Graphics Hardware (2002), pp. 37–46
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.

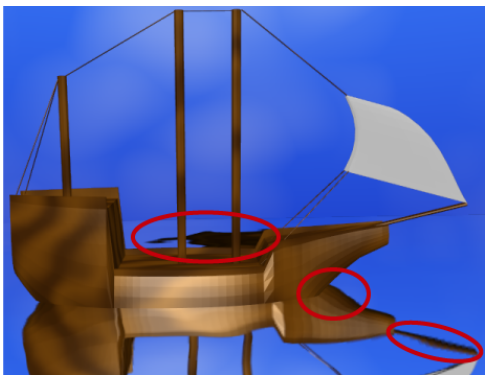


Figure 1: *What is wrong with this environment-mapped picture? (1) The boat does not meet its reflection, (2) the boat is reflected in the water behind it, and (3) some aliasing can be seen in the reflection.*

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.

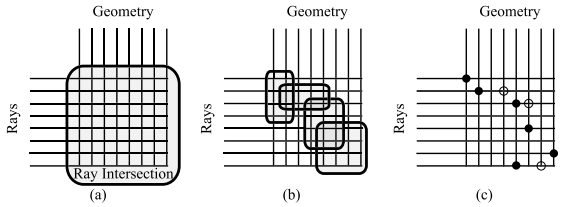


Figure 2: *Ray intersection is a crossbar.*

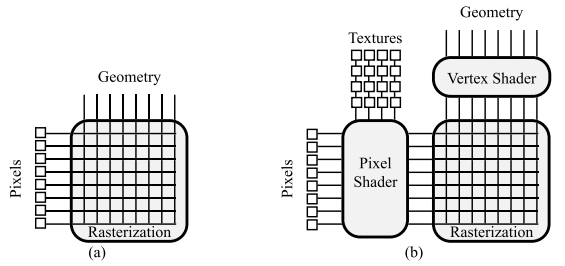


Figure 3: *Programmable pixel shading is a crossbar.*

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

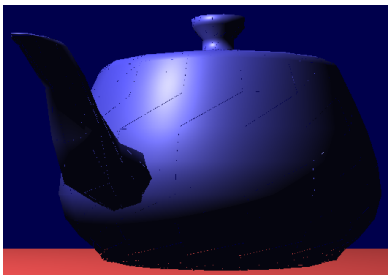


Figure 4: *Leaky teapot, due to the low precision implementation on PS1.4 pixel shaders used to test the performance of ray-triangle intersection. Our simulations using the precision available on upcoming hardware are indistinguishable from software renderings.*

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.

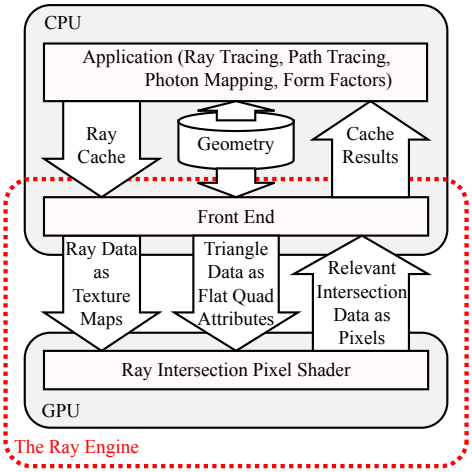


Figure 5: *The organization of the ray engine.*

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.*

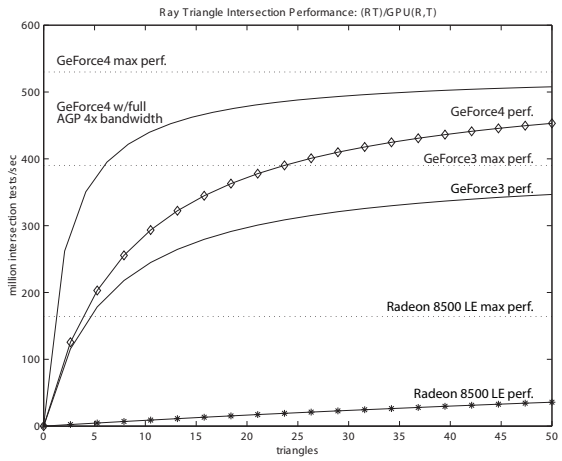


Figure 7: *Theoretical performance in millions of ray-triangle intersection tests per second on the GPU with* $\gamma = 4$.

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.

Graphics Hardware (2003), pp. 51–59
M. Doggett, W. Heidrich, W. Mark, A. Schilling (Editors)

GPU Algorithms for Radiosity and Subsurface Scattering

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

Dept. of Computer Science, University of Illinois, Urbana-Champaign

Abstract

We capitalize on recent advances in modern programmable graphics hardware, originally designed to support advanced local illumination models for shading, to instead perform two different kinds of global illumination models for light transport. We first use the new floating-point texture map formats to find matrix radiosity solutions for light transport in a diffuse environment, and use this example to investigate the differences between GPU and CPU performance on matrix operations. We then examine multiple-scattering subsurface light transport, which can be modeled to resemble a single radiosity gathering step. We use a multiresolution meshed atlas to organize a hierarchy of precomputed subsurface links, and devise a three-pass GPU algorithm to render in real time the subsurface-scattered illumination of an object, with dynamic lighting and viewing.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Subsurface Scattering

1. Introduction

The programmable shading units of graphics processors designed for z-buffered textured triangle rasterization have transformed the GPU into general purpose streaming processors, capable of performing a wider variety of graphics, scientific and general purpose processing.

