Graphics Hardware (2002), pp. 1–10
Thomas Ertl, Wolfgang Heidrich, and Michael Doggett (Editors)

The Ray Engine

Nathan A. Carr Jesse D. Hall John C. Hart

University of Illinois

Abstract

Assisted by recent advances in programmable graphics hardware, fast rasterization-based techniques have made significant progress in photorealistic rendering, but still only render a subset of the effects possible with ray tracing. We are closing this gap with the implementation of ray-triangle intersection as a pixel shader on existing hardware. This GPU ray-intersection implementation reconfigures the geometry engine into a ray engine that efficiently intersects caches of rays for a wide variety of host-based rendering tasks, including ray tracing, path tracing, form factor computation, photon mapping, subsurface scattering and general visibility processing.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

Keywords: *Hardware acceleration, ray caching, ray classification, ray coherence, ray tracing, pixel shaders.*

1. Introduction

Hardware-accelerated rasterization has made great strides in simulating global illumination effects, such as shadows[35, 25, 7], reflection[3], multiple-bounce reflection[5], refraction[9], caustics[29] and even radiosity[13]. Nonetheless some global illumination effects have eluded rasterization solutions, and may continue to do so indefinitely. The environment map provides polygon rasterization with limited global illumination capabilities by approximating the irradiance of all points on an object surface with the irradiance at a single point[3]. This single-point irradiance approximation can result in some visually obvious errors, such as the boat in wavy water shown in Figure 1.

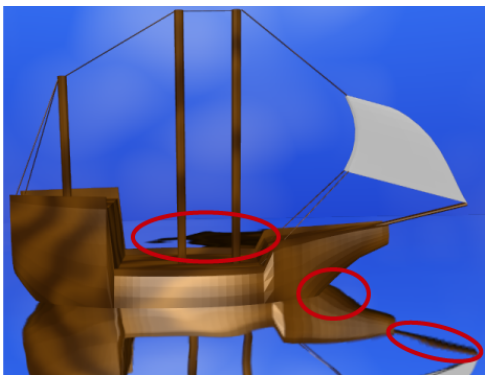

Figure 1: *What is wrong with this environment-mapped picture? (1) The boat does not meet its reflection, (2) the boat is reflected in the water behind it, and (3) some aliasing can be seen in the reflection.*

Ray tracing of course simulates all of these effects and more. It can provide true reflection and refraction, complete with local and multiple bounces. Complex camera models with compound lenses are easier to simulate using ray tracing[15]. Numerous global illumination methods are based on ray tracing including path tracing[12], Monte-Carlo ray tracing[33] and photon mapping[10].

Ray tracing is classically one of the most time consuming operations on the CPU, and the graphics community has been eager to accelerate it using whatever methods possible. Hardware-based accelerations have included CPU-specific tuning, distribution across parallel processors and even construction of special purpose hardware, as reviewed in Section 2.

Graphics cards have recently included support for programmable shading in an effort to increase the realism of their rasterization-based renderers[16]. This added flexibility is transforming the already fast graphics processing unit (GPU) into a supercomputing coprocessor, and its power is being

applied to a wider variety of applications than its developers originally intended.

One such application is ray tracing. Section 3 shows how to configure the graphics processing unit (GPU) to compute ray-triangle intersections, and Section 4 details an implementation. This GPU ray-triangle intersection reconfigures the graphics accelerator into a *ray engine*, described in Section 5, that hides the details of its back-end GPU ray-triangle intersection, allowing the ray engine to be more easily integrated into existing rendering software systems.

The ray engine can make existing rasterization-based renderers look better. A rasterization renderer augmented with the ray engine could trace the rays necessary to achieve effects currently impossible with rasterization-only rendering, including local reflections (Figure 1), true refractions and sub-surface scattering[11].

The ray engine is also designed to be efficiently integrated into existing ray-tracing applications. The ray engine performs best when intersecting caches of coherent rays[21] from host-based rendering tasks. This is a form of load balancing that allows the GPU to do what it does best (perform the same computation on arrays of data), and lets the CPU do what the GPU does worst (reorganize the data into efficient structures whose processing requires flow control). A simple ray tracing system we built using the ray engine is already running at speeds comparable to the fastest known ray tracer, which was carefully tuned to a specific CPU[32]. The ray engine could likewise accelerate Monte Carlo ray tracing, photon mapping, form factor computation and visibility preprocessing.

2. Previous Work

Although classic ray tracing systems support a wide variety of geometric primitives, some recent ray tracers designed to achieve interactive rates (including ours) have limited themselves to triangles. This has not been a severe limitation as geometric models can be tessellated, and the simplicity of the ray-triangle intersection has led to highly efficient implementations[2, 18].

Hardware z-buffered rasterization can quickly determine the visibility of triangles. One early hardware optimization for ray tracing was the first-hit speedup, which replaced eye-ray intersections with a z-buffered rasterization of the scene using object ID as the color[34]. Eye rays are a special case of a coherent bundle of rays. Such rays can likewise be efficiently intersected through z-buffered rasterization for hardware accelerated global illumination[28], of which ray tracing is a subset.

One obvious hardware acceleration of ray tracing is to optimize its implementation for a specific CPU. The current fastest CPU implementation we are aware of is a coherent ray tracer tuned for the Intel Pentium III processor[32]. This ray tracer capitalized on a variety of spatial, ray and memory coherencies to best utilize CPU optimizations such as caching, branch prediction, instruction reordering, speculative execution and SSE instructions. Their implementation ran at an average of 30 million intersections per second on an 800 Mhz Pentium III. They were able to trace between 200K and 1.5M rays per second, which was over ten times faster than POV-Ray and Rayshade.

