RenderAnts: Interactive REYES Rendering on GPUs*

Kun Zhou* Qiming Hou† Zhong Ren‡ Minmin Gong‡ Xin Sun‡ Baining Guo†‡

*Zhejiang University †Tsinghua University ‡Microsoft Research Asia

Abstract

We present RenderAnts, the first system that enables interactive REYES rendering on GPUs. Taking RenderMan scenes and shaders as input, our system first compiles RenderMan shaders to GPU shaders. Then all stages of the basic REYES pipeline, including bounding/splitting, dicing, shading, sampling, compositing and filtering, are executed on GPUs using carefully designed data-parallel algorithms. Advanced effects such as shadows, motion blur and depth-of-field can be also rendered with our system. In order to avoid exhausting GPU memory, we introduce a novel dynamic scheduling algorithm to bound the memory consumption during rendering. The algorithm automatically adjusts the amount of data being processed in parallel at each stage so that all data can be maintained in the available GPU memory. This allows our system to maximize the parallelism in all individual stages of the pipeline and achieve superior performance. We also propose a multi-GPU scheduling technique based on work stealing so that the system can support scalable rendering on multiple GPUs. The scheduler is designed to minimize inter-GPU communication and balance workloads among GPUs.

We demonstrate the potential of RenderAnts using several complex RenderMan scenes and an open source movie entitled Elephants Dream. Compared to Pixar's PRMan, our system can generate images of comparably high quality, but is over one order of magnitude faster. For moderately complex scenes, the system allows the user to change the viewpoint, lights and materials while producing photorealistic results at interactive speed.

Keywords: GPGPU, RenderMan, feature-film rendering, shaders, dynamic scheduling, out-of-core texture fetch

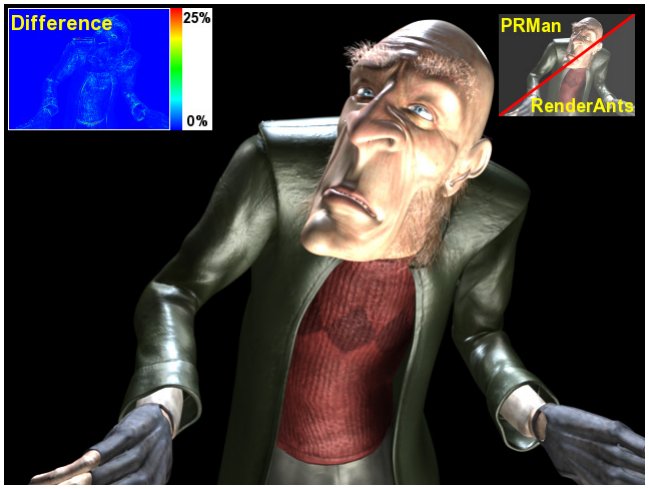


Figure 1: *A character in Elephants Dream, named Proog, is rendered at 640×480 resolution with 8×8 supersampling. The upper-left of the image is rendered with PRMan while the lower-right half is generated using our RenderAnts – the seam is barely visible. The map shown at the top left corner displays error as a percentage of the maximum 8-bit pixel value in the image. RenderAnts renders the image in about 2 seconds on three GeForce GTX 280 (1GB) graphics cards, while PRMan needs 40 seconds on a quad-core CPU. The scene contains 10,886 high-order primitives for the body and clothes, and 9,390 transparent curves for the hair, whiskers, and eyebrows. With RenderAnts, the user can change the viewpoint, lights, and materials while receiving feedback at 2.4 frames per second. See the accompanying video for live demos.*

1 Introduction

The REYES architecture is a successful architecture for photorealistic rendering of complex animated scenes [Cook et al. 1987]. Several REYES implementations, including Pixar's PhotoRealistic RenderMan (PRMan), have become the defacto industrial standard in high-quality rendering, and they have been widely used in film production [Apodaca and Gritz 1999]. While these systems are relatively fast and some of them (e.g., NVIDIA's Gelato) even use GPU acceleration, they are all CPU-based off-line renderers. None of them executes the whole rendering pipeline on the GPU.

In this paper, we present RenderAnts, the first REYES rendering system that runs entirely on GPUs. The name "RenderAnts" refers to the fact that in our system, rendering is performed by tens of thousands of lightweight threads that optimally exploit the massive parallel architecture of modern GPUs. Our system takes RenderMan scenes and shaders as input and generates photorealistic images of high quality comparable to those produced by PRMan. By capitalizing on the GPU's formidable processing power, our system is over one order of magnitude faster than existing CPU-based renderers such as PRMan. With RenderAnts, moderately complex scenes such as those shown in Fig. 1, Fig. 5 and Fig. 12 can be rendered at interactive speed while the user changes the viewpoint, lights and materials.

*http://www.renderants.org

To design a high performance GPU system for REYES rendering, we have to address three major issues. First, we need to map all stages of the basic REYES pipeline to the GPU. While some stages have been successfully implemented in the past [Patney and Owens 2008], handling the other stages remains a challenging issue as noted in [Patney 2008]. Second, we must maximize parallelism while bounding the memory consumption during rendering. CPU-based REYES renderers deal with the memory issue by bucketing. However, a naive adaptation of bucketing to the GPU would lead to suboptimal rendering performance. Finally, we should make the rendering system scalable via multi-GPU rendering so that complex animated scenes can be efficiently processed.

We present a scheme for mapping all stages of the basic REYES rendering pipeline to the GPU using carefully designed data-parallel algorithms. The basic REYES pipeline includes bounding/splitting, dicing, shading, sampling, compositing and filtering stages. Our focus is on stages whose GPU mapping has not been established in previous work. For the shading stage, we develop a shader compiler to convert RenderMan shaders to GPU shaders. Several previously unaddressed issues such as light shader reuse and arbitrary derivative computation are resolved. In addition, the texture pipeline is designed to support out-of-core texture fetches. This is indispensable to feature film production where typical scenes have textures that are too large to be stored in GPU memory. For the sampling stage, we propose a GPU implementa-

tion of the stochastic sampling algorithm described in [Cook 1986]. Besides supporting the basic pipeline, our system can also render advanced effects such as shadows, motion blur and depth-of-field, completely on GPUs.

We also propose a dynamic stage scheduling algorithm to maximize the available parallelism at each individual stage of the pipeline while keeping data in GPU memory. This algorithm significantly improves the rendering performance over a naive adaptation of the bucketing approach. The original REYES system bounds the memory consumption by dividing the screen into small rectangular regions, known as buckets, before entering the rendering pipeline. The buckets are processed one by one during rendering. While this static bucketing works well for CPU-based systems, it is inappropriate for a GPU-based renderer as it significantly slows down rendering. Different rendering stages have different memory requirements. To ensure that a bucket can be successfully processed through all stages, the bucket size must be bounded by the stage with the highest memory requirement. This greatly restricts the available parallelism at other stages and leads to inferior performance. To solve this problem, we add three schedulers to the REYES pipeline, each dynamically adjusting the degree of parallelism (i.e., the amount of data being processed in parallel) in individual stages to ensure that the data fits into the available GPU memory. Thus we can fully exploit the GPU's massive parallelism at all rendering stages without exhausting GPU memory.

Finally, we have designed RenderAnts to support scalable rendering on multiple GPUs by using a multi-GPU scheduling technique to dispatch rendering tasks to individual GPUs. The key to achieving efficient multi-GPU rendering is the minimization of inter-GPU communication, as inter-GPU communication is prohibitively expensive with current hardware architectures. GPUs cannot directly communicate with each other; instead they must communicate through the CPU. This causes several problems. CPU/GPU data transfer is significantly slower than the GPU's processing speed. Moreover, only one GPU can communicate with the CPU at a time, which serializes all communication processes. Our solution is a multi-GPU scheduling algorithm based on work stealing, which can be easily combined with the stage scheduling algorithm described above. The multi-GPU scheduler is also designed to balance workloads among all GPUs.

Our GPU-based REYES renderer has potential in a variety of rendering applications. An immediate example is the acceleration of the preprocessing computation required by recent light preview systems [Pellacini et al. 2005; Ragan-Kelley et al. 2007], which need to cache the visibility information evaluated by the REYES rendering pipeline. Our system also extends the application domain of the REYES architecture from off-line to the interactive domain. With RenderAnts, the user can change the viewpoint, lights and materials while viewing the high-quality rendering results at interactive frame rates. Since the system is linearly scalable to the GPU's computational resources, real-time REYES rendering is achievable in the near future with advances in commodity graphics hardware.

It is important to note that the goal of this paper is not to describe a complete, production-ready REYES rendering system that is functionality-wise competitive to PRMan; instead what we propose is a system that focuses on the basic REYES pipeline on GPUs. The comparison with PRMan is only intended to demonstrate the potential of REYES rendering on GPUs. Although our work only focuses on basic REYES rendering, we believe this is an important step forward because it shows for the first time that it is possible to map all stages of the basic pipeline onto the GPU and significantly improve rendering performance. We are going to release the RenderAnts system as an open platform for future research on advanced REYES rendering on GPUs. Making the source code available online will make research in this promising area more practical for many researchers.