A key challenge in the generalization of the GPU to non-rasterization applications is the mapping of well-known algorithms to the streaming execution model and limited resources of the GPU. Some popular graphics and scientific algorithms and data structures, such as ray tracing [19, 3] as well as preconditioned conjugate gradient and multigrid solvers [2], have nevertheless been implemented on, or at least accelerated by, the GPU. This paper expands the horizon of photorealistic rendering algorithms that the GPU can accelerate to include matrix radiosity and subsurface scattering, and describes how the techniques could eventually lead to a GPU implementation of hierarchical radiosity.

Matrix radiosity is a classic technique for simulating light transport in diffuse scenes [7]. It is capable of synthesizing and depicting the lighting, soft shadowing and color bleeding found in scenes of diffuse materials. Our mapping of radiosity to the GPU uses the floating-point texture format to hold the radiosity matrix, but also pays attention to cache coherence and the order of computation to efficiently perform a Jacobi iteration which gathers radiosities as it iterates toward a solution. Given precomputed form factors, we are thus able to both compute and display a radiosity solution entirely on the GPU. While the geometry is fixed, the emittance is not, and our GPU algorithm can support dynamic relighting as well as dynamic alteration of patch reflectances.

Lensch *et al.*[15] shows how the BSSRDF formulation of subsurface (multiple diffuse) scattering[12] resembles a single radiosity gathering step. Light transport in the object interior is computed by gathering, for each patch, the diffused image of the Fresnel transmitted irradiances from the other patches. The BSSRDF can be integrated to form a *throughput factor*[15] that re-

sembles a form factor. We use this similarity to bridge our GPU implementation of matrix radiosity into a GPU implementation of subsurface scattering.

Subsurface scattering can be computed more efficiently through a multiresolution approach. Whereas others have used an octree representation[12], we have opted for a MIP-mappable texture atlas representation[4] of surface radiosities. This representation allows the BSSRDF to be integrated over the surface of an object by applying a fragment program to the appropriate level of the MIP-mapped atlas. This results in a GPU implementation of multiresolution subsurface multiple diffuse scattering that runs in real time (61Hz) on a mesh with 7K faces.

2. Previous Work

2.1. Radiosity

Early radiosity techniques beginning with Goral *et al.*[7] required computationally intensive solutions, which led to numerous acceleration strategies including shooting[5] and overrelaxation. Though form factor computation is generally considered the bottleneck of radiosity techniques, hardware accelerated methods such as the hemicube[6] have been available for a long time, though recent improvements exist[17].

Graphics researchers have looked to hardware acceleration of radiosity long before commodity graphics processors became programmable[1]. For example, Keller[14] used hardware-accelerated OpenGL to accelerate radiosity computations. He performed a quasi-Monte Carlo particle simulation for light transport on the CPU. He then placed OpenGL point light sources at the particle positions, setting their power to the radiosity represented by the particle's power along the random walk path. He then used OpenGL to render the direct illumination due to these particles, integrating their contribution with an accumulation buffer.

Martin *et al.*[16] computed a coarse-level hierarchical radiosity solution on the CPU, and used graphics hardware to refine the solution by texture mapping the residual.

In each of these cases, graphics hardware is used to accelerate elements of the radiosity solution, but the bulk of the processing occurs on the CPU. Our goal is to port the bulk of the radiosity solution process to the GPU, using the CPU for preprocessing.

2.2. Subsurface Scattering

Hanrahan and Krueger[9] showed subsurface scattering to be an important phenomena when rendering translucent surfaces, and used a path tracing simulation to render skin and leaves. Pharr *et al.*[18] extended these techniques into a general Monte-Carlo ray tracing framework. Jensen and Buhler[13] used a dipole and diffusion to approximate multiple scattering. These methods made clear the importance of subsurface scattering to the graphics community, and led some to consider additional approximations and accelerations.

Jensen *et al.*[12] accelerates subsurface scattering using a hierarchical approach consisting of several passes. Their first pass finds surface irradiances from external light whereas the second pass transfers these irradiances to the other surface patches. Their hierarchical approach uses an octree to volumetrically represent the scattered irradiances in the interior of the translucent substance.

Hao *et al.*[11] approximated subsurface scattering using only local illumination, by bleeding illumination from neighboring vertices to approximate local back scattering. They precompute a scattering term for source lighting expressed in a piecewise linear basis. They reconstructed scattering per-vertex from directional light by linearly interpolating these terms computed from the nearest surrounding samples. Their technique was implemented on the CPU but achieved real-time rendering speeds.

Sloan *et al.*[21] uses a similar precomputation strategy, though using a spherical harmonic basis for source lighting. They precomputed per-vertex a transfer matrix of spherical harmonic coefficients from environment-mapped incoming radiance to per-vertex exit radiance that includes the effects of intra-object shadowing and interreflection in addition to subsurface scattering. They were able to compress these large transfer matrices in order to implement the evaluation of precomputed radiance transfer entirely on the GPU to achieve real-time frame rates.