There have been a large number of implementations of ray tracers on MIMD computers[26]. These implementations focus on issues of load balancing and memory utilization. One recent implementation on 60 processors of an SGI Origin 2000 was able to render at 512^2 resolution scenes of from 20 to 2K patches at rates ranging from two to 20 Hz[19].

Special purpose hardware has also been designed for ray tracing. The AR350 is a production graphics accelerator designed for the off-line (non-real-time) rendering of scenes with sophisticated Renderman shaders[8]. A ray tracing system designed around multiprocessors with smart memory is also in progress[23].

Our ray engine is similar in spirit to another GPU-based ray tracing implementation that simulates a state machine[24]. This state-based approach breaks ray tracing down into several states, including grid traversal, ray-triangle intersection and shading. This approach performs the entire ray tracing algorithm on the GPU, avoiding the slow readback process required for GPU-CPU communication that our approach must deal with. The state-based method however is not particularly efficient on present and near-future GPU's due to the lack of control flow in the fragment program, resulting in a large portion of pixels (from 90% to 99%) remaining idle if they are in a different state than the one currently being executed. Our approach has been designed to organize ray tracing to achieve full utilization of the GPU.

3. Ray Tracing with the GPU

3.1. Ray Casting

The core of any ray tracer is the intersection of rays with geometry. Rays are represented parametrically as $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ where $\mathbf{o}$ is the ray origin, $\mathbf{d}$ is the ray direction and $t \geq 0$ is a real parameter corresponding to points along the ray. The classic implementation of recursive ray tracing casts each ray individually and intersects it against the scene geometry. This process generates a list of parameters t_i corresponding to points of intersection with the scene's geometric primitives. The least positive element of this list is returned as the first intersection, the one closest to the ray origin.

Figure 2(a) illustrates ray casting as a crossbar. This illustration represents the rays with horizontal lines and the (unorganized) geometric primitives (e.g. triangles) with vertical lines. The crossing points of the horizontal and vertical lines represent intersection tests between rays and triangles.

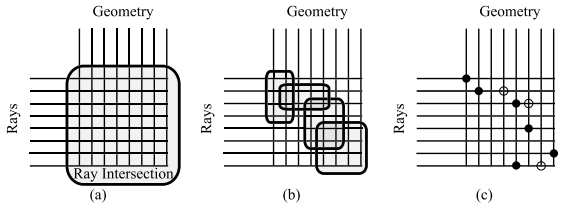

Figure 2: *Ray intersection is a crossbar.*

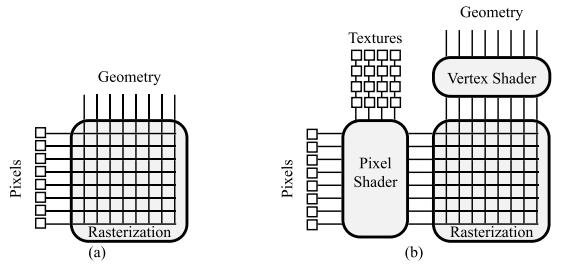

Figure 3: *Programmable pixel shading is a crossbar.*

This crossbar represents an all-pairs check of every ray against every triangle. Since their inception, ray tracers have avoided checking every triangle against every primitive through the use of spatial coherent data structures on both the rays and the geometry. These data structures reorganize the crossbar into a sparse overlapping block structure, as shown in Figure 2(b). Nevertheless the individual blocks are themselves full crossbars that perform an all pairs comparison on their subset of the rays and geometry.

The result of ray casting is the identification of the geometry (if any) intersected first by each ray. This result is a series of points in the crossbar, no greater than one per horizontal line (ray). These first intersections are shown as black disks in Figure 2(c). The other ray-triangle intersections are indicated with open circles and are ignored in simple ray casting.

3.2. Programmable Shading Hardware

Graphics accelerators have been designed to implement a pipeline that converts polygons vertices from model coordinates to viewport coordinates. Once in viewport coordinates, rasterization fills the polygon with pixels, interpolating the depth, color and texture coordinates in a perspective-correct fashion. During rasterization, interpolated texture coordinates index into texture memory to map an image texture onto the polygon.

This rasterization process can also be viewed as a crossbar, as shown in Figure 3(a). The vertical lines represent individual polygons passing through the graphics pipeline whereas the horizontal lines represent the screen pixels.

Consider the case where each polygon, a quadrilateral, exactly covers all of the screen pixels. Then rasterization of these polygons performs an all-pairs combination of every pixel with every polygon.

While even early graphics accelerators were programmable through firmware[4], modern graphics accelerators contain user-programmable elements designed specifically for advanced shading[16]. These programmable elements can be separated into two components, the vertex shader and the pixel shader, as shown in Figure 3(b). The vertex shader is a user-programmable stream processor that can alter the attributes (but not the number) of vertices sent to the rasterizer. The pixel shader can perform arithmetic operations on multiple texture coordinates and fetched texture samples, but does so in isolation and cannot access data stored at any other pixel. Pixel shaders run about an order of magnitude faster than vertex shaders.

3.3. Mapping Ray Casting to Programmable Shading Hardware

We map the ray casting crossbar in Figure 2 to the rasterization crossbar in Figure 3 by distributing the rays across the pixels and broadcasting a stream of triangles to each pixel by sending their coordinates down the geometry pipeline as the vertex attribute data (e.g. color, texture coordinates) of screen filling quadrilaterals.

