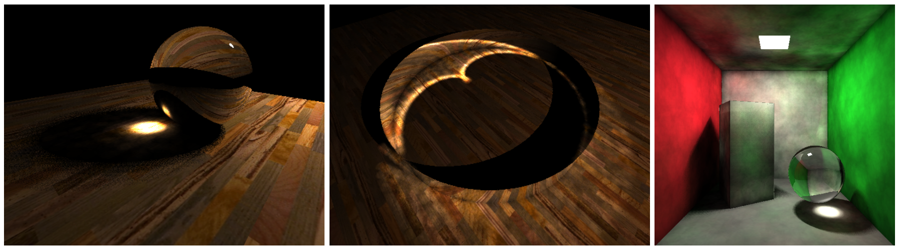
Graphics Hardware (2003)
M. Doggett, W. Heidrich, W. Mark, A. Schilling (Editors)

Photon Mapping on Programmable Graphics Hardware

Timothy J. Purcell[1], Craig Donner[2], Mike Cammarano[1], Henrik Wann Jensen[2] and Pat Hanrahan[1]

[1] Stanford University

[2] University of California, San Diego



Abstract

We present a modified photon mapping algorithm capable of running entirely on GPUs. Our implementation uses breadth-first photon tracing to distribute photons using the GPU. The photons are stored in a grid-based photon map that is constructed directly on the graphics hardware using one of two methods: the first method is a multipass technique that uses fragment programs to directly sort the photons into a compact grid. The second method uses a single rendering pass combining a vertex program and the stencil buffer to route photons to their respective grid cells, producing an approximate photon map. We also present an efficient method for locating the nearest photons in the grid, which makes it possible to compute an estimate of the radiance at any surface location in the scene. Finally, we describe a breadth-first stochastic ray tracer that uses the photon map to simulate full global illumination directly on the graphics hardware. Our implementation demonstrates that current graphics hardware is capable of fully simulating global illumination with progressive, interactive feedback to the user.

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

Keywords: Programmable Graphics Hardware, Global Illumination, Photon Mapping

1. Introduction

Global illumination is essential for realistic image synthesis in general environments. Effects such as shadows, caustics, and indirect illumination are important visual cues that add to the perceived realism of a rendered scene.

Global illumination algorithms have a long history in computer graphics from early work based on radiosity[8] and Monte Carlo ray tracing[12], to more recent algorithms such as photon mapping[10]. Photon mapping is one of the more widely used algorithms, since it is very practical and capable of computing a full global illumination solution efficiently. It is a two-pass technique in which the first pass consists of tracing photons through the scene and recording their interaction with the elements in the scene in a data structure, the photon map. This photon map is used during the second pass, the rendering pass, to estimate diffuse indirect illumination as well as caustics. The illumination at a given point is estimated based on statistics, such as the density, of the nearest photons located in the photon map.

Global illumination algorithms such as photon mapping have traditionally relied on sophisticated software implementations and offline rendering. Using graphics processors (GPUs) for computing a global illumination solution has not previously been possible due to the lack of floating point capability, as well as insufficient programmability. This has changed with the most recent generation of programmable graphics hardware such as the ATI Radeon 9800 Pro[1] and the NVIDIA GeForce FX 5900 Ultra[19]. The programming model for these GPUs is still somewhat limited, mainly due to the lack of random access writes. This prevents efficient construction of most data structures and makes many common algorithms such as sorting difficult to implement efficiently. Nonetheless, several researchers have harnessed the computational power of programmable GPUs to perform computations previously run in software[4, 5, 9, 14, 15, 21]. Similarly, we are interested in using GPUs to simulate global illumination using photon mapping.

Previous research on graphics hardware has explored the idea of simulating global illumination. Ma et al.[16] proposed a technique for approximate nearest neighbor search in the photon map on a GPU using a block hashing scheme. Their scheme is optimized to reduce bandwidth on the hardware, but it requires processing by the CPU to build the data structure. Carr et al.[5] and Purcell et al.[21] used the GPU to speed up ray tracing, and they also simulated global illumination using path tracing. Unfortunately, path tracing takes a significant number of sample rays to converge and even with the use of GPUs it remains a very slow algorithm.

The idea of speeding up global illumination to achieve interactive frame rates has been explored by several researchers in the last few years. Parker et al.[20] demonstrated how ray tracing, and to some extent path tracing, could be made interactive on a 32 processor shared memory SGI machine. This concept was later extended to Linux clusters by Wald et al.[24]. Recently, Wald et al.[23] also demonstrated that photon mapping combined with instant radiosity could be used to simulate global illumination at interactive rates on a Linux cluster. They achieve interactive speeds by biasing the algorithm and by introducing a number of limitations such as a highly optimized photon map data-structure, a hashed grid. By choosing a fixed search radius *apriori*, they set the grid resolution so that all neighbor queries simply need to examine the 8 nearest grid cells. However, this sacrifices one of the major advantages of the *k*-nearest neighbor search technique, the ability to adapt to varying photon density across the scene. By adapting the search radius to the local photon density, Jensen's photon map can maintain a user-controllable trade off between noise (caused by too small a radius yielding an insufficient number of photons) and blur (caused by too large a search radius) in the reconstructed estimate.

