UMass Technical Report UM-CS-2010-064

Importance-Driven Point Projection for GPU-based Final Gathering

David Maletz, Rui Wang, and John C. Bowers

University of Massachusetts Amherst

Abstract

We present a practical importance-driven method for GPU-based final gathering. We take as input a point cloud representing directly illuminated scene geometry; we then project and splat the points to microbuffers which store each shading pixel's occluded radiance field. We select points for projection based on importance, defined as each point's estimated contribution to a shading pixel. We calculate the splat sizes adaptively based on the importance values, requiring no separate hole filling step. Our method differs from existing work in that we do not perform hierarchical traversal of the points; instead, we use importance-driven point sampling, which is fast and suitable for fully dynamic scenes. In addition, our method makes it easy to incorporate other importance factors such as glossy to glossy reflection paths. We also introduce an image-space adaptive sampling method, which combines adaptive image subdivision with joint bilateral upsampling to robustly preserve fine details. We implement our algorithm on the GPU, providing high-quality rendering for dynamic scenes at near interactive rates.

1. Introduction

Interactive global illumination in dynamic scenes continues to present a great challenge in computer graphics. A popular technique in recent years is to perform final gathering from many point lights [WFA*05, HPB07]. The idea is to sample the scene's illumination as many virtual point lights (VPLs) [Kel97], then integrate the contributions from all VPLs to a shading pixel. Because the set of VPLs is the same for every shading points, gathering is fast and suitable for parallel computation. Moreover, approaches such as [Chr08, REG*09] adopt a single point cloud to represent both the illumination and geometry, enabling very fast visibility calculation using point-based rasterization.

In this paper, we present a novel method for GPU-based final gathering. As in [Chr08], we adopt a single point cloud to represent directly illuminated scene geometry. We project and splat these points into each pixel's microbuffer to compute the occluded incoming radiance field. Our main objective is to select points for projection based on importance – their approximated contributions to the shading pixel. Such an importance-driven method provides fast convergence speed and is suitable for GPU processing.

We start with random sampling to estimate the importance function, defined as each scene point's un-occluded contribution to a given shading pixel. We then draw points according to the importance function, and splat the selected points to the pixel's microbuffer. We compute the splat size adaptively using the importance value, providing an estimate of solid angle subtended by the splat. The adaptive splats eliminate the need for a separate hole filling step in the microbuffers. To quickly compute the importance function, we partition the scene points into spatial clusters. Points within each cluster are treated uniformly.

Our method differs from existing work such as [REG*09] in that we do not perform hierarchical traversal of points. Instead, we use importance-driven point sampling. This makes our method suitable for fully dynamic scenes while providing fast computation speed. Moveover, we can easily incorporate other importance factors such as glossy to glossy reflection paths, improving the efficiency when sampling such paths. To reduce the spatial sampling cost, we introduce an image-space adaptive sampling method. It combines adaptive image subdivision with joint bilateral upsampling to robustly preserve fine details and edges.

We have implemented our algorithms on modern GPUs, providing high-quality rendering of dynamic scenes at near interactive rates. We provide a progressive version of the renderer, which allows for fully interactive scene manipulation.

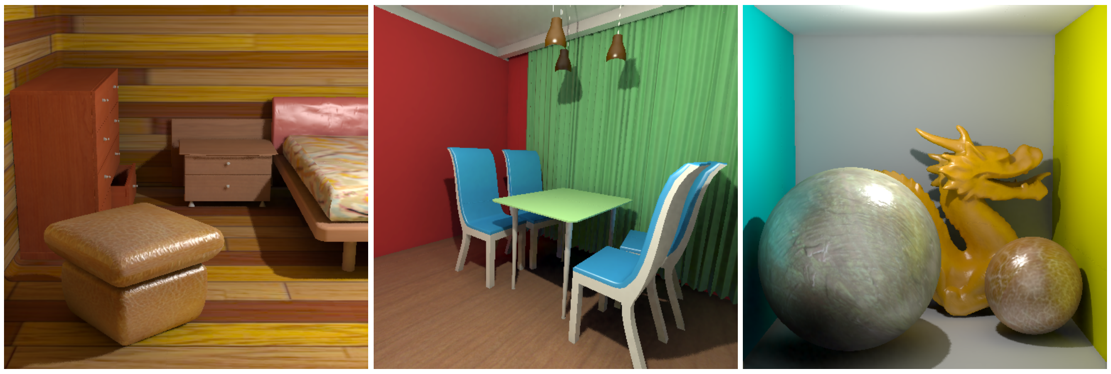

Figure 1: *Global illumination results using our algorithm. These images are rendered at* 512^2 *resolution and are computed in* $3 \sim 4$ *seconds on an NVIDIA 480 GTX. Note the realistic surface reflections and indirect shadows.*

The rendering quality typically converges in a second, while full frame rendering is achieved in $3 \sim 4$ seconds. Figure 1 shows several examples.

2. Related Work

Conventional methods for global illumination, such as Monte Carlo ray tracing or photon mapping, provide high-quality results offline, but are usually too expensive for interactive applications. Precomputation based methods, such as [SKS02], exploit offline computed datasets for fast online rendering, but provide limited support for dynamic scenes. As a user may wish to manipulate objects, light sources, and surface materials all at interactive rates, it is difficult to assume any prior knowledge about the scene.

Point-based Global Illumination. Extensive work has shown that point-based representations are very suitable for global illumination due to their simplicity and intrinsic parallelism. Instant radiosity [Kel97] first proposed to treat indirect illumination as a set of virtual point lights (VPLs). They compute visibility caused by VPLs using shadow mapping, which is feasible only for a small number of VPLs. Incremental instant radiosity [LSK*07] exploits temporal coherence and reuses VPLs over time to greatly improve speed; but it requires smooth movement of the light. Imperfect shadow maps [RGK*08] use a geometry point cloud to compute approximate shadow maps, significantly speeding up the visibility calculation with 1024 VPLs.

In [NPW10], screen-space voxelization is used to efficiently gather illumination from area lights, which are approximated by a small number of VPLs. Recently, [KD10] introduced a lattice-based structure to store light propagation volumes. These methods are aimed for rendering plausible global illumination effects in 3D games.

Lightcuts [WFA*05] represent indirect illumination as a hierarchical point cloud, allowing for efficient integration of all VPLs at sublinear cost. They resolve visibility by using ray tracing. Matrix row-column sampling [HPB07] clusters VPLs by sampling a matrix representing the contribution from each VPL to a subset of the shading pixels. Visibility is resolved using GPU shadow mapping. While fast, it uses the same set of VPL clusters for all pixels.

A technique proposed in [HKWB09] extends VPLs to virtual spherical lights (VSLs), which are suitable for scenes with many glossy materials. Most recently, [DKH*10] combine global and local lights to efficiently render high-rank illumination effects. These methods are accurate, but take a few minutes to render.

By converting polygons to surfels, [Bun05] approximate ambient occlusion on the GPU for dynamic scenes. They build a hierarchy of surfels to accelerate the computation. [Chr08] use a similar representation to compute final gathering in production quality renderings. A point cloud is created to represent directly illuminated scene geometry, and is then hierarchically rasterized to a shading pixel's microbuffer. Micro-rendering [REG*09] is based on a similar approach and achieves interactive rates by exploiting modern GPUs. They also introduced importance-warped [Jen95] microbuffers to efficiently handle glossy BRDFs.

Instead of building and traversing a full hierarchy of points, our goal in this paper is to present a different formulation that uses importance-driven point projection. The main benefits are its simplicity and improved support for dynamic scenes. Moreover, it allows the incorporation of both diffuse and glossy importance factors.

GPU-based Photon Mapping. Photon mapping [Jen01] is widely used to simulate multi-bounce indirect lighting. Since the first GPU-based photon mapper introduced in [PDC*03], a number of recent papers have demonstrated impressive results: [ZHWG08] presented a GPU-based kd-tree for interactive photon mapping; [ML09] introduced a fast image-space photon mapper for global illumination effects. These methods achieve fast computation speed by avoiding final gathering. Recently [WWZ*09] exploit sparse irradiance samples to reduce final gather cost. This approach is fast, but the sparsity of their irradiance samples is not suitable for scenes with complex geometric details.

Importance Sampling. Importance sampling reduces stochastic sampling noise by drawing samples from an importance function that approximates the integrand. An efficient importance function can be defined as the product of the illumination and BRDF. This is called bidirectional importance sampling [BGH05], which has been studied by much recent work [CJAMJ05, CETC06, CAM08, WÅ09]. These methods typically rely on ray tracing to compute the samples' visibility and thus remain offline; in contrast, we use microbuffers implemented on the GPU to allow for interactive rendering.

Caching and Interpolation. Indirect lighting is typically smooth, making it a good candidate for caching or adaptive sampling. [WRC88] progressively cache irradiance samples and reuse them during the computation. [TPWG02] proposed an object-space caching method suitable for dynamic scenes, but require parameterized objects. To reconstruct an image from sparse samples, joint bilateral upsampling [SGNS07] provides efficient nonlinear interpolation that prevent blurring from crossing feature edges. However, a regular grid sampling pattern can lead to loss of fine details around small geometric features. Our method combines joint bilateral upsampling with adaptive image subdivision, which can robustly preserve fine details and features. Moreover, we consider both local geometric changes as well as radiance changes computed on the fly. This produces better sampling on glossy materials compared to techniques such as [WWZ*09], which only consider geometric changes.

3. Algorithms

3.1. Overview

Point Representation. We use $\mathcal{S}$ to denote a point cloud sampled from a scene with directly illuminated radiance. Each point is associated with a position, normal, delta surface area, and illumination radiance. The points can be generated using a variety of methods, such as random sampling, micropolygon subdivision, photon mapping. In our case, we use the real-time Poisson disk surface sampling algorithm provided by [BWWM10] to generate points on scene surfaces. Due to the uniform distribution, all points are assigned the same delta surface area. Each point can emit diffuse as well as glossy radiance.

Microbuffers. Our goal is to project the points to each shading pixel's microbuffer, where we store the depth and color of the closest projection points. Essentially this is using point rasterization to compute a small environment map at a shading pixel representing its incoming radiance field. Since we only need to represent the upper hemisphere, we use a hemi-octahedral map [WNLH06] to store the microbuffer as a 32×32 texture. Each pixel in the map is associated with a direction and a delta solid angle, and all pixels together cover a 2π solid angle.

Stochastic Sampling. To obtain the microbuffer, we could project all points. But this brute-force solution is too slow as there are tens of thousands of points. It is also not necessary as the microbuffer size (32×32) is much smaller than the total number of points. To avoid projecting all points, we draw samples stochastically from $\mathcal{S}$. A simple idea is to pick a point from $\mathcal{S}$ uniformly randomly, thus every point has an equal probability $p = \frac{1}{|\mathcal{S}|}$ to be selected ($|\mathcal{S}|$ is the point set size). However, this usually leads to a very noisy image.

Clearly a more efficient way is to importance sample the points. First, every point is assigned an importance value, defined as its estimated contribution to a given shading pixel. This is the importance PDF function, which we use to draw samples from $\mathcal{S}$. As an example, we can define importance as a point's projected solid angle to a shading pixel. Intuitively, this ensures that points subtending large solid angles are more likely to be selected for projection, while those subtending small solid angles are selected less frequently.

3.2. Importance-Driven Point Splatting

Clustering Points. Since the importance function has to be created for every shading pixel, evaluating it over all points is impractical. Instead, we partition the points into clusters, and assign a single importance value per cluster. Points within each cluster are treated with equal importance. This essentially approximates the importance function as piecewise constants, greatly reducing the evaluation and sampling cost. This approach is inspired by [WÅ09], where they partition lights into clusters driven by the points' diffuse illumination energy. Unlike them, we do not consider illumination when clustering points, because our points are not necessarily diffuse; moreover, since our point cloud represents both illumination and geometry, a point that carries no diffuse energy may still be important as it can occlude other points.

We partition the points using a simple spatial clustering. We typically create 512 clusters, using a process similar to building a kd-tree, except we only need to build the first 9 levels. Once the clusters are created, they are treated independently.

Evaluating Per-Cluster Importance. Next, we need to estimate the importance of each cluster. We do so by drawing N_{rnd} random points from the cluster, and computing the sum of their individual importance, defined as the projected solid angle of the point.

$$p_k = \sum_{\mathbf{s}} \frac{\max(\cos\theta_{\mathbf{p}}, 0) \cdot |\cos\theta_{\mathbf{s}}|}{|\mathbf{s}-\mathbf{p}|^2} \Delta A_{\mathbf{s}} \tag{1}$$

where $\mathbf{p}$ is the a shading pixel being considered, p_k is the estimated importance for cluster $\mathcal{C}_k$, $\mathbf{s}$ is a random point selected from $\mathcal{C}_k$, $\Delta A_{\mathbf{s}}$ is the point's delta surface area. See Figure 3(a)(c) for illustration. Note that the absolute value of

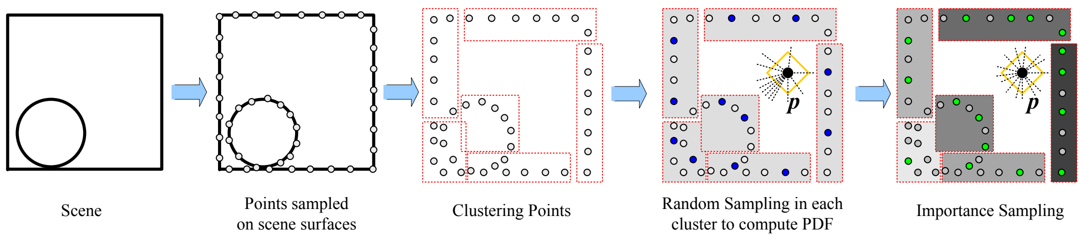
Scene
Points sampled on scene surfaces
Clustering Points
Random Sampling in each cluster to compute PDF
Importance Sampling
p
p

Figure 2: *Algorithm overview. Scene surfaces are sampled as many points, which are then clustered. At a shading pixel* **p**, *we estimate per-cluster importance by drawing random points (blue dots); we then treat the importance as a PDF to draw importance samples (green dots). Both random and importance samples are projected into the microbuffer (yellow diamond). Box shade indicates cluster importance.*

$\cos\theta_{\mathbf{s}}$ is taken because a point facing away from **p** still contributes importance, as it can cast indirect shadows. We typically use $N_{rnd} = 2$ random points per cluster, which works well in practice, and is fast to compute.

Importance Sampling. Once we have the per-cluster importance p_k, we normalize it to become a PDF. To draw samples, we could convert it to a CDF. Then we produce a uniform random number, and use binary search to find where it falls in the CDF. This results in a cluster number. Finally we pick a random point from that cluster since all points within the same cluster are treated equally.

While this is the standard way to perform importance sampling, it is not very GPU-friendly, especially the binary search step. Therefore we propose an alternative method suitable for parallel processing. Given a budget of N_s samples, our goal is to assign a certain number of samples per cluster, proportional to the cluster importance. We start from the first cluster $\mathcal{C}_1$, and compute its expected number of samples $N_s \cdot p_1$. The integer portion of it, $\lfloor N_s \cdot p_1 \rfloor$, is the guaranteed number of samples. We then produce a uniform random number, and compare it with the fractional part of $N_s \cdot p_1$ to decide whether an additional number will be assigned. If it is assigned, we have 'overdrawn', and we discount the overdrawn portion from the next cluster; otherwise, we have a 'surplus', which we deposit to the next cluster. The pseudo code is listed below. This algorithm is fast and ensures exactly N_s samples will be drawn from all clusters, while reducing thread divergence on the GPU.

Projection and Adaptive Splat Size. After selecting a

```
for (carry = 0.0,  k = 0; k < # clusters; k ++) {
    f_k = N_s * p_k + carry;  n_k = (int) max(f_k, 0);  f_k = f_k - n_k;
    if (random() < f_k)  { n_k ++;  carry = f_k – 1.0; }
    else  { carry = fk; }
    // draw n_k random points from this cluster
}
```

point, we project it to the microbuffer, and splat its color if the center of the splat succeeds a depth test. To project, we map the line-of-sight direction $(\mathbf{s}-\mathbf{p})$ to the microbuffer using hemi-octahedral mapping. To do so, we first transform $(\mathbf{s}-\mathbf{p})$ to the local coordinates defined by the shading pixel's normal. Denote this transformed direction ω. We then compute:

$$\omega' = \frac{\omega}{|\omega.x| + |\omega.y| + |\omega.z|}, \quad \text{and} \begin{cases} t_x = \frac{1-\omega'.x+\omega'.z}{2} \\ t_y = \frac{1-\omega'.x-\omega'.z}{2} \end{cases} \tag{2}$$

The resulting $[t_x, t_y] \in [0,1]^2$ are the normalized 2D coordinates corresponding to a pixel location in the microbuffer.

The splat size (the number of microbuffer pixels covered by the projection) has to be carefully computed in order to minimize holes. Assume that a sample **s** comes from cluster $\mathcal{C}_k$: since an expected number of $N_s \cdot p_k$ samples will be drawn from this cluster, each sample shares $\frac{1}{N_s \cdot p_k}$ of the total surface area of the cluster. Therefore, we can estimate the solid angle represented by **s** as:

$$\Omega_{\mathbf{s}} = \frac{|\mathcal{C}_k| \cdot \Delta A_s}{N_s \cdot p_k} \cdot \frac{|\cos\theta_{\mathbf{s}}|}{|\mathbf{s}-\mathbf{p}|^2} \tag{3}$$

We further divide $\Omega_{\mathbf{s}}$ by the delta solid angle that the microbuffer pixel at $[t_x, t_y]$ represents, and the result gives the total number of pixels covered by the splat. We clamp the number to 1 if it is less. Finally, the splat color and depth value are both written to the microbuffer in a square region covered by splat size.

Discussion. Note that the cluster importance p_k appears in the denominator of Eq. 3. This make sense intuitively, since a cluster with small importance is less likely to be sampled, thus any sample drawn from it must represent a large support size. Refer to Figure 4 for illustration. This is analogous to Monte Carlo importance sampling, where an unbiased estimator must divide the probability of drawing a sample.

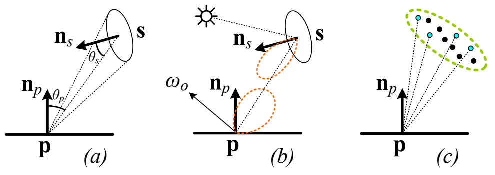

Figure 3: *(a) for diffuse surfaces, the importance of a point* **s** *is defined as its projected solid angle; (b) for glossy surfaces, we additionally include the BRDF into the importance value; (c) cluster importance is estimated by random sampling within the cluster.*

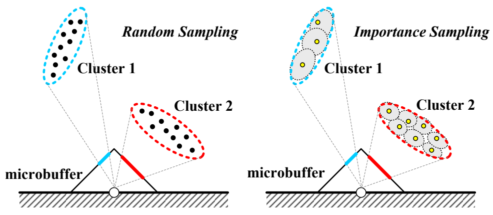

Figure 4: *During random sampling, cluster 1 is found to have small importance, thus is sampled infrequently during importance sampling. Each sample drawn from it represents a large support size.*

Glossy Importance. Because we estimate importance using point sampling, it is easy to include other factors that influence the importance value. For example, for a glossy shading point, we should give higher importance to clusters that fall close to the reflected viewing direction. This can be achieved by simply adding a BRDF term f_r to Eq. 1, which now becomes:

$$p_k = \sum_{\mathbf{s}} \frac{\max(\cos\theta_{\mathbf{p}}, 0) \cdot |\cos\theta_{\mathbf{s}}|}{|\mathbf{s}-\mathbf{p}|^2} f_r(\mathbf{s} \to \mathbf{p}, \omega_o)\Delta A_{\mathbf{s}} \qquad (4)$$

Refer to Figure 3(b). In addition, we can further multiply the importance by the strength of glossy reflection at the source point **s**. This allows us to efficiently include glossy-to-glossy reflection paths.

Note, however, because we use a small set of random points per cluster to sample the importance, our approach is only suitable for moderately glossy BRDFs, such as Phong with exponent less than 50. Highly glossy BRDFs require proper filtering to avoid aliasing, or should perhaps be handled with an entirely different approach.

Summary. Figure 2 summarizes the steps of our algorithm. Note that the points drawn from the random sampling step are also projected into the microbuffer. This makes better use of the computation already required to evaluate these random sample points. The splat size for a random point can be computed from Eq. 3 accordingly, with p_k equal to the inverse of the number of clusters.

In essence, the random sampling and the subsequent importance sampling can be seen as *exploration* and *exploitation* in the context of reinforcement learning. While this idea has been previously explored by [HPB07, HKWB09] for final gathering, they compute a single set of lights and use it for all pixels. In contrast, we sample and evaluate each pixel independently. In addition, we use microbuffers instead of ray tracing to quickly resolve visibility, achieving fast rendering speed.

Comparing to a hierarchical approach such as [REG*09], we note that our method uses a fixed clustering of scene points. However, by assigning different importance values to each cluster, we can achieve similar efficiency as a hierarchical method. Therefore we provide a viable alternative to existing techniques.

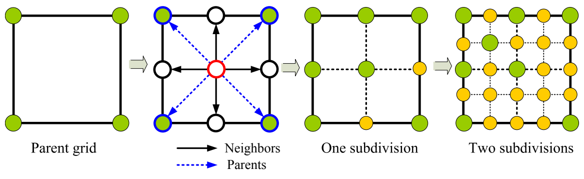

Figure 5: *Adaptive sampling. Green indicates sampled pixels; orange indicates interpolated pixels.*

3.3. Image-Space Adaptive Sampling

Even though our importance-driven point splatting is fast, computing it for every shading pixel is still very expensive, especially for high-resolution, anti-aliased images. Fortunately the indirect lighting in most scenes tends to be smooth, so we can exploit spatial coherence by computing final gathering only at a subset of pixels. These sparsely sampled pixels are then used to reconstruct the other pixels using joint bilateral upsampling [SGNS07]. To select sample pixels, a straightforward way is to use a regular sampling grid, such as 4×4. While this works well in simple scenes, we found it often necessary to perform further sampling around edges and places with small geometric details.

To address this problem, we propose to compute adaptive samples instead. To begin, we rasterize a G-buffer, storing each pixel's position $\mathbf{p}_i$ and normal $\mathbf{n}_i$. We then use a top-down subdivision, starting with pixels on an initial 4×4 sampling grid. Once these pixels are shaded, we subdivide and examine pixels on the 2×2 grid. For each pixel on this grid, we compute the following coherence metric:

$$\max_i \left(\frac{|\mathbf{p}_i - \mathbf{p}_c|}{d/2} + \sqrt{2-2(\mathbf{n}_i \cdot \mathbf{n}_c)} \right) + \lambda \sum_{j,k} |L_o(\mathbf{p}_j) - L_o(\mathbf{p}_k)|$$

where $\mathbf{p}_c$ and $\mathbf{n}_c$ are the position and normal of the current pixel, i loops over its four neighbor pixels on the grid, and d is the length of the scene's bounding box diagonal. Here j and k are pairs of two out of the four parent pixels on the 4×4 grid, and L_o is the radiance computed at the parents pixels. The first term evaluates local geometric changes, the second term evaluates radiance changes, λ weighs the relative importance of the two. We found $\lambda = 8$ to work well practice. Refer to Figure 5 for illustration of the samples. Note that including the radiance term L_o is important for the cases where geometric changes are low but radiance changes are hight, such as glossy highlights on a plane.

If the coherence metric is larger than a given threshold ($\varepsilon = 0.3$), the current pixel must be sampled; otherwise it will be interpolated from the four parents. We perform this checking for every pixel independently, except for those that are already sampled in previous passes. We then use a parallel list compaction to collect all pixels that require sampling at the current subdivision level, and launch GPU threads to shade these pixels in parallel. Next, we launch GPU threads again to interpolate the un-sampled pixels, using joint bilateral upsampling [SGNS07].

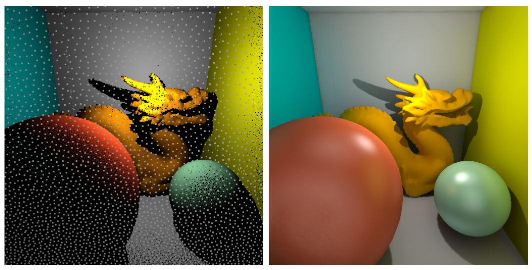

Figure 6: *(L) point cloud (256K points) with direct lighting; (R) global illumination result by gathering from the points.*

We repeat the process for the remaining levels until the grid size becomes 1, or a desired subpixel size if anti-aliasing is turned on. Note that the error threshold ε should be scaled by 2 every time we go down a level. Because the pixels are checked independently in each pass, even if a pixel is interpolated in a previous pass, it can still be requested for sampling at later pass. Therefore our method can robustly preserve edges and small geometric details in the rendering. Figure 11 shows the adaptive samples selected during convergence. Our results show that that samples can quickly capture edges and areas with glossy reflections or under indirect shadows, thus providing improved convergence speed.

4. Implementation Details

We implemented our algorithms on the GPU using NVIDIA CUDA 3.0. We make use of several parallel computation primitives such as global sorting and list compaction, all of which are available in CUDPP 1.1. For random numbers, we precompute a texture storing 4K×4K random numbers and reuse them on the fly.

Scene Points. For all scenes we generated 256K points uniformly distributed on the surfaces. The number of points is sufficient for our test scenes. We have also tried 1M points, which did not produce any observable improvement in the rendering quality. We use a real-time GPU-based Poisson disk sampling algorithm provided in [BWWM10] to generate these points. For each point we store its position as 3 floats, and its normal, diffuse color, specular color, and Phong exponent as bytes. We also store its triangle ID and bary coords, which are used to update the point upon scene manipulation. Unless an object undergoes significant deformation, we do not need to re-generate scene points every frame; instead, we simply update them using their bary coords. Figure 6 shows an example set of points computed for the Cornell box scene.

Materials. For simplicity we currently only support diffuse and Phong BRDFs. It is straightforward to include other BRDFs, as our algorithm makes no assumption about the specific type of BRDF. We also support bump maps and spatially varying BRDF parameters defined using textures. We do, however, restrict the glossiness of the BRDFs to be generally below 50, as highly glossy BRDFs should be handled using a different approach such as ray tracing.

Primary Lights. We allow multiple omni-directional lights as primary light sources of the scene. For each light we rasterize a cube shadow map at $6{\times}512^2$ resolution and use it to compute shadowed direct lighting at both the scene points and the shading pixels.

G-buffer. At the beginning of each frame, we use a shader to rasterize a G-buffer containing each shading pixel's position, normal, and material properties. The data is copied to a pixel buffer object (PBO), which is then bound to CUDA for further processing.

Clustering Scene Points. As described in Section 3.2, we partition the scene points into 512 clusters in order to quickly estimate the importance PDF. To do so, we first apply a median split on the points' position along the longest axis. We then repeatedly split each sub-cluster along a different axis until we have 512 clusters. This is similar to building a kd-tree of the points. We perform each level of split on the GPU using a segmented global sort. This ensures that points within the same cluster remain in that cluster during global sort. Each cluster in the end contains the same number of points. The overall cost of this step is less than 10ms.

Microbuffers. We use 32×32 microbuffers, and compute each microbuffer in a single CUDA block. This allows the entire microbuffer to be stored in shared memory, providing fast access speed. The microbuffer itself requires 3 unsigned chars for storing each of the diffuse and glossy incoming radiance, and 1 ushort for storing depth. The total is 8KB per microbuffer. The reason to separate glossy from diffuse glossy radiance is for better interpolation in textured areas. In those areas, direct color interpolation will lead to loss of texture details. Instead, we interpolate the diffuse and glossy incoming radiance separately, and multiply them with the textured reflectance values to produce the final color.

To estimate per-cluster importance, we launch 512 threads, corresponding to 512 clusters. Each thread k independently draws 2 random points from its cluster $\mathcal{C}_k$ and evaluates p_k (Eq.4). These points are immediately projected to the microbuffer. After this step, thread 0 goes through the PDF and assigns the number of importance samples to each cluster, using the algorithm described in Section 3.2. Next, each thread independently takes 4 budgeted samples, and projects them to the microbuffer using splat sizes estimated via Eq. 3. So in total we project (2+4)×512=3K points to the microbuffer. Every time a point is projected to the microbuffer, we use `atomicMin` to update the color and depth values, ensuring correctness upon concurrent writes. Finally we multiply each microbuffer pixel with the BRDF of the shading point, and perform a parallel reduction to return the total reflected color. The whole algorithm maintains high parallelism with little thread divergence.

For robustness when calculating delta solid angles, we use

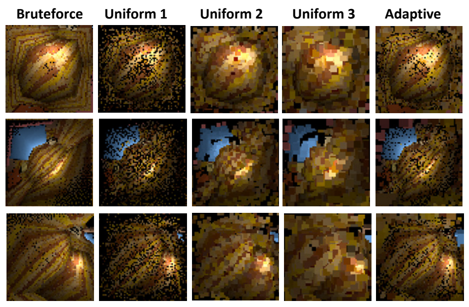

Figure 7: *Comparing microbuffers generated with our adaptive splat sizes (right), non-adaptive (uniform) splats of 1, 2, and 3 pixels, and a brute-force reference (left).*

an approximated disk-to-point solid angle formula:

$$\Omega_{\mathbf{s}} = \frac{2\pi A_{\mathbf{s}} \cos\theta_{\mathbf{s}}}{A_{\mathbf{s}} + 2\pi|\mathbf{s} - \mathbf{p}|^2} \tag{5}$$

This prevents $\Omega_{\mathbf{s}}$ from becoming arbitrarily large when $\mathbf{p}$ and $\mathbf{s}$ are very close to each other.

Glossy-Glossy Reflections. Since our approach is based on point sampling, we can achieve glossy-to-glossy reflections by performing a direct lighting calculation at a scene point when it is requested for projection. Clearly this will increase the computation time. In practice, we assume that there are no more than 4 primary light sources and hence we can store up the glossy reflected lobes together with the scene points. To do so, we run a separate pass that calculates shadowed direct lighting, including a diffuse radiance and up to 4 glossy reflected lobes. These are then used during point projections.

Multi-Bounce Indirect Lighting. We enable multi-bounce indirect lighting by treating the scene points as shading points, and use the same microbuffer algorithm to update the diffuse radiance at each point. This allows us to include an additional bounce of indirect illumination in the final shading. Note, however, we do not account for multi-bounce glossy-to-glossy reflections, as doing so would cause the number of glossy lobes to increase exponentially. Instead, we only update the diffuse radiance at the scene points during multi-bounce calculation. Nonetheless, this approach provides satisfactory results in most cases.

Image-space Adaptive Sampling and Interpolation. To balance work among threads, the adaptive sampling is split into several steps. The first step evaluates which pixels need to be sampled in the current frame, compacts them, and launches CUDA threads to compute their microbuffers in parallel. The results are written into an output buffer, and made available to the next step through a thread synchronization. After all samples are computed, we go through all levels and interpolate pixels that are currently un-sampled.

Progressive Rendering. With adaptive sampling, we can easily enable progressive rendering to improve user interaction speed. During adaptive sampling, the user can manipulate any part of the scene, and the current adaptive sampling step will be interrupted to provide interaction feedback. As soon as the user stops moving, the program will continue to sample the remaining pixels, allowing the full indirect light buffer to be filled in over time. Typically the rendering quality converges in a second, while full frame rendering (i.e. every pixel is shaded) takes $3 \sim 4$ seconds.

When high quality rendering is needed, we enable super-sampling to provide anti-aliasing. In this case, the sampling will proceed to subpixel level. Due to adaptive sample, the most importance pixels (usually those around edges, indirect shadows, and glossy objects) are selected and shaded first. Thus the rendering usually converges quickly within a few seconds, long before the full frame rendering is completed. This allows the rendering cost to grow sublinearly with respect to the total number of super-sampling pixels. To keep track of the status of each pixel, the alpha component is set to 1 for sampled pixels and 0 for interpolated pixels. This allows the program to overwrite interpolated pixels in subsequent passes.

Post-process Filter. We apply a 7×7 joint bilateral filter on the computed indirect lighting buffer as a post-process step. This generally reduces stochastic sampling noise and works very effectively for diffuse scenes. For scenes with many glossy objects, the sampling noise is usually more pronounced, but we cannot use a larger filter to attenuate noise because that would blur out reflection details. Instead, we reduce the filter to 3×3 in size, and rely on anti-aliasing to provide a higher quality image.

5. Results and Discussions

Our results are tested on a PC with Intel Core i7-920 CPU and NVIDIA 480 GTX graphics card. All GPU programs are compiled using CUDA 3.0. Unless specified otherwise, all images and videos are captured at a default resolution of 512×512 without anti-aliasing. The supplemental material includes executable demos which can run on both high-end and low-end graphics cards.

For high-quality rendering, we turn on $2\times$AA, which provides nicer quality but reduces performance by $2\sim3$ times. All scenes are computed with $|\mathcal{S}|$=256K scene points, partitioned into 512 clusters. The microbuffer size is 32×32.

Performance. By default we turn on progressive rendering mode to provide better user interaction. The rendering speed is about $2\sim3$ fps, and the full frame rendering is achieved in $3\sim4$ seconds. With $2\times$AA, the frame rate drops to 1 fps. Note that due to our adaptive sampling, the image quality typically converges in a few seconds, and hence full frame rendering is not necessary to obtain a high-quality rendering.

Since we use point-based illumination, the rendering performance is generally insensitive to the scene complexity, although the indirect shadows can take longer to converge

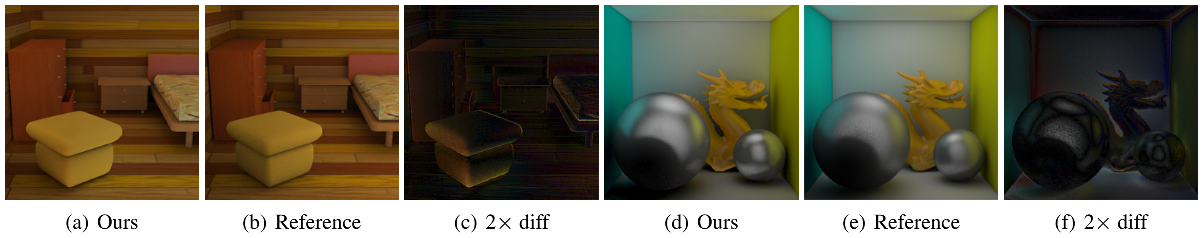

Figure 8: *Comparison of indirect lighting result computed using our method vs. a ray traced reference. For each example we show a 2× difference image to highlight the places of errors.*

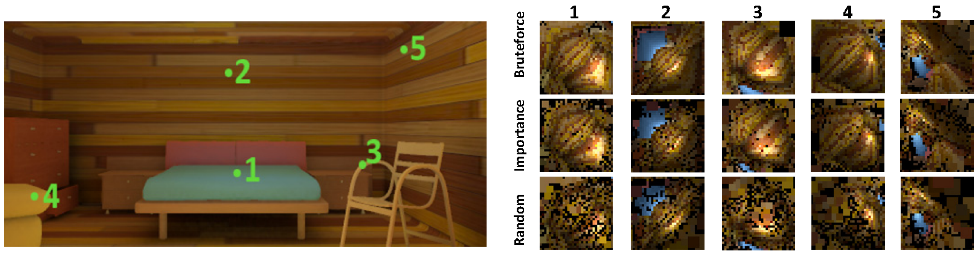

Figure 9: *Comparing microbuffers generated using importance-driven vs. random projection, both with 3K projected points.*

in scenes with high depth complexity. Figure 1 shows three example images, which are all captured at 3 ∼ 4 seconds of rendering. Note the realistic surface reflections and the indirect shadows.

Microbuffers. We first examine the quality of the microbuffers generated. Our reference is a brute force solution that projects all scene points into the microbuffer. In Figure 9 we show 5 selected pixels from the bedroom scene. For each point, we compare our importance-driven point projection vs. purely random point projection. Both are computed by drawing 3K samples from the scene points. Note how the importance-driven method better matches the reference, while the random method leaves many gaps that need to be filled with large adaptive splats.

To see how effective the adaptive splats are, in Figure 7 we compare microbuffers generated using our adaptive splats sizes vs. non-adaptive (uniform) splats of 1, 2, and 3 pixels. Small uniform splats (such as 1) do not cover the microbuffers well, leaving lots of holes; large uniform splats (such as 3) overfills the buffer, causing one splat to spill to adjacent pixels. In contrast, adaptive splats can effectively overcome these issues.

Validation. To verify our algorithm, we used ray tracing instead of microbuffers to generate reference images of two scenes. Results are shown in Figure 8. The bedroom scene contains only diffuse objects, while the Cornell box scene contains both diffuse and glossy objects. We only show indirect lighting. Note that our result looks qualitatively similar to the reference. Some differences are observable, mainly in the tone of the color and the indirect shadows around edges. The color difference is caused by our color quantization (i.e. 8-bit), and the indirect shadow difference is primarily caused by the limited resolution of the microbuffers. We also show a 2× difference image on the right, which highlights the places of errors.

Image adaptive sampling. Our adaptive sampling algorithm considers both local geometric changes and luminance changes, thus it efficiently budgets samples in areas of sharp features or strong radiance changes. In Figure 11 we show the adaptive sample points selected at different time stamps during the rendering. Note how the samples quickly capture area of depth discontinuities as well as indirect shadows, making the sampling process more efficient.

Multi-bounce Indirect Lighting. We enable multiple diffuse bounces by applying the same final gathering algorithm on the scene points. The final bounce of reflection (at shading pixels) is still glossy. It generally takes several seconds to update the diffuse radiance at all scene points. Once it's done, the user can change viewpoints at will, but moving the light source, objects, or changing material parameters will incur a new round of scene points update. Figure 13 show our results for the dining room scene and the Cornell box scene. Note how the additional bounce adds more color saturation, and brightens up the indirect shadow regions.

Glossy-to-glossy Reflections. As we allow the scene points to carry glossy reflected lobes, we enable glossy-glossy reflection effects. Figure 12 shows a comparison between rendering with the glossy components at the scene points enabled vs. disabled. Note the difference in the highlights reflected from the teapot. **Scene Manipulation.** Our algorithm supports dynamic BRDF editing with bump maps and spatially varying BRDF parameters defined by textures. Figure 10 shows the teapot-torus scene edited in real-time with several different materials. We also support online manipulation of scene objects. Upon manipulation, the direct lighting radiance at every scene point will be updated, and we re-

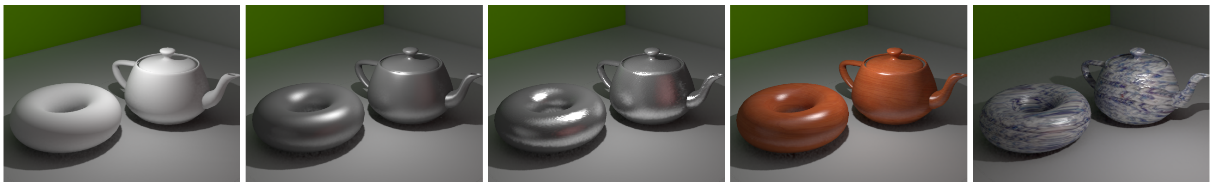
Figure 10: *The teapot-torus scene edited in real-time with several different materials, textures and bumpmaps.*

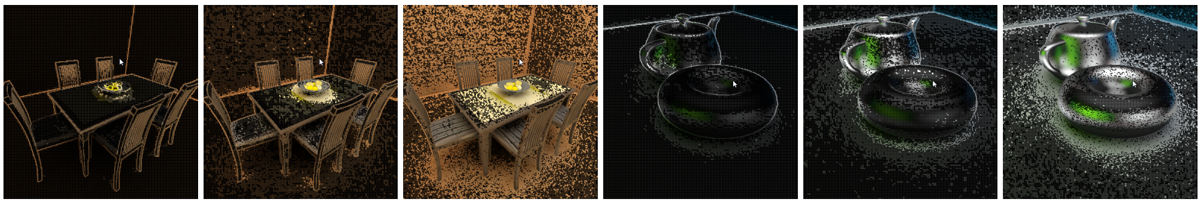
Figure 11: *Image adaptive samples selected at different time stamps as the rendering progresses. Note how the samples quickly capture areas of depth discontinuities and radiance changes (such as indirect shadows and glossy highlights).*

cluster the points. These two steps take less than 12ms. Figure 14 shows an example where the objects in the Cornell box scene were moved at run-time. The same figure shows lighting changes – i.e. move the primary light. All edits were performed and captured online.

6. Limitations and Future Work

To summarize, we have presented an efficient method using importance-driven point projection for GPU-based final gathering. Our method is fast, suitable for fully dynamic scenes, and provides a viable alternative to existing work. Our method is not real-time yet, so it is not immediately applicable to 3D games. However, it serves as useful and practical tool for design applications, as it provides realistic rendering in just a couple of seconds. Like other point-based methods, when dealing with highly glossy materials, the number of scene points can become a limiting factor, which eventually will lead to reflection aliasing artifacts.

There are several directions to explore for future work. First, we plan to include BRDF-warped microbuffers, similar to [REG*09]. This would be a straightforward change since we do not require any particular parametrization of the buffers. Second, we will study how our importance-driven framework can be used to simulate other effects such as translucency. Finally, we would like to explore features

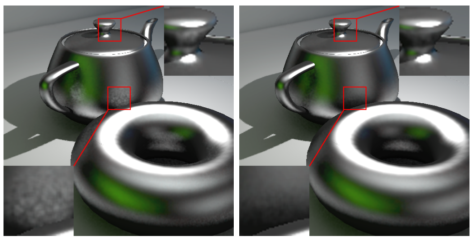
Figure 12: *Renderings with glossy to glossy reflections enabled vs. disabled.*

available in the future generation of GPUs to further improve the speed of our approach.

Supplemental Material. The supplemental material contains an executable demo for our algorithm. The minimum requirement to run the demo is a PC with CUDA-capable GPU (compute capability 1.1 or above).

References

[BGH05] BURKE D., GHOSH A., HEIDRICH W.: Bidirectional importance sampling for direct illumination. In *Proceedings of Eurographics Symposium on Rendering* (2005), pp. 147–156. 3

[Bun05] BUNNEL M.: Dynamic ambient occlusion and indirect lighting. *GPU Gems 2 2* (2005), 223–233. 2

[BWWM10] BOWERS J., WANG R., WEI L.-Y., MALETZ D.: Parallel poisson disk sampling with spectrum analysis on surfaces. *ACM Trans. Graph. 29*, 5 (2010), to appear. 3, 6

[CAM08] CLARBERG P., AKENINE-MÖLLER T.: Practical product importance sampling for direct illumination. *Computer Graphics Forum 27*, 2 (2008), 681–690. 3

[CETC06] CLINE D., EGBERT P. K., TALBOT J. F., CARDON D. L.: Two stage importance sampling for direct lighting. In *Proceedings of Eurographics Symposium on Rendering* (2006), pp. 103–113. 3

[Chr08] CHRISTENSEN P.: *Point-based Approximate Color Bleeding*. Tech. rep., Pixar Technical Memo #08-01, 2008. 1, 2

[CJAMJ05] CLARBERG P., JAROSZ W., AKENINE-MÖLLER T., JENSEN H. W.: Wavelet importance sampling: efficiently evaluating products of complex functions. *ACM Trans. Graph. 24*, 3 (2005), 1166–1175. 3

[DKH*10] DAVIDOVIC T., KRIVANEK J., HASAN M., SLUSALLEK P., BALA K.: Combining global and local lights for high-rank illumination effects. *ACM Trans. Graph. 29*, 5 (2010), to appear. 2

[HKWB09] HAŠAN M., KŘIVÁNEK J., WALTER B., BALA K.: Virtual spherical lights for many-light rendering of glossy scenes. *ACM Trans. Graph. 28*, 5 (2009), to appear. 2, 5

[HPB07] HAŠAN M., PELLACINI F., BALA K.: Matrix row-column sampling for the many-light problem. *ACM Trans. