

Chapter 8. Rendering Algorithms

John Hart

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}, reflection³, multiple-bounce reflection⁵, refraction⁹, caustics²⁹ and even radiosity¹³. 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 point³. This single-point irradiance approximation can result in some visually obvious errors, such as the boat in wavy water shown in Figure 1.

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 tracing¹⁵. Numerous global illumination methods are based on ray tracing including path tracing¹², Monte-Carlo ray tracing³³ and photon mapping¹⁰.

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 con-

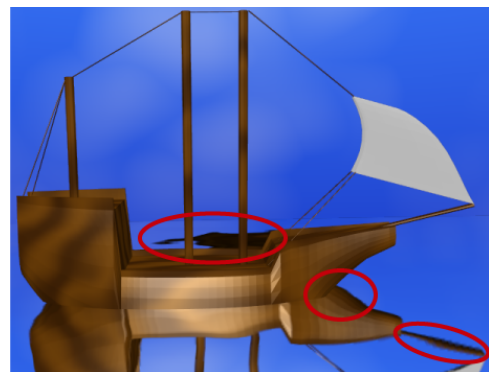


Figure 1: *What is wrong with this environment-mapped picture? (1) The boat does not meet its reflection, (2) the boat is reflected in the water behind it, and (3) some aliasing can be seen in the reflection.*

struction 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 renderers¹⁶. 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 scattering¹¹.

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 rays²¹ 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 CPU³². 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 color³⁴. 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 illumination²⁸, 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 processor³². 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 computers²⁶. 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 512² resolution scenes of from 20 to 2K patches at rates ranging from two to 20 Hz¹⁹.

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 shaders⁸. A ray tracing system designed around multiprocessors with smart memory is also in progress²³.

Our ray engine is similar in spirit to another GPU-based ray tracing implementation that simulates a state machine²⁴. 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.

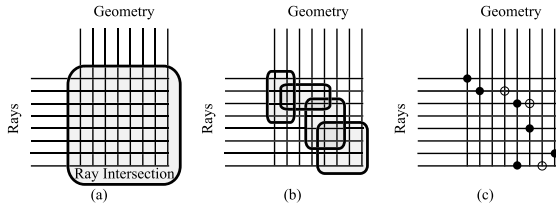


Figure 2: Ray intersection is a crossbar.

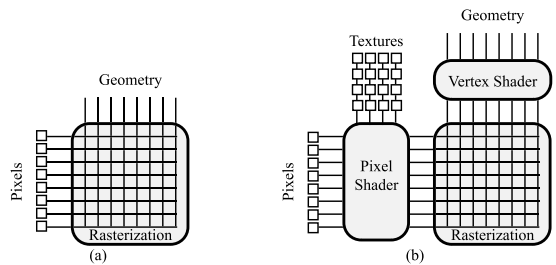


Figure 3: Programmable pixel shading is a crossbar.

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, ex-

actly 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 firmware⁴, modern graphics accelerators contain user-programmable elements designed specifically for advanced shading¹⁶. 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 processor²⁰ 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 processors⁸, which improved load balancing, but inhibited the possible advantages of ray coherence. The coherent ray tracer³² 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 implementation 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 databases³¹.

4. Ray-Triangle Intersection on the GPU

The pixel shader implementation of ray-triangle intersection treats the GPU as a SIMD parallel processor²⁰. 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 pre-computation 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 one¹⁸ to fit in a pixel shader.

Our revised ray-triangle intersection is evaluated as

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

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

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

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

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

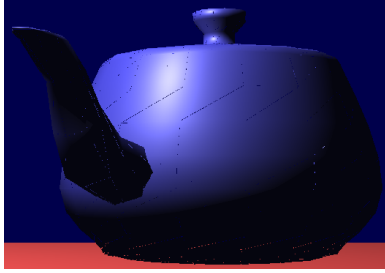


Figure 4: *Leaky teapot, due to the low precision implementation on PS1.4 pixel shaders used to test the performance of ray-triangle intersection. Our simulations using the precision available on upcoming hardware are indistinguishable from software renderings.*

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 III³². 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 GPU²⁴, 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 hardware¹⁷, such hardware is not currently available. Recursive ray tracing can be implemented completely on the GPU²⁴, 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 tracing¹². 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 methods³⁰ 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 scenes²¹. The management of ray caches and the radiances resulting

from 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 shuffling²⁴, 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 systems¹⁴. 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.

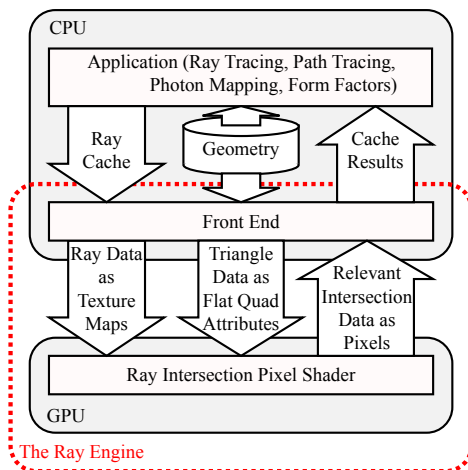


Figure 5: The organization of the ray engine.

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 infrequently (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 tree¹ to maintain ray coherence.

The ray engine works more efficiently when groups of

similar rays intersect a collection of spatially coherent triangles. In order to maintain full buckets of coherent rays, we utilize a ray cache²¹. 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 test¹ 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 performance meets the low end performance of the coherent ray tracer, which was able to trace from 200K to 1.5M rays per second³². 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 $\text{GPU}(R, T)$ and $\text{CPU}(R, T)$. To achieve improved performance, we are only interested in values of R and T for which $\text{GPU}(R, T) \leq \text{CPU}(R, T)$, 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

$$\text{GPU}(R, T) = O(RT). \quad (9)$$

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

$$\text{CPU}(R, T) \leq O(RT). \quad (10)$$

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 $\text{GPU}(R, T)$ with the efficiency of $\text{CPU}(R, T)$.

6.1. The Readback Bottleneck

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

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

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 proportional 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.

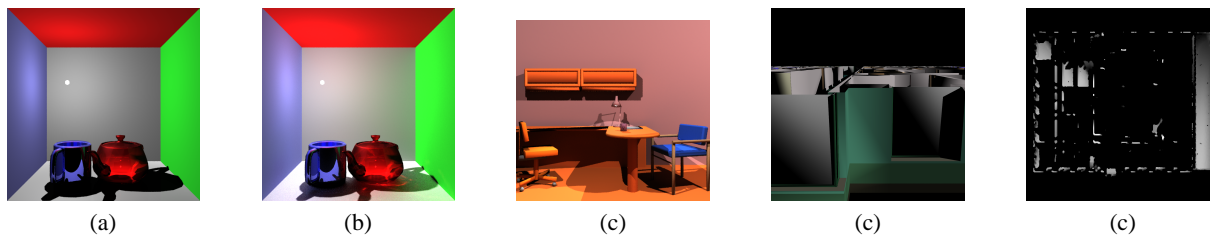


Figure 6: Images tested by the ray engine: teapot Cornell box ray traced classically (a) and Monte Carlo (b), office (c), and Soda Hall side (d) and top (e) views.

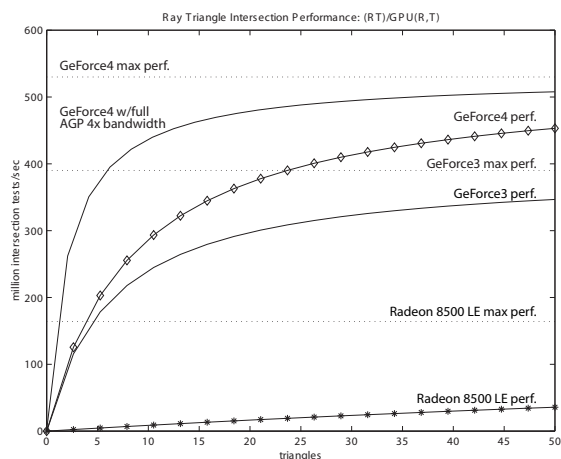


Figure 7: Theoretical performance in millions of ray-triangle intersection tests per second on the GPU with $\gamma = 4$.

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 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 point 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	T	GPU Rays	Rays/sec.	Speedup
Teapot Room Classical	89%	CPU		135,812	
Teapot Room Monte-Carlo	71%	4-16	78%	147,630	8%
Office	65%	5-12	81%	157,839	16%
Soda Hall Top View	70%	5-15	89%	165,098	22%
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 tracers³², 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 per-

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.

form 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.

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 others³². Even though our method for processing the data structures is considered quite efficient²⁷, 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. Glogowski, 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 intersection. *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. M. 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.

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 solvers², 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 scenes⁷. 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) scattering¹² 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 representation¹², we have opted for a MIP-mappable texture atlas representation⁴ 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 shooting⁵ and overrelaxation. Though form factor computation is generally considered the bottleneck of radiosity techniques, hardware accelerated methods such as the hemicycle⁶ have been available for a long time, though recent improvements exist¹⁷.

Graphics researchers have looked to hardware acceleration of radiosity long before commodity graphics processors became programmable¹. For example, Keller¹⁴ 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 Krueger⁹ 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 Buhler¹³ 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 Hart⁴, 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 exist²⁰, 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.

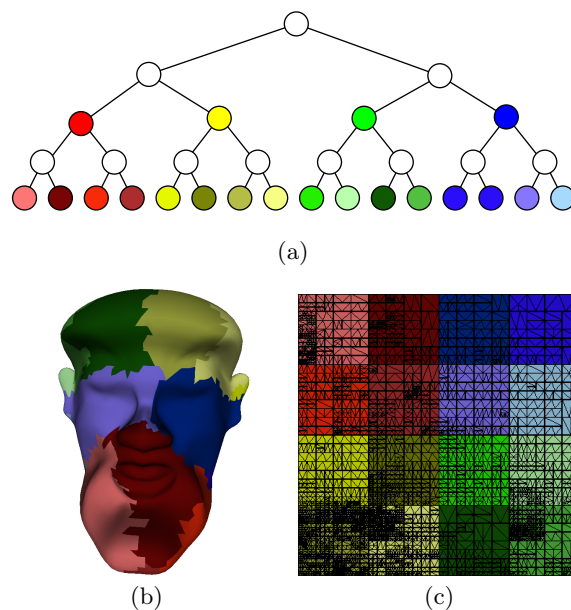


Figure 1: A multiresolution meshed atlas of a cow. Each node in the binary tree (a) corresponds to a cluster of triangles in the model (b) and a rectangular region in the texture domain (c). Each cluster/region is the union of its corresponding node's children's cluster/region.