In this paper we present a modified photon mapping algorithm that runs entirely on the GPU. We have changed the data structure for the photon map to a uniform grid, which can be constructed directly on the hardware. In addition, we have implemented a variant of Elias's algorithm[6] to search the grid for the *k*-nearest neighbors of a sample point (kNN-grid). This is done by incrementally expanding the search radius and examining sets of grid cells concentrically about the query point. For rendering, we have implemented a stochastic ray tracer, based on a fragment program ray tracer like that introduced by Purcell et al.[21]. We use recursive ray tracing for specular reflection and refraction[26] and distributed tracing of shadow rays to resolve soft shadows from area lights[7]. Finally, our ray tracer uses the kNN-grid photon map to compute effects such as indirect illumination and caustics.

Our implementation demonstrates that current graphics hardware is capable of fully simulating global illumination with progressive and even interactive feedback to the user.

The contribution of this paper is a method for obtaining a complete global illumination solution on the GPU using photon maps. To compute various aspects of the global illumination solution, we introduce a number of GPU based algorithms for sorting, routing, and searching.

2. Photon Mapping on the GPU

The following sections present our implementation of photon mapping on the GPU. Section 2.1 briefly describes the tracing of photons into the scene. Section 2.2 describes two different techniques for building the photon map data structures on the GPU. Section 2.3 describes how we compute a radiance estimate from these structures using an incremental *k*-nearest neighbor search. Finally, section 2.4 briefly describes how we render the final image. A flow diagram for our system is found in figure 1.

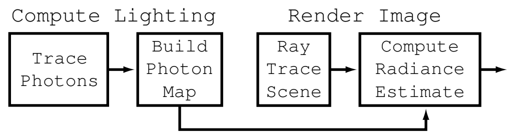


Figure 1: *System flow for our rendering system. Photon tracing and photon map construction only occur when geometry or lighting changes. Ray tracing and radiance estimates occur at every frame.*

Most of our algorithms use fragment programs to simulate a SIMD array of processors. For every processing pass, we draw screen sized quad into a floating point pbuffer, effectively running an identical fragment program at every pixel in the 2D buffer. This setup is common among several systems treating the GPU as a computation engine[4, 5, 21]. When computing the radiance estimate, however, we tile the screen with large points, enabling us to terminate certain tiles sooner than other tiles. The benefits of tiling are examined further in section 3.

2.1. Photon Tracing

Before a photon map can be built, photons must be emitted into the scene. The process of tracing eye rays and tracing photons from a light source is very similar. The most important difference is that at each surface interaction, a photon is stored and another is emitted. Much like tracing reflection rays, this takes several rendering passes to propagate the photons through several bounces. Each bounce of photons is rendered into a non-overlapping portion, or *frame*, of a photon texture, while the results of the previous pass are accessed by reading from the previous frame. The initial frame is simply the positions of the photons on the light source, and their initial random directions. The direction for each photon bounce is computed from a texture of random numbers.

Not all photons generated are valid; some may bounce into space. Current GPUs do not allow us to selectively terminate processing on a given fragment. We are, however, able to mark them as invalid.

2.2. Constructing the Photon Map Data Structure

The original photon map algorithm uses a balanced k-d tree[3] for locating the nearest photons. While this structure makes it possible to quickly locate the nearest photons at any point, it requires random access writes to construct efficiently. Instead we use a uniform grid for storing the photons, and in this section we present two different techniques for building this grid which involves placing the photons into the right grid cells. The first method sorts photons by grid cell using bitonic merge sort. This creates an array of photon indices where all photons in a grid cell are listed consecutively. Binary search is then used to build an array of indices to the first photon in each cell (see figure 4 for an example of the resulting data structure). To reduce the large number of of passes this algorithm requires, we propose a second method for constructing an approximate photon map using the stencil buffer. In this method, we limit the maximum number of photons stored per grid cell, making it possible to route the photons into their destination grid cells in a single pass using a vertex program and the stencil buffer.

2.2.1. Fragment Program Method - Bitonic Merge Sort

One way to index the photons by grid cell is to sort them by cell and then find the index of the first photon in each cell using binary search.

Many common sorting algorithms require the ability to write to arbitrary locations, making them unsuitable for implementation on current GPUs. We can, however, use a deterministic sorting algorithm for which output routing from one step to another is known in advance. Bitonic merge sort[2] has been used for sorting on the Imagine stream processor[13], and meets this constrained output routing requirement of the GPU.

Bitonic merge sort is a parallel sorting algorithm that allows an array of n processors to sort n elements in $O(\log^2 n)$ steps. Each step performs n comparisons and swaps. The algorithm can be directly implemented as a fragment program, with each stage of the sort performed as one rendering pass over an n pixel buffer. Bitonic sort is illustrated graphically in figure 2 and the Cg[17] code we used to implement it is found in figure 3. The result of the sort is a texture of photon indices, ordered by grid cell.

