Real-Time Reyes-Style Adaptive Surface Subdivision

Anjul Patney
University of California, Davis

John D. Owens
University of California, Davis

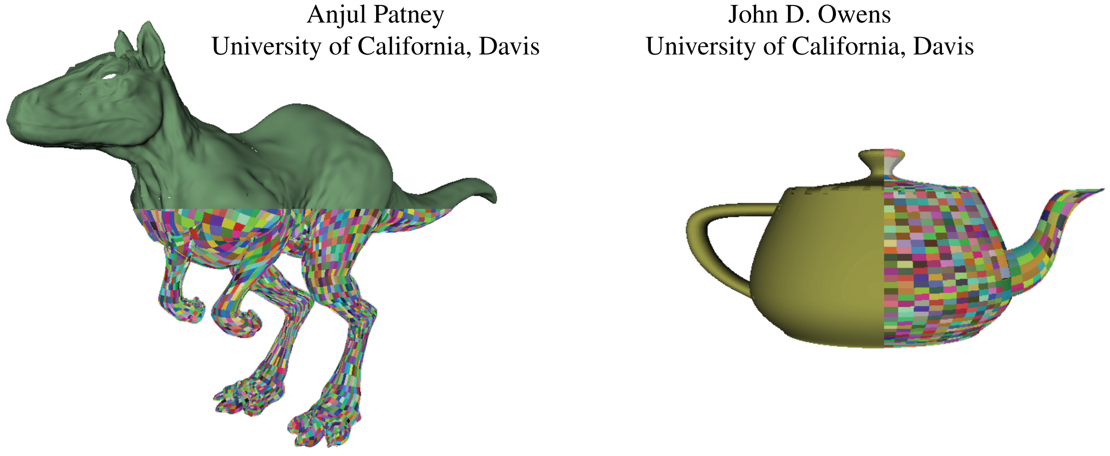

Figure 1: Flat-shaded OpenGL renderings of Reyes-subdivided surfaces, showing eye-space grids generated during the Bound & Split loop.

Abstract

We present a GPU based implementation of Reyes-style adaptive surface subdivision, known in Reyes terminology as the Bound/Split and Dice stages. The performance of this task is important for the Reyes pipeline to map efficiently to graphics hardware, but its recursive nature and irregular and unbounded memory requirements present a challenge to an efficient implementation. Our solution begins by characterizing Reyes subdivision as a work queue with irregular computation, targeted to a massively parallel GPU. We propose efficient solutions to these general problems by casting our solution in terms of the fundamental primitives of prefix-sum and reduction, often encountered in parallel and GPGPU environments.

Our results indicate that real-time Reyes subdivision can indeed be obtained on today's GPUs. We are able to subdivide a complex model to subpixel accuracy within 15 ms. Our measured performance is several times better than that of Pixar's RenderMan. Our implementation scales well with the input size and depth of subdivision. We also address concerns of memory size and bandwidth, and analyze the feasibility of conventional ideas on screen-space buckets.

CR Categories: I.3.7 [Computing Methodologies]: Computer Graphics—Three-Dimensional Graphics and Realism; I.3.1 [Computing Methodologies]: Computer Graphics—Hardware Architecture; I.3.6 [Computing Methodologies]: Computer Graphics—Methodology and Techniques

Keywords: Reyes, graphics hardware, GPGPU, adaptive surface subdivision

Anjul Patney and John D. Owens. *Real-Time Reyes-Style Adaptive Surface Subdivision.* ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia), 27(5), December 2008.

1 Introduction

Today's real-time graphics systems are rapidly migrating from an era of graphics pipelines with a few programmable units to one where the pipeline itself is completely programmable [Pharr et al. 2007]. This jump in programmability allows us to rethink the fundamental primitives used in real-time rendering.

The Reyes architecture [Cook et al. 1987] was originally developed in mid-1980s as an offline renderer optimized for speed, complexity, and visual appeal. Twenty years later, it continues to be one of the most widely used techniques for photo-quality rendering. However, traditional Reyes implementations like Pixar's RenderMan are still primarily non-interactive [Apodaca and Gritz 1999]. The increasing compute capability and programmability of the modern GPU motivates its use for a real-time Reyes implementation. However, all parts of the Reyes pipeline do not map well to such a data-parallel architecture. Within the pipeline we identify three main bottlenecks:

- Surface subdivision in Reyes is a recursive depth-first algorithm, which involves complex and irregular data structure management. Such processing of irregular amounts of data and work per element is an ongoing challenge for highly parallel processors such as today's programmable GPUs.
- The Reyes pipeline generates a huge amount of data in the form of micropolygons, each of which must be shaded independently. Thus, shading is often the computational bottleneck in high-quality offline rendering. Managing such dynamic and unbound data has also been a weak point for traditional parallel hardware.
- Image composition in the form of the Reyes A-buffer is highly specialized and hence, fairly complex for implementation in a real-time system.

Each of the above is an individually challenging problem for modern GPUs. In this work, we address the first challenge, that is, subdivision of higher-order surfaces to subpixel accuracy. This is an important aspect of the pipeline to study, because the irregular nature of its computation forms one of the biggest computational bottlenecks to real-time performance [Owens et al. 2002]. Moreover, a graphics pipeline that can natively render complex surfaces offers larger amounts of freedom to application designers, both in terms of performance and convenience.

On a broad level, we characterize the problem of subdivision as a more general computing challenge: processing irregular and dynamic work queues. The combination of irregular work for a centralized queue is a difficult problem and often a significant bottleneck on a parallel computing system such as the GPU. Our work attempts to solve this general problem using completely parallel operations, and thus has significance beyond surface subdivision.