The rays are stored in two screen-resolution textures. The color of each pixel of the ray-origins texture stores the coordinates of the origin of the ray. The color of each pixel of the ray-directions texture stores the coordinates of the ray direction vector.

An identical copy of the triangle data is stored at each vertex of a screen-filling quadrilateral. Rasterization of this quadrilateral interpolates these attributes at each pixel of its screen projection. Since the attributes are identical at all four vertices, interpolation simply distributes a copy of the triangle data to each pixel.

A pixel shader performs the ray-triangle intersection computation by merging the ray data stored per-pixel in the texture maps with the triangle data distributed per-pixel by the interpolation of the attribute data stored at the vertices of the quadrilateral. The specifics of this implementation will be described further in Section 4.

3.4. Discussion

The decision to store rays in texture and triangles as vertex attributes was based initially on precision. Since rays can be specified with five real values whereas triangles require nine we found it easier and more accurate to store the ray values at the lower texture precisions.

We also chose to implement ray-triangle intersection as a pixel shader instead of a vertex shader. Vertex shaders do

not have direct access to the rasterization crossbar, and hence needed to store ray data as constants in the vertex shader's local memory. The vertex shader is also slower, and was able to compute 4.1M ray-triangle intersections per second, which is much less than what the CPU is currently capable of performing.

Viewing the GPU as a SIMD processor[20] allowed us to compare other SIMD ray tracing implementations. SIMD ray tracers typically distribute rays to the processors and broadcast the geometry, or distribute geometry and broadcast the rays. The AR350 ray tracing hardware utilized a fine-grain ray distribution to isolated processors[8], which improved load balancing, but inhibited the possible advantages of ray coherence. The coherent ray tracer[32] also distributed rays at its lowest level, intersecting each triangle with four coherent rays using SSE whereas an axis-aligned BSP-tree coherently organized the triangles (but required special implemention to efficiently intersect four-ray bundles). Geometry distribution on the other hand seems better suited for handling the special problems due to ray tracing large scene databases[31].

4. Ray-Triangle Intersection on the GPU

The pixel shader implementation of ray-triangle intersection treats the GPU as a SIMD parallel processor[20]. In this model, the framebuffer is treated as an accumulator data array of 5-vectors (r,g,b,α,z), and texture maps are used as data arrays for input and variables. Pixel shaders perform sequences of operations that combine the textures and the framebuffer. While compilers exist for multipass programming[20, 22], the current limitations of pixel shaders required complete knowledge and control of the available instructions and registers to implement ray intersection.

4.1. Input

Ray Data. As mentioned in Section 3.3, the GPU component of the ray engine intersects multiple rays with a single triangle. Every pixel in the data array corresponds to an individual ray. Our implementation stores ray data in two textures: a ray-origins texture and a ray-directions texture. Batches of rays cast from the eyepoint or a point light source will have a constant color ray-origins texture and their texture map could be stored as a single pixel or a pixel shader constant.

Triangle Data. The triangle data is encapsulated in the attributes of the four vertices of a screen filling quad. Let $\mathbf{a},\mathbf{b},\mathbf{c}$ denote the three vertices of the triangle, and $\mathbf{n}$ denote the triangles front facing normal. The triangle id was stored as the quad's color, and the vectors $\mathbf{a},\mathbf{b},\mathbf{n},\mathbf{ab}(=\mathbf{b}-\mathbf{a}),\mathbf{ac},\mathbf{bc}$ were mapped to multi-texture coordinate vectors. The redundant vector information includes ray-independent precomputation that reduces the size and workload of the pixel shader. Our implementation passes only the three vertices of the triangle from the host, and computes the additional redundant values in the vertex shader.

The texture coordinates for texture zero (s_0,t_0) are special and are not constant across the quadrilateral. They are instead set to $(0,0),(1,0),(1,1),(0,1)$ at the four vertices, and rasterization interpolates these values linearly across the quad's pixels. These texture coordinates are required by the pixel shader to access each pixel's corresponding ray in the screen-sized ray-origins and ray-directions textures.

4.2. Output

The output of the ray-triangle intersection needs to be queried by the CPU, which can be an expensive operation due to the asymmetric AGP bus on personal computers (which sends data to the graphics card much faster than it can receive it). The following output format is designed to return as little data as necessary, limiting itself to the index of the triangle that intersects the ray closest to its origin, using the z-buffer to manage the ray parameter t of the intersection.

Color. The color channel contains the color of the first triangle the ray intersects (if any). For typical ray tracing applications, this color will be a unique triangle id. These triangle id's can index into an appearance model for the subsequent shading of the ray-intersection results.

Alpha. Our pixel shader intersection routine conditionally sets the fragments alpha value to indicate ray intersection. The alpha channel can then be used as a mask by other applications if the rays are coherent (e.g. like eye rays through the pixels in the frame buffer).

The t-Buffer. The t-value of each intersection is computed and replaces the pixel's z-value. The built-in z-test is used so the new t-value will overwrite the existing t-value stored in the z-buffer if the new value is smaller. This allows the z-buffer to maintain the least positive solution t for each ray. Since the returned t value is always non-negative, the t-value maintained by the z-buffer always corresponds to the first triangle the ray intersects.

4.3. Intersection

We examined a number of efficient ray-triangle intersection tests[6, 2, 18], and managed to reorganize one[18] to fit in a pixel shader.