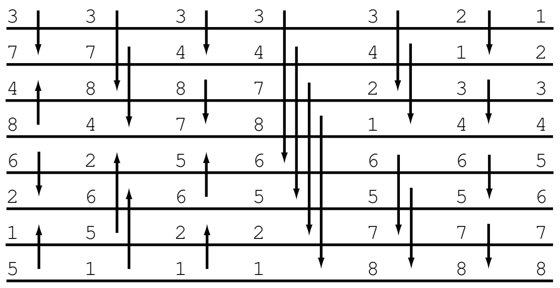


Figure 2: *Stages in a bitonic sort of eight elements. The unsorted input sequence is shown in the left column. For each rendering pass, element comparisons are indicated by the arrows, with items swapping to low and high in the arrow direction. The final sorted sequence is achieved in $O(log^2 n)$ passes.*

```
fragout_float BitonicSort( vf30 In, uniform samplerRECT sortedplist,
                           uniform float offset, uniform float pbufinfo,
                           uniform float stage, uniform float stepno )
{
    fragout_float dst;
    float2 elem2d = floor(In.WPOS.xy);
    float elem1d = elem2d.y*pbufinfo.x + elem2d.x;
    half csign = (fmod(elem1d, stage) < offset) ? 1 : -1;
    half cdir = (fmod(floor(elem1d/stepno), 2) == 0) ? 1 : -1;
    float4 val0 = f4texRECT( sortedplist, elem2d );
    float adr1d = csign*offset + elem1d;
    float2 adr2d = convert1dto2d(adr1d, pbufinfo.x);
    float4 val1 = f4texRECT( sortedplist, adr2d );
    float4 cmin = (val0.y < val1.y) ? val0 : val1;
    float4 cmax = (val0.y > val1.y) ? val0 : val1;
    dst.col = (csign == cdir) ? cmin : cmax;
    return dst;
}
```

Figure 3: *Cg code for the bitonic merge sort fragment program. The function* `convert1dto2d` *maps 1D array addresses into 2D texture addresses.*

Once the photons are sorted, binary search can be used to locate the contiguous block of photons occupying a given grid cell. We compute an array of the indices of the first photon in every cell. If no photon is found for a cell, the first photon in the next grid cell is located. The simple fragment program implementation of binary search requires $O(\log n)$ photon lookups. Because there is no need to output intermediate results, all of the photon lookups can be unrolled into a single rendering pass. An example of the final set of textures used for a grid-based photon map is found in figure 4.

Sorting and indexing is an effective way to build a compact, grid-based photon map. Unfortunately, the sorting step can be quite expensive. Sorting just over a million photons (1024×1024) would require 210 rendering passes, each applied to the full 1024×1024 buffer.

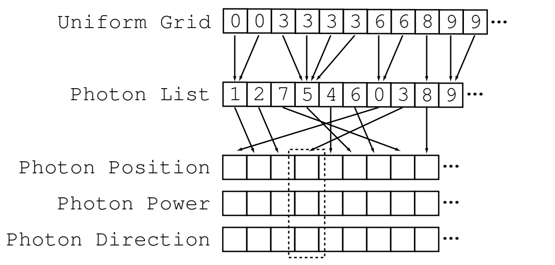


Figure 4: *Resultant textures for a grid-based photon map generated by bitonic sort. The uniform grid texture contains the index of the first photon in that grid cell. The photon list texture contains the list of photon indices, sorted by grid cell. Each photon in the photon list points to its position, power, and incoming direction in the set of photon data textures.*

2.2.2. Vertex Program Method - Stencil Routing

The limiting factor of bitonic merge sort is the $O(\log^2 n)$ rendering passes required to sort the emitted photons. To support global illumination at interactive rates, we would prefer to avoid introducing the latency of several hundred rendering passes when generating the photon map. To address this problem, we have developed an alternate algorithm for constructing a grid-based photon map that runs in a single pass.

We note that vertex programs provide a mechanism for drawing a glPoint to an arbitrary location in a buffer. The ability to write to a computed destination address is known as a scatter operation. If the exact destination address for every photon could be known in advance, then we could route them all into the buffer in a single pass by drawing each photon as a point. Essentially, drawing points allows us to solve a one-to-one routing problem in a single rendering pass.

This method of organizing photons into grid cells is a many-to-one routing problem, as there may be multiple photons to store in each cell. However, if we limit the maximum number of photons that will be stored per cell, we can preallocate the storage for each cell. By knowing this "texture footprint" of each cell in advance, we reduce the problem to a variant of one-to-one routing.

The idea is to draw each photon as a large glPoint over the entire footprint of its destination cell, and use the stencil buffer to route photons to a unique destination within that footprint. Specifically, each grid cell covers an $m \times m$ square set of pixels so each grid cell can contain at most $m \times m$ photons. We draw photons with glPointSize set to m which when transformed by the vertex program will cause the photon to cover every possible photon location in the grid cell. We set the stencil buffer to control the location each photon renders to within each grid cell by allowing at most one fragment of the $m \times m$ fragments to pass for each drawn photon. The stencil buffer is initialized such that each grid cell region contains the increasing pattern from 0 to $m^2 - 1$. The stencil

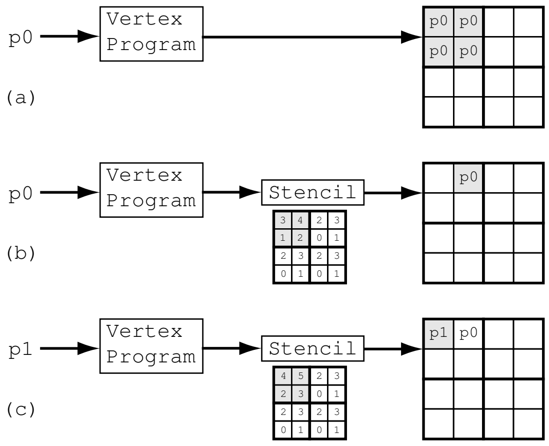


Figure 5: *Building the photon map with stencil routing. For this example, grid cells can hold up to four photons, and photons are rendered as 2×2 points. Photons are transformed by a vertex program to the proper grid cell. In (a), a photon is rendered to a grid cell, but because there is no stencil masking the fragment write, it is stored in all entries in the grid cell. In (b) and (c), the stencil buffer controls the destination written to by each photon.*

test is set to write on equal to $m^2 - 1$, and to always increment. Each time a photon is drawn, the stencil buffer allows only one fragment to pass through, the region of the stencil buffer covering the grid cell all increment, and the next photon will draw to a different region of the grid cell. This allows efficient routing of up to the first m^2 photons to each grid cell. This process is illustrated in figure 5.