The remainder of this paper is organized as follows. The following section reviews related work. Section 3 briefly describes all stages of the RenderAnts system. In Section 4, we introduce the dynamic scheduling algorithm. The data-parallel algorithms for all individual stages of the basic REYES pipeline are described in Section 5. In Section 6, we describe how to extend the system to support rendering on multiple GPUs. Experimental results are presented in Section 7, followed by the conclusion in Section 8.

2 Related Work

The REYES architecture was designed to be able to exploit vectorization and parallelism [Cook et al. 1987]. Over the past few years, much research has been conducted to seek efficient parallel implementations of the entire rendering pipeline or some of its stages.

Owens et al. [2002] compared implementations of the basic REYES pipeline and the OpenGL pipeline on the Imagine stream processor. Their implementation simplifies the original REYES pipeline considerably in many stages. For example, they employed a screen-space dicing approach whereas REYES performs dicing in the eye space. As noted in [Owens et al. 2002], a huge number of micropolygons are generated, which leads to a significant performance overhead. They also used a simple rasterizer in the sampling stage whereas REYES uses stochastic sampling. Moreover, out-of-core texture access is neglected in their implementation. In order to fully exploit modern GPUs' large-scale parallelism at all stages of the pipeline, it is necessary to design new data-parallel algorithms to map these stages to the GPU.

Lazzarino et al. [2002] implemented a REYES renderer on a Parallel Virtual Machine. The renderer consists of a master and several slave processes. The master divides the screen into buckets, which can be processed independently, and dispatches them to slave processes. A bucket-to-slave allocation algorithm is used to achieve load balancing among slaves. PRMan also has a networking rendering scheme, known as *NetRenderMan*, for parallel rendering on many CPU processors [Apodaca 2000]. With this networking renderer, a parallelism-control client program dispatches work in the form of bucket requests to multiple independent rendering server processes. Our system supports REYES rendering on multiple GPUs. We propose a multi-GPU scheduler to minimize inter-GPU communication and balance workloads among GPUs.

NVIDIA's Gelato rendering system is a GPU-accelerated REYES renderer [NVIDIA 2008]. However, only the hidden surface removal stage of the pipeline is accelerated on the GPU [Wexler et al. 2005]. A technique is also proposed to achieve motion blur and depth-of-field by rendering scenes multiple times into an accumulation buffer, with the number of time or lens samples as a user-supplied parameter. Our system uses a similar approach to render motion blur and depth-of-field. However, since we execute the entire rendering pipeline on GPUs, our approach completely runs on GPUs and can achieve higher performance.

Patney and Owens [2008] demonstrated that the bounding/splitting and dicing stages of the REYES pipeline can be performed in real-time on the GPU. Both stages are mapped to the GPU by using fundamental parallel operations of scan and compact [Harris et al. 2007]. Patney [2008] also described the probability of mapping other stages to the GPU and listed challenging issues. Most of these issues are resolved in our work. In addition to mapping the entire REYES pipeline to GPUs using well-designed data-parallel algorithms, we introduce a dynamic scheduling algorithm to maximize the available parallelism in individual stages and thus greatly im-

prove the overall rendering performance. We also design a multi-GPU scheduler for efficient rendering on multiple GPUs.

Recently, several techniques have been developed for high-quality preview of lighting design in feature film production [Pellacini et al. 2005; Ragan-Kelley et al. 2007]. These methods cache visibility information evaluated in the REYES pipeline as deep or indirect framebuffers during preprocessing and later use these framebuffers to perform interactive relighting at runtime. Our work explores a different direction: we focus on implementing the REYES pipeline on GPUs, which can be used to significantly speed up the preprocesses of these techniques. Since our system takes RenderMan scenes and shaders as input, we develop a shader compiler to compile RenderMan shaders to GPU shaders. Although a few methods have been proposed to perform this compilation [Olano and Lastra 1998; Peercy et al. 2000; Bleiweiss and Preetham 2003; Ragan-Kelley et al. 2007], some problems such as light shader reuse and arbitrary derivative computation have not been addressed before. Our shader compiler provides good solutions to these problems. Our out-of-core texture fetching mechanism is similar to GigaVoxel [Cyril et al. 2009]. The key difference is that GigaVoxel only implements the out-of-core function in a specific rendering algorithm, while our system is capable of adding out-of-core support to general, arbitrary shaders.

We implemented the RenderAnts system using BSGP [Hou et al. 2008], a publicly available programming language for general purpose computation on GPUs. BSGP simplifies GPU programming by providing several high level language features. For example, it allows the programmer to pass intermediate values using local variables as well as to call a parallel primitive having multiple kernels in a single statement. We also employed the GPU interrupt mechanism and debugging system described in [Hou et al. 2009] to assist our development. The interrupt mechanism is a compiler technique that allows calling CPU functions from GPU code. As described later in the paper, all of our algorithms can also be implemented using other GPU languages such as CUDA and OpenCL.

3 System Overview

Fig. 2 shows the basic pipeline of RenderAnts running on a single GPU. It follows the REYES pipeline with three additional schedulers (drawn in red). The input of the system is RenderMan scenes and shaders, which are written in the RenderMan Shading Language (RSL), a C-like language designed specifically for shading. After converting all RenderMan shaders to GPU shaders in a preprocess using the shader compiler described in Section 5.2, we execute the following stages to produce the final picture.

- **Bucketing** In the beginning of the pipeline, the screen is divided into small *buckets*, which are processed one at a time. Only those primitives which affect the current bucket are rendered in the subsequent stages. This scheme is used to reduce the memory footprint during rendering.
  In existing CPU-based renderers, the bucket size is bounded by the stage that has the peak memory requirement in the pipeline. In our system, since individual stages have their own schedulers as described later, the bucket size only needs to satisfy the memory requirement of the bounding/splitting stage, i.e., the data size of all primitives in each bucket should be less than the currently available GPU memory. Unless mentioned otherwise, all images shown in this paper are rendered using a single bucket – the whole screen.
- **Bound/Split** For each input primitive whose bounding box overlaps with the current bucket, if the size of its bonding box is greater than a predetermined bound, it is split into smaller primitives, which follow the same procedure recursively. At the end of the stage, all primitives are ready for dicing.

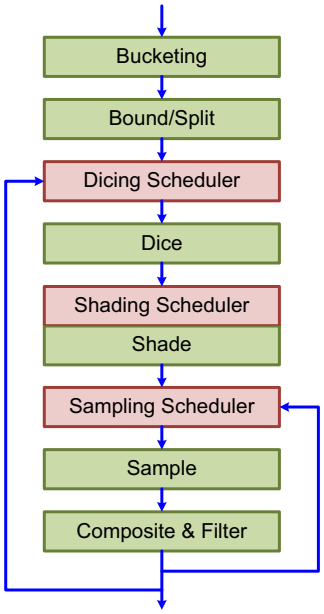


Figure 2: *RenderAnts system pipeline. Three stage schedulers (in red) are added to the basic REYES pipeline.*

- **Dicing Scheduler** The dicing scheduler splits the current bucket into *dicing regions*, which are dispatched to the dicing and subsequent stages one at a time. The memory required to process each dicing region should be less than the currently available GPU memory.
- **Dice** Every primitive in the current dicing region is subdivided into a regular *grid*, each having a number of quads known as *micropolygons*. The micropolygon size in screen space is constrained by the *shading rate*, a user-specified parameter. Unless mentioned otherwise, all rendering results shown in this paper are rendered with shading rate 1.0, which means that the micropolygon is no bigger than one pixel on a side. In our current implementation, each primitive generated from the bounding/splitting stage is no greater than 31 pixels on a side. Therefore each grid has at most 31×31 micropolygons.
- **Shading Scheduler** The shading scheduler works inside the shading stage. For each GPU shader that is converted from a RenderMan shader, the scheduler splits the list of micropolygons into sublists before shader execution. The sublists are to be shaded one by one, and each sublist should be shaded with the currently available GPU memory.
- **Shade** Each vertex of the micropolygon grids generated after dicing is shaded using GPU shaders.
- **Sampling Scheduler** The sampling scheduler splits the current dicing region into *sampling regions*, which are dispatched to the sampling and subsequent stages one at a time. The memory required to process each sampling region should be less than the currently available GPU memory.
- **Sample** All micropolygons in the current sampling region are sampled into a set of *sample points* by using the jittering algorithm described in [Cook 1986].
  Each pixel in the current sampling region is divided into a set of *subpixels*. Each subpixel has only one *sample location*, which is determined by adding a random displacement to the location of the center of the subpixel. Each micropolygon is tested to see

(a) micropolygons (b) dicing regions (c) sampling regions

Figure 3: *Micropolygons, dicing and sampling regions generated when rendering Proog. Note that for the purpose of visualization, micropolygons in (a) are generated at a shading rate of 400, and the dicing/sampling regions are generated at a shading rate of 0.1.*

if it covers any of the sample locations. For any sample location that is covered, the color, opacity, and depth of the micropolygon are interpolated and recorded as a sample point.

- **Composite and Filter** The sample points generated in the sampling stage are composited together to compute the color, opacity and depth values of all subpixels in the current sampling regions. The final pixel colors are then computed by blending the colors and opacities of subpixels. Note that we do not have a scheduler for the compositing and filtering stage because the memory requirement at this stage is similar to that of the sampling stage. The sampling scheduler already takes into account the memory usage at this stage.