We begin by a short description of the Reyes pipeline in Section 2. After a detailed discussion of the hardware issues, we detail our implementation in Section 4. This is followed by performance results in Section 5. We report several noteworthy observations and inferences from these results, which can be found in sections 6 and 7.

2 Background

2.1 The Reyes Pipeline

The Reyes image rendering system was introduced by Pixar in the mid-1980s [Cook et al. 1987]. Pixar's goal with this architecture was the development of an efficient system for high-quality cinematic graphics. At that time, this meant that interactive rendering was not an option. In concept, Reyes has several important differences from a modern-day GPU pipeline. Rendering occurs in units of *micropolygons*, each of which according to the original architecture represents a flat shaded quad, no bigger than 1/2 pixel on a side. Modern implementations often use a more relaxed bound of 1 pixel per side [Owens et al. 2002], and use shading to compensate for image quality loss due to aliasing.

The primary inputs to a Reyes pipeline are higher-order parametric surfaces. These surfaces, according to the original paper, take the following path through the pipeline:

1. **Bound & Split Loop** During the first stage of the pipeline, each of the input primitives is a) *bound* in screen space, b) *split* into smaller primitives based on its screen-size, and c) *culled* if the bound lies outside the screen. The resulting set of primitives follows the same procedure recursively, until all input primitives are smaller than a predetermined bound. Splitting is performed in object space, and ensures no loss of surface precision.

2. **Dicing** After the previous step, all primitives are known to be bound by a constant. From these primitives, it is easy to construct micropolygons by appropriately subdividing in eye space. This subdivision dices primitives into regular *grids*, each having a known number of micropolygons. Dicing is similar to tessellation in a traditional context.

3. **Shading** Each micropolygon is shaded in eye space, using scene-specific and usually programmable shaders.

4. **Sampling** After transforming to screen space, Reyes uses a stochastic sampling technique on the micropolygons to form the final image. Sampled polygons are resolved for visibility using a depth buffer of sub-pixel degree.

Reyes offers several advantages over the conventional OpenGL pipeline [Segal and Akeley 2006]. It supports higher-level surfaces as geometry primitives, allowing artists to incorporate complex static as well as dynamic shapes and surfaces into a scene with more modest bandwidth requirements than OpenGL's triangles, whose number blows up for even moderately complex surfaces. Moreover, triangles are inherently static, and hence render quality is view-dependent. While OpenGL performs shading both in eye space as well as screen space, in Reyes pipeline, all shading is done in a single stage and in eye coordinates. Texturing in OpenGL is a per-fragment operation, requiring special filtering techniques (mipmapping, anisotropic filtering) to avoid artifacts, and suffering from potentially incoherent memory access patterns. Instead, Reyes can support object-space texturing in the form of "coherent access textures" (CATs), avoiding the need for filtering during rendering.

Reyes forms the foundation for Pixar's RenderMan [Apodaca and Gritz 1999], the industry standard in cinematic rendering.

2.2 Existing Implementations

Parallel implementations of the Reyes pipeline are uncommon in the academic literature; the works by Owens et al. [2002] and Lazzarino et al. [2002] are two of the most recent. In the former, the authors compare implementations of Reyes and OpenGL on the Imagine stream processor. Their implementation modifies the split/dice loop and instead employs a screen-space dicing technique to generate micropolygons. This modification is not optimal, since it is non-trivial to guarantee bounds of generated quad sizes. The number of micropolygons can be huge, which would introduce significant performance overheads. They conclude that "subdivision cost dominates the runtime of [their] Reyes scenes". In the latter work, the authors implement a Reyes renderer on a Parallel Virtual Machine [Lazzarino et al. 2002]. Slave nodes of the PVM compute separate screen-space buckets for the master in parallel, and the required geometry is replicated across all of them. The authors explore optimal bucket assignment for load balancing and maximizing locality, and report near linear speedup in the number of nodes used for rendering. We achieve similar scalability for a many-core GPU in Section 5.

The current version of the NVIDIA Gelato renderer is based on the Reyes pipeline [Wexler et al. 2005]. However, it appears that Gelato uses GPU acceleration only for a few stages of the pipeline, primarily sampling and hidden surface removal.

GPU-assisted surface subdivision algorithms are often used to tessellate coarse meshes [Bóo et al. 2001; Boubekeur and Schlick 2008; Shiue et al. 2005]. One of the biggest advantages of Reyes is the freedom to specify the scene as higher order surfaces, independent of size or resolution. Unlike refined meshes, this requires a much smaller set of inputs to represent complex geometry. Moreover, this does not precompute geometry, it instead performs view-dependent subdivision at every step.

With coming generations of rendering APIs [Microsoft Corporation 2008] and graphics hardware anticipated to support hardware-accelerated tessellation, what is the place for software-based surface subdivision in future GPUs? We believe the trend toward programmable pipelines over fixed-function hardware ones [Pharr et al. 2007] will, in time, also apply to subdivision schemes, allowing more flexible and scene-specific subdivision. Interest has also been drawn towards completely programmable graphics processors, which may not include hardware tessellation [Seiler et al. 2008]. Even in the presence of hardware-supported subdivision, however, the median installed graphics system is still years away from carrying such a feature, making it more likely that application developers will continue to explore software-based tessellation techniques in the meantime.

3 Reyes Analysis and Hardware Challenges

In this section, we present an analysis of the stages of Reyes algorithm from an architect's perspective, and their relation to general purpose issues in parallel computing. Note that our present work addresses the problem of efficient acceleration for the Bound/Split and Dice Stages. Although we discuss other stages as well, these two form the focus of this section. We begin by revisiting our Reyes pipeline description from Section 2.1.