3. Matrix Radiosity

The recent support for floating point texture formats provides an obvious application to matrix problems. Matrix radiosity⁷ expresses the illumination of a closed diffuse environment as a linear system

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

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

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

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:

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

The matrix $\text{diag}(M)^{-1}$ is the inverse of the matrix containing only the diagonal elements of M . (For constant elements, as are typically used, $\text{diag}(M) = I$, 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 products⁸.

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.

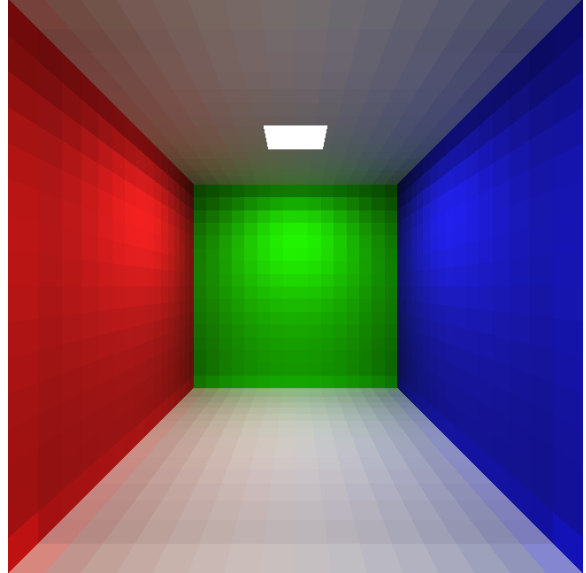


Figure 2: A simple radiosity scene solved and displayed completely on the GPU.

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

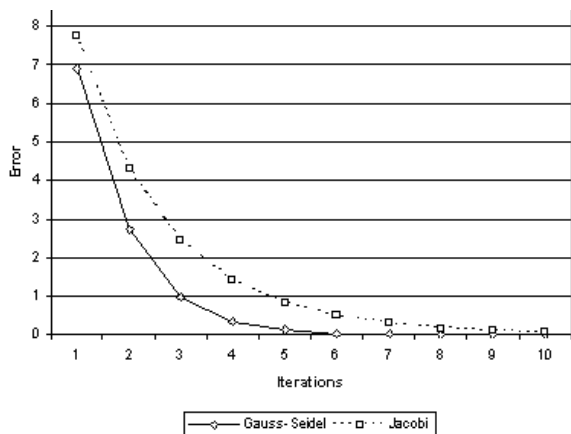


Figure 3: Gauss-Seidel converges faster than Jacobi iteration. Error is measured as the mean squared error of the residual $MB - E$.

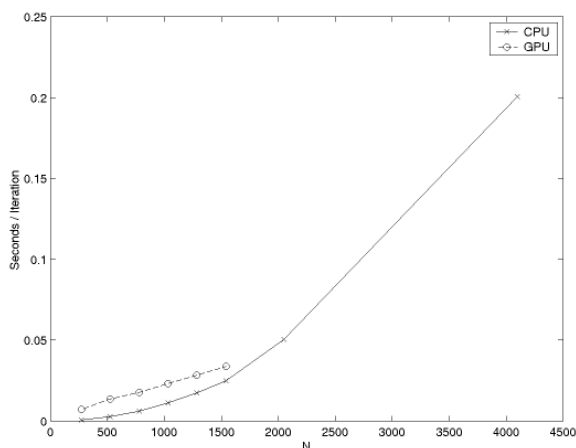


Figure 4: Though the CPU currently outperforms the GPU on this matrix radiosity example, the CPU is memory bandwidth bound and its performance degrades on larger matrices, whereas the GPU is compute bound and its performance scales linearly.

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

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

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 $\vec{\omega}_i$.

Jensen *et al.* noted that single scattering for many common materials only accounts for a small percentage of the outgoing radiance of a material¹². 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

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

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

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

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 Lensch¹⁵. 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 radiosity¹⁰. 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 pre-computed throughput factors and stored into a *scattered irradiance map*.

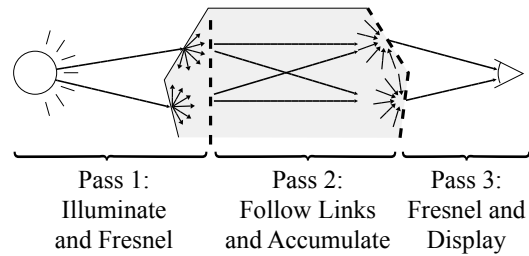


Figure 5: Three passes for rendering subsurface scattering on the GPU.

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.

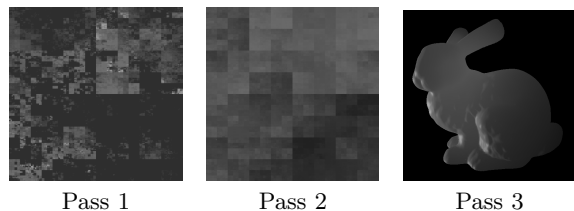


Figure 6: Pass one plots triangles using their texture coordinates (left), interpolating vertex colors set to their direct illumination scaled by a Fresnel transmission term. Pass two transfers these radiances (center) via precomputed scattering links implemented as dependent texture lookups into a MIP-map 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 texturing⁴. 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 hierarchy 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 rgba 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

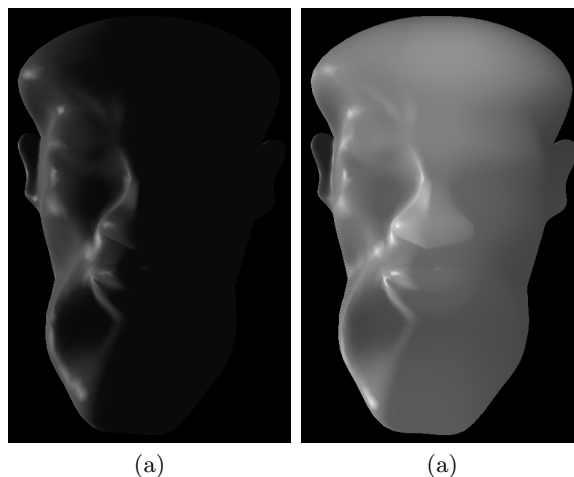


Figure 7: Without subsurface scattering (a), and with subsurface scattering (b) using $\sigma'_s = 2.21$, $\sigma_a = 0.0012$, 16 links, 40fps Nvidia GeForceFX 5800.

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 tessellation complexity.

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 hemisphere: 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).

Clustered Principal Components for Precomputed Radiance Transfer

Peter-Pike Sloan
Microsoft Corporation

Jesse Hall
University of Illinois

John Hart
University of Illinois

John Snyder
Microsoft Research

Abstract

We compress storage and accelerate performance of precomputed radiance transfer (PRT), which captures the way an object shadows, scatters, and reflects light. PRT records over many surface points a transfer matrix. At run-time, this matrix transforms a vector of spherical harmonic coefficients representing distant, low-frequency source lighting into exiting radiance. Per-point transfer matrices form a high-dimensional surface signal that we compress using *clustered principal component analysis* (CPCA), which partitions many samples into fewer clusters each approximating the signal as an affine subspace. CPCA thus reduces the high-dimensional transfer signal to a low-dimensional set of per-point weights on a per-cluster set of representative matrices. Rather than computing a weighted sum of representatives and applying this result to the lighting, we apply the representatives to the lighting per-cluster (on the CPU) and weight these results per-point (on the GPU). Since the output of the matrix is lower-dimensional than the matrix itself, this reduces computation. We also increase the accuracy of encoded radiance functions with a new least-squares optimal projection of spherical harmonics onto the hemisphere. We describe an implementation on graphics hardware that performs real-time rendering of glossy objects with dynamic self-shadowing and interreflection without fixing the view or light as in previous work. Our approach also allows significantly increased lighting frequency when rendering diffuse objects and includes subsurface scattering.

Keywords: Graphics Hardware, Illumination, Monte Carlo Techniques, Rendering, Shadow Algorithms.

1. Introduction

Global illumination effects challenge real-time graphics, especially in area lighting environments that require integration over many light source samples. We seek to illuminate an object from a dynamic, low-frequency lighting environment in real time, including shadowing, interreflection, subsurface scattering, and complex (anisotropic) reflectance.

These effects can be measured as radiance passing through spherical shells about the surface point p in Figure 1. *Source radiance* originates from an infinite sphere (environment map). *Transferred incident radiance* passes through an infinitesimal hemisphere, and equals the source radiance decreased by self-shadowing and increased by interreflection. *Exiting radiance* passes outward through an infinitesimal hemisphere, and results from the BRDF times the transferred incident radiance, plus subsurface scattering.

The spherical harmonic (SH) basis provides a compact, alias-avoiding representation for functions of radiance over a sphere or hemisphere [Cabral *et al.* 1987][Sillion *et al.* 1991][Westin *et al.* 1992][Ramamoorthi and Hanrahan 2001]. Low-frequency source illumination, which small vectors (e.g. $N=25$) of SH coefficients

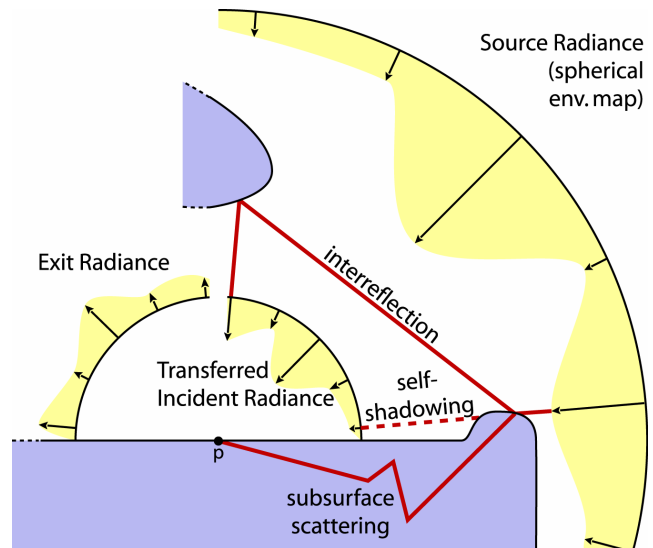


Figure 1: Radiance transfer at p from source to transferred incident to exit.

approximate well [Ramamoorthi and Hanrahan 2001][Sloan *et al.* 2002], is exactly the situation in which integration over the light becomes the bottleneck for traditional rendering methods.

Sloan *et al.* [2002] precompute the radiance transfer of an object in terms of low-order SHs. For a diffuse object, exiting radiance results from dotting a 25-vector, representing the source radiance, with a 25-element radiance transfer vector precomputed and stored at each sample point p . By storing this transfer vector per-vertex, real-time self-shadowing and interreflection results from a simple vertex shader. For a glossy object, [Sloan *et al.* 2002] represents radiance transfer as a linear operator converting a 25D source radiance vector into a 25D transferred radiance vector, via a 625-element transfer matrix that varies for each p . This glossy transfer matrix was too big for graphics hardware. The CPU implementation ran at interactive rates (~ 4 Hz) and could achieve real-time frame rates only for a constant view or lighting, hampering its usefulness for applications like 3D games.

Our method lifts these restrictions, rendering the same glossy objects more than 10-20 times faster. For simpler diffuse transfer, the method allows higher frequency lighting (i.e., higher-order SH projections) for the same computational cost. As in [Lehtinen and Kautz 2003], we precompute and store per-vertex the source-to-exiting radiance transfer matrix, instead of the source-to-incident transfer [Sloan *et al.* 2002]. This also allows us to include the precomputed contribution of the object's subsurface scattering of distant environmental light.

To get real-time performance, we treat the transfer vectors or matrices stored at each vertex as a surface signal and partition them into a few (128-256) clusters. Principal component analysis (PCA) approximates the points in each cluster as a low-dimensional affine subspace (mean plus up to $n'=8$ PCA vectors). We call this approach *clustered principal component analysis* (CPCA). For glossy transfer, CPCA reconstructs a good approximation of its $N \times N$ matrix at each point by storing only the index of its cluster and a few ($n' \ll N \ll N^2$) scalar coordinates of

projection onto the cluster’s PCA vectors. CPCA reduces not only signal storage (n' rather than N^2 scalars per point) but also the runtime computation. Instead of multiplying an $N \times N$ transfer matrix by an N -dimensional light vector at each p , we precompute this multiplication in each cluster for each of its PCA vectors and accumulate weighted versions of the n' resulting N -vectors. CPCA on diffuse transfer provides a similar savings in storage and computation.

We describe two technical contributions which may have wider applicability. The first is a very general signal approximation method using CPCA. Though used before in machine learning applications [Kambhatla and Leen 1994][Kambhatla and Leen 1997][Tipping and Bishop 1999], it is new to computer graphics. To increase spatial coherence, we augment the method by redistributing points to clusters according to an “overdraw” metric. The second contribution is the use of the optimal least-squares projection of the SH basis onto the hemisphere, which significantly reduces error compared to approaches used in the past [Sloan *et al.* 2002][Westin *et al.* 1992].

2. Related Work

Various representations encapsulate precomputed or acquired global illumination. Light fields [Gortler *et al.* 1996][Levoy and Hanrahan 1996] record radiance samples as they pass through a pair of viewing planes whereas surface light fields [Chen *et al.* 2002][Miller *et al.* 1998][Nishino *et al.* 1999][Wood *et al.* 2000] record 4D exiting radiance sampled over an object’s surface. Both techniques support arbitrary views but fix lighting relative to the object.

Precomputed radiance transfer (PRT) [Sloan *et al.* 2002] parameterizes transferred incident radiance in terms of low-frequency source lighting, allowing changes to lighting as well as viewpoint. We build on PRT and its generalization to anisotropic BRDFs [Kautz *et al.* 2002], but speed up performance and reduce error in three ways: we record exiting radiance instead of transferred incident, use least-squares optimal projection of hemispherical functions, and compress using CPCA. We also extend PRT to include subsurface scattering. In parallel work, Lehtinen and Kautz [2003] approximate PRT using PCA. Our CPCA decoding reduces approximation error and maps well to the GPU, resulting in 2-3 times better performance.

Other illumination precomputation methods also support dynamic lighting. Matusik *et al.* [2002] handle limited, non-real-time lighting change with a surface reflectance field measured over a sparsely sampled directional light basis, stored on the visual hull of an acquired object. Hakura *et al.* [2000] support real-time lighting change with parameterized textures, but constrain viewing and lighting changes to a 2D subspace (e.g. a 1D circle of viewpoints \times 1D swing angle of a hanging light source). [Sloan *et al.* 2002] compares PRT to many other precomputed approaches for global illumination.

Precomputed illumination datasets are huge, motivating compression. Light fields were compressed using vector quantization (VQ) and entropy coding [Levoy and Hanrahan 1996], and reflectance fields using block-based PCA [Matusik *et al.* 2002]. Surface light fields have been compressed with the DCT [Miller *et al.* 1998], an eigenbasis (PCA) [Nishino *et al.* 1999], and generalizations of VQ or PCA to irregular sampling patterns [Wood *et al.* 2000]. Our CPCA compression strategy improves [Wood *et al.* 2000] by hybridizing VQ and PCA in a way that reduces error better than either by itself. Unlike [Chen *et al.* 2002] which compresses a 4D surface light field over each 1-ring

mesh neighborhood, our clustering is free to group any number of samples that can be approximated well together regardless of their surface location. Our purpose is real-time rendering with graphics hardware, not minimizing storage space. For example, we avoid entropy coding for which current graphics hardware is ill-suited.

Jensen *et al.* [2002] simulate translucent materials using a diffusion approximation of subsurface scattering accelerated by decoupling the computation of irradiance from a hierarchical evaluation of the diffusion approximation. This paper also experimentally validated when the multiple scattering term dominated. Two recent paper exploit this property and implement interactive rendering techniques based on the idea. Lensch *et al.* [2002] combine spatially varying filters in texture space with vertex-to-vertex transfer to model near and far subsurface transport. Global shadowing and interreflection effects are ignored and only ~ 5 Hz frame rate is obtained. Hao *et al.* [2003] precompute subsurface scattering for a directional light basis. We model smooth, distant lighting environments and include a glossy term to approximate single scattering.

Like PRT, Westin *et al.* [1992] also use matrices which transform lighting into exiting radiance, both expressed in the SH basis. Their matrices encode local effects for BRDF synthesis, not spatially-varying global transport for real-time rendering. They devise a SH projection of hemispherical functions, which we improve via least-squares in the appendix.

Lensch *et al.* [2001] use a similar clustering procedure to reconstruct a spatially-varying BRDF from images. They fit parameters of a BRDF model in each cluster using nonlinear optimization and approximate using a linear combination of the resulting models, one per cluster. We use an independent affine basis per cluster.

3. Radiance Transfer Signal Representation

For diffuse surfaces, PRT encodes a transfer vector, t_p , per surface point p [Sloan *et al.* 2002]. The i -th component of this vector represents the linear contribution of source lighting basis function $y_i(s)$ to the exiting radiance of p . For glossy surfaces, we make several modifications to the glossy transfer matrix defined in [Sloan *et al.* 2002].

3.1 Transferred Incident vs. Exiting Radiance Transfer

PRT in [Sloan *et al.* 2002] represents *transferred incident* radiance (Figure 1). It is derived from a Monte Carlo simulation illuminating geometry by the SH basis functions. This decouples the way an object shadows itself from its reflectance properties, and allows different BRDFs to be substituted at run-time. Here we seek to approximate the transfer signal to reduce computation. To measure approximation error properly, we must know the BRDF. For example, a smooth BRDF weights low-frequency components of transferred radiance more than high-frequency components.

To measure signal errors properly, we include BRDF scaling by encoding the *exiting radiance* transfer matrix at p , M_p . Its component, $M_{p,ij}$, represents the linear influence of source lighting basis function j to exiting radiance basis function i . It can be numerically integrated over light directions s and view directions v over the hemisphere $\mathbf{H} = \{(x, y, z) \mid z \geq 0 \text{ and } x^2 + y^2 + z^2 = 1\}$ via

$$M_{p,ij} = \int_{v \in \mathbf{H}} \int_{s \in \mathbf{H}} y_i(v) T_p(s, y_j(s)) B(v, s) s_z ds dv$$

where T_p represents transport effects like shadowing, B is the BRDF, y are the SH basis functions, and s_z is the “cosine” factor (z component of s). For simple shadowing, $T_p = y_j(s) q_p(s)$ where $q_p(s)$ is 0 if the object occludes itself in direction s and 1 other-

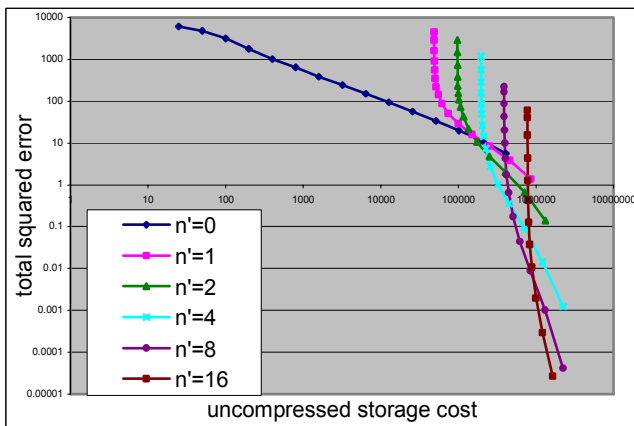


Figure 2: CPCA error analysis using static PCA. Each curve represents how squared error varies with various numbers of clusters (1, 2, 4, ..., 16k) using a given number of principal components in each cluster ($n' = 0, 1, 2, 4, 8,$ and 16). The signal approximated was a 25D shadowed diffuse transfer vector over a bird statue model from [Sloan *et al.* 2002] having 48668 sample points. 20 VQ iterations were used, followed by PCA in each cluster.

wise. For general transport where lighting is specified in a global frame, $M_p = B R_p T_p$ where T_p is the glossy transfer matrix defined in [Sloan *et al.* 2002], R_p is an SH rotation aligning p 's normal to the z axis and its tangents to x and y , and B is the BRDF matrix

$$B_{ij} = \int_{v \in H} \int_{s \in H} y_i(v) y_j(s) B(v, s) s_z ds dv$$

R_p is a $N \times N$ rotation matrix; its computation is outlined in [Kautz *et al.* 2002].

We also add a view-independent subsurface scattering term to the transport, precomputed using the hierarchical diffusion approximation of [Jensen and Buhler 2002] but parameterized by the SH basis for lighting. The result affects only the y_0 (constant) basis function of exiting radiance.

3.2 Representing Radiance over the Hemisphere

Exit and transferred radiance at a surface point are actually functions over a hemisphere, not a sphere. For the SH basis, there is complete freedom in evaluating the function on the “opposite” hemisphere when projecting it to the SH basis. Transfer in [Sloan *et al.* 2002] and the formulas above in Section 3.1 implicitly zero the opposite hemisphere by integrating only over the hemisphere. Westin *et al.* [1992] used a reflection technique. It is also possible to use other bases such as Zernike polynomials lifted to the hemisphere [Koenderink *et al.* 1996].

Our approach uses the least-squares optimal projection of the SH basis onto the hemisphere described in the Appendix. The technique represents any SH-bandlimited spherical function restricted to the hemisphere without error. In contrast, zero-hemisphere projection incurs 35% worst-case and 20% average-case RMS error integrated over the hemisphere for all unit-power spherical signals formed by linear combinations of the 5th order SH basis. The odd reflection technique [Westin *et al.* 1992] is even worse. Beyond theoretical results, we also see visibly better accuracy on our experimental objects using optimal projection (see Figure 7).

Given a vector b which projects a hemispherical function into the SH basis by zeroing out the opposite hemisphere, the optimal hemispherical projection is simply $A^{-1} b$ where A is defined in the appendix. Therefore, the optimally projected exiting radiance transfer matrix is given by

$$M_p = A^{-1} B A^{-1} R_p T_p \quad (1)$$

projecting first transferred radiance, $R_p T_p$, and then exiting radiance. Figure 7 compares results with and without this least-squares “boost” by A^{-1} to reduce error in transferred and exiting radiance.

3.3 Clustered PCA (CPCA) Approximation

We have an n -dimensional signal x_p sampled at points p over a surface. Each x_p represents exiting radiance as a linear operator on a light vector, and takes the form of vectors for diffuse surfaces (e.g., $n=N=25$) or matrices for glossy surfaces (e.g., $n=N^2=625$). To approximate this signal, we partition its samples into a number of *clusters* each of which is approximated by an affine subspace. More precisely, the points in a cluster are approximated by

$$x_p \approx \tilde{x}_p = x_0 + w_p^1 x_1 + w_p^2 x_2 + \dots + w_p^{n'} x_{n'}$$

where the $n'+1$ n -vectors $x_0, x_1, \dots, x_{n'}$ are constant over the cluster and the n' scalar weights $w_p^1, w_p^2, \dots, w_p^{n'}$ vary for each point p on the surface. To reduce signal dimensionality, $n' \ll n$. The vector x_0 is called the *cluster mean*, and the vectors $x_i, i \geq 1$ are called the *cluster PCA vectors*. Together, the cluster's mean and PCA vectors are called its *representative vectors*.

CPCA (called “VQPCA” in [Kambhatla and Leen 1994] [Kambhatla and Leen 1997] and “local PCA” or “piecewise PCA” in the machine learning literature under the general title of “mixtures of linear subspaces”) generalizes PCA (single cluster, $n' > 0$) and VQ (many clusters, $n' = 0$). VQ approximates a signal as a piecewise constant while PCA assumes it is globally linear. CPCA exploits the *local linearity* of our radiance transfer signal by breaking it down into clusters, approximating each with a separate affine subspace.

4. Compressing Surface Signals with CPCA

We review CPCA, beginning with the simplest approach and then describing several enhancements that further reduce error.

4.1 VQ Followed by Static PCA

The simplest CPCA method is to first cluster the points using VQ, and then compute a PCA fit in each of the resulting clusters [Kambhatla and Leen 1994].

VQ Clustering The LBG algorithm [Linde *et al.* 1980] performs the initial clustering. Given a desired number of clusters, the algorithm starts with clusters generated by random points from the signal and then classifies each point into the cluster having minimum distance to its representative. Each cluster representative is then updated to the mean of all its points, and the algorithm iterated until no points are switched or an iteration count is reached.

Per-Cluster PCA We first compute the cluster mean, x_0 . We then compute a $m_k \times n$ matrix of residuals after subtracting the mean, $C = [x_{p1} - x_0, x_{p2} - x_0, \dots, x_{pnk} - x_0]^T$, where m_k is the number of points in cluster k . Computing an SVD yields $C = U D V^T$ where U and V^T are rotation matrices and D is a diagonal matrix whose elements are sorted in decreasing order. The first n' rows of V (columns of V^T) are the cluster PCA vectors. A point p_j 's projection weights (n' -vector w_{p_j}) are given by the first n' columns of row j of UD (they are also given simply by the dot product of $x_{p_j} - x_0$ with each of the PCA vectors). This provides a least-squares optimal linear approximation of C from combinations of n' fixed vectors. Total squared error over all cluster points is given by

$$\sum_{j=1}^{m_k} \|x_{p_j} - \tilde{x}_{p_j}\|^2 = \sum_{i=n'+1}^n D_i^2 = \sum_{j=1}^{m_k} \|x_{p_j} - x_0\|^2 - \sum_{i=1}^{n'} D_i^2$$

The SVD of C can be directly computed using the LAPACK routine `dgesvd`. To reduce matrix size and so computation, we instead convert to normal form. When $m_k \geq n$, we compute the

$n \times n$ matrix $C^T C$ and its eigenvalues (which are the squares of C 's singular values) and eigenvectors (which are equal to C 's right singular vectors V^T and thus the needed PCA vectors). When $m_k < n$, we compute the $m_k \times m_k$ matrix CC^T . Its eigenvalues are still the squares of C 's singular values, but its eigenvectors are C 's left singular vectors, U , from which the right can be derived via $V^T = U^T D^{-1} C$. The LAPACK routine `dsyevx` computes eigenpairs of symmetric matrices like $C^T C$ and CC^T , and saves computation because it can return just the n' eigenpairs having the largest eigenvalues, while `dgesvd` returns *all* singular values.

Experimental Results Figure 2 shows results for this approach on an example diffuse transfer signal (25D) over a bird statue model. Using straight VQ ($n'=0$), errors are only gradually reduced as storage increases. Increasing the number of PCA vectors per cluster provides an approximation that is worse at small numbers of clusters but eventually crosses below the previous curve as the number of clusters increases.

The graphs use a simple cost metric measuring total storage for the approximated signal:

$$m_p n' + m_c (n' + 1) n$$

where m_p is the number of surface samples and m_c is the number of clusters. The first term represents the per-point weight data whereas the second represents the per-cluster representative data. This simple model correlates well with actual rendering cost.

4.2 Iterative PCA

The previous section clusters using distance to the cluster mean $\|x_p - x_0\|^2$ as the classification metric, but as observed in [Kambhatla and Leen 1997], the distance that matters is approximation error, $\|x_p - \tilde{x}_p\|^2$. Iterative PCA [Kambhatla and Leen 1997] exploits this observation by classifying points in the cluster that minimizes approximation error rather than distance to the mean. Also, after every point classification step (instead of only once at the end of the whole clustering process) it computes a PCA of the cluster's current point members to update the affine subspace model.

This approximation error at a point x_p is computed via

$$\|x_p - \tilde{x}_p\|^2 = \|x_p - x_0\|^2 - \sum_{i=1}^{n'} ((x_p - x_0) \cdot x_i)^2.$$

To avoid local minima having high error, we introduce additional PCA vectors one by one, from zero to n' , and do several iterations (typically 10-20) of the generalized LBG algorithm for that number of vectors before adding another.

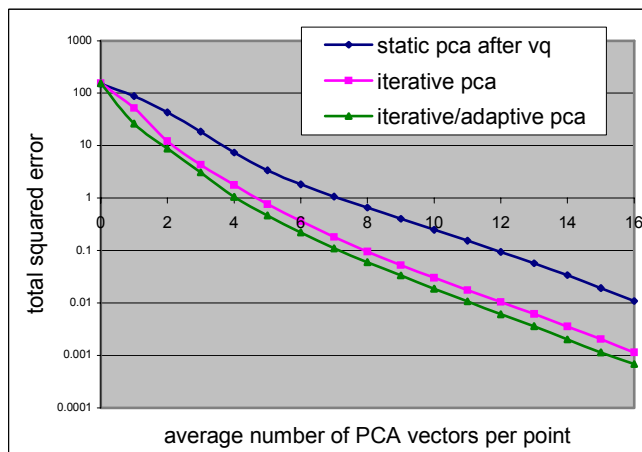


Figure 3: Comparison of error for three CPCA encoding methods. As in Figure 2, the signal encoded is 25D diffuse transfer over a bird model. 256 clusters were used.

Figure 3 and Figure 4 demonstrate the large error reduction from iterative over static PCA. Typically, iterative PCA performs as well as static having 1-4 additional PCA vectors per cluster, but the encoding cost is significantly higher.