Currently all these stages are executed in the GPGPU pipeline via BSGP. While the traditional graphics pipeline (i.e., the hardware rasterizer and vertex/geometry/pixel shaders) is more suitable for certain tasks, currently we are unable to utilize them in the REYES pipeline due to some practical reasons including interoperability issues, the lack of exact rasterizer behavior specification and relatively high switch cost between GPGPU/graphics mode.

4 Dynamic Scheduling

The key idea of dynamic scheduling is to estimate the memory requirements at individual stages of the rendering pipeline and maximize the degree of parallelism in each stage while making sure that the data fits into the available GPU memory.

As described in the last section, we have three schedulers for the dicing, shading and sampling stages, respectively. The dicing stage always consumes much less memory than subsequent stages because the micropolygons generated after dicing consume a lot of memory and these micropolygons need to be retained until the end of the REYES pipeline. Based on this observation, our dicing scheduler first divides the current bucket into a set of dicing regions, which are processed one by one. The schedulers of subsequent stages then operate on the micropolygons in the current dicing region.

Dicing Scheduler The dicing scheduler recursively splits a screen region using binary space partitioning (BSP). It begins with the current bucket and the list of primitives contained in this bucket. For a given region, the scheduler first estimates the peak memory required to process this region. If the peak fits in currently available memory minus a constant amount, the region and all primitives in it are dispatched to the dicing and subsequent stages. Otherwise, the region is split into two subregions at the middle point of the longer axis. The scheduler then recurses to process the two subregions, with the one having fewer primitives being processed first.

```
schedule(quad r, list(prim) l)
    if memoryUse(r,l)<=memoryFree()-C:
        process(r,l)
        return
    (r0,r1) = BSPSplit(r)
    (n0,n1) = countPrimInQuads(l,r0,r1)
    if n0>n1: swap(r0,r1)
    l0 = primInQuad(l,r0)
    schedule(r0,l0)
    delete l0
    //overwrite l
    l = primInQuad(l,r1)
    schedule(r1,l)
```

Listing 1: *Pseudo code of the dicing scheduler.*

The pseudo code of this process is shown in Listing 1.

Fig. 3(b) shows the dicing regions generated by this process. The constant amount of memory (`C` in Listing 1) is reserved for the subsequent shading and sampling stages which have their own schedulers. The value of `C` can be adjusted using an `Option` statement in RIB (RenderMan Interface Bytestream) files. For all examples shown in our paper, `C` is set as 32MB. Note that the finally dispatched dicing regions do not necessarily consume all memory available to the dicing stage (i.e., `memoryFree()-C`). Therefore, the memory available to the subsequent stages is typically much larger than `C`.

The function `memoryUse` estimates the peak memory required to process a region in the dicing stage, which is caused by the micropolygons generated after dicing. Recall that each primitive is subdivided into a grid. The size of micropolygon data in each grid can be computed exactly. The memory peak of the dicing stage thus can be accurately estimated by summing up the micropolygon data sizes of all primitives in the region using a parallel reduce operation.

Note that the scheduler dispatches all tasks in strictly sequential order and operates in a strictly DFS (depth first search) manner. Subregions are dispatched to the dicing and subsequent stages one at a time. After a subregion has been completely processed, its intermediate data is freed and all memory is made available for the next subregion. Currently we do not reuse information across subregions. Primitives overlapping multiple subregions are re-diced in every subregion.

Shading Scheduler Unlike the dicing scheduler which is executed prior to the dicing stage, the shading scheduler works inside the shading stage. For each GPU shader, the scheduler estimates before shader execution the memory peak during shader execution and computes the maximal number of micropolygons that can be processed with the currently available memory. The input micropolygon list is split into sublists according to this number and the

sublists are shaded one by one.

The memory peak in the shading stage is caused by the temporary data structures allocated during shader execution. The temporary data size is always linear to the number of micropolygons. However, the exact coefficients are different for different shaders. A typical scene may contain many different shaders, with significant variation in per-micropolygon temporary data size. Estimating the memory peak of the whole shading stage will result in overly conservative scheduling and thus leads to suboptimal parallelism for many shaders. Therefore, we design the shading scheduler to work for every shader execution instead of the whole shading stage.

Sampling Scheduler Like the dicing scheduler, the sampling scheduler recursively splits a screen region using BSP. The main difference is the peak memory estimation algorithm. The sampling scheduler needs to estimate the memory peak of the sampling, compositing and filtering stages. This peak, reached during the compositing stage, is caused by the subpixel framebuffer and the list of sample points. We estimate the total number of sample points using the same algorithm in the sampling stage (see Section 5.3 for details). The framebuffer size equals the region size.

Another difference is that the sampling scheduler operates within the current dicing region, whereas the dicing scheduler operates within the current bucket. As an example, Fig. 3(c) shows the sampling regions generated by our algorithm for the Proog scene.

Design Motivation Note that the most obvious solution to the memory problem is to have a full virtual memory system with paging. This is actually the first solution we attempted. Based on the GPU interrupt mechanism described in [Hou et al. 2009], we implemented a prototype compiler-based GPU virtual memory system during preliminary feasibility evaluation of the RenderAnts project. However, we found that it is unrealistically expensive to heavily rely on paging in massive data-parallel tasks. Paging is especially inefficient when managing the input/output streams of data-parallel kernels (e.g., micropolygons, sample points) – the page faults are totally predictable, and paging can usually be entirely avoided by simply processing less data in parallel. This observation motivated us to prevent data from growing out of memory rather than just paging them out – leading to our current memory-bounded solution.

5 GPU REYES Rendering

In this section we describe the GPU implementation of each stage of the basic REYES pipeline.

5.1 Bound/Split and Dice

Our GPU implementation of the bounding/splitting and dicing stages follows the algorithm described in [Patney and Owens 2008]. In the bounding/splitting stage, all input primitives are stored in a queue. In every iteration of the bounding/splitting loop, the primitives in this queue are bound and split in parallel. The resulting smaller primitives are written into another queue, which is used as input for the next iteration. The parallel operations of scan and compact [Harris et al. 2007] are used to efficiently manage the irregular queues. When the bounding/splitting stage finishes, all primitives are small enough to be diced.

The dicing stage is much simpler. In parallel, all primitives in the current dicing region are subdivided into grids, each of which has at most 31×31 micropolygons.

Although [Patney and Owens 2008] only handles Bézier patches, our system supports a variety of primitives, including bicubic patches, bilinear patches, NURBS, subdivision surface meshes, triangles/quads, curves, and spheres.

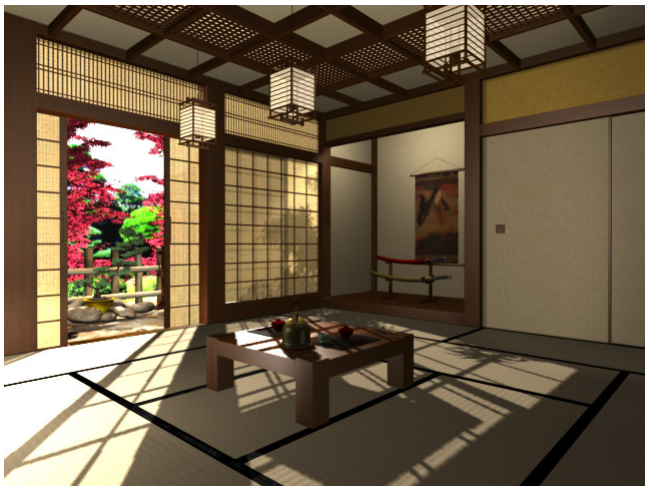


Figure 4: *This indoor scene has about one-half gigabytes of textures and contains 600K polygon primitives. At 640×480 resolution with 4×4 supersampling, our system renders the scene at about 1.3 frames per second when the user is walking around in the room. See the accompanying video for live demos.*

5.2 Shade

To perform shading computations on GPUs, we need a shader compiler to automatically translate RenderMan shaders to GPU shaders.

Our system supports four basic RenderMan shader types: displacement, surface, volume, and light. The first three types of shaders are bound to objects. During the shading stage, they are executed on all vertices of micropolygon grids generated by the dicing stage. The output of these shaders are displaced vertex positions, colors, and opacity values. Light shaders are bound to light sources. They are invoked when a displacement/surface/volume shader executes an *illuminance* loop.

For each object, our renderer composes its displacement, surface, volume shaders, and light shaders from all light sources that illuminate this object into a *shading pipeline*. The shader compiler is called to compile each shading pipeline into a BSGP GPU function. The function is inserted into a BSGP code stub that spawns shading threads and interfaces with the dicer and sampler, yielding a complete BSGP program. This program is then compiled into a DLL (Dynamically Linked Library) and loaded during shading. To maximize parallelism, we spawn one thread for each vertex. Therefore, the function generated by our shader compiler only shades one vertex.

Note that the memory required to shade one micropolygon vertex is proportional to the maximum live variable size at texture/derivative instructions in the shader. In our experiments, this is always less than 400 bytes per vertex. This value grows approximately logarithmically with respect to the shader length. The memory consumption of a 2000-line shader is only a few dozens of bytes larger than a 50-line shader.

In the following, we describe several algorithmic details for implementing our shader compiler and the shading stage.