3.1 Analysis of Reyes Pipeline

When approached from a hardware designer's angle, we see that each stage in the Reyes pipeline is individually challenging to approach.

1. **Bound & Split Loop** In this stage, a relatively few primitives may split into a large number of smaller primitives. This is recursive, i.e. the split primitives might further subdivide. This process must continue until no new primitives are generated. Even though each primitive can be processed completely in parallel, a new iteration can only begin after it has been completely split. This process is serial and irregular, because new work may be generated in every iteration. The stopping criteria for the subdivision is inherently unpredictable.

2. **Dicing** Unlike the previous stage, dicing is a fairly regular and predictable step. Since the input primitives to this stage have guaranteed bounds, sufficiently small micropolygons can be easily obtained. Dicing offers abundant parallelism, as the resulting grids are independent.

3. **Shading** Shading is performed in object space on individual micropolygons. Huge levels of parallelism are thus usually available, and the job is again fairly regular. Shaders can however, be extremely complex. Many Reyes renderings are limited by shading performance.

4. **Sampling** Sampling in the Reyes pipeline is similar to rasterization in a modern GPU. The only difference is that the Reyes renderers sample in a stochastic fashion, while GPU samples are obtained from a regular grid.

Several key observations can be made from the above analysis. First, the stages of dicing, shading and sampling appear extremely well suited for efficient execution on a data-parallel machine. Ample parallelism is available, and the workload is highly regular. As a result, we find appreciable coherence in the execution of these operations. For example, all micropolygons in a grid usually execute the same shader. Thus, these operations are expected to benefit greatly from a massively parallel SPMD (Single-Program Multiple-Data) architecture similar to today's GPUs.

The second observation concerns the irregular nature of the Bound/Split loop. This is a recursive multi-iteration stage, in which each iteration may dynamically allocate and deallocate work for the next. A primitive starts as a single object, but within a few iterations it can produce many individual objects. This problem can be generalized to managing a highly irregular dynamic work queue.

A third observation can be made regarding memory requirements of the Reyes pipeline in general. A huge number of micropolygons are generated during dicing, and storing all of them simultaneously may not be feasible. Moreover, this number is unpredictable and view-dependent, which makes it even harder to manage.

3.2 Implications for Parallel Computing

The two problems of irregular behavior and unbounded memory usage, mentioned above, are by no means new in the domain of parallel computing. The first is often encountered in applications other than graphics, and is easily solved for conventional parallel systems. Computing nodes perform units of work in parallel, and update the queue when they are done. These updates are required to be atomic, i.e., accesses to the queue must be serialized. Unfortunately, this forced serialization adversely affects SIMD threads. It is expected that using atomic operations will be detrimental to Bound/Split performance, since many threads will invariably request queue access at the same time.

Rather than forcing threads to synchronize on accessing the queue, we propose a two-stage solution. First, parallel thread groups work independently on elements from the input queue, accumulating output primitives at independent locations. These outputs must then be merged back into the original queue efficiently. We use the parallel primitives of *scan* and *reduce* [Sengupta et al. 2007; Harris et al. 2007] to achieve this compaction.

Regardless of visibility, on-screen Reyes primitives cannot be discarded until transparency and blending have been resolved. To reduce this problem of potentially unbounded data, most implementations divide the screen into screen-space tiles ("buckets") [Cook et al. 1987] and render those buckets one at a time. Because buckets are usually much smaller than the screen, the memory footprint while rendering is substantially reduced. However, sequential buckets restrict the available parallelism. Running multiple buckets in parallel, on the other hand, increases memory requirements. For a hardware implementation, we expect a designer to balance between the two extremes, by choosing enough buckets to keep the processor busy, but only as much as the memory budget permits. In Section 5, we examine the effects of this tradeoff on our implementation.

4 Implementation on a GPU

In this section, we describe our implementation and how it deals with the issues discussed in Section 3. We present our solution as an implementation on a modern programmable GPU. This poses several hardware-specific challenges in addition to those mentioned above, and we also discuss how we overcome those. The most prominent of these implementation related issues concern the SIMD nature of GPU cores, and the high off-chip memory latency. Our solution takes significant steps to ensure maximal utilization of these scarce resources. The implementation is based on a NVIDIA GeForce 8800 GTX graphics processor. To access this device, we used release 1.1 of the NVIDIA CUDA data-parallel programming framework [NVIDIA Corporation 2007]. We also used reference code from the open-source CUDPP library for efficient implementations of common data-parallel primitives [Harris et al. 2007].

The system takes a collection of scene primitives as its input, and generates adaptively subdivided grids as the output. It accepts these primitives in the form of Bézier bicubic (4×4) patches, which are subdivided in a view-dependent fashion every frame. The following subsections cover this procedure in detail. The obtained set of micropolygons is rendered in a simple manner using OpenGL. Section 4.2 provides a detailed description for the exact approach taken.

4.1 Subdivision Kernels

4.1.1 The Bound & Split Loop

Input primitives are first passed to the *bound/split* procedure. The job of this kernel is to recursively split these primitives into smaller ones, until each primitive is contained in a fixed bound. The kernel is also responsible for culling primitives that do not contribute to the scene. In our first key insight, we transform this operation from a recursive depth-first one to a parallel breadth-first one. Every iteration of *bound/split* is launched as a data-parallel task for all the available primitives. This kernel calculates the bound for each primitive, and based on the result either splits it into multiple ones, or culls it (removes it from the queue of primitives). If enough primitives are present in the scene, this provides ample operations to the many-core GPU. Moreover, as input primitives split into finer ones, the number of parallel operations quickly grows to fill the machine.