4.3 Per-Cluster Adaptation of Number of PCA Vectors

Neither static nor iterative CPCA distribute error homogeneously – some clusters usually have much more error than others. Without increasing the overall amount of per-point data, we can reduce error by allowing clusters with high error to use more PCA vectors and clusters with less error to use fewer. Adaptation like this was used in [Meinicke and Ritter 2001] to avoid local overfitting.

The squared singular value D_i^2 in a cluster represents how much total squared error is reduced by the addition of PCA vector i to that cluster. But clusters do not contain the same number of points; adding an additional PCA vector in a cluster with more points is more expensive than in a cluster with fewer points because an additional weight must be stored per point. So we rank PCA vectors by D_i^2/m_k which represents the rate at which per-point squared error will be reduced by the addition of PCA vector i in cluster k containing m_k points. We sort this quantity in decreasing order over all PCA vectors in all clusters, and add PCA vectors according to this order (greatest error-reduction rate first), until it reaches its total budget of PCA vectors.

The overall algorithm starts with the CPCA result from the previous section (constant number of PCA vectors per cluster). Additional adaptation iterations then perform the following steps:

- 1) classify each point to the cluster having minimum approximation error, using the cluster's current n' ,
- 2) update cluster representatives using PCA (see Section 4.1),
- 3) redistribute the number of PCA vectors over all clusters by sorting over D_i^2/m_k and adding vectors (vector i from cluster k) in decreasing order until $\sum m_k$ reaches its budget. Record the number of PCA vectors allocated in each cluster.

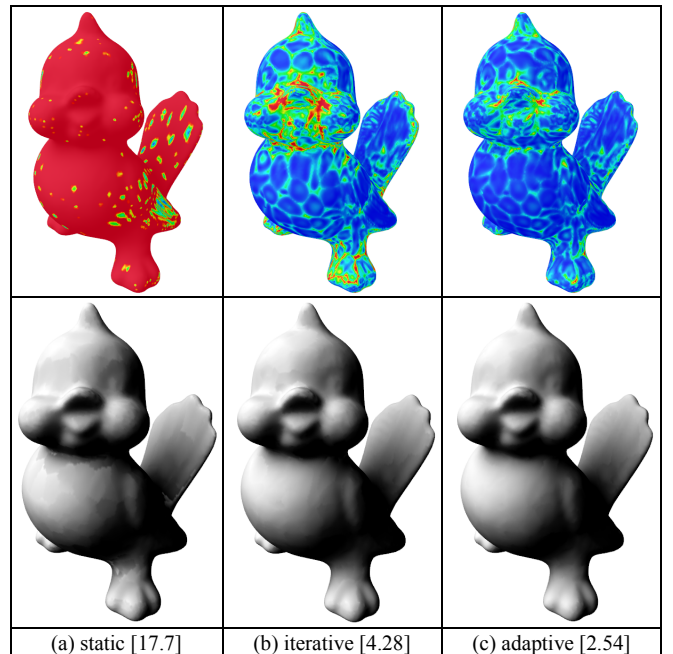
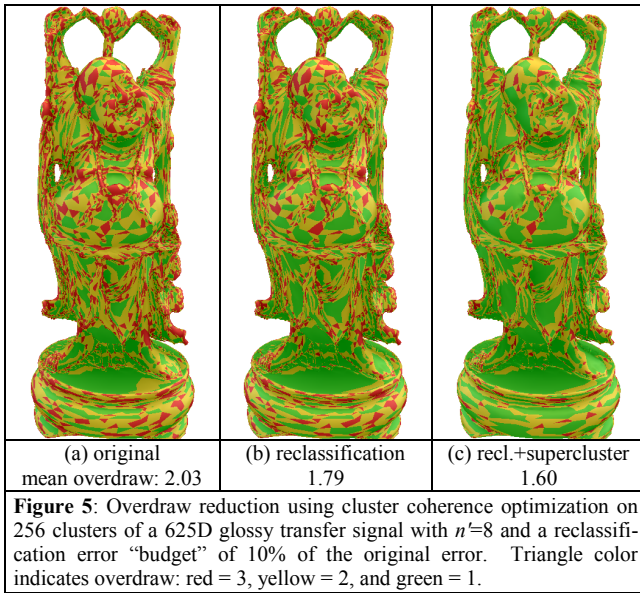


Figure 4: Per-point error distribution for three CPCA methods. A linear blue-cyan-green-yellow-red error scale is used. Rendered images are shown in the second row. The signal is that for Figure 3 with $n=3$. Total squared error of the 25D signal over all 48k vertices is written in brackets.



As shown in Figure 3 and Figure 4, adaptation reduces error, typically producing error as low as non-adaptive PCA with an additional vector.

5. Cluster Coherence Optimization

The clusters from the previous section ignore where samples lie on the object’s surface – clusters can have ragged boundaries or contain multiple components. This leads to rendering inefficiency because triangles whose vertices are in different clusters are drawn multiple times. For each triangle, this *overdraw* is defined as the number of unique clusters its vertices belong to. Overdraw summed over all mesh triangles represents the number of triangles sent to the graphics hardware for rendering (see details in Section 6). We reduce overdraw with two techniques.

The first technique seeks for each vertex a better cluster that reduces overdraw without significantly increasing approximation error. This greedy search tries reclassifying the vertex’s signal in its neighboring vertices’ clusters, and computes the resulting overdraw reduction and error increase. The technique then sorts all vertices by overdraw reduction divided by error increase, and reclusters each vertex in decreasing order of this quotient until reaching a given error increase budget, such as 5-10% of the initial error. Vertex reclassification requires recomputation of the quotient for the vertex and its neighbors. Figure 5(b) shows reclassification results.

The second technique, called *superclustering*, allows the graphics hardware to draw a group of clusters as a single unit. It reduces overdraw because triangles straddling clusters from the same supercluster need not be drawn more than once (see Section 6). Superclustering also ensures that primitive batches are large enough to maximize performance; the number of clusters in a supercluster is limited by the number of registers available in the graphics hardware. Unlike reclassification, superclustering does not increase approximation error.

We form superclusters greedily, initializing them to be the clusters, then repeatedly merging neighboring superclusters in order of overdraw reduction. Two superclusters neighbor each other when at least one triangle has vertices from both. Figure 5(c) demonstrates how well greedy superclustering reduces overdraw.

6. Rendering Using CPCA-Encoded Transfer

To shade a glossy surface at point p using CPCA-encoded transfer, we use a modified version of [Kautz *et al.* 2002], via

$$y^T(v_p) (B R_p T_p) l = y^T(v_p) (M_p) l$$

Here, the column-vector l results of projecting source lighting (in a global coordinate frame) into the SH basis. The matrix T_p converts this source lighting to transferred incident radiance (accounts for self-shadowing and inter-reflection). The rotation matrix R_p aligns the global coordinate system to a local frame defined by p ’s normal and tangent directions. The BRDF matrix B converts local incident radiance into exit. Finally, y is a column-vector (y^T is a row-vector) of SH basis functions evaluated at the view direction at p , v_p , expressed in the local frame. y and l are N -vectors and B , R , and T are $N \times N$ matrices. A fifth-order SH projection, $N=25$, is accurate when the lighting and BRDF are low-frequency.

One can compute the source lighting vector l in various ways [Sloan *et al.* 2002]. We can dynamically rotate a predefined environment to simulate rigid rotations of the object. Graphics hardware can sample radiance near the object which is then SH-projected. Simple light sources like circles can be projected analytically. Spatial variation in l captures local lighting effects but complicates the rendering process.

The approach in [Kautz *et al.* 2002] recorded the spatially varying signal $T'_p = R_p T_p$ and evaluated the matrix-vector product $f_p = T'_p l$ on the CPU. It then evaluated $B'(v_p) = y(v_p) B$ using N texture maps indexed by the view vector v_p , and finally computed a dot product of these two vectors. Texture maps B' in [Kautz *et al.* 2002] were evaluated per-vertex on the CPU because the hardware was unable to interpolate 25D vectors f_p over a triangle nor perform the 25D dot product in a pixel shader. Though the latest graphics hardware now makes it possible to interpolate such large vectors, transfer matrices remain too large to be manipulated on the GPU. Fortunately, the affine approximation used by CPCA solves this problem.

Using CPCA, we encode the entire matrix chain M_p converting source lighting to exiting radiance. This produces the approximation

$$\tilde{M}_p = M_0 + w_p^1 M_1 + w_p^2 M_2 \cdots + w_p^{n'} M_{n'}$$

Multiplying \tilde{M}_p by l then yields exiting radiance projected into the SH basis, e_p , via

$$e_p = M_p l = (M_0 l) + w_p^1 (M_1 l) + w_p^2 (M_2 l) + \cdots + w_p^{n'} (M_{n'} l)$$

We recompute the matrix/vector products for each cluster on the CPU, resulting in $n'+1$ fixed N -vectors, and accumulate them as a sum scaled by the per-point weights, w_p^j , on the GPU. For small $n' < N$, this reduces computation and makes the vertex data small enough for vertex shaders on current graphics cards. For example, for $N=25$ and $n'=5$, we save more than a factor of 4.

Finally, we evaluate the exiting radiance function at v_p by dotting the vector $y(v_p)$ with e_p . We evaluate y at v_p using a texture map in the same way as [Kautz *et al.* 2002] evaluated $y^T(v_p) B$, but we can now perform this evaluation and dot product in a pixel shader.

Diffuse surfaces simplify the computation but CPCA achieves a similar reduction. In this case, $t_p \cdot l$ computes shading where t_p is an N -dimensional transfer vector and l is the lighting’s SH projection as before. Using CPCA, we encode t_p as an affine combination of per-cluster representatives and precompute in each cluster the dot product of the light with these vectors. The final shade is a weighted combination of $n'+1$ scalar values $t_i \cdot l$ which are constant over a cluster, via

$$\tilde{t}_p \cdot l = (t_0 \cdot l) + w_p^1 (t_1 \cdot l) + w_p^2 (t_2 \cdot l) + \dots + w_p^{n'} (t_{n'} \cdot l)$$

This saves computation when $n' < N$. In fact, the per-vertex computation does not depend on N at all! So we can use higher-order projections (e.g., $N=36$ up to $N=100$) as long as the approximation error remains acceptable for small n' (Figure 6). Unlike [Sloan *et al.* 2002], real-time rendering is now possible with such high-order lighting, since the transfer vector is no longer stored on the GPU.

6.1 Non-Square Matrices

M_p need not be square. In an $N_r \times N_i$ matrix, more columns, N_i , provide for greater lighting frequency and thus longer, sharper shadows. More rows, N_r , provide for more specular BRDFs. Interestingly, N_i has little effect on the run-time cost with CPCA, since the transformation of source lighting is done per-cluster to produce vectors whose dimensionality only depends on N_r . Increasing N_i does increase the entropy of the transfer signal, making it harder to encode and likely to require more representatives per cluster.

Non-square transfer matrices are useful in another way. Exiting radiance is a hemispherical function, so we can use the optimal least-squares projection derived in the Appendix to represent M_p . Fifth order optimal projection of the output of M_p can be done with little error using $N_i=24$ basis functions – one of the 25 bases is nearly redundant (see Appendix).

6.2 Implementation Notes

We first describe the simple case of no superclustering. We decompose the mesh into chunks of geometry for each cluster, where a chunk contains all faces containing at least one vertex from that cluster. Since this also includes vertices from other clusters, we store a per-vertex bit, α_p , indicating whether the vertex p is a cluster member. Pseudocode for rendering is

```

Draw the mesh into the zbuffer only ( $rgb=0$ )
Set the blending mode to add
Foreach cluster
  Compute  $n'+1$  per-cluster constants ( $M_i l$  or  $t_i \cdot l$ ) on CPU
  Load per-cluster constants to graphics hardware
DrawCluster

```

DrawCluster sends the cluster’s geometry to the GPU and runs a vertex shader computing the linear combination of the w_p^i with the per-cluster constants. If $\alpha_p = 0$, the w_p^i ’s are also set to zero so that blending vertices from other clusters does not effect the result. In other words, we blend using a linear partition of unity

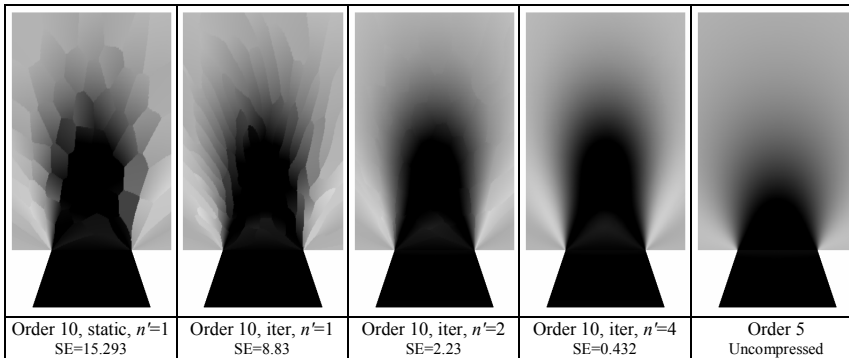


Figure 6: Higher-order lighting for diffuse transfer (simple two-polygon scene). The left four columns show CPCA-encoded results for 10th order lighting ($N=100$) using various numbers of representatives (n') and $m_c=64$. The rightmost column shows uncompressed 5th order lighting ($N=25$) used in [Sloan *et al.* 2002]. Note how shadows are sharpened at higher order and how iterative PCA adapts cluster shapes to the transfer signal better than static PCA (leftmost two columns). CPCA with $n'=4$ provides an accurate approximation that can be rendered in real-time.

over each triangle face that straddles multiple clusters.

Generalizing to superclusters is not much more complicated. We compute the per-cluster constants for all clusters in the supercluster and load them into hardware registers. Every vertex in a supercluster records a cluster index, used by the vertex shader as an index register to look up its cluster’s constants.

For diffuse transfer, the vertex shader produces the final shaded result. Glossy transfer is more complex – its vertex shader requires normal and tangent vectors to transform the global view vector into the local frame at p to obtain v_p . Rasterization interpolates the resulting view vector v_p and exiting radiance vector e_p over the pixels of each triangle. The pixel shader uses the local view vector to index a map of SH basis functions, $y(v_p)$, and then dots this result with e_p . We use a parabolic hemispherical parameterization [Heidrich and Seidel 1999] for the SH map, sampled at 32×32 . Since e_p contains more components than current rasterization hardware can interpolate, we perform three separate passes for glossy transfer – one per color channel.

Diffuse and glossy transfer also differ in their per-cluster state. For each of the $n'+1$ representative vectors, the per-cluster constant is a scalar color, $t_i \cdot l$, for diffuse transfer regardless of the value of N_i . For glossy transfer, this state is a colored N_r -vector, $M_i l$. Current graphics hardware (ATI 9700, Nvidia GeForce 4) supports ~256 registers accessible by vertex shaders where each register contains 4 channels. For nonadaptive PCA, glossy transfer requires $m_s (n' + 1) N_r/4$ registers where m_s is the number of clusters per supercluster. This assumes three pass rendering, one per color channel, and packs 4 components of an N_r -vector into each register. Diffuse transfer requires less state: only $m_s (n' + 1)$ registers per supercluster to compute all three color channels by packing an rgb color per register.

Though the programmable resources of GPUs have increased greatly, they are not yet sufficient to feasibly render adaptive PCA (Section 4.3), which requires data-dependent looping.

7. Results

Figure 10 compares rendering quality of various transfer encodings on an example bird model with a glossy anisotropic BRDF. We experimentally picked a number of clusters for VQ ($n'=0$) and a number of representative vectors for pure PCA ($m_c=1$) such that rendering performance matched that from CPCA with $n'=8$, $m_c=256$. For CPCA, we used iterative PCA encoding from Section 4.2. We applied superclustering (Section 4.3) to both VQ and CPCA to the extent permitted by hardware register limits (it is unnecessary for pure PCA since there is only one cluster). Example images, encoding error, and rendering rates appear in the figure for all three methods as well as the uncompressed original. Methods used before in computer graphics [Nishino *et al.* 1999][Wood *et al.* 2000] perform poorly: pure PCA is smooth but has high error; VQ reduces error but has obvious cluster artifacts. Our CPCA result (third column) is very faithful to the uncompressed image on the far right.

Figure 11 shows the effect on encoding accuracy of varying the per-cluster number of representative vectors (n'). The two rows show results on two models, one smooth (bird, bottom) and one bumpier (Buddha, top). Each column corresponds to a different n' . The signal encoded represents glossy transfer for an anisotropic

BRDF, including multiple bounce interreflections for the Buddha model, but only shadowing for the bird model. With about 8 cluster PCA vectors, we obtain very accurate results that can be rendered quickly. Rendering results using un-

Model	original	Rec	sc	sc+rec
Buddha	1.93/26.8	1.72/29.6	1.61/29.7	1.48/33.2
Bird	1.25/39.3	1.23/39.3	1.14/45.2	1.138/45.3

compressed transfer data (without CPCA encoding) is shown in the far right column. CPCA speeds up rendering by more than a factor of 10 compared to uncompressed rendering [Sloan *et al.* 2002] with little visual loss.

Interestingly, though the Buddha model has higher error per transfer sample than the bird model, error is masked by its high-frequency spatial variation. The Buddha’s $n=4$ result looks quite accurate, whereas the bird’s has cluster artifacts visible in the shadowed neck area. Error on the Buddha reveals itself in the neck/chin shadow and the pedestal shadow between the feet.

Figure 9 and Figure 8 show the quality of real-time rendering results achieved by our method. The transfer signal for Figure 9 represents the sum of a diffuse subsurface scattering term and a isotropic glossy term. The result is a realistically rendered Buddha that includes shadowing, translucency, and glossiness effects that respond to changes in lighting or view in real-time.

Figure 8 includes models from [Sloan *et al.* 2002] which could be rendered quickly only by fixing the light with respect to the object (allowing view change), or fixing the view (allowing light movement). We now render these models with visually identical quality in real-time without constraints. For comparison, uncompressed rendering using 25×25 matrices gives a frame rate of 2.9Hz for the head model, and 2.7Hz for the buddha model, a factor of $20 \times$ and $16 \times$ slower than rendering with CPCA-encoded 16×25 matrices. (For 16×25 matrices, the uncompressed rendering speeds are 5.2Hz and 4.7Hz.) This comparison is fair because 16×25 matrices with least squares optimal projection (Equation (1)) produce results almost indistinguishable from 25×25 matrices with the zero-hemisphere projection (see Figure 7). A Radeon 9800 runs 20% faster with CPCA while uncompressed rendering is 1% faster, showing that CPCA scales well with the GPU.

Table 1 compares encoding results, highlighting the preprocessing times and error values for static PCA (Section 4.1) vs. iterative PCA (Section 4.2). Iterative encoding is expensive, but it often reduces error significantly (see rows for bird model, for example). For the Buddha model, transfer signals tend to be more spatially incoherent, so error reduction from iterative encoding is less dramatic. Using more clusters (increasing m_c) could help matters, but we have done little experimentation with this parameter.

We also measured the effectiveness of cluster coherence optimization (Section 5). Using a 5% error threshold, which has little effect on visual quality, this table shows overdraw/frame rate (in Hz) using reclassification alone (“rec”), superclustering alone (“sc”), and both together (“sc+rec”). Results are for anisotropic glossy transfer (“gloss-anis” from Table 1). We achieve a 15-20% increase in rendering speed on these examples.

8. Conclusions and Future Work

We have shown that CPCA-encoded transfer provides real-time rendering of global transport effects for a variety of geometric models and material characteristics, including glossy/anisotropic BRDFs and translucency. Though they depend on prerecorded transfer data over specific models, these effects are new to real-

model	BRDF	figure	m_p	Transfer $N_f \times N_f$	m_c	n'	Static PCA		Iter PCA		Fps
							cpu	SE	cpu	SE	
Buddha	gloss-iso	11b	49990	16×25	256	6	3m30s	563	1h51m	451	42.8
Buddha	gloss-anis	10	49990	24×25	256	8	6m7s	10996	4h34m	8524	24.2
Buddha	subsurf+gloss-iso	9	49990	25×25	256	8	6m21s	1992	4h32m	1439	27
bird	gloss-anis	8,10	48688	24×25	256	8	6m34s	898	3h43m	294	45
bird	diffuse	video	48688	1×100	256	8	43s	3.14	3m26s	0.668	227
head	gloss-iso	11a	50060	16×25	256	6	4m20s	78.8	2h12m	43.7	58.5
polys	diffuse	6	58624	1×100	32	4	14s	0.492	3m26s	0.432	294

Table 1: Encoding/performance results. SE means squared approximation error over all vertices. Fps is the rendering performance in Hz. All performance measurements were taken from a 2.2Gz Pentium IV with ATI Radeon 9700.

time graphics. CPCA is an effective and very general technique for approximating high-dimensional signals (e.g., transfer matrices) over low-dimensional manifolds (e.g., 3D surfaces). It reduces error better than VQ or PCA for the same storage and yields data granularity in the approximation that better suits GPU implementation. Rather than grouping arbitrarily based on blocks in an image or polygons on a mesh, CPCA adapts cluster size and shape to the nature of the signal. Since the transfer signal is a linear operator on a light vector, representing a cluster containing many samples as a low-dimensional affine subspace not only reduces storage but converts a matrix/vector multiply per point into a weighted combination of a few pre-computed vectors. This is the key to our real-time performance.

In future work, we are interested in using CPCA compression to precompute transfer on deformable models, perhaps by constraining the number of degrees of freedom. We also believe CPCA can be used for surface signals other than radiance transfer of distant source lighting, including simpler signals like surface light fields and more complex ones like transfer for spatially varying illumination. CPCA could be enhanced by an automatic search over the number of clusters variable (m_c), at the cost of additional encoding time. Finally, we are interested in combining our transfer technique, which is specialized for low-frequency lighting, with others handling high-frequency lighting.

Acknowledgments

This research was supported by Microsoft, with additional support from ATI, NVidia, and the NSF ITR grant ACI-0113968. We thank David Thiel for editing the submission video.

References

CABRAL, B, MAX, N, AND SPRINGMEYER, R. 1987. Bidirectional Reflection Functions from Surface Bump Maps, SIGGRAPH 87, 273-281.

CHEN, W-C, BOUGUET, Y-V, CHU, MH, AND GRZESZCZUK, R. 2002. Light Field Mapping: Efficient Representation and Hardware Rendering of Surface Light Fields, SIGGRAPH 2002, 447-456.

GERSHO, A, AND GRAY, R. 1992. Vector Quantization and Signal Compression, Kluwer Academic, Boston, pp. 606-610.

GORTLER, SJ, GRZESZCZUK, R, SZELISKI, R, AND COHEN, M.F. 1996. The Lumigraph, SIGGRAPH 96, 43-54.

HAKURA, Z, LENGUEL, J, AND SNYDER, J. 2000. Parameterized Animation Compression. Eurographics Rendering Workshop, pp.101-112.

HAO, X, BABY, T, VARSHNEY, A. 2003. Interactive Subsurface Scattering for Translucent Meshes, to appear in Symposium on Interactive 3D Graphics.

HEIDRICH, W, SEIDEL H. 1999. Realistic, Hardware-Accelerated Shading and Lighting, SIGGRAPH 99, 171-178.

JENSEN, H, AND BUHLER, J. 2002. A Rapid Hierarchical Rendering Technique for Translucent Material, SIGGRAPH 2002, 576-581.

KAMBHATLA, N, AND LEEN, T. 1994 Fast Non-Linear Dimension Reduction, Advances in Neural Information Processing Systems 6.

KAMBHATLA, N, AND LEEN, T. 1997. Dimension Reduction by Local PCA, Neural Computation, 9, 1493.

KAUTZ, J, SLOAN, P, AND SNYDER J. 2002. Fast, Arbitrary BRDF Shading for Low-Frequency Lighting Using Spherical Harmonics, Eurographics Workshop on Rendering, 291-296.

KOENDERINK, J, VAN DOORN, A, AND STAVRIDIS, M. 1996. Bidirectional Reflection Distribution Function Expressed in terms of surface scattering modes, ECCV.

LEHTINEN, J, AND KAUTZ, J. 2003. Matrix Radiance Transfer, to appear in Symposium on Interactive 3D Graphics.

LENSCH, H, KAUTZ, J, GOESELE, M, HEIDRICH, W, AND SEIDEL, H. 2001. Image-Based Reconstruction of Spatially Varying Materials, Proceedings of Eurographics Rendering Workshop, 104-115.

LENSCH, H, GOESELE, M, BEKAERT, P, KAUTZ, J, MAGNOR, M, LANG, J, SEIDEL, H. 2002. Interactive Rendering of Translucent Objects, Pacific Graphics.

LEVOY, M, AND HANRAHAN, P. 1996. Light Field Rendering, SIGGRAPH 96, August 1996, 31-41

LINDE, Y, BUZO, A, AND GRAY, R. 1980. An algorithm for Vector Quantizer Design, IEEE Transactions on Communication COM-28, 84-95.

MATUSIK, W, PFISTER, H, NGAN, A, BEARDSLEY, P, ZIEGLER, R, AND McMILLAN L. 2002. Image-Based 3D Photography using Opacity Hulls. SIGGRAPH 02, 427-437.

MEINICKE, P, AND RITTER, H. 2001. Resolution-Based Complexity Control for Gaussian Mixture Models, Neural Computation, 13(2), 453-475.

MILLER, G, RUBIN, S, AND PONCELEN, D. 1998. Lazy Decompression of Surface Light Fields for Pre-computed Global Illumination, In 9th Eurographics Rendering Workshop, June, pp. 281-292.

NISHINO, K, SATO, Y, AND IKEUCHI, K. 1999. Eigen-Texture Method: Appearance Compression based on 3D Model, Proceedings of 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. Fort Collins, CO, June, pp. 618-24 Vol. 1.

RAMAMOORTHI, R, AND HANRAHAN, P. 2001. An Efficient Representation for Irradiance Environment Maps, SIGGRAPH 2001, 497-500.

SILLION, F, ARVO, J, WESTIN, S, AND GREENBERG, D. 1991. A Global Illumination Solution for General Reflectance Distributions, SIGGRAPH 91, 187-196.

SLOAN, P, KAUTZ, J, AND SNYDER J. 2002. Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments, SIGGRAPH 2002, 527-536.

TIPPING, M, AND BISHOP, C. 1999. Mixtures of Probabilistic Principal Component Analysers, Neural Computation, 11(2), 443-482.

WESTIN, S, ARVO, J, TORRANCE, K. 1992. Predicting Reflectance Functions from Complex Surfaces, SIGGRAPH 92, 255-264.

WOOD, D, AZUMA, D, ALDINGER, K, CURLESS, B, DUCHAMP, T, SALESIN, D, AND STUETZLE, W. 2000. Surface Light Fields for 3D Photography, SIGGRAPH 2000, 287-296.

9. Appendix: Hemispherical SH Projection

9.1 Least-Squares Optimal Projection

Let $f(s)$ be a function over the hemisphere $s=(x,y,z)$, $s \in H$. We approximate f as a linear combination of SH basis functions $y_i(s)$ restricted to H where these basis functions are no longer orthogonal. So we seek

$$f(s) \approx \sum_i c_i y_i(s)$$

such that this approximation has minimum squared error over H . We call this vector c the *least-squares optimal hemispherical projection* of f .

To derive the coefficients c_i of this projection, we minimize squared error

$$E = \int_H (f(s) - \sum_i c_i y_i(s))^2 ds$$

This is an unconstrained minimization problem with the c_i forming the degrees of freedom. So we take $\partial E / \partial c_k$ and set it to 0:

$$\partial E / \partial c_k = \int_H 2 (f(s) - \sum_i c_i y_i(s)) y_k(s) ds = 0$$

$$\Rightarrow \sum_i c_i \int_H y_i(s) y_k(s) ds = \int_H f(s) y_k(s) ds$$

This reduces to $Ac=b$ or $c=A^{-1} b$ where A is the symmetric matrix

$$A_{ik} = \int_H y_i(s) y_k(s) ds$$

and b is the vector of integrals over the hemisphere of $f(s)$ multiplied by the SH basis functions

$$b_k = \int_H f(s) y_k(s) ds$$

Alternatively, b can be thought of as the standard SH projection of a spherical extension of f which returns 0 when evaluated on the other half of the sphere, called the *zero-hemisphere hemispherical projection*. Note that A can be inverted once regardless of the function $f(s)$. Note also that A is the identity matrix when integrating over the entire sphere.

Readers familiar with biorthogonal bases used for wavelets will find this familiar; $y(s)$ is the primal basis and $A^{-1} y(s)$ forms its dual basis.

For 5th order SH projection (25 basis functions), the matrix A is nearly singular – its smallest singular value is 6.59×10^{-6} whereas its largest singular value is 1 (for comparison, the second smallest singular value is 3.10×10^{-4}). We can therefore discard one of the SH basis functions, since at least one is very well approximated as a linear combination of the others when restricted to a single hemisphere. A simple analysis shows that discarding the $l=1, m=0$ SH basis function (i.e., the SH basis function that is linear in z) has the smallest squared error, 1.48×10^{-5} , when approximated as a linear combination of the other basis functions.

9.2 Error Analysis of Various Projections

We first compare the difference between the zero-hemisphere and least-squares optimal projections. The integral, $\int_H (\sum_i c_i y_i(s))^2 ds$, of the squared value of an approximated function specified by its least-squares

optimal coefficient vector c is given by $c^T A c$. If, as before, b is the zero-hemisphere hemispherical projection of f , then $c = A^{-1} b$ is the optimal least-squares hemispherical projection of f . The squared difference between these two projections integrated over H is

$$E_l = (c-b)^T A (c-b) = c^T [(A-I)^T A (A-I)] c = c^T Q_l c$$

where I is the identity matrix. E_l attains a maximum value of 0.125 and an average value of 0.0402 over all signals formed by linear combinations of up to 5th order SH basis functions having unit squared integral over the sphere; i.e., over all unit-length 25D vectors c . Worst- and average-case errors are derived as the largest and average singular value of the symmetric matrix Q_l . These are large differences as a fraction of the original unit-length signal; using the RMS norm enlarges them still more via a square root. Optimal projection represents any element of this function space without error.

Another way of restricting the SH basis to the hemisphere ([Westin *et al.* 1992]) is to reflect f 's value about z to form a function defined over the whole sphere, via

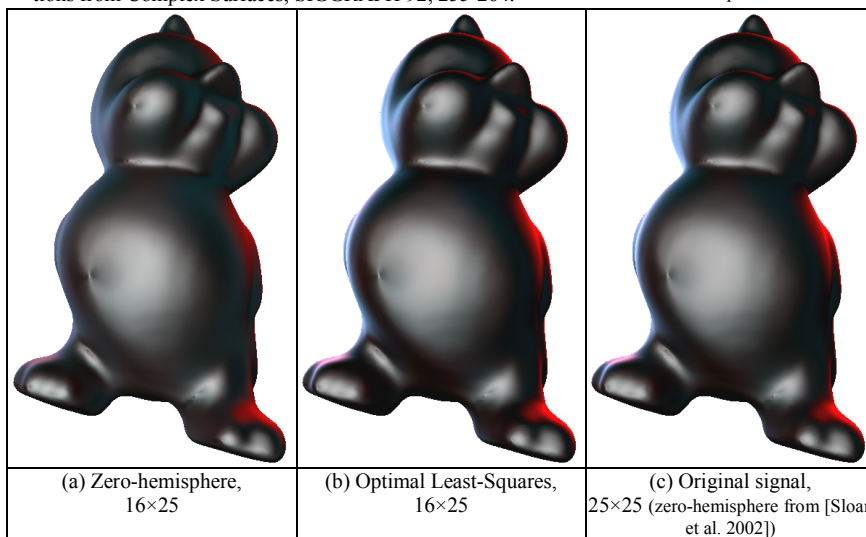


Figure 7: Projection comparison for glossy transfer matrices. Note the increased fidelity of the optimal least-squares projection (b) compared to zero-hemisphere (a) especially at the silhouettes (blue and red from colored light sources) where the Fresnel factor in the BRDF has high energy. Essentially, using optimal least-squares matches accuracy of a 25×25 matrix from [Sloan *et al.* 2002] via a 16×25 matrix (compare b and c).

$$f_{\text{odd}}(x, y, z) = \begin{cases} f(x, y, z), & \text{if } z \geq 0 \\ -f(x, y, -z), & \text{otherwise} \end{cases}$$

We can then derive a hemispherical projection of f as the coefficient vector given by the standard SH projection of f_{odd} . We call this method the *odd reflection hemispherical projection*. It is easy to see that a spherical function given by its SH coefficient vector c and then restricted to the $z \geq 0$ hemisphere yields a projection coefficient of $2c_i$ for the *odd* SH basis functions, for which $y_i(x, y, z) = y_i(x, y, -z)$, and 0 for the *even* SH basis functions, for which $y_i(x, y, z) = -y_i(x, y, -z)$ (all SH basis functions are either odd or even in z).

We analyze reflection projection in terms of squared error in the same way as for zero-hemisphere projection. Since the projection should have a comparable number (i.e., at least 25) of nonzero projection coefficients, we use SH basis functions up to order 8, which includes 28 odd (and thus nonzero) basis functions and 36 even ones, for a total of 64. Using this projection method for the same 5th order function space of interest, represented by the coefficient vector c , yields error E_2 defined as

$$E_2 = c^T [(D^*A - I)^T A (D^*A - I)] c = c^T Q_2 c$$

where D^* is a 64×64 diagonal matrix which scales odd basis functions by 2 and even ones by 0, and A is the symmetric matrix defined previously but now for up to 8th order (64×64). Using an SVD of the upper left 25×25 block of the symmetric matrix Q_2 , the worst case error over all unit-length 25D vectors c is given by its largest singular value and equals 0.145. The average squared error is given by the average singular value of the upper-left block of Q_2 and equals 0.044. In other words, odd reflection is worse than zero-hemisphere projection in both worst-case and average-case, even though it has more projection coefficients.

A similar analysis can be applied to even reflection, by projecting the even reflection extension of f defined as

$$f_{\text{even}}(x, y, z) = \begin{cases} f(x, y, z), & \text{if } z \geq 0 \\ f(x, y, -z), & \text{otherwise} \end{cases}$$

For 7th order SH basis functions, 28 are even and thus produce nonzero coefficients. An error measure for even reflection is identical to E_2 except that its diagonal matrix D^* scales by 2 the even basis functions and zeroes the odd. This projection provides worst-case error over all unit-length signals c of 0.022 and average-case error of 0.0036; still significant but far less than either the zero-hemisphere or odd reflection projections. Interestingly, even reflection using a smaller 5th order basis with only 15 relevant basis functions provides 0.193 worst-case and 0.030 average-case error – better average-case error than zero-hemisphere projection with many fewer coefficients.



Figure 8: Glossy phong models. We get a performance speedup of 16-20x over the method in [Sloan *et al.* 2002] without noticeable degradation, by encoding with CPCA and using least-squares optimal projection to reduce matrix rows from 25 to 16.

So even reflection is better than zero-hemisphere which in turn is better than odd reflection to approximate functions over the hemisphere. This can be understood because odd reflection produces a discontinuous spherical extension, while even reflection is continuous. Zeroing out the hemisphere is at least continuous for a portion of its basis functions – the odd ones. [Westin *et al.* 1992] also included scaling of the SH basis function by z , so that scaled odd reflection provides a continuous spherical extension. But such scaling yields high approximation error unless f roughly decreases as $z \rightarrow 0$ and $f(x, y, 0) = 0$. This is not generally true of our exiting radiance functions.



Figure 9: Translucent+glossy Buddha in two lighting environments. These images were rendered at 27Hz.





			
PCA , $m_c=1$, $n'=20$ 45.7Hz, SE=101304	VQ , $m_c=1792$, $n'=0$ 41.8Hz, SE=14799	CPCA , $m_c=256$, $n'=8$ 45.5Hz, SE=294.4	Uncompressed 3.7Hz, SE=0

Figure 10: VQ vs. PCA vs. CPCA quality results for matched rendering performance. The transfer signal encoded was a 24×25 (600D) glossy transfer matrix for an anisotropic BRDF. CPCA achieves much better visual and quantitative accuracy than VQ and pure PCA. Rendering frame rates and error measurements are listed below each of the four columns. CPCA was encoded using the iterative method of Section 4.2.

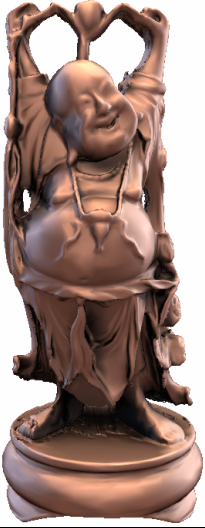
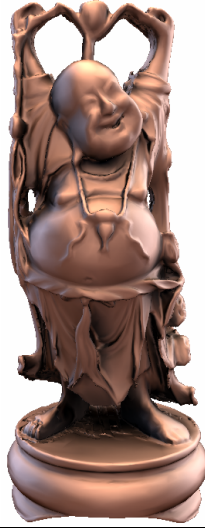
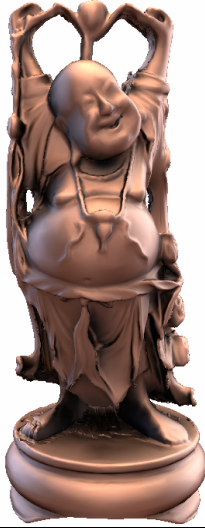
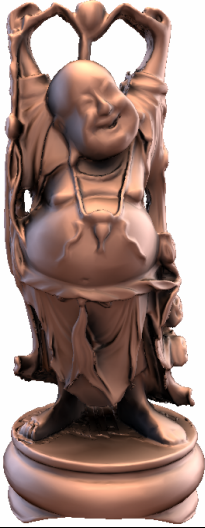
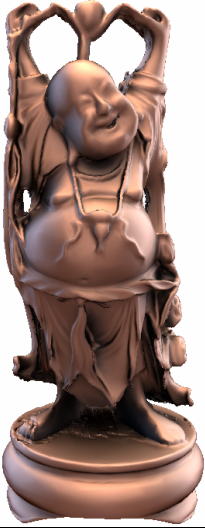



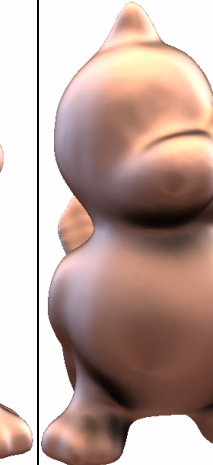
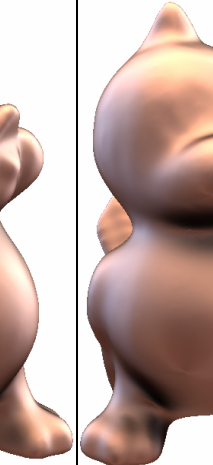
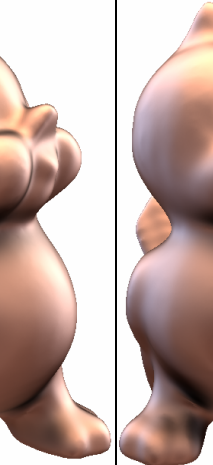
					
40.4Hz, SE= 40353.5	36.4Hz, SE=21077.5	24.2Hz, SE=8524.1	18.7Hz, SE= 4413.01	3.3Hz, SE=0	
					
58.9Hz, SE=9510.75	57.1Hz, SE=2353.09	45.5Hz, SE=294.421	31.9Hz, SE=66.7495	3.7Hz, SE=0	
CPCA , $n'=2$	CPCA , $n'=4$	CPCA , $n'=8$	CPCA , $n'=12$	uncompressed	

Figure 11: Varying the number of representatives per cluster (n'). Signal is glossy 24×25 transfer with anisotropic BRDF, $m_c=256$.