Our revised ray-triangle intersection is evaluated as

$$\mathbf{ao} = \mathbf{o}-\mathbf{a}, \quad (1) \qquad \mathbf{bod} = \mathbf{bo}\times\mathbf{d}, \quad (5)$$

$$\mathbf{bo} = \mathbf{o}-\mathbf{b}, \quad (2) \qquad u = \mathbf{ac}\cdot\mathbf{aod}, \quad (6)$$

$$t = -\frac{\mathbf{n}\cdot\mathbf{ao}}{\mathbf{n}\cdot\mathbf{d}}, \quad (3) \qquad v = -\mathbf{ab}\cdot\mathbf{aod}, \quad (7)$$

$$\mathbf{aod} = \mathbf{ao}\times\mathbf{d}, \quad (4) \qquad w = \mathbf{bc}\cdot\mathbf{bod}. \quad (8)$$

The intersection passes only if all three (unnormalized) barycentric coordinates u,v and w are non-negative. If the ray does not intersect the triangles, the alpha channel for that

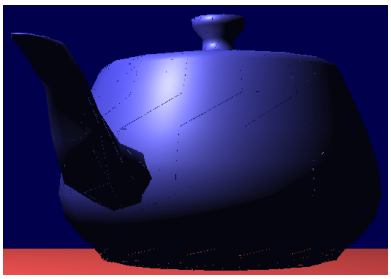

Figure 4: *Leaky teapot, due to the low precision implementation on PS1.4 pixel shaders used to test the performance of ray-triangle intersection. Our simulations using the precision available on upcoming hardware are indistinguishable from software renderings.*

pixel is set to zero and the pixel is killed. The parameter t is also tested against the current value in the z-buffer, and if it fails the pixel is also killed. Surviving pixels are written to the framebuffer as the ray intersection currently closest to the ray origin.

This implementation reduces cross products, which require multiple pixel shader operations to compute. The quotient (3) was implemented using the *texdepth* instruction, which implements the "depth replace" texture shader.

4.4. Results

We tested the PS1.4 implementation of the ray-triangle intersection using the ATI R200 chipset on the Radeon 8500 graphics card. The limited numerical precision of its pixel shader (16-bit fixed point, with a range of ± 8) led to some image artifacts shown in Figure 4, this implementation did suffice to measure the speed of an actual hardware pixel shader on the task of ray intersection.

We clocked our GPU implementation of ray intersection at 114M intersections per second. The fastest CPU-based ray tracer was able to compute between 20M and 40M intersections per second on an 800Mhz Pentium III[32]. Even doubling the CPU numbers to estimate performance on today's computers, our GPU ray-triangle intersection performance already exceeds that of the CPU, and we expect the gap to widen as GPU performance growth continues to outpace CPU performance growth.

5. Ray Engine Organization

This section outlines the encapsulation of the GPU ray-intersection into a ray engine. It begins with a discussion of why the CPU is a better choice for the management of rays during the rendering process. Since the CPU is managing the rays, the ray engine is packaged to provide easy access to the GPU ray-intersection acceleration through a front-end interface. This interface accepts rays in coherent bundles, which can be efficiently traced by the GPU ray-intersection implementation.

5.1. The Role of the CPU

We structured the ray engine to perform ray intersection on the GPU and let the host organize the casting of rays and manage the resulting radiance samples. Since the bulk of the computational resources used by a ray tracer are spent on ray intersection, the management of rays and their results is a relatively small overhead for the CPU, certainly smaller than performing the entire ray tracing on the CPU.

The pixel shader on the GPU is a streaming SIMD processor good at running the same algorithm on all elements of a data array. The CPU is a fast scalar processor that is better at organizing and querying more sophisticated data structures, and is capable of more sophisticated algorithmic tools such as recursion. Others have implemented the entire ray tracer on the GPU[24], but such implementations can be cumbersome and inefficient.

For example, recursive ray tracing uses a stack. While some have proposed the addition of state stacks in programmable shader hardware[17], such hardware is not currently available. Recursive ray tracing can be implemented completely on the GPU[24], but apparently at the expense of generating two frame buffers full of reflection and refraction rays at each intersection, which are then managed by the host.

The need for a stack can be avoided by path tracing[12]. Paths originating from the eyepoint passing through a pixel can accumulate its intermediate results at the same location in texture maps. Path tracing requires importance sampling to be efficient, even with fast ray intersection. Sophisticated importance sampling methods[30] use global queries into the scene database, as well as queries into previous radiance results in the scene. Such queries are still performed more efficiently on the CPU than on the GPU.

Some ray tracers also organize rays and geometry into coherent caches that are cast in an arbitrary order to more efficiently render large scenes[21]. The management of ray caches and the radiances resulting

rom their batched tracing requires a lot of data shuffling. An implementation on the GPU would require all of the pixels in the image returned by the batch ray intersection algorithm to be shuffled to contribute to the radiance of the previously cast rays. While dependent texturing can be used to perform this shuffling[24], the GPU is ill-designed to organize and set up this mapping.

We used the NV_FENCE extension to overlap the computation of the CPU and GPU. This allows the CPU to test whether the GPU is busy working on a ray-triangle bundle so the CPU can continue to work simultaneously on ray caching.

5.2. The Ray Engine Interface

Organizing high-performance rendering services to be transparent makes them easier to integrate into existing rendering systems[14]. We structured the ray engine as both a front end driver that runs on the host and interfaces with the application, and a back end component that runs on the GPU to perform ray intersections.

The front end of the ray engine accepts a cache of rays from a host application. This front end converts the ray cache into the texture map data for the pixel shader to use for intersection. The front end then sends the geometry (from a shared database with the application) down the geometry pipeline to the pixel shader. The pixel shader is treated as a back end of this system that intersects the rays with the triangles passed to it. The front end grabs the results of ray intersection (triangle id, t-value and, if supported, the barycentric coordinates) and returns them to the application in a more appropriate format.

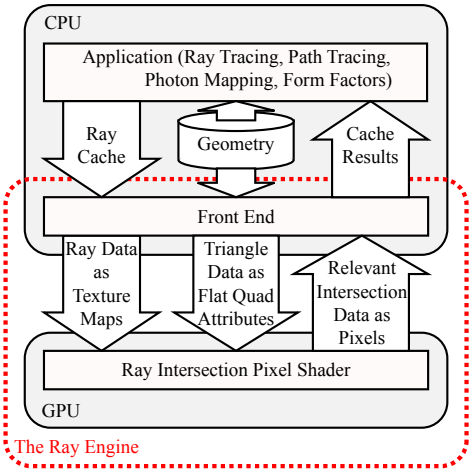

Figure 5: *The organization of the ray engine.*

The main drawback of implementing ray casting applications on the host is the slow readback bandwidth of the AGP bus when transferring data from the GPU back to the CPU. This bottleneck is addressed by the ray engine system with compact data that is returned infrequenty (once after all triangles have been sent to the GPU).

5.3. The Ray Cache

Accelerating the implementation of ray intersection is not enough to make ray tracing efficient. The number of ray intersections needs to be reduced as well. The ray engine uses an octree to maintain geometry coherence and a 5-D ray tree[1] to maintain ray coherence.

The ray engine works more efficiently when groups of similar rays intersect a collection of spatially coeherent triangles. In order to maintain full buckets of coherent rays, we utilize a ray cache[21]. Ray caching allows some rays to be cast in arbitrary order such that intersection computations can be performed as batch processes.

As rays are generated, they are added to the cache, which collects them into buckets of rays with coherent origins and directions. For maximum performance on the ray engine, each bucket should contain some optimal hardware-dependent number of rays. Our bucket size was 256 rays, organized as two 64×4 ray-origin and ray-direction textures. Textures on graphics cards are commonly stored in small blocks instead of in scanline order to better capitalize on spatial coherence by placing more relevant texture samples into the texture cache of the GPU. The size of these texture blocks is GPU-dependent and can be found through experimentation.

If adding a ray makes a bucket larger than the optimal bucket size then the node is split into four subnodes along the axis of greatest variance centered at the using the mean values of the ray origins and directions. We also add rays to the cache in random order which helps keep the tree balanced.

When the ray tracer needs a result or the entire ray cache becomes full, a bucket is sent to the ray engine to be intersected with geometry. We send the fullest buckets first to maximize utilization of the ray engine resources. Each node of the tree contains the total number of rays contained in the buckets below it. Our search traverses down the largest valued nodes until a bucket is reached. While this simple greedy search is not guaranteed to find the largest bucket, it is fast and works well in practice since the buckets share the same maximum size. This greedy search also tends to balance the tree.

Once the search has chosen a bucket, rays are stolen from that node's siblings to fill the bucket to avoid wasting intersection computations. Due to the greedy search and the node merging described next, this ensures that buckets sent to the ray engine are always as full as possible, even though in the ray tree they are typically only 50-80% full.

Once a bucket has been removed from the tree and traced, it can often leave neighboring buckets containing only a few rays. Our algorithm walks back up the tree from the removed bucket leaf node, collecting subtrees into a single bucket leaf node if the number of rays in the subtree has fallen below a threshold. Our tests showed that this process typically merged only a single level of the tree.

The CPU performs a ray bucket intersection test[1] against the octree cells to determine which should be sent to the GPU. We also used the vertex shader to cull back-facing triangles as well as triangles outside the ray bucket from intersection consideration. The vertex shader cannot change the number of vertices passing through it, but it can transform

the screen-filling quad containing the triangle data to an off-screen location which causes it to be clipped.

5.4. Results

We implemented the ray engine on a simulator for an upcoming GPU based on the expected precision and capabilities needed to support the Direct3D 9 specification. These capabilities allow us to produce full precision images that lack the artifacts shown earlier in Figure 4.

We used the ray engine to classically ray trace a teapot room and an office scene, shown in Figure 6(a) and (c). We applied the ray engine to a Monte-Carlo ray tracer that implemented distributed ray tracing and photon mapping, which resulted in Figure 6(b). The ray engine was also used to ray trace two views of one floor from the Soda Hall dataset, shown in Figures 6(d) and (e).

The performance is shown in Figure 1. Since our implementation is on a non-realtime simulator, we have estimated our performance using the execution rates measured on the GeForce 4. We measured the performance in rays per second, which measures the number of rays intersected with the entire scene per second. This figure includes the expensive traversal of the ray-tree and triangle octree as well as the ray-triangle intersections.

Scene	Polygons	Rays/sec.
Teapot Room Classical	2,650	206,905
Teapot Room Monte-Carlo	2,650	149,233
Office	34,000	114,499
Soda Hall Top View	11,052	129,485
Soda Hall Side View	11,052	131,302

Table 1: *Rays per second across a variety of scenes and applications.*

This perfomance meets the low end performance of the coherent ray tracer, which was able to trace from 200K to 1.5M rays per second[32]. It too used coherent data structures to increase performance, in this case an axis aligned BSP tree organized specifically to be efficiently traversed by the CPU. Our ray traversal implementation is likely not as carefully optimized as theirs.

6. Analysis and Tuning

Suppose we are given a set of R rays and a set of T triangles for performing ray-triangle intersection tests. We denote the time to run the tests on the GPU and CPU respectively as $\mathrm{GPU}(R,T)$ and $\mathrm{CPU}(R,T)$. To acheive improved performance, we are only interested in values of R and T for which $\mathrm{GPU}(R,T) \leq \mathrm{CPU}(R,T)$, suggesting the right problem granularity for which the GPU performs best.