Out-of-core Texture Fetch Typical scenes in film production have a few large textures within a single shot. It is impossible to store all texture data in the GPU memory. Simply translating RSL

Figure 5: *RenderAnts rendering of a car parked in front of a house. The scene is illuminated by 22 lights with 8.8K lines of surface shaders and 700 lines of light shaders. At 720×405 resolution with 3×3 supersampling, our system renders at about one frame per second when the user is changing the viewpoint. Top right: changing light positions results in different shadows. Bottom right: viewing the scene from a different viewpoint.*

texture fetches into native GPU instructions does not work for such scenes because it requires all texture data to reside in the GPU memory. We need a mechanism to handle out-of-core texture fetch.

REYES uses an out-of-core algorithm to manage textures [Peachey 1990]. Textures are split into fixed-sized 2D tiles. Whenever a texture fetch accesses a non-cached tile, the tile is loaded into the memory cache. This mechanism allows arbitrarily large textures to be efficiently accessed through a relatively small memory cache.

To map this algorithm to the GPU, we split the texture pipeline into a GPU texture fetcher and a CPU-side cache manager. Our compiler compiles each RSL texture fetch into an inline function call to the GPU fetcher, while the cache manager is a static component shared by all compiled shaders. Whenever a texel is required in a shader, the GPU fetcher is called to fetch the texel from a GPU-side texture cache which contains a number of tile slots packed as a large hardware 2D texture. The GPU fetcher uses a hash table to map the texture file name and tile position to texture coordinates on the cache texture. If the requested tile is not in the cache, the fetcher calls a GPU interrupt [Hou et al. 2009] to fetch the requested tile. The interrupt handler computes a list of required tile IDs and calls the CPU-side cache manager. The cache manager then reads the required tiles from the disk, copies them to the GPU, and rebuilds the mapping hash table.

Raw textures are preprocessed into mipmaps stored in tiles before rendering begins. We let neighboring tiles overlap by one texel so that we can utilize the GPU's hardware bilinear filtering. Both the cache texture and the address translation table have fixed sizes. They are allocated in the beginning of our pipeline and thus do not interfere with scheduling.

Light Shader Reuse A typical RSL surface shader may call illuminance loops multiple times to compute various shading components. This is especially true for shaders generated from shade trees [Cook 1984]. In such shaders, individual shading components such as diffuse and specular terms are computed in individual functions and each function has a separate illuminance loop. As a result, each illuminance loop would execute light shaders for all light sources, which is very inefficient. This problem is illustrated in Listing 2.

Conventional CPU-based renderers solve this problem by caching light shader results during the first execution and reusing it in subsequent illuminance loops with equivalent receiver configurations. This caching approach, however, is inappropriate for the GPU. While we know all light shaders at compile time, we do not know the number of lights that use each shader. Therefore, the size required for the light cache is known only at runtime. Current GPUs do not support dynamic memory allocation, which makes runtime light caching impractical.

Original code	After illuminance merging

`//DiffusePart`
`color Cd=0;`
`illuminance(P){`
`  Cd+=lambert();`
`}`
`//SpecularPart`
`float r=0.2;`
`color Cp=0;`
`illuminance(P){`
`  Cp+=blinn(r);`
`}`
`//combination`
`Ci=Cd+Cp;`
	`color Cd=0;`
	`float r=0.2;`
	`color Cp=0;`
	`illuminance(P){`
	`  //merged loop`
	`  Cd+=lambert();`
	`  Cp+=blinn(r);`
	`}`
	`//combination`
	`Ci=Cd+Cp;`

Listing 2: *Pseudo code demonstrating illuminance merging.*

To address this issue, we seek to reduce light shader execution using compile time analysis. Specifically, we find illuminance loops with equivalent receiver configurations and merge them into a single loop. During merging, we first concatenate all loop bodies. Then we find all values read in the concatenated loop body and place their assignments before the merged loop. This is illustrated in Listing 2. Note that the variables `Cp` and `r` are used in the later specular illuminance loop and they have to be placed before the merged illuminance.

The merge may fail in cases where one illuminance loop uses a value defined by another illuminance, e.g., if a surface has its specularity computed from its diffuse shading. We check for this sort of data dependency prior to illuminance merging as a precautionary measure. In practice, such data dependencies are not physically meaningful. They have never occurred in any of our shaders. Our compiler consistently succeeds in merging all illuminance loops. Optimal light reuse is achieved without any additional storage.

Our shader compiler first compiles shaders to static single assign-

Figure 6: *696K blades of grasses rendered at 2048×1536 with 11×11 supersampling. This generates 30.1M micropolygons and 4.7G sample points. The rendering time of RenderAnts and PRMan are 23 and 1038 seconds, respectively.*

ment (SSA) form as in [Cytron et al. 1991] and then performs dataflow analysis on this SSA form for light reuse and derivative computation as described below. Note that the term *light shader reuse* has a different meaning here when compared with the *light reuse* in previous lighting preview systems such as [Pellacini et al. 2005; Ragan-Kelley et al. 2007]. In our system, light shader reuse refers to the reuse of light shader output across different shading components during shader execution. In a lighting preview system, light reuse refers to reusing the shading result from unadjusted light sources during lighting design. They are completely different techniques used in completely different rendering stages.

Derivative Computation Modern RSL shaders use derivatives intensively to compute texture filter sizes for the purpose of anti-aliasing. Derivative computation needs to get values from neighborhood vertices, which we have to fetch through inter-thread communication.

We use a temporary array in global memory to get values from neighborhood vertices. An alternative is to use CUDA shared memory. While shared memory is more efficient, it is limited to threads within the same CUDA thread block. For derivatives, this implies that each grid has to fit in a single CUDA block. Unfortunately, our grids are up to 32×32 in size and do not always fit in a block. In addition, we find that the performance gain of using larger grids outweighs the entire derivative computation. Therefore, the communication required for derivatives has to use global memory.

Inter-thread communication via global memory requires barrier synchronization to ensure that all threads have computed the values to exchange. This poses a problem: barriers cannot be used in non-uniform flow control structures, whereas derivative instructions are not subjected to this limitation. To address this issue, our shader compiler relocates all derivative computation to valid barrier positions.

For each derivative instruction, there may be multiple valid positions for relocation. We need a way to find an optimal relocation so that the number of required barriers is minimized. Observing that consecutive derivative instructions can be relocated to the same barrier, we find an optimal relocation by minimizing the number of distinct target positions to relocate derivatives to. To do this, we first eliminate trivially redundant derivatives, i.e., multiple derivative instructions of the same value. After that, we find all valid relocation positions for each derivative. A graph is constructed for the derivatives. Each derivative corresponds to a node in the graph. An edge is created for each pair of derivatives that can be relocated to the same position. The minimal barrier derivative relocation corresponds to the minimal clique cover of this graph. The number of derivative instructions is typically very small after eliminating trivially redundant ones. We simply use an exhaustive search to compute the optimal solution.

Note that derivatives are only well-defined for variables that have a defined value for all vertices in a grid. This guarantees our derivative relocation to be successful. A minor problem is that BSGP does not allow placing `barrier` statements in uniform flow-control structures. In such cases, we implement the synchronization using a GPU interrupt [Hou et al. 2009].

Listing 3 illustrates our derivative relocation process. In the original code, there are four derivative instructions `Du`. All of them are written in flow control structure. After the derivative relocation, the derivatives are pulled out and redundant ones are eliminated. The compiler can then proceed to insert barriers and `thread.get` calls to compute these derivatives.

Original code	After derivative relocation

`if(swapst!=0){`
`  dsdu=Du(t);`
`  dtdu=Du(s);`
`}else{`
`  dsdu=Du(s);`
`  dtdu=Du(t);`
`}`
	`tmp0=Du(s);`
	`tmp1=Du(t);`
	`if(swapst!=0){`
	`  dsdu=tmp1;`
	`  dtdu=tmp0;`
	`}else{`
	`  dsdu=tmp0;`
	`  dtdu=tmp1;`
	`}`

Listing 3: *Pseudo code demonstrating derivative relocation.*

Other Features RSL shaders use strings to index textures and matrices. We follow the approach in [Ragan-Kelley et al. 2007] to implement strings as integer tokens. Corresponding texture information and matrices are organized into arrays indexed by these tokens and sent to the GPU prior to shading.

Shaders within a shading pipeline may communicate with each other by exchanging variable values through message passing. Since we compile each shading pipeline into a single function, variables in different shaders actually belong to the same local scope in the final BSGP function. We simply replace message passing functions with variable assignments after inline expansion.

Our system currently does not support imager shaders written in RSL. Instead, a post-processing function written in BSGP is substituted in place of the imager shader in the original pipeline. After rendering each frame, the renderer calls this function with a pointer to the output image, and overwrites the output image with the post-processed image. The user can write his/her own post-processing function to implement any desired effect. This post-processing feature is used to compute the color adjustment and HDR glowing in rendering Elephants Dream shots. Note that the PRMan version 13 that we use also provides its own scriptable compositor tool "it" for post-render processing, and does not support RSL imager shaders [Pixar 2007].

5.3 Sample

The sampling stage stochastically samples micropolygons into a set of sample points. Each micropolygon is first bounded in the screen space. If the micropolygon is completely outside of the current sampling region, it is culled. Otherwise, we test the micropolygon to see if it covers any of the predetermined sample locations

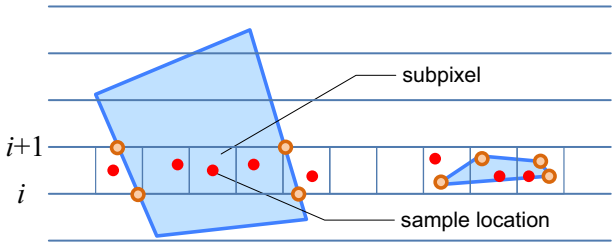


Figure 7: *Two micropolygons are sampled at sample locations between the i-th and $i+1$-th scan lines.*

in its bounding box. For any sample location that is covered, the color, opacity, and depth of the micropolygon are interpolated and recorded as a sample point. We use the jittering algorithm [Cook 1986] to determine the sample locations.

As described in Section 3, the jittering algorithm divides each pixel into subpixels. Each subpixel has only one sample location, which is determined by adding a random displacement to the location of the center of the subpixel. The random displacement is computed solely from the subpixel's coordinates in the screen space. To map this algorithm to the GPU, we take a two-step approach. The first pass conservatively estimates the number of sample points of each micropolygon, computes the required memory size for all sample points, and allocates the memory. The second pass computes and stores the actual sample points. In both steps, we parallelize the computation over all micropolygons.

In the first step, we scan the bounding box of each micropolygon line-by-line from bottom to top. The interval between adjacent lines is set to be 1 subpixel. For the i-th line, the intersections of the line and the micropolygon in screen space are computed as shown in Fig. 7. Suppose that the set of intersections is represented as P_i. The number of sample points of the micropolygon lying between the i-th and i+1-th lines is no greater than $R_i - L_i + 1$, where

$$R_i = \lceil \max\{p.x, p \in P_i \cup P_{i+1}\} \rceil \text{ and}$$
$$L_i = \lfloor \min\{p.x, p \in P_i \cup P_{i+1}\} \rfloor .$$

$p.x$ denotes the horizontal coordinate of point p in the screen space. Note that the horizontal coordinates of the micropolygon's vertices that are located between the i-th and i+1-th lines are also included in the above formula to estimate the number of sample points.

After estimating the number of sample points of each micropolygon, we use a parallel scan operation to compute the required memory size for sample points of all micropolygons and compute the starting addresses of each micropolygon's sample points in memory. Finally, a global memory buffer of the required size is allocated for sample points.

In the second step, we scan each micropolygon again in the same manner as in the first step. For the i-th line, L_i and R_i are computed again. For each subpixel between L_i and R_i, we compute the sample location by adding a random displacement to the location of the center of the subpixel and then test if the micropolygon covers the sample location. If the sample location is covered, a sample point is generated by interpolating the color, opacity and depth values of the micropolygon, and the sample point is contiguously written into the sample point buffer. Note that the sample point needs to record the index of its associated subpixel. The starting address of the micropolygon is used to ensure that its sample points are written into the right places without conflicting with other micropolygons' sample points. At the end of this step, the sample points of all micropolygons are stored in the sample point buffer, ordered by the indices of micropolygons. Note that for opaque micropolygons, we atomically update the depth values and colors of the covered subpixels using atomic operations supported by the NVIDIA G86 (or above) family of graphics cards, and do not generate the sample points. The depth values of subpixels are stored in the z buffer.

Note that there are other methods for estimating the number of sample points of a micropolygon. For example, we can simply compute an upper bound of sample points based on the area of the micropolygon's bounding box. This saves the line scanning process in the first step, but leads to a larger memory footprint and a higher sorting cost in the compositing stage (see Section 5.4). Our approach is able to compute a tighter bound, resulting in an overall gain in performance.

5.4 Composite & Filter

In this stage, the sample points generated in the sampling stage are composited together to compute the final color, opacity and depth values of all subpixels. The final pixel colors are then computed by blending the colors and opacities of subpixels.

Composite In order to perform composition for each subpixel, we need to know the list of its sample points, sorted by depth. To get this information, we sort the sample points using their associated subpixel indices and depths as the sort key. In particular, the depth values are first converted to 32-bit integers and packed with the subpixel indices into 64-bit code. The lower 32 bits indicate the depth value and the higher 32 bits are for the subpixel index. Then the binary search based merge sort algorithm described in [Hou et al. 2008] is used to sort the sample points. After sorting, sample points belonging to the same subpixel are located contiguously in the buffer, sorted by depth.

Note that some elements in the sample point buffer may not contain any sample point because the number of sample points of each micropolygon is over-estimated in the sampling stage. After sorting, these empty elements will be contiguously located in the rear of the buffer since their subpixel indices are initialized to be -1 during memory allocation. They will not be processed in subsequent computation.

After sorting, we generate a unique subpixel buffer by removing elements having the same subpixel indices in the sorted sample point buffer. We do this through the following steps. First, for each element of the sorted buffer, the element is marked as invalid if its subpixel index equals that of the preceding element in the buffer. Then, the compact primitive provided in BSGP is used to generate the unique subpixel buffer which does not contain invalid elements. During this process, for each element of the subpixel buffer, we record the number of sample points belonging to this subpixel and the index of the first sample point in the sample point buffer.

Finally, in parallel for all subpixels in the subpixel buffer, the sample points belonging to each subpixel are composited together in a front-to-back order until the depth of the sample point is greater than the depth of the subpixel in the z buffer.

Filter We perform filtering for all pixels in the current sampling region in parallel. For each pixel, the color and opacity values of its subpixels are retrieved and blended to generate the color and opacity of the pixel. The depth value of the pixel is determined by properly processing the depth values of its subpixels according to the depth filter option (e.g., min, max, or average). The pixels are sent to the display system to be put into a file or a framebuffer.

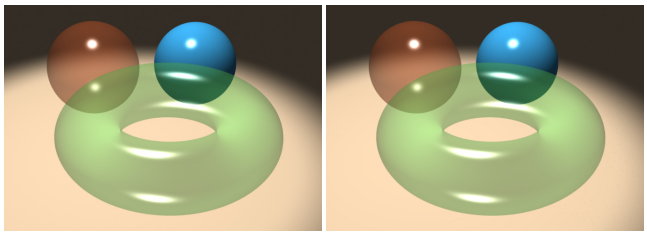


Figure 8: *Transparency. Left: PRMan. Right: RenderAnts. The two images are visually identical, with the root mean squared error (RMSE) equal to 1.41.*

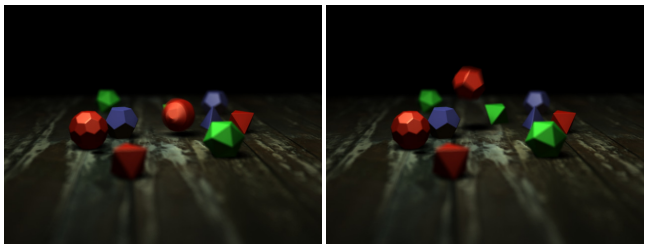


Figure 9: *Motion blur and depth-of-field: two frames from an animation. The scene is rendered at 640×480 resolution with 8×8 supersampling. The rendering time of RenderAnts and PRMan are 1.37 and 13 seconds, respectively.*

5.5 Advanced Features

Besides the basic REYES rendering pipeline, our system can also render shadows, motion blur and depth-of-field directly on GPUs.

We render shadows using shadow maps with percentage closer filtering [Reeves et al. 1987]. Shadow maps are generated in *shadow passes*. In each shadow pass, a depth map is rendered from the point of view of a light source, using the basic REYES pipeline. Shadow maps are managed using the out-of-core texture fetching system. Therefore, the number of shadow maps is not constrained by the number of texture units on hardware.

We implement motion blur and depth-of-field by adapting the accumulation buffer algorithm [Haeberli and Akeley 1990] to the REYES pipeline. Here we use motion blur as an example to illustrate our implementation. Each subpixel is assigned a unique sample time according to a randomly-created prototype pattern [Cook 1986]. Primitives are interpolated and rendered multiple times for a series of sample times. At each rendering time, only those subpixels whose sample time is equal to the current rendering time are sampled. Then the same compositing and filtering stage described above is applied to generate the final results. Depth-of-field can be similarly handled. Each subpixel is assigned a sample lens position and primitives are rendered from various lens positions.

Note that unlike the stochastic sampling algorithm described in [Cook 1986], which jitters the prototype sample time of each subpixel, we directly use the prototype sample time in the sample stage. This is a tradeoff between rendering quality and performance – if the prototype sample time is jittered, we need to estimate the number of sample points covered by micropolygons over a period of time, which is very expensive.

Our implementation of motion blur and depth-of-field needs to render primitives multiple times. Although this always gives accurate rendering results for arbitrary motions, the computation is expensive, especially for scenes with complex shaders and many light sources. In the future, we are interested in investigating methods to shade moving primitives only at the start of their motion as described in [Apodaca 2000].