Lensch *et al.*[15] approximated back scattering by filtering the incident illumination stored in a texture atlas. The shape of these kernels is surface dependent and precomputed before lighting is applied. They also approximated forward scattering by precomputing a vertex-to-vertex throughput factor, which resembles a form factor. Forward scattering is rendered by performing a step similar to radiosity gathering, by collecting for a given vertex the irradiance from the other vertices scaled by the throughput factor.

While Lensch *et al.*[15] benefited from some hardware acceleration, for example using a vertex shader to accumulate external irradiances, the application of the vertex-to-vertex throughput factors to estimate forward scattering, and the atlas kernel filtering to estimate backward scattering is performed on the CPU

(actually a pair of CPUs). They expressed a desire to explore full GPU implementations of these techniques, and our GPU implementation of subsurface scattering is an extension of their technique. But as we have discovered and documented in this paper, their technique requires novel extensions to be efficiently implemented given the limited resources of a modern GPU.

2.3. Texture Atlas Generation

A texture atlas is a one-to-one mapping from an object surface into a 2-D texture map image. Texture atlases arise from a parameterization of an object surface, and provide a mechanism for navigating complex surfaces by navigating its charts in a flat texture space. The texture atlas also provides method for applying the pixel shader processors of a GPU to a sampling of an entire object surface (instead of just the visible portions of the surface).

Following Carr and Hart[4], we use a multiresolution meshed atlas (MMA) to distribute available texture samples evenly across the entire object surface. Unlike more common atlases, the MMA is discrete, packing each triangle individually into the texture map, independently from its neighbors. Seam artifacts that can occur when a triangle's sample is inadvertently drawn from a neighboring texel in the texture map are avoided by carefully matching the rules of rasterization with the rules of texture sampling. A half-pixel gutter surrounds each texture map triangle to support bilinear texture interpolation to avoid magnification aliasing.

The MMA is based on a binary face clustering where each node in a binary tree represents a simply connected cluster of mesh triangles. The two children of a node represent a disjoint partitioning of the parent cluster into two simply connected (neighboring) subsets. Leaf nodes correspond to individual surface mesh triangles. Hence the binary tree contains exactly $2T-1$ nodes, where T is the number of triangles in our mesh.

The MMA creates a correspondence between each triangle cluster corresponding to a node in the binary tree with a rectangular (either 2:1 or square) region of the texture map. Hence the face cluster hierarchy is packed as a quadtree hierarchy in the texture map. This organization allows the texture map to be MIP-mapped such that a each texel value (color) in a lower-resolution level of the MIP-map corresponds to an average cluster value (color) in the mesh. Other MIP-mappable atlas methods exist[20], but the MMA more completely utilizes available texture samples. This MIP-map allows the MMA to reduce minification aliasing, as well as providing a data structure to store and quickly recall precomputed sums or averages over mesh regions.

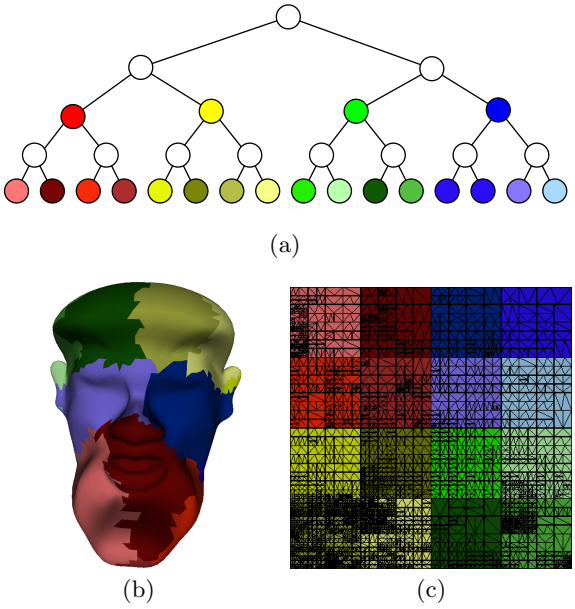


Figure 1: *A multiresolution meshed atlas of a cow. Each node in the binary tree (a) corresponds to a cluster of triangles in the model (b) and a rectangular region in the texture domain (c). Each cluster/region is the union of its corresponding node's children's cluster/region.*

3. Matrix Radiosity

The recent support for floating point texture formats provides an obvious application to matrix problems. Matrix radiosity[7] expresses the illumination of a closed diffuse environment as a linear system

$$B_i = E_i + \rho_i \sum_{j=1}^{N} F_{ij} B_j \qquad (1)$$

where B is a column vector of N radiosities, E are the emittances, ρ are the diffuse reflectivities and F_{ij} is the form factor. This system is solved in $Ax = b$ form as

$$MB = E \qquad (2)$$

where M is a matrix formed by the reflectivities and form factors.

A variety of techniques have been used to solve (2) in computer graphics, including standard numerical techniques such as Jacobi and Gauss-Seidel iteration. Gauss-Seidel is typically preferred because it requires less space and converges about twice as fast as Jacobi. However, Gauss-Seidel is inherently serial, whereas Jacobi iteration offers significant parallelism, and hence takes better advantage of the streaming na-

ture of GPU architectures. The typical Jacobi iteration formula is

$$B_i^{(k+1)} = E_i - \sum_{j \neq i} M_{ij} \frac{B_j^{(k)}}{M_{ii}} . \quad (3)$$

To avoid the conditional execution caused by the $j \neq i$ condition (which cannot be done efficiently with current fragment processing architectures), we do the full matrix-vector product:

$$B^{(k+1)} = B^{(k)} + E - \mathrm{diag}(M)^{-1} M B^{(k)} . \quad (4)$$

The matrix $\mathrm{diag}(M)^{-1}$ is the inverse of the matrix containing only the diagonal elements of M. (For constant elements, as are typically used, $\mathrm{diag}(M) = I$, and the denominator also drops out of (3).)

We used one texel per matrix element. The radiosity system is computed per wavelength of light, so an RGB texture can simultaneously represent a red, green, and blue matrix. When solving a single-channel luminance (as opposed to RGB) matrix system, one can pack the vectors and matrices into all four color channels, such that four elements of a vector (or four columns of one row of a row-major matrix) can be accessed in a single texture lookup. We have found this organization works best for matrix-vector products, whereas a block-channel packing (where e.g. the upper-left submatrix is stored in a texture's red channel) works better for single-channel matrix-matrix products[8].

Each pass operates on a block of the matrix-vector product, where the block size is dictated by the number of pixel shader instructions available per pass. For example, our GeForce FX implementation uses a block size of 254 elements, since each four-element texel requires four instructions (two texture fetches, a multiply-add, and an update to the texture coordinates), and the GeForce FX pixel shader supports a maximum of 1024 instructions (there are a few instructions of overhead for each pass). Thus for an N-patch radiosity solution, each Jacobi iteration takes $\lceil N/254 \rceil$ passes.

3.1. Results

We tested the solver on a simple Cornell box without any occluders, since form factor complexity should not significantly affect solution time. Figure 2 demonstrates the solution, which was solved and displayed (without interpolation) entirely on the graphics card, using precomputed form factors. The vector E and the matrix M were formed by a CPU preprocess, and loaded as a 1-D and 2-D texture, respectively. A pixel shader performed the Jacobi iteration to solve for the unknown radiosities B which were also stored as a 1-D texture. The iterations can be observed by displaying the scene using texture coordinates based on the 1-D radiosity texture, though a 1-D texture prevents bilinear interpolation of the displayed radiosities.

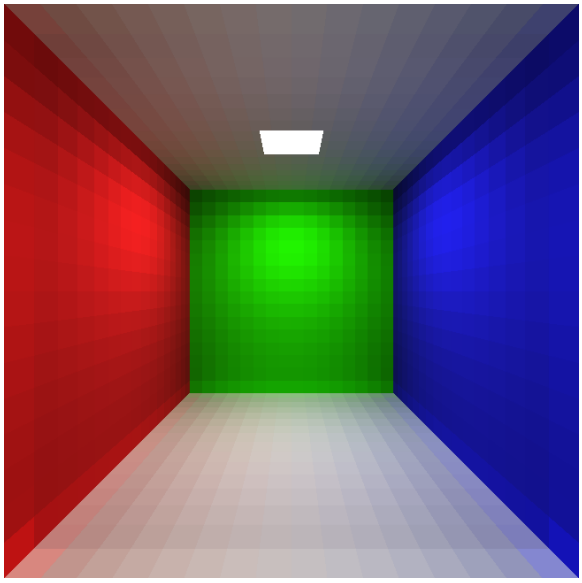


Figure 2: *A simple radiosity scene solved and displayed completely on the GPU.*

In order to smoothly interpolate the displayed radiosities, the 1-D radiosity texture B would need to be resampled into a 2-D displayed radiosity texture. An easy way to perform this resampling is to create an atlas of the scene, such as the meshed atlas described in Section 2.3, and render the texture image of the atlas using 1-D texture coordinates corresponding to each vertex's index in the radiosity vector.

As Figure 3 shows, our GPU Jacobi radiosity implementation takes about twice as many iterations to converge as does a Gauss-Seidel solution on the CPU. Moreover, each Jacobi iteration of our GPU solver (29.7 iterations/sec. on a NVIDIA GeForce FX 5900 Ultra) takes longer to run than a Gauss-Seidel iteration on the CPU (40 iterations/sec. on an AMD Athlon 2800+). This corresponds to 141 MFLOPS/s and 190 MFLOPS/s for the GPU and CPU, respectively (the floating point operations for indexing on the GPU are not included in this number).

We found, however, that the CPU implementation is limited by memory bandwidth, while the GPU implementation is only using a fraction (perhaps as little as 10%) of its available bandwidth. This difference can be observed by comparing the super-linear CPU curve to the linear GPU curve in Figure 4. We believe

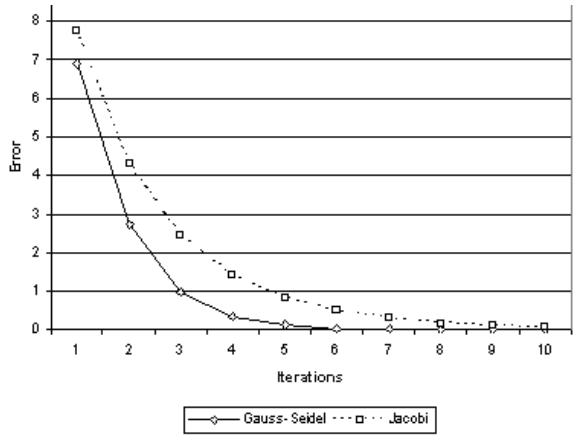


Figure 3: *Gauss-Seidel converges faster than Jacobi iteration. Error is measured as the mean squared error of the residual $MB - E$.*

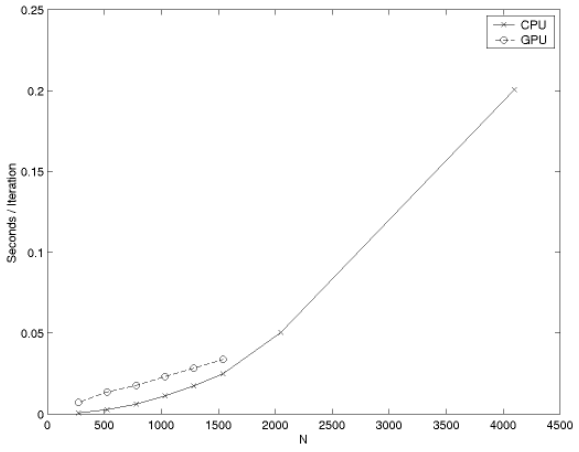


Figure 4: *Though the CPU currently outperforms the GPU on this matrix radiosity example, the CPU is memory bandwidth bound and its performance degrades on larger matrices, whereas the GPU is compute bound and its performance scales linearly.*

the GPU curve overtakes the CPU curve for matrices larger than 2000 elements, but we were limited by a maximum single texture resolution of 2048. As computation speeds have historically increased faster than memory bandwidth, we also expect the GPU will readily outperform the CPU in the near future.

4. Subsurface Scattering

We base our subsurface scattering scheme on the method derived by Jensen *et al.*[13]. The standard rendering equation using a BRDF approximation has been used for many years to model light transport and the reflectance of surfaces. However, the BRDF formulation assumes that light entering a material at a point also leaves the material at the same point. For many real-world surface this approximation is sufficient, but for numerous translucent materials (skin, milk, marble, etc..), much of their appearance comes from internal scattering of light. To account for subsurface scattering, the BRDF is replaced with a more general model known as the BSSRDF (bidirectional surface scattering reflectance distribution function). The amount of radiance leaving a point x_o due to subsurface scattering can be expressed as

$$L_o(x_o, \vec{\omega}_o) = \int_A \int_\Omega S(x_i, \vec{\omega}_i; x_o, \vec{w}_o) L_i(x_i, \vec{\omega}_i)(\vec{n}_i \cdot \vec{\omega}_i) d\vec{\omega}_i dA(x_i). \quad (5)$$

Integration is performed over the surface A at points x_i and all incident light directions. The term S is the BSSRDF, which relates the amount of outgoing radiance at point x_o in direction $\vec{\omega}_o$, given that there is incoming radiance at some other surface point x_i in direction $\vec{\omega}_o$.

Jensen et. al. noted that single scattering for many common materials only accounts for a small percentage of the outgoing radiance of a material[12]. Also, light tends to become heavily scattered when it enters materials, removing the relationship between incident and exitant light directions. This simplifies the BSSRDF to a four dimensional function R_d known as the diffuse BSSRDF. Reformulating the subsurface scattering equation (5) to only account for subsurface scattering we have

$$L_o(x_o, \vec{\omega}_o) = \frac{1}{\pi} F_t(\eta, \vec{\omega}_o) B(x_o) \quad (6)$$

$$B(x_o) = \int_{x_i \in S} E(x_i) R_d(x_i, x_o) dA(x_i) \quad (7)$$

$$E(x_i) = \int_\Omega L_i(x_i, \vec{\omega}_i) F_t(\eta, \vec{\omega}_i) |\vec{n}_i \cdot \vec{\omega}_i| d\vec{\omega}_i \quad (8)$$

where $R_d(x_i, x_o)$ is the diffuse subsurface scattering reflectance. It encodes geometric information and also the volumetric material properties anywhere in the object dealing with light transport from x_i to x_o. For R_d we use the dipole approximation model detailed by Jensen et. al. [12], and also used in Lensch [15]. Also found in Jensen et. al. are the scattering and absorption coefficients: σ_s', σ_a for common materials that are used in this model. For brevity we have omitted the details here.

Lensch et. al. noted this strong similarity between the radiosity formula and the formula in (8). This equation may be solved by discretizing the scene into patches.

4.1. Real-Time Subsurface Approximation Algorithm

We start by discretizing our object into a collection of N patches where P_i and P_j are patches on the surface S. We can reformulate (8) into its discretized form as follows:

$$B_i = \sum_{j=1}^{N} F_{ij} E_j \quad (9)$$

which resembles a single transport step of radiosity transport (1). The multiple diffuse scattering throughput factor F_{ij} is expressed as:

$$F_{ij} = \frac{1}{A_i} \int_{x_i \in P_i} \int_{x_j \in P_j} R_d(x_j, x_i) dx_j dx_i. \quad (10)$$

For a static model, we can precompute the F_{ij} factors between all pairs of patches. Using (9), the radiosity due to diffuse multiple scattering now reduces to a simple inner product for each patch resulting in $O(N^2)$ operations to compute the incident scattered irradiance for all patches.

A simple way to reduce the number of interactions is to turn to a clustering strategy like that used to solve the N-body problem and hierarchical radiosity[10]. This is particularly applicable to subsurface scattering since the amount of scattered radiance drops rapidly as it travels further through the media. This implies that patches that are far away from the patch whose scattered radiosity we are computing may be clustered together and be replaced by a single term to approximate the interaction.