Since the GPU performs all pairs intersection test between the rays and triangles passed to it, its performance is independent of scene structure

$$\mathrm{GPU}(R,T) = O(RT). \tag{9}$$

The running time for $\mathrm{CPU}(R,T)$ is dependent on both scene and camera (sampling) structure since partitioning structures in both triangle and ray space may be used to reduce computation

$$\mathrm{CPU}(R,T) \leq O(RT). \tag{10}$$

As Section 4.4 shows, the constant of proportionality in the $O(RT)$ in (9) is smaller (by at least a factor of two) than the one in (10). Tuning the ray engine will require balancing the raw speed of $\mathrm{GPU}(R,T)$ with the efficiency of $\mathrm{CPU}(R,T)$.

6.1. The Readback Bottleneck

We can model $\mathrm{GPU}(R,T)$ by analyzing the steps in the GPU ray-triangle intersection in terms of GPU operations, and empirically measuring the speed of these operatrions. A simple version of this model sufficient for our analysis is

$$\mathrm{GPU}(R,T) = TR\,\mathrm{fill}^{-1} + R\gamma\,\mathrm{readback}^{-1}, \tag{11}$$

where γ is the number of bytes read back from the graphics card per ray. This model shows that the GPU ray-triangle intersection time is linearly dependent on the number of rays and affinely dependent on the number of triangles. This model does not include the triangle rate, which would add a negligible term proportianal to T to the model. Once we determine values for fill and readback we can then determine the smallest number of triangles T_{min} needed to make GPU ray-triangle intersection practical.

The fill rate is measured in pixels per second (which includes the cost of the fragment shader execution) whereas the readback rate is measured in bytes per second. The fill rate is measured pixels per second instead of bytes per second because it is non-linear in the number of bytes transferred (modern graphics cards can for example multitexture two textures simultaneously). Since our ray engine uses two ray textures (an origins texture and a directions texture) we simply divide the number of rays (pixels) by the fill rate (pixels per second) to get the fragment shader execution time.

We determine values for the fill and readback rates empirically. For example, the GeForce3 achieves a fill rate of 390 MP/sec. (dual-textured pixels) and an AGP 4x readback rate of 250 MB/sec (which is only one quarter of the 1 GB/sec that should be available on the AGP bus). Returning a single 64-bit triangle ID uses a γ of four, whereas returning an additional three single-precision floating-point barycentric coordinates sets γ to 16. Hence we can return triangle ID's at a rate of 62.5M/sec., but when we include barycentrics the rate drops to 15.6M/sec. We can further increase performance by reducing the number of bytes used for the index of each triangle, especially since the ray engine sends smaller buckets of coherent triangles to the GPU.

(a) (b) (c) (c) (c)

Figure 6: *Images tested by the ray engine: teapot Cornell box ray traced classically (a) and Monte Carlo (b), office (c), and Soda Hall side (d) and top (e) views.*

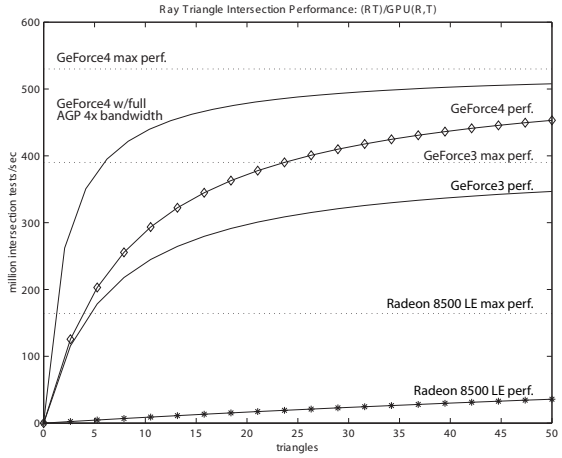

Figure 7: *Theoretical performance in millions of ray-triangle intersection tests per second on the GPU with* $\gamma = 4$.

For small values of T the performance is limited by the readback rate. As T increases, the constant cost of readback is amortized over a larger number of intersections tests. (When we measured peak ray-triangle intersection rates on the Radeon 8500, we sent thousands of triangles to the GPU.) In each case, the curve asymptotically approaches the fill rate, which is listed as the maximum performance possible. Realistically, only smaller values of T should be considered since the GPU intersection routine is an inefficient all-pairs $O(RT)$ solution and our goal is to only send coherent rays and triangles to it.

Figure 7 shows that even for small value of T, the performance is quite competitive with that of a CPU based implementation in spite of the read back rate limitation. For example, the ray-triangle intersections per second for ten triangles clock at 240M on the GeForce3 and 286M on the GeForce4 Ti4600 (if they had the necessary fragment processing capabilities). The recent availability of AGP 8x, and the upcoming AGP 3.0 standard will further reduce the impact of the readback bottleneck and further validate this form of general GPU processing.

6.2. Avoiding Forced Coherence

The previous section constructed a model for the efficiency of the GPU ray-triangle intersection. We must now determine when it is more efficient to use the CPU instead of the GPU.

It is important to exploit triangle and ray coherence only where it exists, and not to force it where it does not. We hence identify the locations in the triangle octree where the ray-triangle coherence is high enough to support efficient GPU intersection. This preprocess occurs after the triangle octree construction, and involves an additional traversal of the octree, identifying cells that represent at least $T_{\min}$ triangles. Since these cells are ideal for GPU processing we refer to these cells as GPU cells.

