Interactive Global Illumination

Abhishek Singh, Anjali Kataria
Information Technology Department
Dronacharya College of Engineering, Farrukh Nagar, Khentawas, Gurgoan, Haryana

Abstract- **The matter and the interaction of light in the world surrounding us is of striking complexity and magnificent. Since the beginning of computer graphics, adequate modeling of these processes and extensive computation is an intensively studied research topic. This paper presents a GPU-based method for interactive global illumination that integrates complex effects such as multi-bounce indirect lighting, glossy reflections, caustics, and arbitrary specular paths.**

Our method builds upon scattered data sampling and interpolation on the GPU. We start with ray traced shading points and partition them into coherent shading clusters using adaptive seeding followed by k-means. At each cluster center we apply final gather to evaluate its incident irradiance using GPU-based photon mapping. We approximate the entire photon trees as a compact illumination cut, thus reducing the final gather cost for each ray. The sampled irradiance values are then interpolated at all shading points to produce rendering.

Our method exploits the spatial coherence of illumination to reduce sampling cost. Therefore our method is both fast and preserves high rendering quality. In contrast, we select sample points adaptively in a single pass, enabling parallel computation. As a result, our algorithm runs entirely on the GPU, achieving interactive rates for scenes with complex illumination effects.

Index Terms- **global illumination, GPU, photon mapping, final gather, k-means, and illumination cut.**

I. INTRODUCTION

Interactive computation of global illumination is a major challenge in computer graphics research today. Effects such as multi-bounce indirect lighting, caustics, and complex surface reflections are important visual cues for the perception of photorealism in synthesized images. However, with conventional CPU-based algorithms the simulation costs of such effects are often too high to permit dynamic interactions, where the user can simultaneously change the lighting, viewpoint, materials, and geometry, and expect accurate real-time feedbacks of such changes. With the rapidly increasing computation power of modern GPUs, much attention has been directed to exploiting the GPU to achieve interactive global illumination. Existing methods, however, typically focus on a very limited set of illumination effects. For instance, GPU-based ambient occlusion simulates global shadowing effects but ignores interreflections.

The goal of our work is to study a GPU-based global illumination method that integrates a wide range of illumination effects in a single algorithm. To allow for sufficient flexibility, we build upon kd tree based photon mapping. Recent work has successfully implemented fast kd-trees on modern GPUs, achieving interactive caustics and specular reflections. However, incorporating full global illumination effects remains challenging. The main difficulty resides in the final gather step, which involves computing a large number of secondary rays that can quickly saturate the GPU's computation power. It is well-known that the final gather step can benefit greatly from exploiting the spatial coherence in illumination changes, such as by using irradiance caching. Unfortunately, irradiance caching is not naturally amenable to the GPU's data-parallel processing model because it requires sequential computation of spatial sample points and frequent updates to a spatial structure that stores these points. As a result, even though irradiance caching is a common practice in CPU-based ray tracing, very few GPU-based solutions have been able to utilize it; those that do typically have to re-formulate the algorithm and ignore indirect shadows.

Similar to irradiance caching, our solution builds upon spatial data caching and interpolation to reduce sampling cost. However, to derive a practical GPU-based solution, our main contribution is a robust method that selects irradiance sample points before the final gather starts. This allows us to perform all steps of global illumination in parallel. To do so, we start by partitioning ray traced shading points into spatially coherent clusters. We introduce a method

based on adaptive sample seeding and k-means to compute this partitioning efficiently on the GPU. Each cluster center now becomes a spatial irradiance sample point, and our results show that the distribution of these points conforms to the actual irradiance changes. Next, at each cluster center, we perform final gather to sample its incident radiance field due to indirect lighting. This step uses a fast GPU-based photon mapping algorithm. To accelerate the density estimation in final gather, we approximate the entire photon tree as a compact illumination cut that is computed dynamically on the GPU. We then interpolate the sampled irradiance values at each shading point to produce the final rendering result. Our method also handles one-bounce of low-frequency glossy reflections by approximating both the sampled radiance fields and BRDFs onto a spherical harmonics (SH) basis set. By selecting irradiance sample points adaptively in a single pass, we eliminate the sequential computation required by standard caching based methods. As a result, our algorithm enables parallel computation and is implemented entirely on the GPU, achieving interactive frame rates for a variety of complex global illumination effects. Figure 1 shows an example.

Our GPU-based solution can be easily adapted to other many-core platforms such as Intel's upcoming Larrabee. As today's chip design is quickly switching to streaming and massively parallel processor models, we believe that it is essential to develop new algorithms that exploit such architecture to achieve interactive global illumination.