Managing the above irregular work queue efficiently on a data-parallel machine is an important implementation hurdle. For rea-

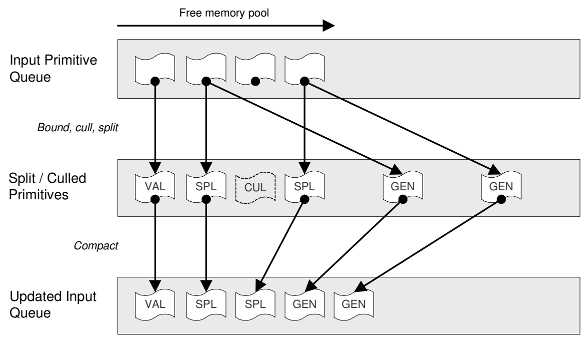

Figure 2: An iteration of the Bound & Split loop (Algorithm 1) for a collection of primitives. The gray region represents a pool of free memory, which is occupied by the queue of input primitives. Initially, all primitives are processed by *bound/split* to generate 0 (CUL), 1 (VAL), or 2 (SPL) primitives each. The resulting queue is processed by the *compact* kernel to yield a contiguous memory region, which is used as input for the next iteration.

sons already mentioned, we must obtain maximum instruction and data throughput for our kernel. For high instruction throughput, we decided to unroll each input primitive across 16 CUDA threads, each responsible for processing one control point. The execution path for each of these threads is the same, because they all correspond to the same primitive. This ensures high SIMD coherence among warps (32 threads), each of which only contains two primitives. Divergence occurs in the relatively rare case when these adjacent primitives take different paths during the *split* part of the kernel. The *bound* portion, however, is uniform across all primitives, so the worst case penalty is minimal.

We also make sure that the available memory bandwidth is highly utilized. To do this, we follow every iteration with a scan-based *compact* kernel [Sengupta et al. 2007]. This ensures that any holes introduced by the creation of new primitives or destruction of old ones are eliminated. That is, the work queue always represents a contiguous region of memory. Consequently, all memory accesses made by a group of 16 threads get merged into a single contiguous access, completely utilizing the available bandwidth.

For a pseudocode detailing the above description, please refer to Algorithm 1. Steps 2, 16, and 17 each represent a different CUDA kernel.

In order to reduce their memory footprint, Reyes implementations often use screen-space buckets to sort grids before rendering [Wexler et al. 2005]. This significantly reduces the memory footprint of the pipeline, since only micropolygons from one bucket stay in flight at a time. On serial hardware, this has negligible effect on performance, since the processor maintains its workload. On a GPU-like parallel environment, however, buckets induce serialization in the subdivision procedure. Unless a single bucket can populate all the available execution units, the GPU will not be fully utilized, and performance will be adversely affected. To monitor the relative gains and costs in this procedure, we added buckets to our implementation. This works by a straightforward augmentation to the Bound & Split loop, which simply culls any primitives that lie outside the current bucket.

4.1.2 The Dicing Kernel

Once all input primitives have been split to the desired bound, they are sent to the *dice* routine. This kernel takes each of these grids, and uniformly dices (tessellates) them into micropolygons. Again, since individual micropolygons are independent of each other, each thread can generate a micropolygon. Dicing is a highly parallel workload that suits the architecture very well. Pseudocode for the dicing operation can be found in Algorithm 2.

Each grid is assigned to a thread block composed of 16×16 threads each. These threads share input data using the on-chip shared memory, and subdivide the input grid into 256 micropolygons. Since dicing is a uniform operation, high efficiency is achieved both in terms of SIMD occupancy and memory bandwidth.

Algorithm 1 Bound & Split Loop

```
1:  repeat
2:    for all input primitives do {in parallel}
3:      Bound each primitive prim in object space using its
        Bézier control points
4:      Transform the bounds to screen space; evaluate screen
        space bound
5:      if bounds lie outside the drawing region then
6:        Mark prim as CUL (culled)
7:      else
8:        Mark prim as VAL (valid)
9:        if the screen space bound is larger than 8x8 pixels then
10:         Mark prim as SPL (split)
11:         Split prim into two smaller primitives (prim1,
            prim2) using De Casteljau's algorithm
12:         Update self with prim1, and store prim2 at an in-
            dependent offset from the end of the work queue;
            mark prim2 as GEN (generated)
13:       end if
14:     end if
15:   end for
16:   Perform a prefix-sum (scan) operation on the updated queue
      of primitives, ignoring ones were marked as CUL
17:   Use the scanned queue values to copy all primitives to a con-
      tiguous memory region (compact)
18: until there are no GEN or CUL primitives
```

Algorithm 2 Dicing

```
1: for all grids obtained from bound/split do {in parallel}
2:   Subdivide the grid into $16 \times 16$ micropolygons, equally dis-
     tributed in parametric space
3: end for
```

4.2 Rendering Micropolygons

In this research, our focus lies in accelerating the two previous algorithms. Thus, we eschew a CUDA implementation of the shading and sampling stages. Instead, we forward the collection of diced micropolygons as a vertex buffer to the OpenGL pipeline, which is then rendered in a regular fashion. This simple choice serves as an example for the situations where this subdivision scheme may be used, that is, in conjunction with conventional rendering. Under a unified shader architecture, this has minimal performance penalty and shading can be done in the vertex stage.

CUDA permits the above interoperability with OpenGL. However, at the time of writing, there are known issues with the performance of this interface. These issues significantly affect our rendering performance, even though surface subdivision is much faster. Moreover, only one of CUDA and OpenGL can be executed on the GPU at a time, which serializes rendering, further affecting performance. Analysis in Section 6 addresses these overheads and their impact in detail. We use flat shading on micropolygons. Normals are generated using a simple kernel that evaluates cross product normals for

Figure 3: One of the scenes used in performance analysis; this frame renders 10 randomly generated bicubic patches.

each micropolygon in a grid.

