also introduce additional control complexity. This is especially true for adaptive subdivision schemes because crack prevention traditionally requires elaborate stitching schemes and non-local knowledge of the surface. We address these problems by introducing a novel crack prevention algorithm that stores edge equations of the micropolygons instead of their vertices.

Finally, we compare our implementation of the Reyes pipeline with an OpenGL pipeline on the Imagine Stream Processor[9]. Mapping applications to Imagine naturally exposes the characteristics we identified for an efficient pipeline implementation. Using Imagine as a common substrate for comparison on several scenes, we identify the relative strengths and weakness of both approaches. We show that on several scenes with complex shading, OpenGL delivers superior performance to Reyes. Specifically, we found that the high computational cost of the subdivision and the large number of micropolygons produced that did not contribute to the final image are the primary impediments to making a Reyes implementation competitive with OpenGL.

We begin in Section 2 by describing the Reyes and OpenGL pipelines. Section 3 describes the fundamentals of our implementation and Section 4 the details of our Reyes implementation. In Section 5 we explain our experimental setup and our test scenes. Finally, Section 6 analyzes and compares the results of our implementations of the OpenGL and Reyes pipelines.

2. OpenGL and Reyes rendering pipelines

2.1. OpenGL

The OpenGL graphics system[16] is a widely used standard for interactive graphics applications. OpenGL is well suited for hardware implementations and most modern real-time graphics hardware supports an OpenGL interface.

The OpenGL pipeline consists of the following stages:

Transformations and vertex operations Objects are specified in object space and are transformed to eye space, where per-vertex operations, such as lighting and other shading operations, are performed. Recent hardware has added user programmability to this stage[10].

Assemble/Clip/Project Triangles are assembled from vertices, transformed to clip space, clipped against the view frustum, and projected to the screen.

Rasterize Screen-space triangles are rasterized to fragments, all in screen space. Per-vertex parameters are interpolated across the triangle.

Fragment operations Per-fragment operations, such as texturing and blending, are applied to each fragment. Like the vertex operations stage, this stage supports increasing user programmability.

Visibility/Filter Visibility is resolved in this stage, usually through a depth buffer, and fragments are filtered and composited into a final image.

2.2. Reyes

The Reyes image rendering system[4] was developed at Lucasfilm and Pixar for high-quality rendering of complex scenes. The system, developed in the mid-1980's, was not designed at the time for real-time rendering but instead for rendering more complex scenes with higher image quality and rendering times from minutes to hours. Reyes is the basis for Pixar's implementation of the RenderMan rendering interface[17].

The Reyes pipeline has four main stages, described below. The primary rendering primitive used by Reyes is the *micropolygon* or *quad*, a flat-shaded quadrilateral. In the original Reyes implementation, quads were no larger than 1/2 pixel on a side[†], but typical quads in modern Reyes-like implementations are on the order of 1 pixel in area because modern shaders are self-antialiasing.

Dice/Split Inputs to the Reyes pipeline are typically higher-order surfaces such as bicubic patches, but Reyes supports a large number of input primitives. Primitives can split themselves into other primitives, but each primitive must ultimately dice itself into micropolygons. Dicing is performed in eye space[†].

Shade Shading is also done in eye space by procedurally specified shaders. Because micropolygons are small, each micropolygon is flat-shaded.

Sample Micropolygons are projected to screen space, sampled, and clipped to the visible view area. Reyes uses a stochastic sampling method with a variable number of subpixels per pixel (16 in the original Reyes description[†]).

Visibility/Filter Visibility is resolved using a depth buffer with one sample per subpixel, then the visible surface subpixel values are filtered to form the final image.

2.3. Differences between OpenGL and Reyes

Both pipelines are similar in function, as shown in Figure 1. In both, the application produces geometry in object coordinates, which are then shaded, projected into screen space, rasterized into fragments, and composited using a depth buffer. However, the pipelines differ in three important ways: shading space, texture access characteristics, and the method of rasterization.

- The OpenGL pipeline shades at two stages: on vertices in eye coordinates (typically lighting calculations, and more recently programmable vertex shaders), and on fragments in screen coordinates (typically textures and blends, and more recently programmable fragment shaders). The Reyes pipeline supports a single shading stage on micropolygon vertices in eye coordinates.
- In OpenGL, texturing is a per-fragment screen-space operation; to avoid visual artifacts, textures are filtered using mipmapping[19]. Texture accesses are incoherent from texel

[†] Our implementation differs from the traditional Reyes approach in quad size, subsampling, and dice space, as described in Section 4.

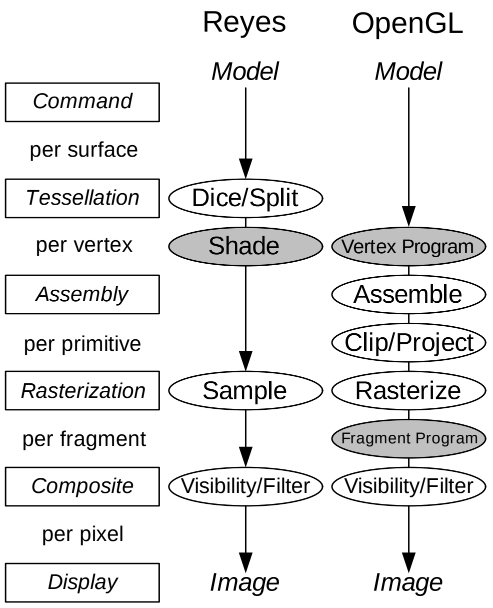

Figure 1: *Reyes vs. OpenGL. The left column shows stages in a generic pipeline; the middle and right columns show the specific stages in Reyes and OpenGL. Shaded stages are programmable.*

to texel. In Reyes, texturing is a per-vertex eye-space operation. Reyes supports “coherent access textures” (CATs) for many classes of textures including standard projective textures. CATs require surface dicing at power-of-two resolutions but cause texels in the texture pyramid to align exactly with the vertices of the quads. Therefore, when accessing CATs, texture filtering is unnecessary, and texels in adjacent micropolygons can be accessed sequentially with significant savings in computation and memory bandwidth. Not all textures can be accessed coherently — non-CATs include environment maps, bump maps, and decals.

- OpenGL’s primitive is a triangle. Objects are specified in triangles, and the rasterization stage of OpenGL must be able to rasterize triangles of arbitrary size. The primitive for the Reyes pipeline is the micropolygon, whose size is bounded in screen space to a half-pixel on a side. Micropolygons are created in eye space during the dice stage of the pipeline, so the dicer must make estimates of the screen coverage of the generated micropolygons.

The fundamental differences in Reyes — single shader, coherent texture accesses, and bounded primitives — are all desirable properties for hardware implementation. Supporting only one programmable unit rather than two is a simpler task for hardware designers. Coherent accessed textures both reduce texture demand from the memory system and increase achievable memory bandwidth.

Bounded-size primitives (particularly small ones such as those in Reyes) have several advantages. The rasterizer does not have to handle the complexity of arbitrary sized primitives, so bounded-sized primitives can be rasterized with simpler algorithms and hardware than unbounded ones. Moreover, every sample within a bounded primitive can be computed in parallel because the total number of possible samples is small and bounded. Evaluating several bounded-size primitives in parallel load-balances better than unbounded primitives. And storage requirements for generated samples are much more easily determined with bounded primitives than with unbounded ones.

2.4. Graphics Trends and Issues

How do these pipelines cope with the issues facing graphics hardware designers of today?

2.4.1. Decreasing Triangle Size

As graphics hardware has increased in capability, models have become more detailed, and triangle size has decreased. The efficiency of factoring work between the vertex and fragment levels in OpenGL is one of the primary reasons that it is the dominant hardware organization today.

The shading work in OpenGL pipelines is divided between vertices and fragments. Vertex-level shading calculations are performed on each vertex. These results are interpolated during rasterization and then used as inputs to the fragment shader, which evaluates a shading function on each fragment.

Because interpolation is cheaper than evaluating the entire shading function at each fragment, and because the number of fragments in a scene is typically many times the number of triangles, this factorization of shading work into the vertex and fragment levels reduces the overall amount of work for the scene.

However, as triangle size continues to decrease, the benefits of this factorization become less significant. For scenes in which the average triangle size is 1 (the numbers of triangles and fragments are equal), there is no benefit.

2.4.2. Host and Memory Bandwidth

Host and memory bandwidth are both precious in modern graphics processors. Reducing the necessary memory bandwidth is achieved by a variety of techniques as texture caches[5], prefetching[6], and memory reference reordering[15]. Parallel interfaces are among the methods used to reduce host bandwidth[7].

The Reyes pipeline, by natively supporting higher-order datatypes, can reduce host bandwidth over sending a stream of triangles. And Reyes’ coherent access textures can also help reduce the necessary bandwidth to texture memory.

3. Implementation Fundamentals

To compare the Reyes and OpenGL pipelines, we implemented each pipeline in the stream programming model for execution on the Imagine Stream Processor. The programmable shading and lighting code was generated from the Stanford Real-Time Shading Language compiler.

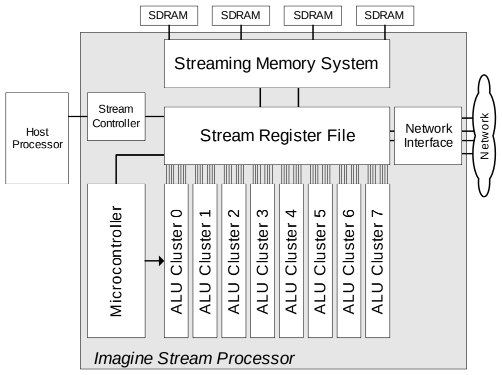

Figure 2: *The Imagine Stream Processor.*

3.1. The Imagine Stream Processor

The Imagine Stream Processor[9] is a high-performance stream coprocessor designed to run media applications. Imagine's architectural primitive is the *stream*, a set of ordered elements of the same datatype. Streams are processed by *kernels* which typically evaluate a function on each element of a stream.

Imagine's block diagram is shown in Figure 2. It consists of a 128 KB stream register file (SRF), 8 SIMD-controlled VLIW arithmetic clusters each containing multiple functional units and controlled by a single microcontroller, and a memory system interface to off-chip DRAM. These modules are all controlled by an on-chip stream controller under the direction of an external host processor.

In an Imagine application, the working set of streams is located in the SRF, which is connected to all modules on Imagine. Stream loads and stores occur between the memory system and the SRF, and the SRF provides the stream inputs to kernels and stores the kernels' stream outputs.

The kernels are evaluated in the 8 arithmetic clusters. Each cluster contains several functional units (providing instruction-level parallelism). The 8 clusters (providing data-level parallelism) are controlled by the microcontroller, which supplies the same instruction stream to each cluster.

On Imagine, streams are stored as contiguous blocks of memory in the SRF or in memory. Kernels are implemented as programs run on the arithmetic clusters. Kernel microcode is stored in the microcontroller. An Imagine application consists of a chain of kernels that process one or more streams. The kernels are run one at a time, processing their input streams and producing output streams. After each kernel is complete, its output is typically input into the next kernel.

Imagine chips were delivered in April 2002; system bringup is actively under way.

3.2. Imagine vs. Graphics Processors

Imagine has many similarities with modern graphics processors in using many of the techniques of modern special-purpose hardware to achieve good performance.

First, both Imagine and modern graphics processors exploit the native parallelism in graphics applications to achieve high performance. Both processors use instruction-level parallelism (functional units working in parallel on the same data element) and data-level parallelism (multiple data elements processed at the same time). Also, modern graphics processors often exhibit task-level parallelism, working on different parts of the graphics pipeline at the same time. Even the APIs are becoming explicitly parallel — NVIDIA's vertex programs[10] are designed for SIMD execution, and Imagine's KernelC kernel language is targeted at SIMD-controlled arithmetic clusters.

In a lengthy pipeline such as rendering, maintaining load balance between stages is crucial to high performance. In modern graphics processors, data buffers between pipeline stages smooth out short-term load imbalances. Imagine's stream register file also buffers intermediate results, although the SRF is used to buffer between stages separated by time instead of between different functional units on the same chip.

In both systems, these buffers are also used to capture the producer-consumer locality of intermediate data on the same chip and to hide latency in the pipeline, particularly memory system latency.

One of the major differences between Imagine and modern graphics processors is Imagine's lack of specialization for rendering. It contains no special purpose hardware directed at graphics applications. Though the lack of specialization hurts Imagine's performance compared to modern graphics processors, when comparing graphics algorithms, it does make Imagine performance-neutral to the algorithms employed. Algorithms that run efficiently on Imagine are likely to run efficiently in hardware specialized to that task, and Imagine's opcounts and runtimes are an accurate assessment of the work involved in running the algorithm. For these reasons, Imagine is an appropriate platform for comparing different rendering algorithms toward an eventual goal of high-performance hardware implementations.

3.3. The Stanford Real-Time Shading Language

Programmable shading is supported by most modern graphics hardware, but the interfaces to the shading functionality differ greatly from vendor to vendor and also change over time. In addition, the native interfaces to this hardware are at a low level.

The Stanford Real-Time Shading Language (RTSL) allows programmers to specify a high-level shading description which can then be targeted to multiple pieces of hardware, including ATI, NVIDIA, and x86 processors[13]. The shading description is parsed into an intermediate representation and then compiled to one or more hardware targets.

We developed back ends to the RTSL front end that target Imagine OpenGL and Reyes shading stages. The OpenGL back end generates code for programs run on each vertex and each fragment, and the Reyes back end generates code that runs on each vertex. The same RTSL shader descriptions can generate both OpenGL and Reyes target code.

3.4. Imagine OpenGL Implementation

Our OpenGL implementation on Imagine is based on the implementation of Owens et al.[12] with several differences. All shading is programmable, with kernel code generated directly by our RTSL back end. The entire pipeline is structured around this programmable shading, whose SIMD nature matches well with Imagine's capabilities. The performance impact of adding generated shader code as opposed to custom shader code is negligible. The rasterizer is now implemented with a barycentric algorithm instead of the previous scanline algorithm. Clipping is also supported. Finally, stream-level code is scheduled by a profiling stream scheduler[8] instead of our previous macrocode implementation.

4. Imagine Reyes Implementation

Our Reyes implementation follows the description in Section 2.2. We begin by projecting the control points of the input B-spline to screen space. We then subdivide in screen space, ensuring that no quad is larger than a fixed size. Because in this implementation we do not support supersampling, we do not subdivide all the way to Reyes' traditional 0.5 pixel area limit. The effects of different subdivision limits are discussed in Section 6.2.

After subdivision, we transform the resulting quad positions and normals back into eye space for shading. This differs from the traditional Reyes implementation, which subdivides in eye space with knowledge of screen space. Quads are then shaded in eye space using the RTSL-generated shader. Next, the sampling kernel inputs quads and outputs fragments, which are composited to make the final image.

4.1. Subdivision

The subdivision step of the Reyes pipeline is responsible for dividing the high-level primitives into micropolygons. We chose to implement the Catmull-Clark[2] subdivision rules to refine these high-level surfaces[‡], allowing native support of subdivision, B-Spline, and Bezier surfaces. The subdivision boundary rules are also included, so effects such as sharp creases in the subdivision surfaces are possible.

Subdivision begins with a collection of control points for the surface. In general, these points can be arranged in an arbitrary topology, but for our implementation we assume a quadrilateral mesh. The subdivision kernel begins by selecting an initial quad from the control mesh and computing the side lengths of that quad. If all the side lengths fall below the stopping threshold[§], the quad is considered complete and is output by the kernel. Otherwise, the quad must be subdivided further, producing four, smaller quads. Now, one of these quads is selected and tested for completion. This process continues as a depth-first traversal of the quad-tree associated with the original control mesh with the leaf depth determined by the screen size of the associated quad. Since each stage of the subdivision requires testing the side lengths in pixels of the quads, this step is naturally performed in screen space.

The depth-first traversal has a key storage advantage — the number of live data values needed at any one time to produce a final set of N fragments is $O(\log N)$. This is in contrast to a triangle-based approach, where each rasterization primitive is part of the input set and therefore $O(N)$ live values are needed to produce the N fragments of a complex surface. While the storage efficiency is useful in any Reyes implementation, it is especially important in an efficient hardware implementation: the ability to produce a large number of fragments from a small amount of control information saves critical memory bandwidth.

However, realizing this savings in memory bandwidth presents several implementation challenges. A naive adaptive subdivision algorithm could use a completely local stopping criterion. However, if neighboring quads are subdivided at different levels, cracks in the final surface can appear. Typical algorithms for eliminating cracks involve a stitching pattern to reconnect the surface[11], which can be used in concert with a global rule such as limiting the difference in subdivision levels for neighboring quads[20]. However, these approaches are not attractive in stream processors for several reasons. First, the control decisions and number of possible stitching patterns make an efficient data-parallel implementation difficult. Secondly, using global information to determine the stitching pattern can defeat the $O(\log N)$ storage requirement and also increase the complexity of memory access patterns, while also decreasing the efficiency of the stream implementation.

We tackle the problem of surface cracks by implementing a novel and completely localized solution: instead of describing the final micropolygons using their corner vertices, they are represented using four edge equations. During subdivision, edge lengths are continually tested to determine if a quadrilateral requires further refinement. Instead of waiting until all four edges meet the length threshold, our approach freezes the final edge equations of a quad *immediately* after they fall below the threshold. Once all four edges have been stored, the final quad is output. This implies that the four edge equations may come from different levels of refinement. However, the edges of a quad are always consistent with its neighbors because the length criterion used on an edge shared between two quads is consistent. This consistency between shared edges is sufficient to prevent cracks in the final surface.

An example of this algorithm in shown in Figure 3. First, the control points of the mesh at level i are shown as unfilled circles. At level i of the refinement, all the edges of the right quad have fallen below the stopping threshold. The

[‡] Other subdivision schemes present similar design issues.

[§] The stopping threshold in our implementation is 1.5 pixels; the effects of different thresholds are discussed in Section 6.2.

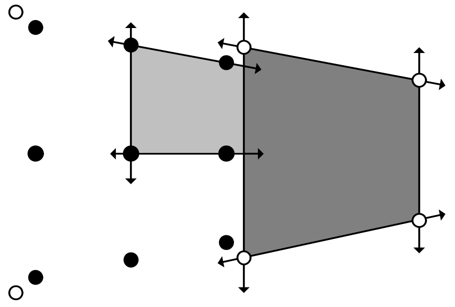

Figure 3: *An example of crack prevention using edge equations. The dark gray quad is completed at the i^{th} subdivision level, while the light gray quad is completed at the $i + 1^{th}$ level. By storing the shared edge as soon as it meets the subdivision criterion, no cracks are created.*

right quad, whose interior is shown in dark gray, is then complete and output at the i^{th} level of refinement. The right edge of the left quad has fallen below the threshold, so it is stored. However, the other edges of the left quad require further subdivision, so refinement continues. The vertices produced at the $i + 1^{th}$ level of refinement are shown as filled circles. The subdivision algorithm perturbs the four vertices of the previous quad and also introduces five new vertices. At this point, the edge lengths of the upper-right sub-quad are tested and found to be below the threshold. This quad, whose interior is shown in light gray, is then output, using three edge equations from the $i + 1^{th}$ subdivision level and one from the i^{th}. Now the importance of storing edge equations becomes clear. The four vertices of the second quad do not necessarily abut the first quad because the refinement can perturb them. However, by using the edge equation from the previous subdivision level to define the right side of the second quad, a crack between the two quads is avoided.

4.2. Shading

Shaders are generated from the same RTSL description as in the OpenGL implementation. However, because Reyes has only one stage in the pipeline for shading, RTSL's vertex and fragment frequencies are collapsed into a single frequency. The generated shader kernel projects the screen-space vertices and normals back into eye space, computes the shading function (using coherent access textures if necessary), and outputs a single color per vertex.

4.3. Sampling

The sampling stage was implemented as a simple bounding-box rasterizer. Since the subdivider described above guarantees that the micropolygon to be drawn is under a certain size, only a small number of pixel locations need to be tested against the four line equations for the micropolygon. The bounded size of the micropolygons leads to two performance improvements over a rasterizer found in the OpenGL pipeline: good load balancing when run under SIMD control and flat shading within the micropolygon with no necessary interpolation. The original Reyes pipeline used stochastic sampling[3], providing many subsamples per pixel with slightly non-regular sample locations. This scheme was not used for our implementation in order to provide a fair comparison with the OpenGL pipeline. However, it would be straightforward to extend the current sampler to support stochastic sampling.

Reyes is particularly well suited to more complex sampling effects such as depth of field and motion blur. The cost of implementing these effects is merely reprojecting and resampling with no reshading, a much smaller cost than accomplishing the same effects in OpenGL using the accumulation buffer. Our implementation could easily be extended to support these effects.

4.4. Composite and Filter

This stage is identical to our OpenGL implementation. Filtering is not currently implemented but could be added in one of two ways. First, subpixels could be composited separately (effectively, a framebuffer with higher resolution) without maintaining a concurrent image at final resolution, and at the end of a frame, subpixels would be filtered as a postpass to create the final image. Second, subpixels are still composited separately but a image at final resolution is concurrently maintained. The second method requires an extra color read and write (for maintaining the image at final resolution) for each sample so potentially uses more memory bandwidth for high-depth complexity scenes. However, it alleviates the massive burst of memory bandwidth necessary if the final image is generated at the end of the scene.

5. Experimental Setup

For the results in this paper we use the Imagine cycle-accurate simulator `isim` and functional simulator `idebug`.

`Isim` models the complete Imagine architecture, including computation, stream and kernel level control, and memory traffic and control, and has been validated against our RTL models and circuit studies. These simulations assume a 400 MHz Imagine stream processor with external SDRAM clocked at 133 MHz, reflecting the actual Imagine system clock speeds. Our OpenGL scenes are simulated in `isim`.

`Idebug` is a faster, higher-level simulator used to develop applications that accurately models kernel runtime but does not model kernel stalls or cluster occupancy effects. Because of these effects, `isim` results are on average 20% slower than `idebug`, so all `idebug` results are scaled by 20% to match the more accurate simulator. Our Reyes scenes are simulated in `idebug`.

Our Reyes implementation also made slight changes to the simulated Imagine hardware. The most significant was increasing the size of the dynamically addressable scratchpad memories in each cluster from 256 to 512 words. These scratchpads are used to implement the working set of quads in the depth-first traversal during adaptive subdivision, and having a larger scratchpad was vital for kernel efficiency.

Scene	Visible Frags	Tris	Average Tri Size	Patches	Quads
TEAPOT-20	137k	23.5k	11.6	28	574k
TEAPOT-30	137k	52.1k	5.26		
TEAPOT-40	137k	91.8k	2.99		
TEAPOT-64	137k	233k	1.18		
PIN	80.0k	91.6k	2.29	12	486k
ARMADILLO	48.9k	93.7k	3.83	1632	328k

Table 1: *Statistics for OpenGL and Reyes scenes.*

Second, the size of the microcode store and the local cluster register files were increased. We expect future improvements to the Reyes implementation will allow us to return the microcode store and cluster register file sizes to the same size as the Imagine hardware.

5.1. Scenes

All scenes are rendered into a 720×720 window. Datasets, textures, cleared depth and color buffers, and kernels are located in Imagine memory at the beginning of the scene. For OpenGL scenes, the input dataset is expressed as subdivided triangle meshes; for Reyes scenes, the input dataset consists of B-spline control points.

We compare three scenes:

- TEAPOT renders the Utah teapot lit by three positional lights with diffuse and specular lighting components. The OpenGL version of the teapot is drawn at several subdivision levels (indicated as TEAPOT-N, where each patch is diced into $2N^2$ triangles) to show performance as a function of triangle size. References to TEAPOT in the context of OpenGL are to TEAPOT-20.
- PIN draws the bowling pin from the UNC Perfect Strike dataset. 5 textures are applied to the pin, which is also lit by a single light with diffuse and specular components. In OpenGL, we render this scene as PIN-1 and PIN-8, which use point sampled and mipmapped textures, respectively. The Reyes version uses a single coherent access per texture per fragment.
- ARMADILLO renders the Stanford armadillo with a single light and a complex marble procedural shader. The shader calculates a turbulence function involving 4 noise calculations[18] per fragment and applies over 1200 floating-point operations to each fragment (OpenGL) or vertex (Reyes).

Details for the scenes are summarized in Table 1.

6. Results and Discussion

Figure 4 shows our simulated performance for OpenGL and Reyes scenes. We see that OpenGL scenes enjoy a significant performance advantage over their Reyes counterparts. There are two reasons for this. First, in Reyes, scenes spend the majority of their time in subdivision, a stage not present in the OpenGL pipeline. Second, our Reyes implementation produces many quads that cover no pixels and do not contribute to the final image. We discuss these points in more detail below in Sections 6.1 and 6.2.

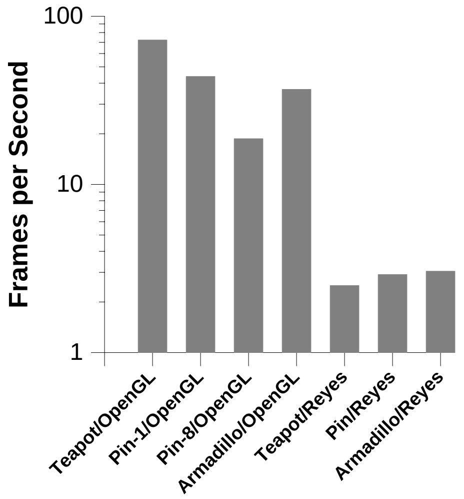

Figure 4: *Simulated runtime for our scenes. OpenGL scenes run an order of magnitude faster than Reyes scenes.*

6.1. Kernel Breakdown

Figure 5 classifies the time spent computing each scene into five categories: geometry, rasterization, composition, and the programmable vertex and fragment programs. Subdivision is considered part of the geometry stage, and the vertex program in Reyes encompasses all the shading work for the scene because Reyes does not have a fragment program.

We see that Reyes runtime is dominated by geometry processing, in particular the subdivision kernel. On average, this kernel takes 82% of the runtime.

OpenGL does not have a subdivision stage because its primitives are subdivided either at compile time or by the host. When subdivision is removed from the Reyes accounting, the two pipelines have similar stage breakdowns. The Reyes pipelines spend comparatively more time in the shading stages than do the OpenGL pipelines, with the exception of the OpenGL PIN-8 scene. Mipmapping is an expensive operation computationally (simply adding mipmapping to PIN-1 cut the resulting OpenGL performance in half), so the amount of time spent in shading in this scene was greater than its Reyes counterpart, which did not require texture filtering.

Rasterization is considerably simpler in the Reyes scenes for two reasons. First, determining pixel coverage of bounded primitives is computationally easier and more parallelizable than unbounded primitives. Second, the primitives in Reyes are smaller and have less computation. OpenGL implementations must carry all their interpolants through their rasterizers, while a Reyes sampler must only carry a single color.

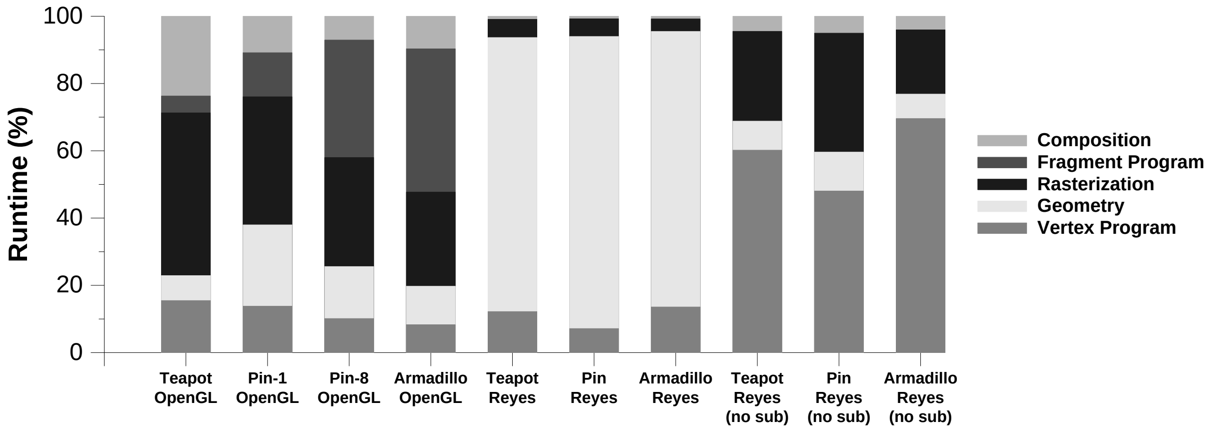

Figure 5: *Stage breakdown of work in each scene. All scene runtimes are normalized to 100%. The first 4 scenes are OpenGL scenes; the next 3 are Reyes scenes; and the final 3 are Reyes scenes with the subdivision runtime (normally part of the Geometry stage) removed.*

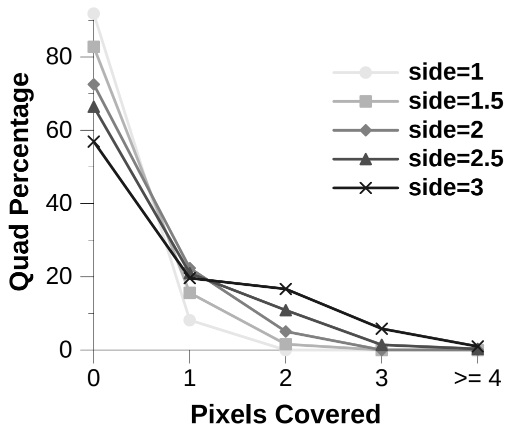

Figure 6: *Quad Size for* PIN. *Other scenes have similar characteristics. Each data point represents the percentage of quads in* PIN *that cover a certain number of pixels given a certain stop length. Lines indicate points associated with the same subdivision stop length. Our implementation has a stop length of 1.5 pixels.*

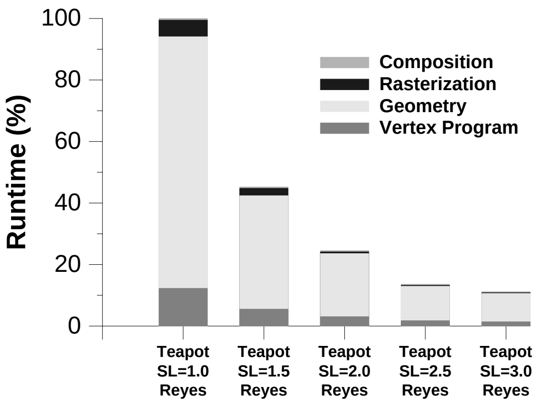

Figure 7: TEAPOT *performance at several stop lengths, normalized to a stop length of 1.0 pixels. Our implementation has a stop length of 1.5 pixels.*

6.2. Reyes: Subdivision Effects

Even with a zero-cost subdivision, the Reyes scenes are still about half the performance of their OpenGL equivalents. This cost is largely due to shading and rasterization work performed on quads that cover no pixels. Ideally, each quad would cover a single pixel. Quads that cover more than one pixel introduce artifacts, while quads that cover zero pixels do not contribute to the final image.

Figure 6 shows the distribution of pixels covered by quads for PIN for several different subdivision stopping criteria (no quad side greater than a certain length). Our implementation stops subdividing when all quad sides are less than 1.5 pixels in length.

In practice, the majority of quads cover no pixels at all, even for larger stop lengths. On our three scenes, with a stop length of 1.5 pixels, zero-pixel quads comprise 73–83% of all generated quads. Even when we double the stop length to 3.0 pixels, we still find that over half of all quads generated do not cover any pixels. As a result, our implementation spends a significant amount of time shading and sampling quads that produce no fragments.

To improve performance, we could consider reducing the number of zero-pixel quads that are passed through the latter half of the pipeline. At the cost of additional computation, a test could be performed before shading and sampling that tested whether the quad covered any fragments. Alternatively, at the cost of slightly more artifacts due to multiple fragments covered by the same quad, the stop length could be increased, resulting in a significant decrease in the number of quads and hence a decrease in runtime. Increasing the stop length from 1 to 1.5 for PIN, for instance, cuts the number of quads produced by more than a factor of 2 (1,121k

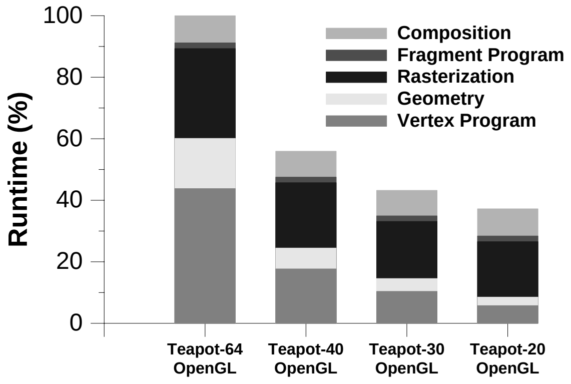

Figure 8: TEAPOT *performance at several subdivision levels, normalized to* TEAPOT-64 *= 100%. The benchmark described in Section 5 is* TEAPOT-20.

to 486k). Doubling the stop length to 3 further decreases the number of quads (to 121k).

Figure 7 shows the performance impact of varying the stop length. Subdivision and geometry/vertex operations decrease with an increased stop length. Because the number of quads decreases (though the total number of fragments covered does not), rasterization work also declines, although not at the same rate. Composition is unaffected.

Also important are datasets that are well-behaved under subdivision. Many of our patches, when subdivided, generated quads with irregular aspect ratios that covered no pixels. Partially this is because when we subdivide a quad, we always generate 4 quads; at the cost of additional computation, subdividing in only one direction instead of both would significantly aid the quality of the generated quads. Choosing both a subdivision scheme that produces well-behaved data and a dataset that conforms well to the subdivision scheme is vital for achieving high efficiency.

6.3. OpenGL: Triangle Size Effects

Similarly, OpenGL performance degrades as triangles become smaller. Figure 8 shows TEAPOT's performance at different subdivision levels. As triangles shrink, the vertex program, geometry, and rasterization cost grows rapidly. TEAPOT-64, with Reyes-sized primitives (an average triangle size of 1.18 pixels), has more than twice the runtime of TEAPOT-20, our benchmark scene.

Small triangles make OpenGL's performance suffer for the same reasons that Reyes' performance is poor. The shading work increases with the number of triangles, and much of the rasterization work is also per-triangle.

6.4. Toward Reyes in Hardware

Many parts of our pipeline are well-suited for programmable stream hardware such as Imagine. The vertex programs for our three Reyes scenes, for instance, sustain an average of 24.5 operations per cycle in their main loops. The sampling algorithm is also efficient, and both would benefit in future stream hardware from more functional units to exploit further levels of instruction-level parallelism.

But subdivision cost dominates the runtime of our Reyes scenes, so continued investigation of subdivision algorithms and hardware is vital. The ideal subdivider has several properties:

Adaptive Uniform subdivision, while simple to implement, is inappropriate for a general subdivider. Uniformly subdividing a patch with part of that patch requiring a fine subdivision means that the entire patch will also be divided finely. This could lead to a huge number of produced quads, most of which would not contribute to the final image.

High performance Ideally, the subdivider would not dominate the runtime of the entire scene.

Artifact free Subdividers must take care that neighboring quads at different subdivision levels do not allow cracks to form as a result of the different levels. Our algorithm, with its use of line equations to represent quad boundaries, guarantees cracks will not occur.

Efficient The ideal subdivider would not output any quads that did not contribute to the final image. Our subdivider does poorly on this point, but could potentially improve at the cost of more computation by testing for pixel coverage before outputting quads or by improving quad quality by allowing subdivision in only one direction.

In the future, we hope to explore other subdivision algorithms that might better address some of the above points. As well, other subdivision schemes and algorithms may be better candidates for our hardware and programming system. For example, Pulli and Segal explore a Loop subdivision scheme that is amenable to hardware acceleration[14]; Bischoff et al. exploit the polynomial characteristics of the Loop scheme with another algorithm for efficient subdivision[1].

Investigating what functional units and operations would allow stream hardware to better perform subdivision would be an interesting topic for future research. Alternatively, our pipelines are implemented in programmable hardware, but due to its large computational costs and regular computation, subdivision may be better suited for special purpose hardware. Hybrid stream-graphics architectures, with high-performance programmable stream hardware evaluating programmable elements such as shading and special-purpose hardware performing fixed tasks such as subdivision, may be attractive organizations for future graphics hardware.

7. Conclusion

In this paper, we have presented implementations of OpenGL and Reyes pipelines running complex programmable shaders on the Imagine Stream Processor. We have shown that although Reyes has several desirable characteristics — bounded-size primitives, a single shader stage, and coherent access textures — the cost of subdivision in the Reyes pipeline allows the OpenGL pipelines to demonstrate superior performance. Continued work in the area of efficient and powerful subdivision algorithms is necessary to allow a Reyes pipeline to demonstrate comparable performance to its OpenGL counterpart.

As triangle size continues to decrease, Reyes pipelines will look more attractive. And though the shaders we have implemented are relatively sophisticated for today's real-time hardware, they are much less complex than the shaders of many thousands of lines of code used in movie production. When graphics hardware is able to run such complex shaders in real time, and the cost of rendering is largely determined by the time spent shading, we must consider pipelines such as Reyes that are designed for efficient shading.

Furthermore, as graphics hardware becomes more flexible, multiple pipelines could be supported on the same hardware, as we have done with our implementation on Imagine. Both the OpenGL and Reyes pipelines in our implementation use the same API, the Stanford Real-Time Shading Language, for their programmable elements. Such flexibility will allow graphics hardware of the future to support multiple pipelines with the same interface or multiple pipelines with multiple interfaces, giving graphics programmers and users a wide range of options in both performance and visual fidelity.

Acknowledgements

The Imagine project was primarily supported by the Defense Advanced Research Projects Agency under ARPA order E254 and monitored by the Army Intelligence Center under contract DABT63-96-C0037. The authors were supported by the DARPA Polymorphous Computing Architectures Project (monitored by the Department of the Air Force under contract F29601-00-2-0085), an Intel Foundation Fellowship, and the Interconnect Focus Center Program for Gigascale Integration (contracted by the Georgia Tech Research Corporation under DARPA Grant MDA972-99-1-0002 as Georgia Tech Project Number B-12-D00). Kekoa Proudfoot and Bill Mark were instrumental in developing and using the RTSL compiler for this work, and Pat Hanrahan helped to develop the subdivision algorithm. David Ebert contributed the marble shader. Matt Pharr's comments on this work were particularly insightful.

References

1. Stephan Bischoff, Leif P. Kobbelt, and Hans-Peter Seidel. Towards hardware implementation of Loop subdivision. In *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 41–50, August 2000.
2. E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350–355, September 1978.
3. Robert L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, January 1986.
4. Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes image rendering architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, volume 21, pages 95–102, July 1987.
5. Ziyad S. Hakura and Anoop Gupta. The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 108–120, 1997.
6. Homan Igehy, Matthew Eldridge, and Kekoa Proudfoot. Prefetching in a texture cache architecture. In *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 133–142. ACM SIGGRAPH / Eurographics / ACM Press, August 1998.
7. Homan Igehy, Gordon Stoll, and Patrick M. Hanrahan. The design of a parallel graphics interface. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 141–150. ACM SIGGRAPH / Addison Wesley, July 1998.
8. Ujval J. Kapasi, Peter Mattson, William J. Dally, John D. Owens, and Brian Towles. Stream scheduling. In *Proceedings of the 3rd Workshop on Media and Streaming Processors*, pages 101–106, 2001.
9. Brucek Khailany, William J. Dally, Scott Rixner, Ujval J. Kapasi, Peter Mattson, Jin Namkoong, John D. Owens, Brian Towles, and Andrew Chang. Imagine: Media processing with streams. *IEEE Micro*, pages 35–46, Mar/Apr 2001.
10. Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 149–158, August 2001.
11. Kerstin Müller and Sven Havemann. Subdivision surface tesselation on the fly using a versatile mesh data structure. *Computer Graphics Forum*, 19(3), August 2000.
12. John D. Owens, William J. Dally, Ujval J. Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery. Polygon rendering on a stream architecture. In *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 23–32, August 2000.
13. Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 159–170, August 2001.
14. Kari Pulli and Mark Segal. Fast rendering of subdivision surfaces. In *Rendering Techniques '96 (Proceedings of the 7th Eurographics Workshop on Rendering)*, pages 61–70, 1996.
15. Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 128–138, June 2000.
16. Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.2.1)*. 1999.
17. Steve Upstill. *The Renderman Companion*. Addison-Wesley, 1990.
18. Greg Ward. *Graphics Gems II*, chapter VIII. 10 (A Recursive Implementation of the Perlin Noise Function), pages 396–401. AP Professional, 1991.
19. Lance Williams. Pyramidal parametrics. In *Computer Graphics (Proceedings of SIGGRAPH 83)*, volume 17, pages 1–11, July 1983.
20. Denis Zorin and Peter Schröder. *Subdivision for Modeling and Animation: Implementing Subdivision and MultiResolution Surfaces*, chapter 5, pages 105–115.