We generally use a 1024×1024 stencil buffer with m set to 16, leaving approximately 40^3 grid cells. In regions of high photon density, many more photons than can be stored will map to a single grid cell. To reduce the artifacts of this method, we redistribute the power of the surplus photons across those that are stored. Note that the the stencil buffer maintains a count of how many photons were destined for each grid cell, and we assume that all our stored photons have roughly the same power. Hence, we can scale the power of the stored photons by the ratio between the number of photons destined for a cell and the number actually stored. This redistribution of power is an approximation, but the potential performance benefits of the fast routing method can be worthwhile. The idea of redistributing power of some photons to limit the local density of photons stored is discussed more generally in Suykens and Willems[22].

By storing a fixed number of photons per cell instead of a variable length list, we can use a vertex program to route photons to grid cells in a single rendering pass. There are two main drawbacks to this method. First, the photons must be read from the photon texture and drawn as points, which currently requires a costly readback. Second, the pre-allocation of storage for each grid cell limits the method's flexibility and space-efficiency. Redistribution of power is needed to represent cells containing more than m^2 photons, and space is wasted for cells with fewer photons (including empty cells).

2.3. The Radiance Estimate

To estimate radiance at a given surface location we need to locate the nearest photons around this location. For this purpose we have developed a kNN-grid method, which is a variant of Elias's algorithm for finding the k-nearest neighbors to a sample point in a uniform grid[6]. First, the grid cell containing the query point is explored, and all of its photons are examined. As each photon is examined, it will either be added to the running radiance estimate, or rejected. A photon is always rejected if it is outside a predefined maximum search radius. Otherwise, rejection is based on the current state of the search. If the number of photons contributing to the running radiance estimate is less than the number requested, the power of the new photon is added to the running estimate and the search radius is expanded to include that photon. If a sufficient number of photons have already been accumulated, the search radius no longer expands. Photons within the current search radius will still be added to the estimate, but those outside will be rejected.

Grid cells are explored in concentric sets centered about the query point. The photon search continues until either a sufficient number of photons have been accumulated, or a predefined maximum search radius is reached. Figure 6 illustrates the kNN-grid algorithm.

The kNN-grid always finds a set of nearest neighbor photons – that is, all the photons within a sphere centered about the query point. It will find at least k nearest photons (or as many as can be found within the maximum search radius). This means that the radius over which photons are accumulated will be at least as large as in Jensen's implementation[11], which uses a priority queue to select only the k-nearest neighbors. Accumulating photons over a larger radius could potentially introduce more blur into our reconstructed estimates. In practice, however, image quality does not seem to suffer from this.

2.4. Rendering

To generate an image we use a stochastic ray tracer written using a fragment program. The output of the ray tracer is is a texture with all the hit points, normals, and colors for a given ray depth. This texture is used as input to several additional fragment programs. One program computes the direct illumination using one or more shadow rays to estimate the visibility of the light sources. Another program invokes the ray tracer to compute reflections and refractions. Finally, we use the fragment program described in the previous section to compute the radiance estimates for all the hits generated by the ray tracer. We display the running radiance estimate maintained by the kNN-grid algorithm, providing progressive feedback about the global illumination of the scene.

3. Results

All of our results are generated using a GeForce FX 5900 Ultra and a 3.0 GHz Pentium 4 CPU with Hyper Threading and 2.0 GB RAM. The operating system was Microsoft Windows XP, with version 43.51 of the NVIDIA drivers. All of our kernels are written in Cg[17] and compiled with cgc version 1.1 to native fp30 assembly.

3.1. Rendered Test Scenes

In order to simplify the evaluation of the photon mapping algorithm we used scenes with no acceleration structures. For each scene, we write a ray-scene intersection in Cg that calls ray-quadric and ray-polygon intersection functions for each of the component primitives. For these simple scenes, the majority of our system's time is spent building the photon map, and computing radiance estimates. Very little time is spent on ray intersection. Purcell et al.[21] discuss ray tracing of more complex scenes using acceleration structures on the

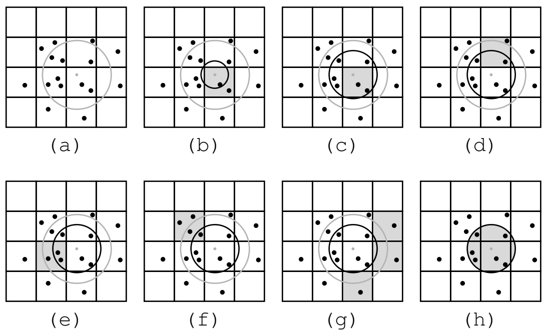


Figure 6: *Computing the radiance estimate with the kNN-grid. For this example, four photons are desired in the radiance estimate. The initial sample point and the maximum search radius are shown in (a). The first grid cell searched (shaded in (b) and (c) contributes two photons and expands the search radius. The next cell searched (d) has one photon added to the radiance estimate, and the other rejected since it is outside the predefined maximum search radius. The photon outside the search radius in (e) is rejected because the running radiance estimate has the requested number of photons, causing the search radius to stop expanding. The cell in (f) contributes one photon to the estimate. None of the other cells searched in (g) have photons that contribute to the radiance estimate. The final photons and search radius used for the radiance estimate are shown in (h).*

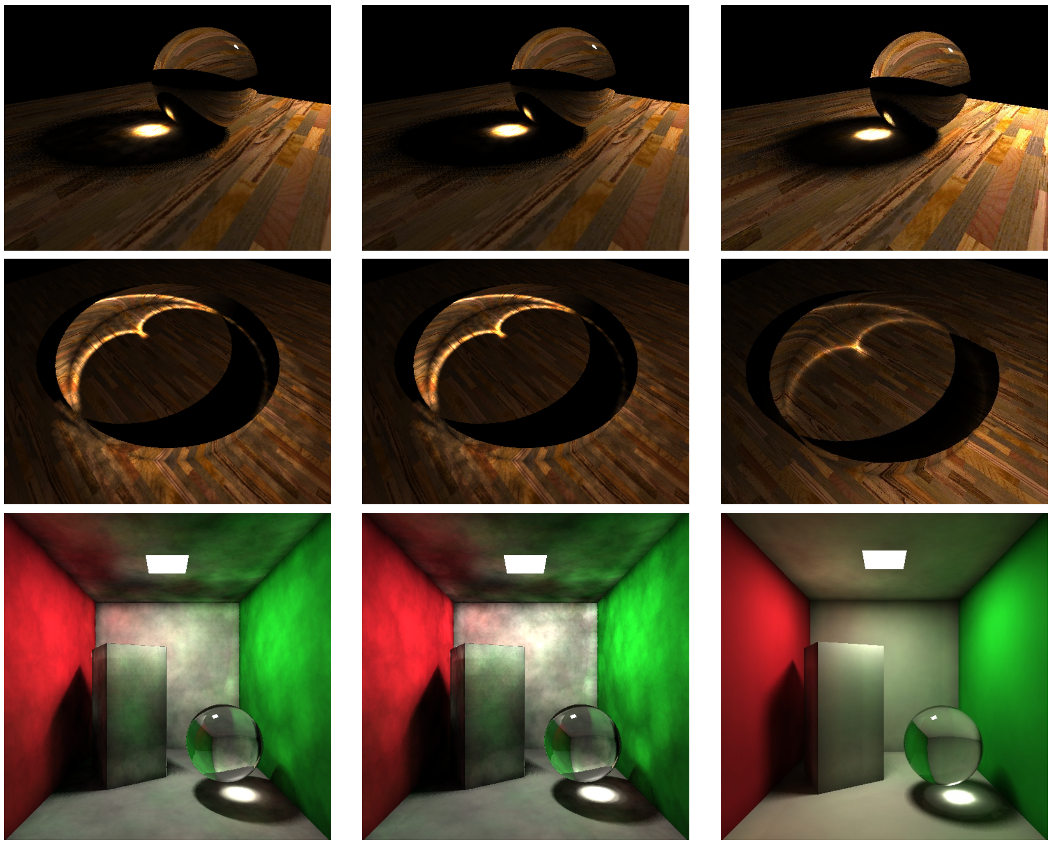


Figure 7: *Test scene renderings. Both (a) and (b) were rendered on the GPU using bitonic sort and stencil routing respectively. Software renderings are shown in (c) for reference.*

	Bitonic Sort				Stencil Routing			
Scene Name	Trace Photons	Build Map	Trace Rays	Radiance Estimate	Trace Photons	Build Map	Trace Rays	Radiance Estimate
GLASS BALL	1.2s	0.8s	0.5s	14.9s	1.2s	1.8s	0.5s	7.8s
RING	1.3s	0.8s	0.4s	6.5s	1.3s	1.8s	0.4s	4.6s
CORNELL BOX	2.1s	1.4s	8.4s	52.4s	2.1s	1.7s	8.4s	35.0s

Table 1: *GPU render times in seconds for the scenes shown in figure 7, broken down by type of computation. Ray tracing time includes shooting eye rays and shadow rays.*

GPU. We will examine the performance impact of complex scenes later in section 4.

We have rendered several test scenes on the GPU using our photon mapping implementation. Figure 7 shows three sets of images of our test scenes. The first column shows the images produced by the GPU when using the kNN-grid on a photon map generated by bitonic sort. The second shows the results of using stencil routing and power redistribution when rendering the scenes. The third column shows a software rendered reference image.

All of our test scenes are rendered with a single eye ray per pixel. The GLASS BALL and CORNELL BOX scenes have area lights which are randomly sampled by the ray tracer when computing shadows. The GLASS BALL scene samples the light source four times per pixel, and the CORNELL BOX scene samples the light source 32 times per pixel. The RING scene uses a point light source and only shoots one shadow ray per pixel.

The GLASS BALL scene was rendered at 512×384 pixels using a $250 \times 1 \times 250$ grid with 5,000 photons stored in the photon map and 32 photons were sought for the radiance estimate. The RING scene was rendered at 512×384 pixels using a $250 \times 1 \times 250$ grid with 16,000 photons stored in the photon map and 64 photons were sought for the radiance estimate. Finally, the CORNELL BOX scene was rendered at 512×512 pixels using a $25 \times 25 \times 50$ grid with 65,000 photons stored and 500 sought for the radiance estimate.