II. RELATED WORK

CPU-based Global Illumination A fundamental difficulty in global illumination is the high computation cost incurred by indirect lighting, where all surfaces contribute illumination to the scene. Instant radiosity is a popular technique that converts indirect illumination to a small set of virtual point lights (VPLs). This approach drastically reduces the computation cost, but is limited to a small number of VPLs and is only suitable for primarily diffuse materials. Photon mapping is another popular solution. It involves a photon scattering pass and a final gather pass to accurately simulate indirect lighting. For complex scenes, a large number of photons are needed, leading to high computation cost. In [2002], Wald et al. implemented interactive global illumination using a cluster of PCs. Recently, presented Light Cuts – a scalable method that computes illumination from a large number of point lights at sub linear cost. Their work is aimed at high-quality rendering and runs offline. Caching and Interpolation Since global illumination involves computing many rays; it is common to exploit the coherence among rays to reduce radiance sampling cost. This is especially true for indirect illumination, which changes smoothly and makes a good candidate for sparse sampling. Irradiance caching (IC) is a popular technique that progressively caches diffuse irradiance samples into an octree, and reuses them along the computation. Radiance caching extends IC by recording directional radiance using spherical harmonics. In [1999], Bala et al. presented a general approach to exploiting both spatial and temporal coherence of rays for radiance interpolation. Recently [Arikan et al. 2005] introduced a fast approximation to global illumination by decomposing radiance fields into far- and near-field components, which are computed separately to improve efficiency. In general, these caching schemes require sequentially inserting spatial sample points into a data structure, which has to be frequently queried and updated during the computation. This makes them unsuitable for the GPU's data parallel computation model. The Render Cache reuses radiance samples from previous frames by reprojecting them to the current frame, followed by the insertion of a small number of new samples. It exploits temporal coherence but the rendering takes a long time to converge. The Shading

Figure 1: *Global illumination result rendered using our algorithm. The user can dynamically manipulate any part of the scene, including lighting, viewpoint, materials, and geometry, at 1.5 fps.*

Cache performs caching and interpolation in object space, leading to faster convergence and improved rendering quality. However, it requires scene geometry to be subdivided into patches, which is

unsuitable for complex models. Furthermore, lighting and geometry changes lead to noticeable temporal artifacts.

GPU-based Global Illumination With the rapidly increasing computation power of modern GPUs, much recent work has focused on GPU-based solutions for global illumination. By using a low-resolution of fixed samples, this method does not provide high accuracy. Reflective shadow maps treat shadow map pixels projected from the direct light source as indirect sources illuminating the scene. Using image space computation, this method enables fast one-bounce indirect lighting, but does not handle multi-bounce interreflections. Similarly, radiance cache splatting uses image-based final gathering to approximate irradiance caching. While fast, these methods ignore occlusions in the final gather step, which are important for producing indirect shadows. Matrix row-column sampling converts illumination to many point lights, then exploits GPU shadow mapping to cluster them into representative lights and compute shadowed illumination from each. This method does not yet perform at interactive rates. Imperfect shadow maps evaluate approximate shadow maps for many point lights in a single pass on the GPU, dramatically improving the computation speed. Due to the lack of a ray tracing framework, these methods are only suitable for primarily diffuse surfaces, and cannot handle caustics or perfectly specular materials. Photon mapping was first implemented on the GPU by using a uniform grid structure to store photons; their rendering takes several seconds to converge. [Hachisuka 2005] introduced a fast acceleration method for final gather by using GPU rasterization. This greatly improves offline rendering speed. [Wyman and Davis 2006] proposed an image-space caustics algorithm, capable of simulating plausible caustics effects in real-time. This method is not aimed to include other global illumination effects. However, incorporating general illumination effects remains unsolved in their framework. Our method builds upon this algorithm, and we extend their work to include complex illumination effects such as multi-bounce Interreflections and caustics-incurred indirect lighting.

III. ALGORITHMS

This section provides an overview of our algorithm. We focus on scenes where the direct light source is a point light, so the direct illumination can be computed in real-time using GPU-based ray tracing. Incorporating area or environment lighting is possible but would require efficient methods to compute shadowed direct illumination from them. Therefore in the following we focus on discussing the computation of indirect illumination. We assume the scene materials to be either diffuse or perfectly specular. As in standard methods, we can extend diffuse reflections to one final bounce of glossy reflections using spherical harmonics. According to the rendering equation, the outgoing radiance Lo at a shading point x in view direction w_o is computed by the following integral:

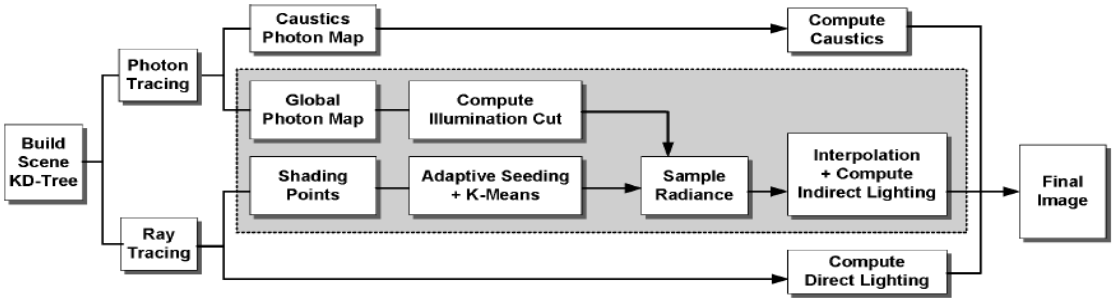


Figure 2: *A diagram showing our main building blocks. The shaded box marks the main contributions of this paper.*

$$L_o(x) = \rho(x) \int_{H^2} L_i(x, \vec{\omega}_i)\,(\vec{\omega}_i \cdot \vec{n})\, d\vec{\omega}_i = \rho(x) \cdot E(x) \quad (1)$$