4.2. A Three Pass GPU Method

We now formalize a solution the diffuse subsurface scattering equation (9) as a three pass GPU scheme (as shown in Fig. 5) preceded by a pre-computation phase. By assuming that our geometry is not deforming we are able to precompute all of our throughput factors F_{ij} between interacting surfaces.

Our first pass to the GPU computes the amount of light incident on every patch of our model, and scales this by the Fresnel term, storing the radiosity that each patch emits internal to the object. This map forms our *radiosity map*.

Our second pass acts as a gathering phase evaluating equation (9). For every patch/texel the transmitted radiosity is gathered and scaled by the precomputed throughput factors and stored into a *scattered irradiance map*.

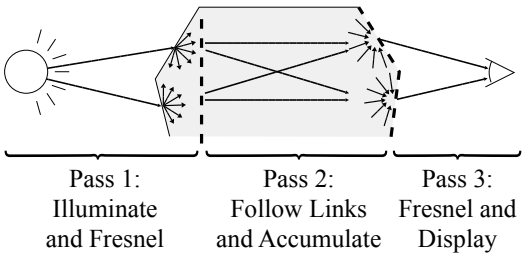


Figure 5: *Three passes for rendering subsurface scattering on the GPU.*

In our third and final pass we render our geometry to the screen using the standard OpenGL lighting model. The contribution from subsurface scattered light is added in by applying the scattered irradiance texture map to the surface of the object scaled by the Fresnel term.

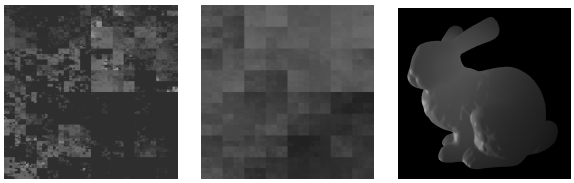


Figure 6: *Pass one plots triangles using their texture coordinates (left), interpolating vertex colors set to their direct illumination scaled by a Fresnel transmission term. Pass two transfers these radiances (center) via precomputed scattering links implemented as dependent texture lookups into a MIP-map of the previous pass result (left). Pass three scales the resulting radiances by a Fresnel transmission factor and texture maps the result onto the original model.*

4.2.1. Pre-computation Phase

We start by forming a hierarchical disjoint face clustering and texture atlas parameterization of our mesh with a method previously developed for real-time procedural solid texturing[4]. Every texel in our texture atlas corresponds to a patch on the surface of our model. This parameterization method is ideal for a GPU solution to the subsurface scattering problem for a number of reasons. First, it provides a natural face cluster hierarchy necessary for a hierarchical approach. Secondly, it works directly with the GPU rasterization rules allowing the GPU to perform surface computation in a seam-free manner. Thirdly, it is MIP-mappable, allowing the GPU to compute fast average and sums

over the surface hierarchy. Lastly, by using a parameterization scheme as a domain to store and compute our surface samples, the number of surface samples is independent of both the tessellation of our geometry, and the resolution of the screen we are rendering to. This marks a difference between earlier interactive subsurface scattering approaches where vertex to vertex interactions were used to discretize the scattering equation.

Full evaluation of equation (9) of patch to patch throughput factors would require P^2 interactions, where P is the number of patches (and also texels in our texture atlas). Each interaction forms a link. Since all of our patches are in texture space, we need only store a u, v location and a throughput factor for each link. By adding an LOD term into the link structure we can access any level in the MIP-map surface hierarchy. Based on our computation and space restrictions we assign some maximum number of links L that we store for each patch P_b in the base level of our texture map.

For a non-adaptive approach we can choose some level l in the hierarchy. We then build all links from patches P_b to P_l where P_b are the patches at the lowest level in the hierarchy, and P_l are patches at level l in the hierarchy.

An adaptive top-down approach for the construction of links may be done similar to that of hierarchical radiosity. For every patch in the lowest level of our hierarch P_b, we start by placing the root patch P_r of our hierarchy onto a priority queue (with highest throughput factor at the top). We then recursively remove the patch at the top of the queue, compute the throughput factors from P_b to its four children, and insert the four children into the priority queue. This process is repeated until the queue size grows to the threshold number of links desired.

Once L adaptive links for each patch/texel in our atlas have been created, we store the result into L textures maps that are $\sqrt{P_b} \times \sqrt{P_b}$ in size. We use an fp16 (16-bit floating point) texture format supported on the GeForceFX to pack the link information: u, v, F_{ij},LOD into the four color channels to be used during pass two of our rendering stage. To reduce storage in favor of more links, we opted to reduce our form factor to a single scalar. Form factors per color channel may be supported at a cost of space and bandwidth.

In the case of a non-adaptive approach, the link locations and LOD are the same for every patch P_b, we therefore store this information in constant registers on the graphics card during the second pass. The throughput factors, however, vary per-link. We store the form factor information into $L/4$ texture maps where each texel holds four throughput factors (corresponding to 4 links) in the rgbα channels.

4.2.2. Pass 1: Radiosity Map

Given our face cluster hierarchy and MIP-mappable parameterization, we must first compute the E_j's for the patches in our scene. To do this, we start by computing a single incident irradiance for every texel in our texture atlas, thereby evaluating lighting incident on all sides of the model. We scale the result of the incident illumination by the Fresnel term, storing the result in the texture atlas. Each texel now holds the amount of irradiance that is transferred through the interface to the inside of the model. This step is similar to the *illumination map* used in Lensch *et al.*[15].