4.3 Implementation Characteristics

Let us now enumerate the salient features of our implementation. On an algorithmic level, we have identified a methodology to efficiently carry out *bound/split* and *dice* operations over Reyes primitives on a massively data-parallel device. By noting the independence of operations in the originally recursive subdivision, we have provided a way to treat it as a parallel breadth-first operation with a more regular behavior. We have also proposed some key implementation-level solutions to obtaining high performance out of the subdivision kernels. By assigning 16 threads to a primitive during *bound/split*, we have obtained a high utilization for the SIMD hardware units. Divergence is rare and has minimal impact on performance, ensuring high instruction throughput. By maintaining a contiguous list of primitives throughout the subdivision, we have also preserved the memory bandwidth. We used the CUDA profiler to help us quantize these achievements. Taking a time average over the entire subdivision procedure, we found that 99.50% of all memory fetches were perfectly coalesced, and 90.16% of all branches taken were SIMD-coherent. These figures stayed consistent over all scenes rendered, and given hardware trends, will continue to be important aspects of efficient parallel solutions for future parallel hardware as well.

To sum-up, the key achievement of our implementation is the reduction of a seemingly irregular algorithm to fundamental parallel operations like scan and compact. By using these operations, we can efficiently extract non-local aggregate information about large amounts of parallel data. This has enabled us to manage a dynamic shared work queue with negligible performance overheads. Our compute-independently-then-compress strategy obviates the need for the potential bottleneck from using atomics or locks for shared queue management.

We have also alleviated the problem of memory capacity by using screen-space buckets, which are extremely simple to support in our system without loss in performance of the individual kernels. Although a bounded memory requirement still is not guaranteed, bucketing significantly reduces the requirement in the average case.

5 Results

Because our primary interest lies in the feasibility of a Reyes-like hardware pipeline, we present our results from that perspective. We expect that the initial impact of the work will be in rendering particular Reyes-style objects within a larger (non-Reyes) scene, though our results scale to larger screen sizes and more objects.

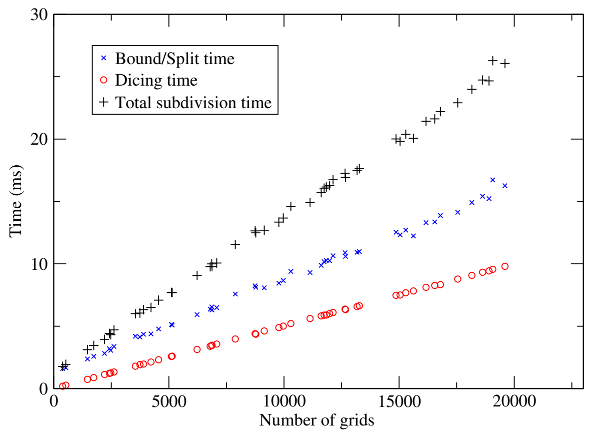

Figure 4: *Bound/Split* and *Dice* timings for various randomly generated inputs (Figure 3) show roughly linear behavior with increasing number of grids.

Thus we initially report raw numbers for performance of subdivision on two sample objects: the Utah Teapot and the Killeroo[1] (both shown in Figure 1). The former is the well-known teapot model formed by 32 Bézier patches, and the latter is a complex model of a creature formed by about 11.5K patches. The fundamental difference between the two is the number and average size of patches. In Teapot, a few individual patches smoothly span a significant object area, while in Killeroo, a large number of small patches cause subtle variations in the model's skin. The two are representative of the range of scenes that might be encountered by a Reyes-like renderer. We also study many scenes composed of a varying number of randomly generated patches, like those shown in Figure 3.

We compare the obtained performance to that of a CPU, by obtaining approximate subdivision times for Pixar's RenderMan. To measure the scalability of our implementation over diverse workloads, we monitor performance for varying input size and depth of subdivision. Lastly, we also study the performance–memory tradeoffs in using screen space buckets. We use these results to propose an optimal bucket size for the Killeroo scene.

5.1 Raw Performance and Comparison

Table 1 shows our measured performance for a render resolution of 512×512 pixels. Also, Figure 4 shows these numbers for several random datasets. It can be seen that the time taken to render these primitives has roughly a linear dependence on the number of grids generated during subdivision. Also noteworthy is the fact that subdivision time is less than 25 ms for most inputs. On current hardware, then, we can perform subdivision of a moderately-sized object up to 40 times a second.

Our implementation of a renderer sustains an average performance of 12.41 frames per second for Teapot and 4.15 frames per second for Killeroo. Also, a scene with 30 randomly placed patches renders at 5.48 frames per second. Since rendering performance is substantially lower than subdivision, we conclude that we are limited by overheads. The breakdown of frame time is shown in Figure 5, and the corresponding discussion can be found in Section 6.

[1]Killeroo NURBS model supplied by headus 3D tools, `http://headus.com.au/`.

Scene	Killeroo	Teapot	Random Scene (30 patches)
Number of grids	14426	4823	9810
Subdivision depth	5	11	17
Bound/Split	6.99 ms	3.46 ms	8.81 ms
Dicing	7.21 ms	2.42 ms	5.02 ms
Total subdivision time	14.21 ms	5.88 ms	13.83 ms

Table 1: Subdivision statistics and time for three input scenes.