Where Li is the incident radiance due to indirect illumination, $\vec{\omega}_i$ is an incident direction, p is the surface BRDF, H^2 is hemisphere centered on the surface normal, and E is the irradiance. Using photon mapping we can evaluate E in two passes. The first pass emits and scatters photons from the main light source, building a global photon map and a caustics

photon map. Each photon map is stored in a kd-tree for efficient access. The second pass performs final gather by sending many secondary rays at x to sample its incident radiance field Li. The radiance along each ray is evaluated using density estimation in the photon map, and then integrated with the cosine function according to Eq 1. For accurate rendering, a large number of final gather rays needs to be computed. As a result, evaluating this integral at every shading point is impractical for interactive settings. Our goal is to reduce spatial sampling cost by evaluating incident radiance only at a small number of shading points. As indirect illumination changes smoothly over the scene, these sparse samples provide a good approximation for interpolation at the remaining shading points. While the idea of spatial caching has long been exploited in irradiance caching and adaptive sampling methods, these methods require sample points to be progressively inserted into a spatial structure. (i.e. kd-tree), which is repeatedly updated during the computation. This computation model is unsuitable for a direct GPU mapping. To avoid the sequential computation, we select irradiance sample points ahead of time in a single pass, allowing us to perform the subsequent computation in parallel on the GPU. Even with spatial caching, the density estimation for each secondary ray is still quite expensive, due to the large number of photons stored in the kd-tree. To reduce this cost, we introduce an efficient approach that approximates the photon tree as an illumination cut which is dynamically constructed on the GPU. The center of each cut node represents an illumination cluster, and we use radial basis functions (RBFs) to smoothly interpolate between them. As the illumination cut size is much smaller than the total number of photons, we achieve significant speedup in density estimation. Figure 2 shows the main building blocks of our algorithm. First, we build a kd-tree of the scene on the GPU, and utilize it for ray tracing and photon mapping. These steps follow the kd-tree algorithms presented it. Next, we take the shading points computed from ray tracing, and partition them into 1000 ~ 6000 shading clusters based on each point's geometric information. This clustering is computed using GPU based adaptive sample seeding followed by k-means. We then perform final gather at each cluster center to sample its incident radiance field. In order to reduce the cost of density estimation, we approximate the global photon map as a compact illumination cut containing 4000~8000 nodes. At each node's center we evaluate and cache an irradiance value by using density estimation in the global photon map. As a result, performing density estimation with the illumination cut is significantly faster than with the full photon tree. Finally we interpolate the sampled irradiance values at all shading points to produce indirect lighting. The results are further combined with direct lighting and caustics to produce the final image.

3.1 GPU-based KD-Tree

Our algorithm requires the use of kd-tree in a number of steps, including ray tracing, photon mapping, k-means clustering, and radiance interpolation. Therefore an efficient kd-tree implementation is important. We follow the method to build kdtrees in real time using

NVIDIA's CUDA. This method constructs the kd-tree in breadth first order (BFS) using a greedy top-down approach, which recursively splits the current kd-tree node into two sub-nodes. For efficient GPU processing, nodes are classified as either large or small based on the number of geometric primitives belonging to them. For fast traversal in the kd-tree, a standard stackbased method is implemented by maintaining a local stack for each thread in the GPU's shared memory. Using this algorithm, we achieved real-time performance for direct lighting and caustics.

Direct Lighting

Following use ray tracing to compute direct lighting with the following steps:

1) Build a kd tree of the scene, and trace eye rays in parallel;

2) Collect rays that hit non-specular surfaces using a parallel list compaction

3) Similarly, collect rays that hit specular surfaces, and then spawn reflected and refracted rays for them.

4) Repeat steps 2 and 3 for additional bounces; 5) for all non-specular hit points, perform shadow tests and compute direct shading in parallel.

Shading Points

The collection of non-specular hit points in step 5) Above defines our shading points: they represent all the points that are either directly visible, or are visible after specular paths. Note that in the absence of specular objects, we can directly produce shading points via rasterization. However, with specular

objects, computing shading points requires ray tracing.

Building Photon Maps

We shoot photons from the direct light source, scatter them in the scene, and store them in photon maps when they hit non-specular surfaces. A photon is stored in the global photon map if its immediate previous hit is on a non-specular surface; otherwise we store it in the caustics photon map. We build a kd-tree for each photon map afterwards.

3.2 Selecting Irradiance Sample Points

Our next step is to select, among all shading points, a small set of representative points for sampling incident irradiance. Because these points will be used for interpolation at the remaining shading points, they need to be carefully selected to preserve the underlying illumination changes. In standard caching based schemes, this is achieved by progressively inserting sample points into an existing set; and the decision to insert more samples is based on the local variations of irradiance samples already stored in the set. This reduces data parallelism since the insertion of samples and the irradiance evaluation are dependent upon each other. An importance sampling method is introduced to distribute gather points for direct-to-indirect transfer computation. It uses a variation of photon mapping to estimate the importance of triangles in the scene. While not requiring progressive insertions, this method is unsuitable for on-the-fly sample generation due to the expensive computation. Our solution is to find an error metric that can reliably predict the actual irradiance changes based on the scene's geometric information. This allows us to select sample points in advance, before the irradiance evaluation step begins. Using this metric, we can partition shading points into coherent shading clusters, and the center of each cluster then becomes an irradiance sample point. To do so, we use a modified version of the illumination change term introduced in irradiance caching to define our error metric:

$$\varepsilon = \alpha \left\| x_i - x_k \right\| + \sqrt{2 - 2(\vec{n}_i \cdot \vec{n}_k)} \quad (2)$$