Rendering employs a standard recursive octree traversal routine. When a ray ray traverses through a cell not tagged as a GPU cell, the standard CPU based ray-triangle intersection is performed. If a ray encounters a GPU cell during its traversal, the ray's traversal is terminated and it is placed in the ray cache for that cell for future processing. When the ray cache corresponding to a given GPU cell reaches $R_{\min}$ rays, its rays and triangles are sent to the GPU for processing using the ray-intersection kernel.

A point may be reached where the ray engine has receive all known rays from the application to be processed. At this poing there may exist GPU cells whose ray cache is non-empty, but containing less than $R_{\min}$ rays. A policy may be chose to select a GPU cell and force its ray cache to be send to the CPU instead of the GPU. This allows the ray engine to continually advance towards completion for rendering the scene.

6.3. Results

We have performed numerous tests to tune the parameters of the geometry engine to eek out the highest performance.

Table 2 demonstrates the utilization of the GPU. As mentioned earlier, only reasonably sized collections of coherent

Scene	% GPU Rays
Teapot Room Classical	89%
Teapot Room Monte-Carlo	71%
Office	65%
Soda Hall Top View	70%
Soda Hall Side View	89%

Table 2: *Percentage of rays sent to the GPU across a variety of scenes and applications.*

rays and triangles are sent to the GPU. The remaining rays and triangles are traced by the CPU. The best performers resulted from classical ray tracing of the teapot room and the ray casting of the Soda Hall side view. The numerous bounces from Monte Carlo ray tracing likely reduce the coherence on all but the eye rays. Coherence was reduced in the office scene due to the numerous small triangles that filled the triangle cache before the ray cache could be optimally filled. The Soda Hall top view contains a lot of disjoint small "silhouette" wall polygons that likely failed to fill the triangle cache for a given optimally filled ray cache.

System	Rays/sec.	Speedup
CPU only	135,812	
plus GPU	165,098	22%
Asynch. Readback	183,273	34%
Infinitely Fast GPU	234,102	73%

Table 3: *Speedup by using the GPU to render the teapot room.*

Table 3 illustrates the efficiency of the ray engine. The readback delay was only responsible for 12% of the potential speedup of 34%. One feature that would allow us to recover that 12% is to be able to issue an asynchronous readback (as is suggested in OpenGL 2.0), such that the CPU and GPU can continue to work during the readback process. The NV_FENCE mechanism could then report when the readback is complete. This feature could possibly be added through the use of threads, but this idea has been left for future research.

The last row of Table 3 shows the estimated speed if we had an infinitely fast GPU, which shows that most of our time is spent on the CPU reorganizing the geometry and rays into coherent structures. This effect has been observed in similar ray tracers[32], where BSP tree traversal is "typically 2-3 times as costly as ray-triangle intersection."

Table 4 shows the effect of tuning the number of triangles that get sent to the GPU. In each of these cases, the number of rays intersected by each GPU pass was set to 64.

Table 5 shows that the number of rays in each bucket can also be varied to achieve peak efficiency. Tuning the ray engine to assign more rays to the GPU frees the CPU to perform more caching. For example, for the teapot room classical ray tracing, we were able to achieve a 52% speedup over the CPU by setting R to 256 and hand tuning the octree resolution.

T	GPU Rays	Rays/sec.	Speedup
CPU		135,812	
4–16	78%	147,630	8%
5–12	81%	157,839	16%
5–15	89%	165,098	22%

Table 4: *Tuning the ray engine by varying the range of triangles T sent to the GPU, measured on the teapot room.*

R	Rays/sec.	Speedup
CPU	135,812	
64	165,098	22%
128	177,647	31%
256	180,558	33%
512	175,904	29%

Table 5: *Tuning the number of rays R sent to the GPU for intersection.*

7. Conclusions

We have added ray tracing to the growing list of applications accelerated by the programmable shaders found in modern graphics cards. Our ray engine performed at speeds comparable to the fastest CPU ray tracers. We expect the GPU will become the high-performance ray-tracing platform of choice due to the rapid growth rate of GPU performance.

By partitioning computation between the CPU and GPU, we combined the best features of both, at the expense of the slow readback of data and the overhead of ray caching. The AGP graphics bus supports high-bandwidth transmission from the CPU to the GPU, but less bandwidth for recovery of results. We expect future bus designs and driver implementations will soon ameliorate this roadblock.

The overhead of ray caching limited the performance speedup of GPU to less than double that of the CPU only, and this overhead as also burdened others[32]. Even though our method for processing the data structures is considered quite efficient[27], we are anxious to explore alternative structures that can more efficiently organize rays and geometry for batch processing by the GPU.

Acknowledgements

This research was supported in part by the NSF grant #ACI-0113968, and by NVidia. The idea of using fragment programs for ray-triangle intersection and the crossbar formalism resulted originally from conversations with Michael McCool.

References

1. ARVO, J., AND KIRK, D. B. Fast ray tracing by ray classification. *Proc. SIGGRAPH 87* (July 1987), 55–64.
2. BADOUEL, D. An efficient ray-polygon intersection. In *Graphics Gems*. Academic Press, Boston, 1990, pp. 390–393, 735.
3. BLINN, J. F., AND NEWELL, M. E. Texture and reflection in computer generated images. *Comm. ACM 19*, 10 (Oct. 1976), 542–547.
4. CLARK, J. The geometry engine: A VLSI geometry system for graphics. *Proc. SIGGRAPH 82* (July 1982), 127–133.
5. DIEFENBACH, P. J., AND BADLER, N. I. Multi-pass pipeline rendering: Realism for dynamic environments. In *Proc. Symposium on Interactive 3D Graphics* (Apr. 1997), ACM SIGGRAPH, pp. 59–70.
6. ERICKSON, J. Pluecker coordinates. *Ray Tracing News 10*, 3 (1997), 11. www.acm.org-/tog/resources/RTNews/html/rtnv10n3.html#art11.
7. FERNANDO, R., FERNANDEZ, S., BALA, K., AND GREENBERG, D. P. Adaptive shadow maps. *Proc. SIGGRAPH 2001* (Aug. 2001), 387–390.
8. HALL, D. The AR350: Today's ray trace rendering processor. In *Hot 3D Presentations*, P. N. Glagowski, Ed. Graphics Hardware 2001, Aug. 2001, pp. 13–19.
9. HEIDRICH, W., LENSCH, H., AND SEIDEL, H.-P. Light field-based reflections and refractions. *Eurographics Rendering Workshop* (1999).
10. JENSEN, H. W. Importance driven path tracing using the photon map. *Proc. Eurographics Rendering Workshop* (Jun. 1995), 326–335.
11. JENSEN, H. W., MARSCHNER, S. R., LEVOY, M., AND HANRAHAN, P. A practical model for subsurface light transport. *Proc. SIGGRAPH 2001* (Aug. 2001), 511–518.
12. KAJIYA, J. T. The rendering equation. *Proc. SIGGRAPH 86* (Aug. 1986), 143–150.
13. KELLER, A. Instant radiosity. *Proc. SIGGRAPH 97* (Aug. 1997), 49–56.
14. KIPFER, P., AND SLUSALLEK, P. Transparent distributed processing for rendering. *Proc. Parallel Visualization and Graphics Symposium* (1999), 39–46.
15. KOLB, C., HANRAHAN, P. M., AND MITCHELL, D. A realistic camera model for computer graphics. *Proc. SIGGRAPH 95* (Aug. 1995), 317–324.
16. LINDHOLM, E., KILGARD, M. J., AND MORETON, H. A user-programmable vertex engine. *Proc. SIGGRAPH 2001* (July 2001), 149–158.
17. MCCOOL, M. D., AND HEIDRICH, W. Texture shaders. In *Proc. Graphics Hardware 99* (August 1999), SIGGRAPH/Eurographics Workshop, pp. 117–126.
18. MÖLLER, T., AND TRUMBORE, B. Fast, minimum storage ray-triangle intersectuion. *Journal of Graphics Tools 2*, 1 (1997), 21–28.
19. PARKER, S., MARTIN, W., SLOAN, P.-P. J., SHIRLEY, P. S., SMITS, B., AND HANSEN, C. Interactive ray tracing. In *1999 ACM Symposium on Interactive 3D Graphics* (Apr. 1999), ACM SIGGRAPH, pp. 119–126.
20. PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. Interactive multi-pass programmable shading. *Proc. SIGGRAPH 2000* (2000), 425–432.
21. PHARR, M., KOLB, C., GERSHBEIN, R., AND HANRAHAN, P. M. Rendering complex scenes with memory-coherent ray tracing. *Proc. SIGGRAPH 97* (Aug. 1997), 101–108.
22. PROUDFOOT, K., MARK, W. R., TZVETKOV, S., AND HANRAHAN, P. A real-time procedural shading system for programmable graphics hardware. *Proc. SIGGRAPH 2001* (2001), 159–170.
23. PURCELL, T. J. SHARP ray tracing architecture. SIGGRAPH 2001 Real-Time Ray Tracing Course Notes, Aug. 2001.
24. PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. Ray tracing on programmable graphics hardware. *Proc. SIGGRAPH 2002* (July 2002).
25. REEVES, W. T., SALESIN, D. H., AND COOK, R. L. Rendering antialiased shadows with depth maps. *Proc. of SIGGRAPH 87* (Jul. 1987), 283–291.
26. REINHARD, E., CHALMERS, A., AND JANSEN, F. Overview of parallel photorealistic graphics. *Eurographics '98 STAR* (Sep. 1998), 1–25.
27. REVELLES, J., URENA, C., AND LASTRA, M. An efficient parametric algorithm for octree traversal. *Proc. Winter School on Computer Graphics* (2000).
28. SZIRMAY-KALOS, L., AND PURGATHOFER, W. Global ray-bundle tracing with hardware acceleration. *Proc. Eurographics Rendering Workshop* (June 1998), 247–258.
29. TRENDALL, C., AND STEWART, A. J. General calculations using graphics hardware with applications to interactive caustics. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering* (Jun. 2000), Eurographics, pp. 287–298.
30. VEACH, E., AND GUIBAS, L. J. Metropolis light transport. *Proc. SIGGRAPH 97* (Aug. 1997), 65–76.
31. WALD, I., SLUSALLEK, P., AND BENTHIN, C. Interactive distributed ray tracing of highly complex models. In *Rendering Techniques 2001* (2001), Eurographics Rendering Workshop, pp. 277–288.
32. WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3 (2001), 153–164.
33. WARD, G. J. The radiance lighting simulation and rendering system. *Proc. SIGGRAPH 94* (Jul. 1994), 459–472.
34. WEGHORST, H., HOOPER, G., AND GREENBERG, D. Improved computational methods for ray tracing. *ACM Trans. on Graphics 3*, 1 (Jan. 1984), 52–69.
35. WILLIAMS, L. Casting curved shadows on curved surfaces. *Proc. SIGGRAPH 78* (Aug. 1978), 270–274.