To accomplish this efficiently on the GPU, we use the standard OpenGL lighting model in a vertex program on the GPU. Using the OpenGL render-to-texture facility, we send our geometry down the graphics pipeline. The vertex program computes the lighting on the vertices scaled by the Fresnel term placing it in the color channel and swaps the texture coordinates for the vertices on output. The radiosity stored in the color channel is then linearly interpolated as the triangle gets rendered into the texture atlas. Our method does not prevent the use of more advanced lighting models and techniques for evaluating the incident irradiance on the surface the object.

As an alternative to computing our transmitted radiosity in a vertex program, we could perform the computation entirely in a fragment program per-texel. In addition to higher accuracy, this approach may have improved performance for high triangle count models. A geometry image (e.g. every texel stores surface position) and a normal map may be precomputed for our object and stored as textures on the GPU. Rendering the radiosity map would only entail sending a single quadrilateral down the graphics pipeline texture mapped with the geometry image and normal map. The lighting model and Fresnel computation can take place directly in the fragment shader.

We use the automatic MIP-mapping feature available in recent OpenGL version to compute the average radiosity at all levels of our surface hierarchy. The radiosity map is then used in the next pass of this process.

4.2.3. Pass 2: Scattered Irradiance Map

This pass involves evaluating equation (9) for every texel in our texture atlas. We render directly to the texture atlas, by sending a single quadrilateral to the GPU. In a fragment program, for each texel, we traverse the L links stored during the pre-computation

phase. Texture lookups are performed to get the link information. Using the u,v and LOD link information a dependent texture lookup is performed on the mip-mapped Radiosity Map. The result of this is scaled by the links form factor. All links are traversed for a given texel and the accumulated radiosities form the incident scattered irradiance for the texel.

This pass can be the most costly of the three passes depending on the number of links chosen. For our adaptive approach, $2*L$ texture lookups must be performed in the fragment shader. For our non-adaptive scheme we were able to pack four links into a single texture lookup resulting in $1.25 * L$ lookups.

4.2.4. Pass 3: Final Rendering

Pass three begins with the scattered irradiance map resulting from pass two. This pass multiplies the scattered irradiance texture with a texture of inverted Fresnel terms to get a texture of external radiosities on the outside of the translucent object's surface. This texture product is further modulated by direct illumination resulting from a standard OpenGL lighting pass, evaluated either per-vertex or per-fragment. This results in the final rendering of the translucent object.

4.3. Subsurface Scattering Results

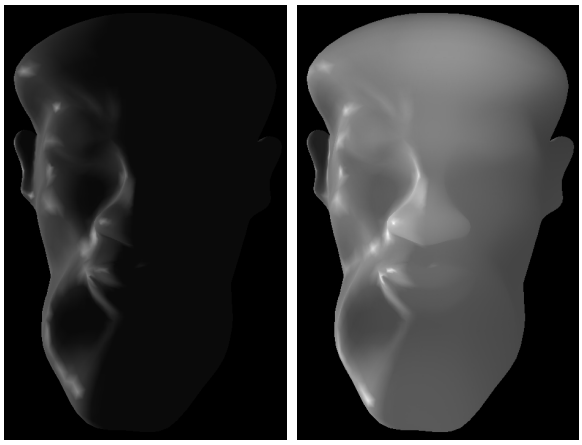
Figure 7 shows the result of our subsurface scattering algorithm running on a GeForce FX card. To achieve real-time performance we are running with either a 512^2 or 1024^2 texture atlas with 16 links per texel. We initially tried to use our adaptive approach for assigning links but found that the adaptivity when using such a course number of links led to visible discontinuities during rendering along the mip-map boundaries. As we increased the number of links, the seams diminished due to the improved approximation.

Precomputation of the links was performed entirely on the CPU and took approximately nine seconds for the 1024^2 resolution.

Model	res.	fps	Pass 1	Pass 2	Pass 3
Head	512^2	61.10	11%	82%	7%
Dolphin	512^2	30.35	43%	43%	14%
Bunny	512^2	30.33	37%	34%	28%
Head	1024^2	15.40	13%	85%	2%
Dolphin	1024^2	15.09	8%	85%	7%
Bunny	1024^2	12.05	18%	68%	14%

Table 1: *Subsurface scattering performance on a GeForce FX 5900.*

We tested the algorithm using an NVidia GeForce FX 5900 on three models. The "head" model with 7,232 faces, a "dolphin" model with 21,952 faces and the "bunny" model with 69,451 faces. The results of the GPU subsurface scattering algorithm is shown in Table 1. At the 1024^2 texture resolution, Pass 2 dominates, and since this pass is a texture-to-texture pass, the use of a texture atlas has effectively decoupled the total shading cost from the tesselation complexity.



(a) (a)

Figure 7: *Without subsurface scattering (a), and with subsurface scattering (b) using* $\sigma_s' = 2.21$, $\sigma_a = 0.0012$, *16 links, 40fps Nvidia GeForceFX 5800.*

5. Conclusion

We have examined the implementation of matrix radiosity on the GPU and used it as an example to examine the performance of the GPU on scientific applications, specifically those involving linear systems.

Our GPU subsurface scattering result is much more successful, yielding full real-time (61 Hz) performance on a present day GPU. Our implementation is novel compared to other real-time subsurface scattering results that are implemented primarily on the CPU. Moreover, our subsurface scattering method is based on a multiresolution meshed atlas, and this multiresolution approach applied to a surface cluster hierarchy is also novel.

The key to our multiresolution surface approach to subsurface multiple diffuse scattering is the assignments of gathering links. These directional links are formed between clusters at different levels in the hierarchy, and a directed link indicates the irradiance over one cluster is gathered from the scattered radiosity of another cluster.

These links are similar to the links used for hierarchical radiosity. Hierarchical radiosity would follow

these links to gather radiosities from other clusters at appropriate levels in the radiosity MIP-map, and radiosities formed at one level would be averaged or subdivided to form radiosities at other MIP-map levels. But hierarchical radiosity gains its greatest speed and accuracy improvement from its ability to dynamically re-allocate these links based on changes in the cluster-to-cluster form factors. The implementation of this dynamic link reassignment on the GPU is of great interest but appears quite challenging, and forms the primary focus of our future work.

Acknowledgments This work was supported in part by NVIDIA and the NSF under the ITR grant #ACI-0113968.

References

1. D. R. Baum and J. M. Winget. Real time radiosity through parallel processing and hardware acceleration. *Computer Graphics*, 24(2):67–75, 1990. (Proc. Interactive 3D Graphics 1990).
2. J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Trans. on Graphics*, 22(3):to appear, July 2003. (Proc. SIGGRAPH 2003).
3. N. A. Carr, J. D. Hall, and J. C. Hart. The ray engine. In *Proc. Graphics Hardware 2002*, pages 37–46, Sep. 2002.
4. N. A. Carr and J. C. Hart. Meshed atlases for real-time procedural solid texturing. *ACM Transactions on Graphics*, 21(2):106–131, 2002.
5. M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg. A progressive refinement approach to fast radiosity image generation. *Computer Graphics*, 22(4):75–84, 1988. (Proc. SIGGRAPH 88).
6. M. F. Cohen and D. P. Greenberg. The hemicube: A radiosity solution for complex environments. *Computer Graphics*, 19(3):31–40, 1985. (Proc. SIGGRAPH 85).
7. C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modelling the interaction of light between diffuse surfaces. *Computer Graphics*, 18(3):213–222, 1984. (Proc. SIGGRAPH 84).
8. J. D. Hall, N. A. Carr, and J. C. Hart. Cache and bandwidth aware matrix multiplication on the GPU. Technical Report UIUCDCS-R-2003-2328, University of Illinois, Apr. 2003. At ftp.cs.uiuc.edu in /pub/dept/tech_reports/2003/ as UIUCDCS-R-2003-2328.ps.gz.
9. P. Hanrahan and W. Krueger. Reflection from layered surfaces due to subsurface scattering. In *Proc. SIGGRAPH 93*, pages 165–174, 1993.
10. P. Hanrahan, D. Salzman, and L. Aupperle. A rapid hierarchical radiosity algorithm. *Computer Graphics*, 25(4):197–206, July 1991. (Proc. SIGGRAPH 91).
11. X. Hao, T. Baby, and A. Varshney. Interactive subsurface scattering for translucent meshes. In *Proc. Interactive 3D Graphics*, pages 75–82, 2003.
12. H. W. Jensen and J. Buhler. A rapid hierarchical rendering technique for translucent materials. *ACM Trans. on Graphics*, 21(3):576–581, 2002. (Proc. SIGGRAPH 2002).
13. H. W. Jensen, S. R. Marschner, M. Levoy, and P. Hanrahan. A practical model for subsurface light transport. In *Proc. SIGGRAPH 2001*, pages 511–518, 2001.
14. A. Keller. Instant radiosity. In *Proc. SIGGRAPH 97*, pages 49–56, 1997.
15. H. P. A. Lensch, M. Goesele, P. Bekaert, J. Kautz, M. A. Magnor, J. Lang, and H.-P. Seidel. Interactive rendering of translucent objects. In *Proc. Pacific Graphics 2002*, pages 214–224, Oct. 2002.
16. I. Martin, X. Pueyo, and D. Tost. A two-pass hardware-based method for hierarchical radiosity. *Computer Graphics Forum*, 17(3):159–164, Sep. 1998.
17. K. H. Nielsen and N. J. Christensen. Fast texture-based form factor calculations for radiosity using graphics hardware. *J. of Graphics Tools*, 6(4):1–12, 2001.
18. M. Pharr and P. Hanrahan. Monte Carlo evaluation of non-linear scattering equations for subsurface reflection. In *Proc. SIGGRAPH 2000*, pages 75–84, 2000.
19. T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. on Graphics*, 21(3):703–712, July 2002. (Proc. SIGGRAPH 2002).
20. P. V. Sander, J. Snyder, S. J. Gortler, and H. Hoppe. Texture mapping progressive meshes. In *Proc. ACM SIGGRAPH 2001*, pages 409–416, 2001.
21. P.-P. Sloan, J. Hall, J. C. Hart, and J. Snyder. Clustered principal components for precomputed radiance transfer. *ACM Trans. on Graphics*, 22(3):to appear, July 2003. (Proc. SIGGRAPH 2003).