Where x_i is a shading point to be classified, x_k is the center position of a cluster Ck, and $\vec{n}$ denotes a surface normal. α is a weighting factor that determines the relative importance of position and normal incurred changes, and it typically varies between 0:1 ~0:5 in our experiments (after the scene geometry scale is normalized). Similar to irradiance caching, this metric is computed purely from geometric properties. The intuition is that points that are geometrically close to each other are likely to receive similar incident illumination. Note that we have omitted the harmonic distance term Rk in the original definition from [Ward et al. 1988]. This is because evaluating Rk requires final gathering, which is what we would like to avoid computing at this stage. However, we introduce an adaptive sample seeding approach to account for the effect of the Rk term, as explained below.

Adaptive Seeding of Sample Points: The Rk term in the original irradiance cache formulation accounts for the effect that areas close to other surfaces are likely to experience rapid illumination changes, therefore more sample points should be distributed there. To account for this effect, we use an approximation that distributes initial sample points according to the local geometric variations in the scene. In order to seed K sample points, we use a hierarchical histogram-based method to evaluate the geometric variations in image space. We start by constructing a static, screen-space quadtree of all the shading points. This can be done trivially since the shading points have a one-to-one mapping to screen pixels. For each quadtree node q, we compute its geometric variation "q according to Eq 2 by treating all shading points belonging to quadtree node q as a cluster. We initialize xk and ~nk to be the average position and normal at each node q. We compute this efficiently on the GPU by evaluating "q for all quadtree nodes in parallel. Following this step, we seed K initial sample points hierarchically according to the magnitude of "q. Specifically, we start from the root node of the quadtree, and at each node we distribute sample points to its four children nodes in proportion to the "q of its child nodes. When a node has only one sample to distribute, we use jittered sampling to randomly select a sample point within the quad. We process each level of the quadtree in parallel.

K-Means Clustering After the initial seeding, we refine the sample points by using k-means to partition all shading points into coherent clusters. To do so, we treat the seeded points as our initial cluster centers, and follow standard k-means to classify every

shading point to its closest center using the metric defined in Equation 2. After every shading point is classified, we recomputed the cluster center by averaging the position and normal of all cluster members. We then repeat these steps until either the clustering converges or a maximum number of iterations has been reached.

Figure 3: *(a) shows our clustering result for the Cornell box scene computed with 1600 clusters; (b) displays the cluster centers on top of a raytraced reference showing the actual irradiance values.*

Finally, in each Cluster, we pick the shading point with the smallest error as our irradiance sample point. Even on the GPU, the above algorithm is still quite expensive to run. The main bottleneck is in finding the nearest cluster center for every shading point, which incurs a computation cost linear to K using the brute-force approach. To reduce this cost, we need to avoid searching among all cluster centers. To do so, we build a kdtree of all cluster centers at the beginning of each k-means iteration; then for every shading point to be classified, we utilize the kd-tree to constrain the search within a small Euclidean distance to that shading point, thereby significantly reducing the search range. Figure 3(a) shows our clustering result for a Cornell box scene computed with 1600 clusters; (b) shows the cluster centers on top of an accurate irradiance image computed using offline ray tracing. Note that areas with rapid illumination changes are sampled densely with small clusters, while those with slow changes are sampled sparsely by large clusters. This example demonstrates that the samples we select conform with the underlying illumination changes. Temporal Coherence Since we recomputed irradiance sample points among shading points at every frame, our method is view dependent and is subject to temporal artifacts. For example, if the camera moves quickly, the selected irradiance sample points will differ from frame to frame, causing potential flickering artifacts. To reduce such artifacts, we attempt to preserve the temporal coherence among sample points. At the beginning of each new frame, we fix cluster centers xk computed from the previous frame, then estimate whether every shading point in the new frame can be classified into an existing cluster xk. To do so, we keep at each cluster the largest error (Eq 2) of its current members with the cluster center, then use this value as a threshold to check whether new shading points can be classified into this cluster. At the end of this estimation, we eliminate those clusters in xk that do not have any shading point associated with them. On the other hand, we create new clusters for all shading points that cannot be classified. The new clusters are generated using the same algorithm described above. Overall we maintain roughly the same number of shading clusters
(Hence irradiance sample points) at each frame.

3.3 Reducing the Cost of Final Gather

Once the irradiance sample points are selected, we sample their incident radiance fields by sending 250~ 500 final gather rays distributed according to the PDF of a cosine function. To evaluate the radiance for each final gather ray, standard photon mapping requires density estimation, which performs a KNN search in the photon tree to find nearby photons around a ray's hit point. Due to the large number of photons, the search cost is typically quite high. To reduce this cost, our goal is to use a smaller set of samples to approximate the illumination represented

by the large number of photons. The idea is that performing KNN search with this reduced set of points significantly improves the search speed. We achieve the reduction by dynamically computing an illumination cut from the photon tree. Although [Walter et al. 2005] have introduced an efficient method for computing cuts from a set of diffuse point lights, here we present a new approach that computes an illumination cut directly from the photon map, and is more amenable to the GPU. To do so, we first estimate an approximate irradiance for each photon tree node p by computing its illumination density, defined as:

$$\widetilde{E}_p = \frac{\sum_{i \in p} (\vec{\omega}_i \cdot \vec{n}_p)\, \Phi_i}{{r_p}^2} \quad (3)$$

Where Φ_i is the power of a photon belonging to node p, $\vec{\omega}_i$ is the incoming direction of the photon, $\vec{n}_p$ is the normal at the node center, and rp is the maximum side length of the node's bounding box. This is equivalent to performing density estimation but using all the photons in p and rp 2 as the area for estimation. This overestimates the search neighborhood but reduces the estimation cost. Note that e Ep is computed and stored when the kd-tree of the photon map is built, thus we do not have to compute it separately. Our next step is to find a cut through the tree to approximate the illumination. Given the approximated irradiance at each node, a coarse tree cut is selected from the photon tree in parallel: first, nodes with irradiance larger than a predefined threshold Emin are selected and added in the cut. Then, different levels of the tree are traversed iteratively to ensure the construction of a valid and complete tree-cut. This is done by deleting nodes whose children are already in the cut, and adding nodes to cover leaf nodes that do not have parents representing them in the cut. We pick the threshold E min as the average e ~Ep of nodes at the 12~13-th level of the photon tree. This is based on the heuristic that the initial illumination cut should be close to its target size, which is approximately 4K_8K. Given the coarse tree cut, we perform an optimization to improve its accuracy. This is done by comparing the accurate irradiance, E_p, evaluated at node p's center using standard density estimation, with the approximated irradiance e E_p computed above. If the difference between the two is larger than a user defined threshold Δ_E (which we typically set to 1:2 e E_p), the node is removed from the cut and replaced by its children nodes, thereby improving its accuracy. We perform 3_5 iterations of this refinement. During each iteration, nodes are independently computed in parallel. Once the cut is selected, each node of the cut becomes a sample point for illumination. We therefore cache the accurate irradiance value E_p at the center of each node and use these values for smooth interpolation. We use a set of spatial radial basis functions (RBFs) centered at each cut node as an interpolation basis. Specifically, given a hit point y of a final gather ray, the radiance of the ray is evaluated by locating a set of nearby cut nodes p_J, and interpolating from them using the following weight:

$$w_j = K\left(\frac{\|p_j - y\|}{r(p_j)}\right) \cdot \max(0,\, \vec{n}(p_j) \cdot \vec{n}_y), \quad (4)$$

$$K(r) = \begin{cases} 1 - r^2, r \le 1 \\ 0, r > 1 \end{cases} \quad (5)$$

Where r (p_J) is the maximum side length of node p_J's bounding box. Note that the sub tree defined from the root of the photon tree to the illumination cut can be used directly as a kd-tree for neighborhood search. Intuitively, each node p_J influences its surrounding space, up to the size of its bounding box, by a quadratic falloff weight.

3.4 Interpolation and Rendering

Representing Radiance Fields using SH In order to allow for glossy BRDFs in the final bounce, similar to previous work, we approximate the directional information of the sampled radiance fields

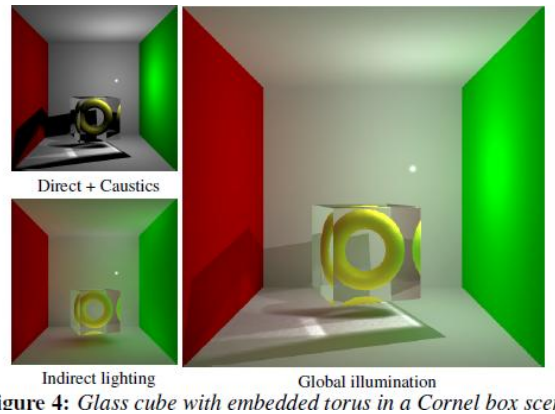


Figure 4: *Glass cube with embedded torus in a Cornel box scene.*

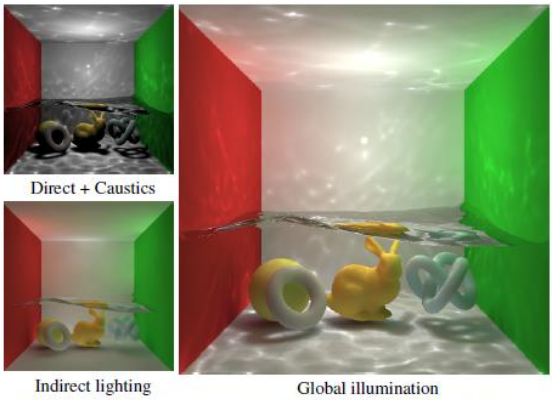


Figure 5: *Cornell box with an animated water surface.*

$L_i(x, \vec{\omega}_i)$ using a spherical harmonics (SH) basis $Y_l^m(\vec{\omega}_i)$ where l and m are the degree and order of an SH function respectively. We use 4-th order SH in our experiments, resulting in 16 projected coefficients, represented as a vector $L_i^{l,m}(x)$. For diffuse materials, we can treat their sampled irradiance value as an SH vector with a non-zero DC term while all other terms are zero.

$$L_i^{l,m}(x) = \frac{\sum_{j \in S} w(s_j) L_i^{l,m}(s_j)}{\sum_{j \in S} w(s_j)}$$

Interpolation To render, we interpolate irradiance values stored at each irradiance sample point. For glossy materials, we interpolate the SH coefficients instead, and integrate the results with glossy BRDFs

$$w(s_j) = \frac{1}{\frac{||x - s_j||}{R_j} + \sqrt{2 - 2\,(\vec{n}(x) \cdot \vec{n}(s_j))}}$$

(also projected onto SH). This step requires scattered data interpolation, which involves finding the nearby irradiance sample points for each shading point, and performing smooth interpolation. As before, we use a kd-tree of the irradiance sample points for efficient neighborhood search. We then perform interpolation in the same way as irradiance caching

Where S is the set of nearby irradiance sample points located around shading point x, sj is a sample in S, $L_i^{l,m}(s_j)$ denotes the SH coefficients and w (s_j) is the interpolation weight, computed by:

Where Rj is the harmonic mean distance to objects visible from s_J. It is evaluated during final gather, by keeping track of the intersection distance of each final gather ray, and performing a parallel reduction afterwards to compute the harmonic distance.

3.5 Implementation

Figure 6: *Comparison of indirect illumination results with reference images. The first row shows the kitchen scene: (a) is computed with 5k irr-adiance samples and the full photon map; (c) is computed at every shading point with an illumination cut size of 4k; (b) and (d) show magnified error images of (a) and (c) against reference (e). The second row shows the same comparison for two additional scenes where the results used for comparison are computed using parameters listed in Table 1.*

Scene	Tris	G/C photons	S.C.	I. cut	FPS
Box w/ torus	0.6K	239k/100k	1.6k	4k	4.2
Box w/ water	17k	268k/277k	1.8k	8k	2.7
Elephant 1	87k	256k/24k	3k	4k	3.1
Elephant 2	84k	290k/107k	4k	4k	2.5
Kitchen	21k	470k/109k	5k	8k	1.5

Table 1: *This table reports for each scene the number of triangles, global/caustics photons, shading clusters (i.e. irradiance sample points), illumination cut size, and rendering fps.*

We have implemented all our algorithms using BSGP – a general-purpose C programming interface suitable for many-core architecture such as the GPU. BSGP builds on top of NVIDIA's CUDA, but is easier to read, write, and maintain. Our algorithms make frequent use of standard parallel computation primitives such as scan, reduction, list compaction, and segmented versions of them. These are directly available in the BSGP library. In addition, we require an efficient GPU implementation of the kd tree. Most of our data elements are stored as dynamic arrays in the GPU linear memory. To avoid the high overheads of repeated allocation of memory, we implement a standard dynamic array management routine, which reserves an initial amount of memory on the GPU, and doubles the array size whenever the space runs out. For structures with many fields such as the k-means clusters, we use structure of arrays (SoA) instead of array of structures (AoS) for achieving global memory coalescing on the GPU. Algorithm details and various parameters we use can be found in Sections 3.1 to 3.4. For clarity, we have included pseudo-code of our algorithms in the supplemental material attached to this paper.

Light Paths

In general we can handle all light paths in the form of L (S|D)* (S|D|G) where L, S, D, G stands for lighting, specular, diffuse, and (low-frequency) glossy respectively. In practice, we limit the number of indirect bounces to 2. Therefore we compute 3 bounce altogether, including the final gather bounce. Our results show that the improvement in quality beyond 3 bounces is small.

IV. Results

In the following, we present our results computed on a PC with Intel 2.0 GHz CPU and an NVIDIA GeForce 280 GTX graphics card. Since we compute everything from scratch in each frame, the user can dynamically manipulate any element in the scene, including lighting, viewpoint, material, and geometry. Unless mentioned otherwise, all images reported are rendered at a super sampled resolution of 1024 X1024, and then down sampled to 512 X 512 final resolutions. The direct light source is a point or spot light.

Demo Scenes

Table 1 summarizes the 5 scenes used for demonstration. The first scene, a Cornell box with a glass cube and an embedded torus (Figure 4), demonstrates complex caustics effects from the cube and global illumination effects on the torus inside the box. The second scene, containing an animated water surface (Figure 5), demonstrates both reflective and refractive caustics effects from the water surface. Note that the illumination of objects below the water comes entirely from caustics. This type of light paths (L S+ D+) would be very difficult to simulate with previous interactive methods. For both scenes we show the decomposition of global illumination into a direct caustics component, and an indirect lighting component, shown on the left of each image. The third scene (Figure 8(a,b)) shows an elephant model with dynamically changing material properties. This scene demonstrates the capability of our solution to handle view-dependent indirect lighting at the final bounce. Glossy reflections are enabled by projecting both sampled incident radiance fields and BRDFs onto 4-th order SH. Note the color bleeding effects from the elephant to the wall and floor. Because the direct light source is pointing away from the elephant, these color bleeding effects are only enabled through multi bounce interreflections. The fourth scene (Figure 8(c, d)) contains an animated elephant model with rotating mirrors. It demonstrates global illumination effects caused by photons reflected from the mirrors and bounced inside the box. Our final scene (Figure 1, 7) is a kitchen model with a moving robot and dining table. Note the caustics effects cast from the dining table, and the overall indirect illumination. In Figure 7 we also include four sub images showing the direct caustics component, the indirect lighting component, the shading clusters, and a visualization of the cluster centers. For all scenes we achieve reasonable frame rates, ranging from 1.5~4.2 fps.

Parameters Table 1 list the main rendering parameters we use for each scene, including the number of global/caustics photons, number of shading clusters, and the illumination cut size.

Simple scenes that contain low visibility complexity require only a small number of shading clusters; more complicated scenes such as the kitchen require a larger number of shading clusters to more accurately represent the underlying illumination changes. For all

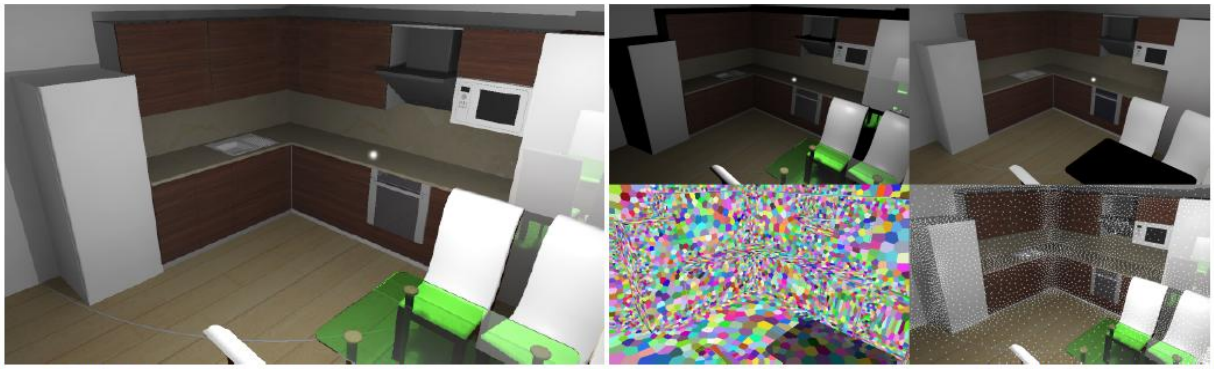
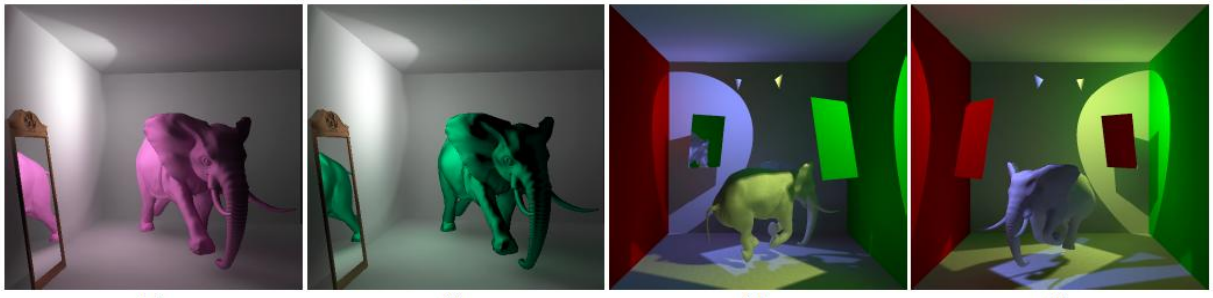


Figure 7: *The left image shows the global illumination result of the kitchen scene; the four subimages on the right show: direct lighting+caustics, indirect lighting, clusters of the shading points, and the distribution of cluster centers (i.e. irradiance sample points).*



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

Figure 8: *(a) and (b) show a glossy elephant with a mirror as the user changes the material property of the elephant. (c) and (d) show an animated elephant with two rotating mirrors in a Cornell box scene; note how the reflected lights from the mirrors illuminate the objects.*

scenes we compute final gather using 250 ~500 final gather rays.

Accuracy our algorithm consists of two main approximations: irradiance sampling and interpolation, and the approximation of photon map as an illumination cut. In Figure 6 (a-e) we examine the accuracy of each approximation individually. To focus on the analysis of indirect illumination, we have changed all materials in this scene to be entirely diffuse. (a) is computed with 5k irradiance samples while the final gather is computed by accurate density estimation in the full photon map; (c) is computed with an illumination cut size of 4k while the final gather is evaluated at every shading point; (e) shows a reference image computed by disabling both approximations, thus performing accurate final gather (with 1024 secondary rays) at every shading point. In (b) and (d) we show magnified absolute error images of (a) and (c) against (e).

In both approximations we found the rendering errors to be relatively small. The errors in (b) occur primarily in places where the irradiance is changing rapidly due to occlusion by nearby objects, such as in regions shadowed by adjacent geometry. Using a higher number of shading clusters will help because more clusters will be allocated to those regions, thereby improving the sampling accuracy. Similarly, the errors in (d) occur primarily in occluded regions where the incident radiance is low. This is mainly because our illumination cut approximates strong source lighting (i.e. photon tree nodes with high powers) more accurately than weak source lighting. Since occluded regions in an image are receiving most of their illumination from weak source lighting, they get larger errors which could be improved by increasing the illumination cut size. Note that due to the limited number of shading clusters and the granularity of our adaptive seeding method, our algorithm can under sample areas with small nearby geometric elements that produce occlusion. One example is the contact shadow regions at the bottom of the chair and table legs. From the magnified error images, it can be seen that these regions are computed with the highest errors compared to other regions. In practice, however, we found the perceived difference caused by such errors is relatively small, as can be observed by directly comparing (a) and (e). In Figure 6 (f-k) we perform similar accuracy comparisons for two additional scenes, where our results are rendered with both approximations enabled and with parameters specified in Table 1.

REFERENCES

[1] ARIKAN, O., FORSYTH, D. A., AND O'BRIEN, J. F. 2005. Fast and detailed approximate global illumination by irradiance decomposition.

[2] ACM Trans. Graph. 24, 3, 1108–1114.

[3] BALA, K., DORSEY, J., AND TELLER, S. 1999. Radiance inters polants for accelerated bounded-error ray tracing. ACM Trans. Graph. 18, 3, 213–256.

[4] DACHSBACHER, C., AND STAMMINGER, M. 2005. Reflective shadow maps. In Proc. SI3D '05, 203–231.

[5] GAUTRON, P., K˘RIV´ANEK, J., BOUATOUCH, K., AND PATTANAIK, S. N. 2005. Radiance cache splatting: A GPU-friendly global illumination algorithm. In Proc. EGSR '05, 55–64.

[6] HACHISUKA, T. 2005. GPU Gems 2 – High-Quality Global Illumination Rendering Using Rasterization. 615–634. HARRIS, M., SENGUPTA, S., AND OWENS, J. 2007. GPU Gems

[7] 3 – Parallel Prefix Sum (Scan) with CUDA. 851–876.

[8] HA˘SAN, M., PELLACINI, F., AND BALA, K. 2006. Direct-to indirect transfer for cinematic relighting. ACM Trans. Graph. 25, 3, 1089–1097.

[9] HA˘SAN, M., PELLACINI, F., AND BALA, K. 2007. Matrix Row-column sampling for the many-light problem. ACM Trans. Graph. 26, 3, 26:1–10.

[10] HOU, Q., ZHOU, K., AND GUO, B. 2008. BSGP: bulk synchronous GPU programming.

[11] ACM Trans. Graph. 27, 3, 1– 12.

[12] JENSEN, H. W. 2001. Realistic image synthesis using photon mapping. A. K. Peters, Ltd.

[13] KAJIYA, J. T. 1986. The rendering equation. In Proc. SIGGRAPH ’86, 143–150.

[14] KELLER, A. 1997. Instant radiosity. In Proc. SIGGRAPH ’97, 49–56.

[15] K˘RIV´ANEK, J., AND GAUTRON, P. 2005. Radiance caching for efficient global illumination computation. IEEE Trans. Visualization and Computer Graphics 11, 5, 550–561.

[16] NIJASURE, M., PATTANAIK, S. N., AND GOEL, V. 2005. Real-time global illumination on GPUs. Journal of Graphics Tools 10, 2, 55–71.

[17] PURCELL, T. J., DONNER, C., CAMMARANO, M., JENSEN, H. W., AND HANRAHAN, P. 2003. Photon mapping on programmable graphics hardware. In Proc. Graphics Hardware, 41–50.

[18] RITSCHEL, T., GROSCH, T., KIM, M. H., SEIDEL, H.-P., DACHSBACHER, C., AND

[19] KAUTZ, J. 2008. Imperfect shadow maps for efficient computation of indirect illumination. ACM Trans. Graph. 27, 5, 129:1–8.

[20] RITSCHEL, T., GROSCH, T., AND SEIDEL, H.-P. 2009. Approximating dynamic global illumination in image space. In Proc. SI3D ’09, to appear.

[21] SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: a many-core x86 architecture for visual computing. ACM Trans. Graph. 27, 3, 1–15.

[22] SHANMUGAM, P., AND ARIKAN, O. 2007. Hardware accelerated ambient occlusion techniques on GPUs. In Proc. SI3D ’07, 73– 80.

[23] SLOAN, P.-P., KAUTZ, J., AND SNYDER, J. 2002. Precomputed radiance transfer for real time rendering in dynamic, low frequency lighting environments. ACM Trans. Graph. 21, 3, 527 536.

[24] TOLE, P., PELLACINI, F., WALTER, B., AND GREENBERG, D. P. 2002. Interactive global illumination in dynamic scenes. ACM Trans. Graph. 21, 3, 537–546. WALD, I., KOLLIG, T.,

[25] BENTHIN, C., KELLER, A., AND SLUSALLEK, P. 2002. Interactive global illumination using fast ray tracing. In Proc. Euro graphics Workshop on Rendering, 15– 24.

[26] WALTER, B., DRETTAKIS, G., AND PARKER, S. 1999. Interactive rendering using the render cache. In Proc. Euro graphics Workshop on Rendering, 235–246.