The rendering times for our test scenes vary between 8.1 seconds for the RING scene and 64.3 seconds for the CORNELL BOX scene. Table 1 summarizes the rendering times for the images, broken down by computation type.

The majority of our render time is spent performing the radiance estimate. The time listed in table 1 is for every pixel to finish computation. However, for our example scenes we find that the system reaches visual convergence (that is, produces images indistinguishable from the final output) after a much shorter time. In the GLASS BALL scene, a photon map built with bitonic sort will visually converge in 4 seconds – nearly four times as fast as the time listed for full convergence would suggest. This happens for two reasons: First, dark areas of the scene require many passes to explore all the grid cells out to the maximum search radius, but few photons are found so the radiance estimate changes little. Second, bright regions have lots of photons to search through, but often saturate to maximum intensity fairly early. Once a pixel is saturated, further photons found do not contribute to its final color. Note that these disparities between visual and total convergence are not manifested when the photon map is built using stencil routing. Under that method, the grid cells contain a more uniform distribution of photons, and saturation corresponds to convergence.

3.2. Kernel Instruction Use

A breakdown of how the kernels spend time is important for isolating and eliminating bottlenecks. The instruction breakdown tells us whether we are limited by computation or texture resources, and how much performance is lost due to architectural restrictions. Table 2 shows the length of each compiled kernel. These instruction costs are for performing one iteration of each computation (e.g. a single step of binary search or a single photon lookup for the radiance estimate). The table further enumerates the number of instructions dedicated to texture lookups, address arithmetic, and packing and unpacking of data into a single output.

We see at least 20 arithmetic operations for every texture access. It may be surprising to note that our kernels are limited by computation rather than memory bandwidth. Generally, we would expect sorting and searching to be bandwidth-limited operations. There are several factors that lead our kernels to require so many arithmetic operations:

Kernel	Inst	TEX	Addr	Pack
Bitonic Sort	52	2	13	0
Binary Search	18	1	13	0
Rad. Estimate	202	6	47	41
Stencil Routing	42	0	25	0
Rad. Estimate	193	5	20	41

Table 2: *Instruction use within each kernel. Inst is the total number of instructions generated by the Cg compiler for one iteration with no loop unrolling. Also shown are the number of texture fetches (TEX), address arithmetic instructions (Addr), and bit packing instructions (Pack).*

- Limits on the size of 1D textures require large arrays to be stored as 2D textures. A large fraction of our instructions are spent converting 1D array addresses into 2D texture coordinates.
- The lack of integer arithmetic operations means that many potentially simple calculations must be implemented with extra instructions for truncation.
- The output from an fp30 fragment program is limited to 128 bits. This means that many instructions are spent packing and unpacking the multiple outputs of the radiance estimate in order to represent the components in the available space.

Our kernel analysis reveals the challenges of mapping traditional algorithms onto GPUs. In cases like sorting, the limited functionality of the GPU forces us to use algorithms asymptotically more expensive than those we would use on processors permitting more general memory access. In other cases, the limitations of the GPU force us to expend computation on overhead, reducing the effective compute power available. In section 4, we discuss several possible architectural changes that would improve the performance of algorithms like photon mapping.

It should be noted that hand coding can still produce kernels much smaller than those generated by the Cg compiler. For example, we have hand coded a bitonic sort kernel that uses only 19 instructions instead of the 52 produced by Cg. However, we determined that the benefits of using Cg during development outweighed the tighter code that could be achieved by hand coding. As the Cg optimizer improves, we anticipate a substantial reduction in the number of operations required for many of our kernels.

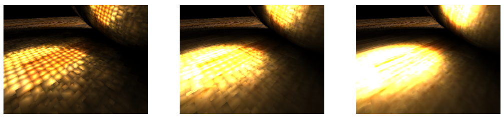

(a) 0.5s (b) 1.0s (c) 2.0s

Figure 8: *A detailed image of the* GLASS BALL *caustic over time. Reasonably high quality estimates are available much sooner than a fully converged solution.*

3.3. SIMD Overhead

Our radiance estimate kernel is run by tiling the screen with large points instead of with a single quad. Using the NV_OCCLUSION_QUERY extension, we are able to stop drawing a tile once all its pixels have finished. By terminating some tiles early, we are able to reduce the amount of SIMD overhead in our radiance estimate kernel.

This early termination of tiles substantially reduced the time required for our scenes to converge. We found tiling the screen with 16×16 points resulted in the largest improvements in convergence time. The CORNELL BOX scene saw the least improvement, with the time for the radiance estimate to fully converge dropping from 104 seconds to 52.4 seconds. Full convergence of the GLASS BALL scene was more dramatically affected, dropping from 102 seconds down to 14.9 seconds. These results are expected as the CORNELL BOX scene has a fairly uniform photon distribution but the GLASS BALL scene has high variance in photon density. Ideas for a more general way to reduce SIMD overhead via a fine-grained “computation mask” are discussed in section 4.

3.4. Interactive Feedback

One advantage of the incremental radiance estimate is that intermediate results can be drawn directly to the screen. The images in figure 7 required several seconds to fully converge. However, initial estimates of the global illumination are available very rapidly. Figure 8 shows various stages in the convergence of the radiance estimate for the full resolution GLASS BALL scene.

For smaller image windows, our system can provide interactive feedback. When rendering a 160×160 window, we can interactively manipulate the camera, scene geometry, and light source. Once interaction stops, the photon map is rebuilt and the global illumination converges in only one or two seconds.

4. Discussion and Future Work

In this section we discuss the limitations of the current system and areas for future work.

4.1. Fragment Program Instruction Set

The overhead of address conversion, simulating integer arithmetic, and packing is a dominant cost in many of our kernels. Addressing overhead accounts for nearly 60% of the cost of the stencil routing, and over 72% of the cost of the binary search. Similarly, the radiance-estimate kernels currently spend a third to a half of their time on overhead. Native support for integer arithmetic and addressing of large 1D arrays need not substantially complicate GPU design, but would dramatically reduce the amount of overhead computation needed in these kernels. Providing multiple outputs would remove the need for aggressive packing of values in the radiance estimates as well. Even with the overhead eliminated from the radiance estimate kernels, they still execute several arithmetic instructions and would continue to benefit from increased floating point performance without being limited by memory bandwidth.

4.2. Memory Bottlenecks

Texture readback and copy can impose significant performance penalties. We have shown timings for renderings with tens of thousands of photons. The stencil routing is particularly subject to readback performance since we currently must readback the texture of photons in order to use them as input to the vertex processor. With a low number of photons, texture readback consumes about 10% of the photon map construction time. However, as the number of photons increases, the fraction of time dedicated to photon readback increases to 60% and more of the total map construction time. The DirectX 9 API[18] supports displacement mapping, effectively permitting texture data to control point locations. We anticipate that similar functionality will appear as an extension to OpenGL, which would eliminate the need for readback in our stencil sort.

4.3. Parallel Computation Model

We mentioned in section 3 that we gained a significant performance increase when computing the radiance estimate by tiling the screen with large points instead of a full screen quad. Unfortunately, tiling is only practical when relatively few tiles are used and when pixels with long computations are clustered so that they do not overlap too many tiles. One natural solution to reducing the SIMD overhead for pixels with varying workloads is what we call a “computation mask”. Similar to the early fragment kill discussed by Purcell et al.[21], a user controllable mask could be set for each pixel in an image. The mask would indicate pixels where work has completed, allowing subsequent fragments at that location to be discarded immediately. We showed a performance gain from two to ten using a coarse tiling, and observe that a computation mask with single pixel granularity would be even more efficient.

4.4. Uniform Grid Scalability

One issue associated with rendering more complex scenes is that the resolution of the grid used for the photon map needs to increase if we want to resolve illumination details. At some point a high density uniform grid becomes too large to store or address on the GPU, and empty cells end up dominating the memory usage. One fix is to simply store the photons in a hash table based on their grid cell address[23]. High density grids no longer have empty cell overhead or addressability issues. Handling hash table collisions would add some overhead to the radiance estimate, however, as photons in the hash bucket not associated with the current grid cell must be examined and ignored. An additional problem for our stencil routing approach is that power redistribution becomes non-trivial.

4.5. Indirect Lighting and Adaptive Sampling

Our current implementation directly visualizes the photon map for indirect lighting and caustics. While this works well for caustics, the indirect lighting can look splotchy when few photons are used. A large number of photons are needed to obtain a smooth radiance estimate when the photon map is visualized directly. Instead, it is often desirable to use distributed ray tracing to sample incident lighting at the first diffuse hit point, and use the photon map to provide fast estimates of illumination only for the secondary rays. This final gather approach is more expensive, although the cost for tracing indirect rays can often be reduced using techniques like irradiance gradients[25] or adaptive sampling.

We have considered an adaptive sampling algorithm that initially computes a low resolution image and then builds successively higher resolution images by interpolating in low variance areas and tracing additional rays in high variance areas. Our initial studies have shown that this can reduce the total number of samples that need to be computed by a factor of 10. However, such a scheme cannot be implemented effectively without support for a fine-grained computation mask like that described in section 4.3.

5. Conclusions

We have demonstrated methods to construct a grid-based photon map, and how to perform a search for at least k-nearest neighbors using the grid, entirely on the GPU. All of our algorithms are compute bound, meaning that photon mapping performance will continue to improve as next-generation GPUs increase their floating point performance. We have also proposed several refinements for extending future graphics hardware to support these algorithms more efficiently.

We hope that by demonstrating the feasibility of a global illumination algorithm running completely in hardware, GPUs will evolve to more easily enable and support these types of algorithms.

6. Acknowledgments

We would like to thank Kurt Akeley and Matt Papakipos for helpful discussions contributing to development of the stencil routing method. Kekoa Proudfoot and Ian Buck both contributed ideas and algorithms related to sorting, searching, and counting on graphics hardware. We are grateful to Mike Houston for advice on debugging, and to John Owens for offering many suggestions on a draft of this paper. David Kirk and Nick Triantos provided us with the hardware and drivers that made this work possible.

We would also like to thank the organizations that provided us with individual funding. Tim Purcell is a recipient of an NVIDIA graduate fellowship. Mike Cammarano is supported by a National Science Foundation Fellowship and a 3Com Corporation Stanford Graduate Fellowship. Additional support was provided by DARPA contract F29601-01-2-0085.

References

1. ATI. Radeon 9800 Pro product web site, 2003. http://mirror.ati.com/products/pc/radeon9800pro/index.html.
2. Kenneth E. Batcher. Sorting networks and their applications. *Proceedings of AFIPS Spring Joint Computing Conference*, 32:307–314, 1968.
3. Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.
4. Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics*, 2003. (To appear in Proceedings of ACM SIGGRAPH 2003).
5. Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *Graphics Hardware*, pages 37–46, 2002.
6. John Gerald Cleary. Analysis of an algorithm for finding nearest neighbors in Euclidean space. *ACM Transactions on Mathematical Software (TOMS)*, 5(2):183–192, 1979.
7. Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '84)*, pages 137–145, 1984.
8. Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modelling the interaction of light between diffuse surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 213–222, July 1984.
9. Mark Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual

simulation on graphics hardware. In *Graphics Hardware*, pages 109–118, 2002.

10. Henrik Wann Jensen. Global illumination using photon maps. In *Rendering Techniques '96: 7th Eurographics Workshop on Rendering*, pages 21–30, 1996.

11. Henrik Wann Jensen. *Realistic Image Synthesis using Photon Mapping*. A K Peters, 2001. ISBN 1568811470.

12. James T. Kajiya. The rendering equation. In *Computer Graphics (Proceedings of ACM SIGGRAPH 86)*, pages 143–150, 1986.

13. Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, and Brucek Khailany. Efficient conditional operations for data-parallel architectures. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 159–170, 2000.

14. Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Transactions on Graphics*, 2003. (To appear in Proceedings of ACM SIGGRAPH 2003).

15. E. Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Supercomputing 2001*, page 55, 2001.

16. Vincent C. H. Ma and Michael D. McCool. Low latency photon mapping using block hashing. In *Graphics Hardware (2002)*, pages 89–98, 2002.

17. William R. Mark, Steve Glanville, and Kurt Akeley. Cg: A system for programming graphics hardware in a c-like language. *ACM Transactions on Graphics*, 2003. (To appear in Proceedings of ACM SIGGRAPH 2003).

18. Microsoft. DirectX home page, 2003. http://www.microsoft.com/directx/.

19. NVIDIA. Geforce FX 5900 product web site, 2003. http://nvidia.com/view.asp?PAGE=fx_5900.

20. Steven Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen. Interactive ray tracing. In *1999 ACM Symposium on Interactive 3D Graphics*, pages 119–126, 1999.

21. Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

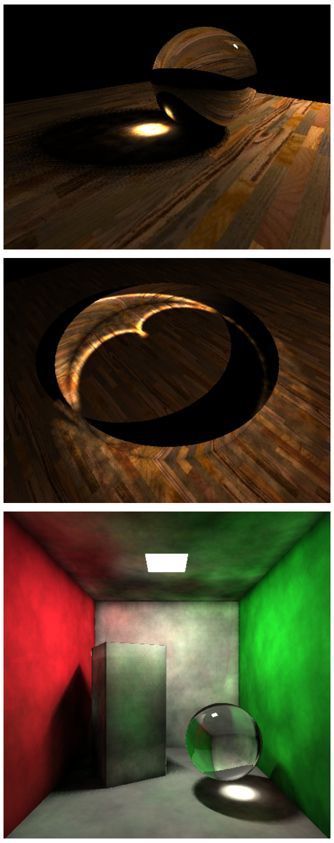
22. Frank Suykens and Yves D. Willems. Density control for photon maps. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 23–34, 2000.

23. Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. Interactive global illumination using fast ray tracing. In *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering*, pages 15–24, 2002.

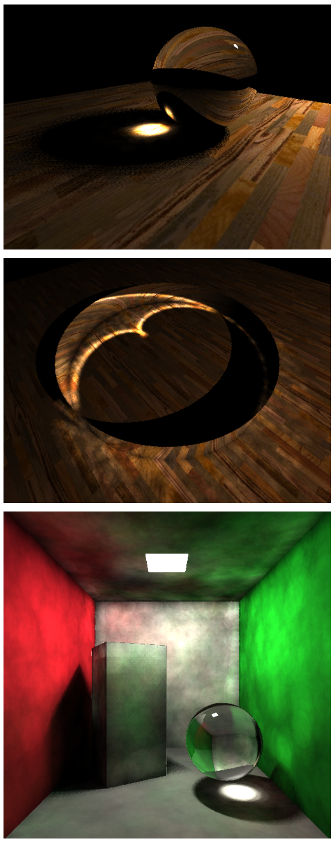
24. Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. *Computer Graphics Forum*, 20(3):153–164, 2001.

25. Greg Ward and Paul Heckbert. Irradiance gradients. In *Eurographics Rendering Workshop*, pages 85–98, May 1992.

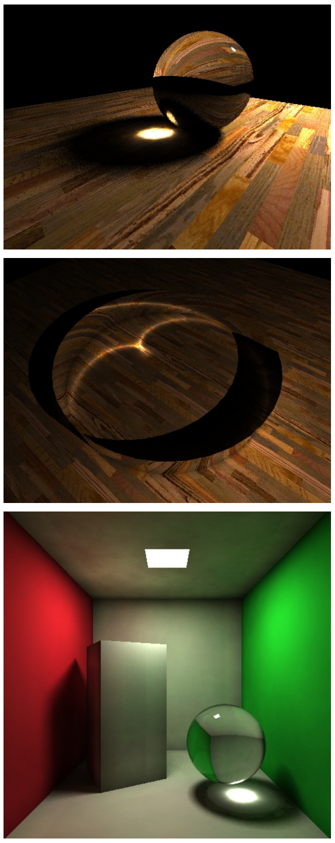
26. Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.



(a) Bitonic Sort



(b) Stencil Routing



(c) Software Reference

Figure 7: *Test scene renderings. Both (a) and (b) were rendered on the GPU using bitonic sort and stencil routing respectively. Software renderings are shown in (c) for reference.*