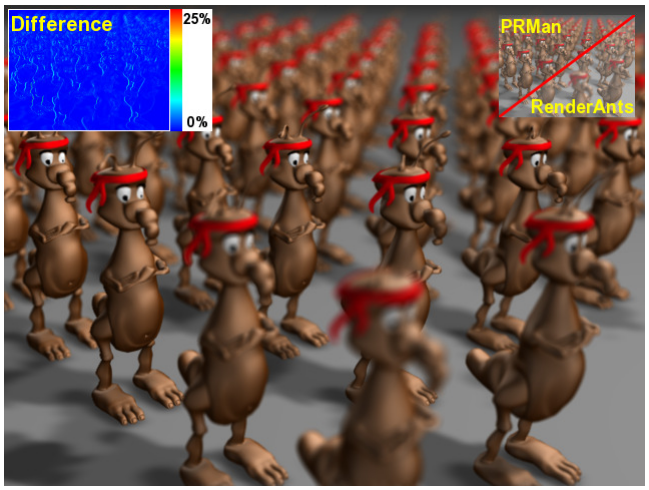


Figure 10: *Depth-of-field: an army of 100 ants rendered at 640×480 resolution with 13×13 supersampling. In total, RenderAnts renders the scene 169 times, shades 545.5M micropolygons, and generates 328.1M sample points in producing this image. Our rendering time on three GPUs is 26 seconds, compared to 133 seconds with PRMan on a quad-core CPU.*

6 Multi-GPU Rendering

In this section we describe the extension of our system to support efficient REYES rendering on multiple GPUs.

As shown in Fig. 11, the dicing scheduler on each GPU is enhanced by a multi-GPU scheduler which is responsible for dispatching rendering tasks to individual GPUs. All other schedulers and stages remain unchanged. To design an efficient multi-GPU scheduler, we need to solve two key problems: minimizing inter-GPU communication and load balancing among GPUs.

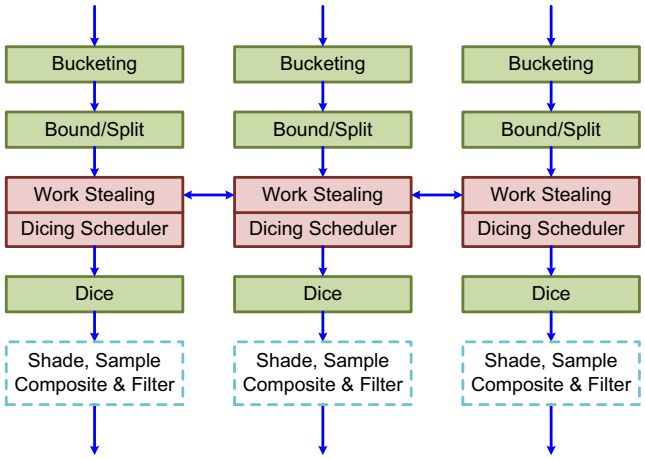


Figure 11: *Multi-GPU rendering with RenderAnts.*

Our multi-GPU scheduler is based on work stealing [Blumofe et al. 1995] and is combined with the dicing scheduler. The dicing scheduler runs on each GPU as usual. Whenever a GPU becomes idle (i.e., its DFS stack of unprocessed subregions becomes empty), it checks other GPUs' DFS stacks for unprocessed subregions. If such

```
thread local storage:
    list(quad) stack
multi_schedule(quad r, list(prim) l)
    if memoryUse(r,l)<memoryFree()-C:
        process(r,l)
        return
    (r0,r1) = BSPSplit(r)
    (n0,n1) = countPrimInQuads(l,r0,r1)
    if n0>n1: swap(r0,r1)
    //push the larger region explicitly
    stack.push(r1)
    l0 = primInQuad(l,r0)
    schedule(r0,l0)
    delete l0
    //return if the other region is stolen
    if stack.empty(): return
    r1 = stack.pop()
    l = primInQuad(l,r1)
    schedule(r1,l)

multi_main(list(prim) l)
    if inMainThread():
        r = bucketQuad()
        multi_schedule(r,l)
    while renderNotDone():
        r = stealTask()
        multi_schedule(r,primInQuad(l,r))
```

Listing 4: *Pseudo code of the multi-GPU scheduling algorithm.*

a region is found, the idle GPU steals a region from the stack *bottom*. It adds the region to its own stack and removes it from the original one. The GPU then proceeds to process the stolen region. The pseudo code of the multi-GPU scheduler is shown in Listing 4. Note that this work stealing scheduler does not involve any significant computation and is implemented on the CPU. One CPU scheduling thread is created to manage each GPU. All stack operations are done by these CPU threads.

Recall that the dicing scheduler requires a region and a list of primitives contained in this region. Stealing the primitive list along with the region requires more inter-GPU communication, which is expensive and intrinsically sequential. To avoid this problem, we maintain a complete list of all primitives on all GPUs. When a GPU steals a region, it recomputes the list of primitives in this region using the complete list. This way, work stealing only requires transferring one region description, a simple 2D quad.

Some preparations are required to set up this work stealing scheduler. At the beginning of the pipeline, all scene data is sent to all GPUs. Each GPU performs the bounding/splitting stage once to compute the complete primitive list. This redundant computation is designed to avoid inter-GPU communication. Before executing the scheduler, a region equal to the current bucket is pushed onto the first GPU's stack and other GPUs are set to idle.

Another important problem is load balancing. For the work stealing scheduler to achieve good load balance, the total number of subregions cannot be too small. Otherwise, some GPUs cannot get regions and will remain idle. Generating many very small subregions would not be good either because that would lead to suboptimal parallelism on each individual GPU. Our scheduler deals with this issue using an adaptive subregion splitting criterion. We first set a primitive count threshold n_{min} such that good load balancing can be expected if all subregions contain no more than n_{min} primitives. Subregions that fit in memory and contain fewer than n_{min} primitives are never split. When a scheduling thread encounters a subregion that fits in available memory while containing more than n_{min} primitives, it checks whether the work queue of any other GPU is empty. If such a GPU is found, the subregion is split. Otherwise, it is dispatched for processing. This strategy allows an adaptive tradeoff between parallelism and load balancing. It worked well in all our experiments.

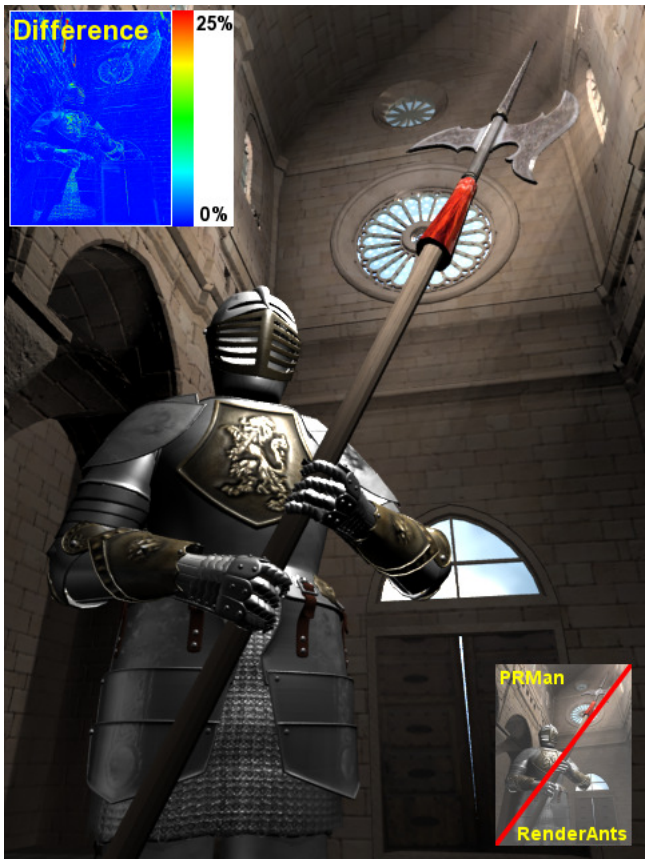


Figure 12: *Armor with abundant geometric details is placed in a church. Lights shining through windows cast volumetric shadows, or light shafts. At 480×640, RenderAnts shades 2.7M micropolygons and composites 41.4M sample points in rendering this image. The view pass time with RenderAnts is about 1.2 seconds, allowing the user to change the viewpoint at 0.8 frames per second.*

Once a region finishes its rendering on a GPU, the rendering result (i.e., the pixel colors) is sent to the CPU and stored. After all regions have finished rendering, the final image is put into a file or sent to a GPU for display.

Design Motivation Note that our multi-GPU scheduling strategy comes out of our hard learned lessons. Our initial design was a general task stealing scheduler aimed at balancing workloads in all stages. However, this design did not work out. For a long period, we were unable to achieve any performance gain, as the inter-GPU task migration cost consistently canceled out any improvement in load balancing. We eventually switched strategy and carefully redesigned the scheduler to eliminate all significant communication. The task representation was designed to allow each GPU to quickly re-compute all necessary data from very little information, and non-profitable parallelization was replaced by redundant computation. The current strategy works reasonably well on our large test scenes (see Fig. 14 in Section 7).

7 Results and Discussions

We have implemented and tested the RenderAnts system on an AMD 9950 Phenom X4 Quad-Core 2.6GHz processor with 4GB RAM, and three NVIDIA GeForce GTX 280 (1GB) graphics cards.

Rendering Quality We use our system to render a variety of scenes

	Proog (Fig. 1)	**Ants** (Fig. 10)	**Blur** (Fig. 9)	**Indoor** (Fig. 4)	**Grass** (Fig. 6)	**Hair** (Fig. 16)
Resolution	640×480	640×480	640×480	640×480	2048×1536	1600×1200
Supersampling	8×8	13×13	8×8	4×4	11×11	13×13
Lights	12	6	2	30	2	4
Light shader length	188	74	160	1,789	75	75
Surface shader length	266	132	113	7,555	266	154
Total texture size	368MB	–	80MB	491MB	3.4MB	–
PRMan						
4 CPU cores	40s	133s	13s	197s	1038s	3988s
Gelato						
1 GPU	29.92s	246.32s	20.74s	–	–	–
RenderAnts						
1 GPU	2.43s	71.82s	2.47s	10.12s	48.94s	700.73s
2 GPUs	2.26s	37.32s	1.64s	9.47s	27.46s	360.24s
3 GPUs	2.11s	25.71s	1.37s	9.26s	22.85s	256.02s
Rendering rates	2.4 fps	–	1.0 fps	1.3 fps	–	–
Shader compilation	41.52s	4.26s	8.11s	147.80s	26.61s	17.86s
Micropolygons	1.0M	545.5M	29.7M	2.9M	30.1M	442.4M
Sample points	56.1M	328.1M	24.6M	48.9M	4.7G	24.0G

Table 1: *Measurements of scene complexity and rendering performance of PRMan 13.0.6, Gelato 2.2, and RenderAnts. For all renderers, the rendering time includes the file loading/parsing time, the shadow pass time, and the view pass time. For RenderAnts, we also report the rendering rates on three GPUs when the user is changing the viewpoint (i.e., the reciprocal of the view pass time), the shader compilation time, the number of shaded micropolygons, and the number of sample points. Note that shader compilation is executed only once for all shaders. Also note that Gelato crashed and reported insufficient memory for the grass and hair scenes.*

including characters, and outdoor and indoor scenes. Visual effects including transparency, shadows, light shafts, motion blur, and depth-of-field have been rendered. For all scenes, our system generates high-quality pictures visually comparable to those generated by PRMan, with slight differences due to different implementation details of individual algorithms (e.g., shadow maps). Note that RenderAnts could be implemented to produce pictures visually identical to those of PRMan if we strictly follow the implementation details of PRMan. Our current results, on the other hand, are already convincing and could be useful in many applications.

Rendering Performance As shown in Table 1, our system outperforms PRMan by over one order of magnitude for most scenes. In the ants scene (Fig. 10), the performance gain is only around five times. This is because our current implementation of depth-of-field needs to render scenes multiple times while PRMan only renders once as described in Section 5.5.

RenderAnts is capable of rendering several moderately complex scenes (Fig. 1, Fig. 4, Fig. 5 and Fig. 12) at interactive frame rates on three GPUs when the user is changing the viewpoint. In this case, the shadow pass time is saved – we do not need to re-render shadow maps if only the viewpoint and materials are changed. Also, since in practice only one light is modified at a time, only the shadow map of this light needs to be re-rendered and other shadow maps remain unchanged. This allows us to modify the viewpoint, lights and materials while viewing high-quality results on the fly.

We also compared RenderAnts with Gelato on three scenes, the Proog, the ants, and the motion blur scene provided in Gelato's tutorial. As shown in Table 1, RenderAnts is about 12 times faster than Gelato for the Proog scene. For the ants scene, RenderAnts outperforms Gelato by a factor of three. Note that Gelato is not a RenderMan compliant renderer and does not directly support RIB file format and RenderMan shaders. Although some tools have been developed by third parties to allow Gelato to read RIB scene files [Lancaster 2006], these tools have limited functions and do not work for our scenes. To perform a comparison, we had to load a scene into Maya and manually replace all surface shaders using Maya's materials.

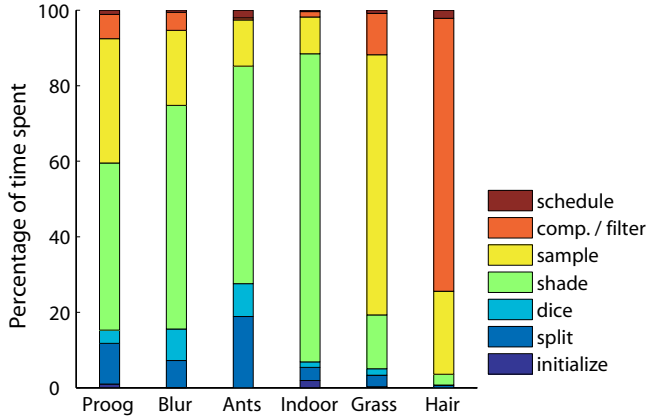


Figure 13: *Breakdown of the rendering time on a single GPU. The initialization time is the time for data loading (i.e., copying data from the CPU to the GPU).*

Performance Analysis Fig. 13 shows the percentage of each stage's running time in the rendering time. Just like in traditional CPU-based renderers, shading accounts for a large portion of the rendering time for most scenes in our system. The grass and hair scenes contain a lot of fine geometry, resulting in a huge number of sample points. Therefore, the sampling and compositing/filtering stages take a lot of time. The percentage for initialization is quite different for different scenes. For the indoor scene (Fig. 4), copying data from the CPU to the GPU consumes considerable time since it contains 600K polygon primitives. On the other hand, although the ants scene contains 100 ants, only one ant model needs to be copied to the GPU – others are all instances of this model. The initialization time for this scene is thus negligible.

Note that the scheduling time is insignificant for all scenes, which demonstrates the efficiency of our stage scheduling algorithm. In our experiments, the scheduling algorithm can improve the overall rendering performance by 30%-300% over the bucketing approach, depending on scene complexity and rendering resolution. For ex-

	Fig. 15 (top)	Fig. 15 (middle)	Fig. 15 (bottom)
Lights	12	19	22
Texture size	1.24GB	1.38GB	1.19GB
PRMan			
4 CPU cores	303s	440s	329s
RenderAnts			
1 GPU	17.24s	16.45s	21.53s
2 GPUs	11.32s	10.34s	12.60s
3 GPUs	8.91s	8.84s	9.92s
Micropolygons	15.4M	12.2M	29.1M
Sample points	1.5G	1.2G	2.8G
Out-of-core			
#fetches	3.15K	1.43K	2.53K
#threads	65.54M	12.15M	44.17M

Table 2: *Statistics of three shots from Elephants Dreams. To evaluate the overhead of out-of-core texture fetches, we count the total number of out-of-core texture fetches (#fetches) and the total number of interrupted shading threads (#threads).*

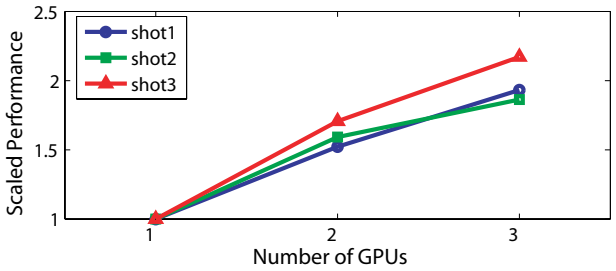


Figure 14: *Scalability study: rendering performance of three images shown in Fig. 15 on 1 to 3 GPUs, with each shot's results plotted related to the performance on one GPU.*

ample, for the Proog and the ants scenes rendered at 640×480 resolution, our algorithm improves the performance by 38% and 164%, respectively. If the two scenes are rendered at 1920×1440 resolution, the improvements increase to about 106% and 255%, respectively.

The overhead of out-of-core texture fetches consists of two parts – context saving at interrupts and texture copy from CPU to GPU. We evaluate this overhead on three complex scenes shown in Fig. 15, each of which has more than one gigabyte of textures. Table 2 gives the total number of interrupted threads and the total number of out-of-core texture fetches when rendering these scenes. For each interrupted thread, a maximum of 236 bytes need to be stored. Assuming the memory bandwidth is 100GB/s, context saving time should be 20-150ms. Each out-of-core texture fetch copies a 128×128 texture tile (64KB) from CPU to GPU. Assuming CPU-to-GPU copy has a 1.5GB/s bandwidth and 10μs per call overhead, the copy time should be 70-170ms. The total estimated overhead is thus less than 5% of the total rendering time.

The vast majority of our system is implemented as GPU code and runs on GPUs. GPU memory allocation, kernel launch configuration, and some operations (e.g., stack) in the schedulers are necessarily performed on the CPU with negligible costs. Since our system is to maximize parallelism within available GPU memory, it always consumes all available GPU memory to achieve high rendering performance.

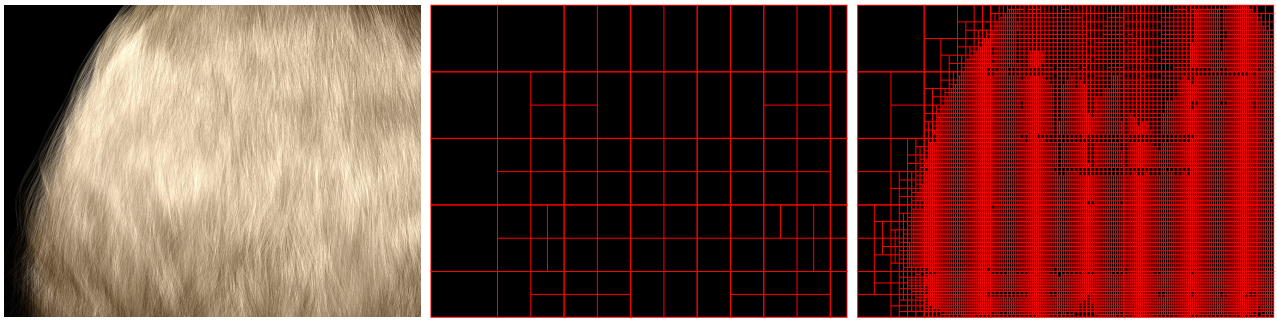
Performance Scalability The scalability of our system with respect to the number of GPUs depends on the scene. As shown in Table 1, the performance scales well for complex scenes such as

Figure 15: *Three frames of Elephants Dream, rendered with RenderAnts at 1920×1080 resolution with 13×13 supersampling.*

the ants, grass and hair scenes. For scenes like the indoor scene, the initial preparation required to set up the multi-GPU scheduler takes a considerable portion of the running time, leading to much less performance gain using multiple GPUs.

To better demonstrate the scalability of our system, we render three shots (see the images in Fig. 15) exported from an open source movie entitled Elephants Dream [Blender Foundation 2006], at 1920×1080 resolution with 13×13 supersampling. These scenes are reasonably complex – the scene shown in the middle row of Fig. 15 contains 407K primitives that are diced into 12.2M micropolygons, generating 1.2G sample points, which is comparable to some examples shown in previous papers such as [Pellacini et al. 2005] and [Ragan-Kelley et al. 2007]. Table 2 and Fig. 14 show the rendering time for the three shots, using 1 to 3 GPUs. For these complex scenes, the performance scales well with the GPU number, although the scaling is not perfectly linear.

Animation Rendering Since the animations of Elephants Dream were produced using Blender, we use RIB MOSAIC [WHiTeRaBBiT 2008] to export Blender files to RIB files. Fig. 15 shows three rendered pictures from the three shots. These shots contain 656



(a) rendering result (b) dicing regions (c) sampling regions

Figure 16: *215K transparent, long hairs ($\alpha = 0.3$) rendered at 1600×1200 with 13×13 supersampling. The bucket size is set as 256×256. RenderAnts shades 442.4M micropolygons in 256 seconds on three GPUs, which is about 15 times faster than PRman on a quad-core CPU. Due to the highly complex geometry, a huge number of sample points (24.0G) are created – many individual pixels contain substantial sample points, resulting in lots of small sampling regions as illustrated in (c). Note that our scheduling algorithm still significantly improves the rendering performance – the simple bucketing approach works with the bucket size 16×16 and takes 451 seconds to render the image.*

frames in total and were rendered in about one and a half hours on three GPUs using RenderAnts, including the rendering time and file input/output time. Note that our rendering results are different from the movie released by the Blender Foundation due to different rendering algorithms and file conversion problems.

Limitations Currently there are two major limitations in the RenderAnts system. The first is the geometry scalability. We assume grids generated during bound-split and their bounding boxes fit in GPU memory. This may be problematic for production scenes that contain a large number of primitives. For instance, increasing the number of hairs in Fig. 16 to 600K would break the assumption and make the system crash. Also, a huge number of sample points will be created for scenes that contain a lot of transparent and fine geometry. For example, 24.0G samples are generated in the hair scene. The sampling scheduler splits the image region into small sampling regions (see Fig. 16(c)). Increasing the number of hairs would result in more sample points and smaller sampling regions, greatly reducing the degree of parallelism and slowing down the rendering performance. In the extreme case, the system will crash if a single pixel contains substantial sample points that are too many to be stored in GPU memory. A complete solution to this limitation is a virtual memory system with paging and is left for future research. Another limitation is motion/focal blur. Our current accumulation buffer based algorithm is more of a proof-of-concept approach and is not intended for actual production. Efficient production-quality motion/focal blur on the GPU is a non-trivial task that should be investigated in future work.

8 Conclusion and Future Work

We have presented RenderAnts, a system that enables interactive REYES rendering on GPUs. We make three major contributions in designing the system: mapping all stages of the basic pipeline to the GPU, scheduling parallelism at individual stages to maximize performance, and supporting scalable rendering on multiple GPUs by minimizing inter-GPU communication and balancing workloads. As far as we know, RenderAnts is the first REYES renderer that entirely runs on GPUs. It can render photorealistic pictures of quality comparable to those generated by PRMan and is over one order of magnitude faster than PRMan. For moderately complex scenes, it allows the user to change the viewpoint, lights and materials while providing feedback interactively.

Based on the RenderAnts system, there exist a number of interesting directions for further investigation. First, some algorithms in our system can be improved. For example, the current dicing/sampling schedulers simply split a region at the middle point of the longer axis. We believe that a splitting scheme which balances the number of primitives/micropolygons contained in the two subregions will generate a better partitioning of the region and improve performance. We also wish to avoid patch cracks which are caused by various errors in the approximation of primitives by their tessellations. These cracks will introduce rendering artifacts.

Second, we are interested in incorporating more advanced features into the system, such as deep shadow maps, ambient occlusion, subsurface scattering, ray tracing and photon mapping. Some features can be added to RenderAnts by adapting existing algorithms. For example, GPU ray tracing of production scenes has already been demonstrated in [Budge et al. 2009]. For photon mapping, Hachisuka et al [2008] proposed a progressive algorithm that can achieve arbitrarily high quality within bounded memory, which should fit well in our pipeline. Culling is also an important feature in modern REYES implementations. It is possible to incorporate some traditional culling techniques into our system, like computing depth buffers prior to shading.

References

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

APODACA, T. 2000. How PhotoRealistic RenderMan works. *ACM SIGGRAPH 2000 Course Notes*.

BLEIWEISS, A., AND PREETHAM, A. 2003. Ashli - advanced shading language interface. *ACM SIGGRAPH Course Notes*.

BLENDER FOUNDATION, 2006. Elephants Dream home page. http://orange.blender.org.

BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. 1995. Cilk: an efficient multithreaded runtime system. *ACM SIGPLAN Notices 30*, 8, 207–216.

BUDGE, B. C., BERNARDIN, T., SENGUPTA, S., JOY, K. I., AND OWENS, J. D. 2009. Out-of-core data management for path tracing on hybrid resources. In *Proceedings of Eurographics 2009*.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The Reyes image rendering architecture. In *SIGGRAPH'87*, 95–102.

COOK, R. L. 1984. Shade trees. In *SIGGRAPH'84*, 223–231.

COOK, R. L. 1986. Stochastic sampling in computer graphics. *ACM Trans. Gr. 5*, 1, 51–72.

CYRIL, C., FABRICE, N., SYLVAIN, L., AND ELMAR, E. 2009. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *I3D'09*.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4, 451–490.

HAEBERLI, P., AND AKELEY, K. 1990. The accumulation buffer: hardware support for high-quality rendering. In *SIGGRAPH'90*, 309–318.

HARRIS, M., OWENS, J., SENGUPTA, S., ZHANG, Y., AND DAVIDSON, A., 2007. CUDPP homepage. http://www.gpgpu.org/developer/cudpp/.

HOU, Q., ZHOU, K., AND GUO, B. 2008. BSGP: Bulk-Synchronous GPU Programming. *ACM Trans. Gr. 27*, 3, 9.

HOU, Q., ZHOU, K., AND GUO, B. 2009. Debugging GPU stream programs through automatic dataflow recording and visualization. Tech. rep., May, 2009.

LANCASTER, T., 2006. RenderMan/Gelato utilities. http://www.renderman.org/RMR/Utils/gelato/index.html.

LAZZARINO, O., SANNA, A., ZUNINO, C., AND LAMBERTI, F. 2002. A PVM-based parallel implementation of the Reyes image rendering architecture. *Lecture Notes In Computer Science 2474*, 165–173.

NVIDIA, 2008. Gelato home page. http://www.nvidia.com/page/gz_home.html.

OLANO, M., AND LASTRA, A. 1998. A shading language on graphics hardware: the pixelflow shading system. In *SIGGRAPH'98*, 159–168.

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.

PATNEY, A., AND OWENS, J. D. 2008. Real-time Reyes-style adaptive surface subdivision. *ACM Trans. Gr. 27*, 5, 143.

PATNEY, A. 2008. Real-time Reyes: Programmable pipelines and research challenges. *ACM SIGGRAPH Asia 2008 Course Notes*.

PEACHEY, D. 1990. Texture on demand. Tech. rep., Pixar Technical Memo #217.

PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. 2000. Interactive multi-pass programmable shading. In *SIGGRAPH 2000*, 425–432.

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 Trans. Gr. 24*, 3, 464–470.

PIXAR. 2007. *PRMan User's Manual*.

RAGAN-KELLEY, J., KILPATRICK, C., SMITH, B. W., EPPS, D., GREEN, P., HERY, C., AND DURAND, F. 2007. The lightspeed automatic interactive lighting preview system. *ACM Trans. Gr. 26*, 3, 25.

REEVES, W. T., SALESIN, D. H., AND COOK, R. L. 1987. Rendering antialiased shadows with depth maps. In *SIGGRAPH'87*, 283–291.

TOSHIYA, H., SHINJI, O., AND JENSE, H. W. 2008. Progressive photon mapping. *ACM Trans. Gr. 27*, 5, 127.

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

WHITERABBIT, 2008. RIB MOSAIC home page. http://ribmosaic.wiki.sourceforge.net/.