Scene	Killeroo	Teapot	Random Scene (30 patches)
Geometry/data allocation	3.13 ms	2.91 ms	1.68 ms
Data transfer	2.41 ms	0.33 ms	0.33 ms
Split/dice time	14.21 ms	5.88 ms	13.83 ms
Vertex buffer allocation	143.44 ms	47.44 ms	107.89 ms
Total subdivision time	163.19 ms	56.56 ms	123.73 ms

Table 2: GPU allocation and transfer overheads.

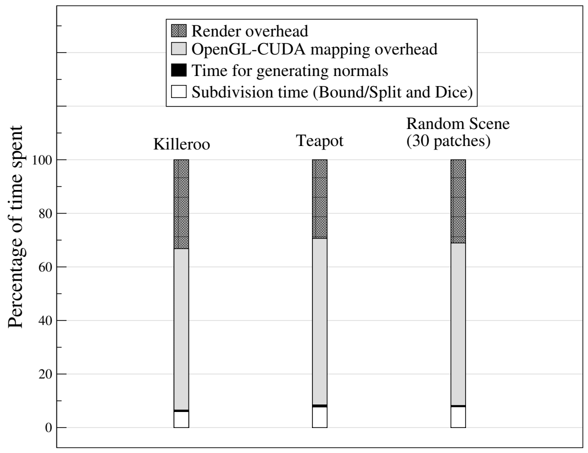

Figure 5: Breakdown of overall execution time: subdivision accounts for less than 10% of the total frame time in each case.

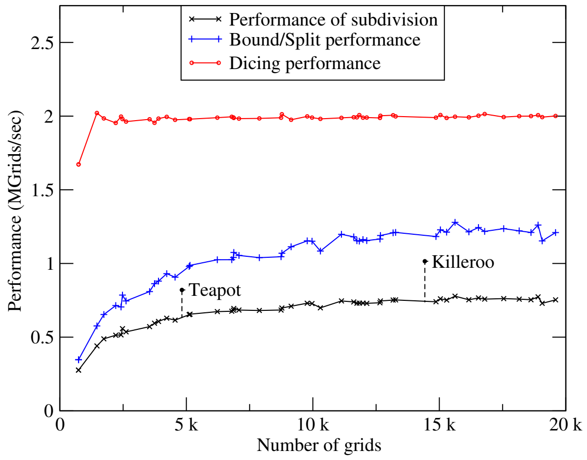

Figure 6: Performance scaling for our implementation: Dicing maintains high utilization for most input sizes, whereas the efficiency of *bound/split* increases with increasing input size. The aggregate trend follows that of the latter.

To put these performance numbers in perspective, we performed a broad level performance comparison with a known reference: the bound/split and dice timings from Pixar's Photorealistic RenderMan (PRMan). In order to do so, we first recreated our exact scenes as inputs to PRMan. Then, to minimize any overheads due to shading, special effects and visibility testing, we used null shaders, configured a single sample per pixel, and turned off all culling and hiding units. This way, we reduced PRMan execution to geometry evaluation, which primarily consists of split and dice procedures. To avoid any further unknown overheads including input parsing, we instantiated the scene several times, and took an average of the total execution time. We believe that this provided us a reasonable estimate of the time taken for geometry evaluation. PRMan 13.0.3, running on a single core of an AMD Opteron (2.4 GHz) system with 16 GB memory, took 154.9 ms to subdivide the Teapot. This is 26.35 times slower than our implementation. Similar numbers for the Killeroo are 488.8 ms on PRMan and 14.21 ms on our implementation, indicating a speedup of 34.4.

Note that the above comparison is approximate, and presented primarily for completeness. Even after our best efforts, a few portions of PRMan's execution time could not be accounted for in our implementation. PRMan spends some time stitching cracks resulting from non-uniform dicing across grids. Currently, our implementation does not explicitly stitch such cracks. Also, PRMan supports several classes of primitives in addition to Bézier patches, which is expected to add to the execution overhead. However by taking the average of a batched subdivision for several instances of the scene, we have tried to minimize this overhead. Finally, the splitting criterion used by PRMan is another point of difference. While PRMan averages lengths of isolines sampled over a grid to decide whether to split a primitive, we use the screen-space bound for making this decision. We expect these overheads to be small though significant parts of the total subdivision time. However, even if they amount to as much as half of the total, our implementation is still an order of magnitude faster. Since the goal of our work is not to rival an offline renderer but to provide comparable quality of geometry at real-time performance, we are satisfied with this comparison.

CPU-GPU transfer time for the input geometry has not been included in the above results. This is a fair assumption in our target scenario, because even for dynamic surfaces, negligible data needs to be transferred from the CPU to the GPU after the initial system setup. The input model stays in the GPU memory for as long as it is needed. All subdivisions after the first frame, irrespective of the viewpoint, entirely use this data. To be thorough, however, we still report these numbers. Execution time with various setup overheads can be found in Table 2. Including all the discussed overheads, the teapot takes 56.56 ms on our system, which is 2.74 times faster than the estimate for PRMan. Subdivision of the Killeroo takes 163.19 ms on our implementation, 2.99 times faster than PRMan. Again, these comparisons provide only an approximate picture. Also note that most of the execution overhead is due to vertex buffer allocation. We believe that it is because of the slow CUDA-OpenGL interface, an issue which is further discussed in Section 6.

5.2 Scalability

The feasibility of a hardware Reyes renderer depends heavily on how robust it is in taking advantage of future improvements in silicon. To this end, we note how performance scales with two input parameters of our implementation, the primitive count and the screen size.

Primitive Count The input size significantly affects the performance of a massively parallel system, because it determines how effectively the available resources are utilized. Thus, we noted the

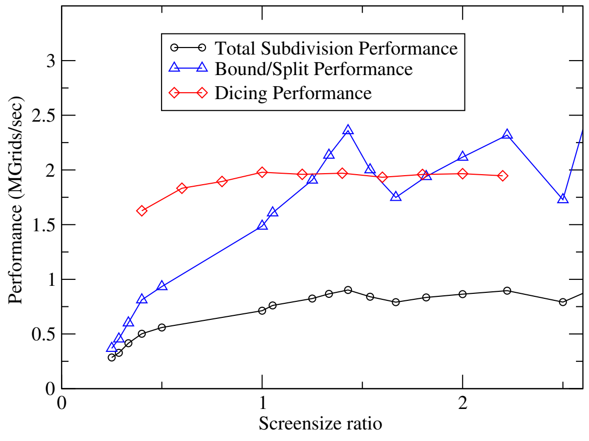

Figure 7: Performance with varying screen size follows roughly the same trend as with varying input size. While dicing attains its peak quickly for small screen sizes, *bound/split* scales to high utilization as the screen size of the model increases.

variation in performance of our system for varying sizes of input (number of grids). Figure 6 shows this trend for split and dice kernels, and for the aggregate subdivision. In each case, it can be seen that increasing input size is not detrimental to performance. While utilization remains fairly consistent for dicing, it increases with the input sizes for bound/split, and hence, for the combination.

Screen Size In Reyes, the screen size of a model is closely related to the scene complexity, because it affects the depth of subdivision. Generating enough micropolygons requires deeper subdivision for a larger image. To study the corresponding effect on performance, we carried out subdivision for the teapot at varying screen-sizes, from about 0.25 to 2.5 times its original size. Beyond that, we were unable to allocate vertex buffers of sufficient size. The resulting performance variation is plotted in Figure 7. For increasing complexity of subdivision, the performance of *bound/split* increases until it roughly saturates, and shows occasional dips in performance whenever the maximum depth of subdivision increases. The dicing kernel occupies most GPU resources even for a small number of grids, and hence maintains consistent performance. The combined performance of subdivision follows *bound/split*.

5.3 Memory Usage and Buckets

Section 3.2 described the tradeoff in bucketing: smaller buckets use less memory but present less parallelism. When rendering an entire scene, our kernels use a lot of device memory to store micropolygons, up to 50 MB for some objects. To explore this tradeoff, we experimented with several bucket sizes, and for each, noted the performance of subdivision and the memory requirement per bucket for that size. These plots are shown together in Figure 8. Maximum memory usage increases roughly linearly with the bucket width, while bucket sizes of approximately 100×100 pixels or more are necessary to fully utilize the available parallelism.

6 Discussion

Our results indicate a number of implications for a real-time Reyes implementation. We have demonstrated that subdivision can run at real-time rates efficiently on a the GPU. Presently, our rendering throughput is limited by overheads of the OpenGL and CUDA interface. In Figure 5, we can see that this overhead can account for more than 50% of the frame time. Because no memory transfer is ideally required, this should have been negligible. This issue with performance of data transfer between OpenGL and CUDA has been acknowledged by NVIDIA, and will significantly reduce in future systems. Also, architectures that allow graphics and computation to seamlessly interact are also expected to be soon available [Seiler et al. 2008]. For conventional GPU-rendered scenes, the result should be frame rates commensurate with the subdivision performance. The behavior of our implementation is desirable in three main respects:

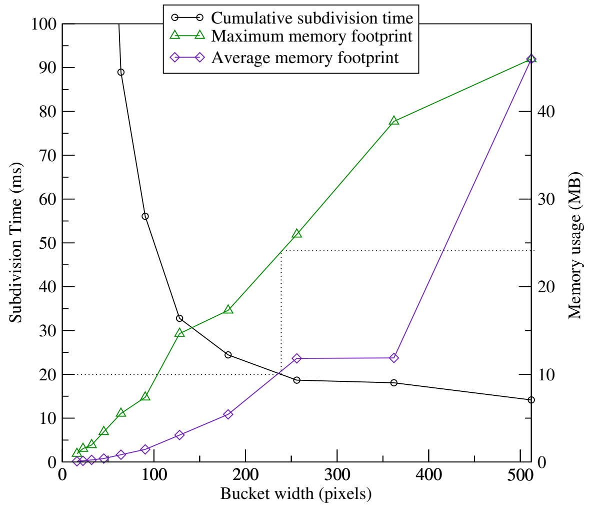

Figure 8: Varying sizes of screen-space buckets show a strong relationship between subdivision performance and memory demand. Bigger buckets mean higher performance from the parallel machine, but at the cost of requiring more storage per bucket.

- The performance is fairly scalable with the complexity of the input and the screen size of the model. We expect our kernels to perform better with improvements in the underlying hardware.
- Our experiments with screen space buckets indicate that there is a strong relationship between the performance of subdivision and the memory requirements. This conclusion is different for buckets on a CPU, because the available parallelism on a GPU must be utilized well to achieve high performance, and buckets discourage this. Only a finite number of buckets may be in flight at the same time to preserve available memory. Fortunately, this relationship is one that can be tuned by the programmer, even on a per-scene basis, to match the requirements of the graphics system. In a simple experiment to this end, we use the plot in Figure 8 to estimate a good bucket size for the Killeroo. Let us assume a performance budget of around 20 ms per subdivision, for a modest real-time rendering goal. We find that to achieve this performance goal, we must have a minimum bucket size of about 240×240 pixels. This also gives us the average and the maximum memory requirements for this setting, which in this case are about 10 MB and 25 MB, respectively (much less than the peak requirement of 45 MB).
- Our implementation subdivides the input surfaces once every rendered frame. Since we only tessellate to pixel accuracy, minimal work is wasted with this approach. Interesting future work includes caching split geometry across frames.

A possible outcome of this work is that the surface subdivision portion of PRMan could be mapped to a GPU coprocessor for preview systems. Coupled with the known ability of the GPU to perform quality, high-performance shading [Pellacini et al. 2005] and the availability of hardware queues that eliminate issues of unbounded data, this opens new possibilities for both GPU-based Reyes accelerators and standalone renderers.

Another important piece of future work is crack avoidance, which is an orthogonal task to an efficient implementation of subdivision. As of now, our implementation of Reyes subdivision does not include support for filling cracks and pinholes generated during adaptive subdivision. We expect to explore three main approaches in this investigation: local methods (edge equations) [Owens et al. 2002]; neighbor information (identifying hanging vertices) [Moreton 2001]; and stitching [Christensen et al. 2006], which is the method used by RenderMan itself.

We also plan to implement variable dicing rates to avoid excessive micropolygon counts. This is a simple exercise in specializing the present kernel and is not expected to cause any performance drops.

7 Conclusion

The problem of real-time surface subdivision in a Reyes pipeline is known to have irregular behavior and does not map well to a hardware implementation. We identify two core issues with the original algorithm: managing irregular work queues, and handling unbounded data patterns. Our implementation offers an efficient solution to the first problem, and our experiments with buckets indicate a reasonable trade-off for the second. By using the fundamental parallel primitives of scan and compact to manage the shared queue of rendering primitives, we have demonstrated that a Reyes-like subdivision scheme can indeed run in real-time.

Our ongoing research aims to address issues of cracks and rendering overheads in the near future. We also plan to incorporate more parts of the Reyes pipeline in our implementation.

The age of programmable graphics motivates the exploration of a much broader range of graphics primitives in future rendering systems, and we hope that our work will serve as a stepping stone towards tomorrow's fully-featured and high-visual-quality systems.

Acknowledgments

Work presented in this paper would not have been possible without generous support and advice from several people: many thanks to Per Christensen, Charles Loop, Dave Luebke, Matt Pharr, and Daniel Wexler for their valuable feedback and suggestions during the research. Their feedback was extremely helpful in shaping the approach in this work and its direction for the future.

The authors gratefully acknowledge funding from the Department of Energy's Early Career Principal Investigator Award (DE-FG02-04ER25609), the National Science Foundation (Award 0541448), and the SciDAC Institute for Ultrascale Visualization, and equipment donations from NVIDIA.

References

APODACA, A. A., AND GRITZ, L. 1999. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann Publishers Inc.

BÓO, M., AMOR, M., DOGGETT, M., HIRCHE, J., AND STRASSER, W. 2001. Hardware support for adaptive subdivision surface rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, 33–40.

BOUBEKEUR, T., AND SCHLICK, C. 2008. A flexible kernel for adaptive mesh refinement on GPU. *Computer Graphics Forum 27*, 1 (Mar.), 102–113.

CHRISTENSEN, P. H., FONG, J., LAUR, D. M., AND BATALI, D. 2006. Ray tracing for the movie "Cars". *IEEE Symposium on Interactive Ray Tracing 2006* (Sept.), 1–6.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The Reyes image rendering architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, 95–102.

HARRIS, M., OWENS, J. D., SENGUPTA, S., ZHANG, Y., AND DAVIDSON, A. 2007. CUDPP: CUDA data parallel primitives library. `http://www.gpgpu.org/developer/cudpp/`, Aug.

LAZZARINO, O., SANNA, A., ZUNINO, C., AND LAMBERTI, F. 2002. A PVM-based parallel implementation of the REYES image rendering architecture. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Springer-Verlag, 165–173.

MICROSOFT CORPORATION. 2008. Introduction to the Direct3D 11 graphics pipeline. `http://www.microsoft.com/downloads/details.aspx?familyid=E410716F-12BF-4E8F-AC41-97B4440C3B90`.

MORETON, H. 2001. Watertight tessellation using forward differencing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, 25–32.

NVIDIA CORPORATION. 2007. NVIDIA CUDA: Compute unified device architecture. `http://developer.nvidia.com/cuda`, Jan.

OWENS, J. D., KHAILANY, B., TOWLES, B., AND DALLY, W. J. 2002. Comparing Reyes and OpenGL on a stream architecture. In *Graphics Hardware 2002*, 47–56.

PELLACINI, F., VIDIMČE, K., LEFOHN, A., MOHR, A., LEONE, M., AND WARREN, J. 2005. Lpics: a hybrid hardware-accelerated relighting engine for computer cinematography. *ACM Transactions on Graphics 24*, 3 (Aug.), 464–470.

PHARR, M., LEFOHN, A., KOLB, C., LALONDE, P., FOLEY, T., AND BERRY, G. 2007. Programmable graphics—the future of interactive rendering. Tech. rep., Neoptica, Mar. `http://www.pharr.org/matt/NeopticaWhitepaper.pdf`.

SEGAL, M., AND AKELEY, K. 2006. The OpenGL® graphics system: A specification. `http://www.opengl.org/documentation/specs`, Dec.

SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics 27*, 3 (Aug.), 18:1–18:15.

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan primitives for GPU computing. In *Graphics Hardware 2007*, 97–106.

SHIUE, L.-J., JONES, I., AND PETERS, J. 2005. A realtime GPU subdivision kernel. *ACM Transactions on Graphics 24*, 3 (Aug.), 1010–1015.

WEXLER, D., GRITZ, L., ENDERTON, E., AND RICE, J. 2005. GPU-accelerated high-quality hidden surface removal. In *Graphics Hardware 2005*, 7–14.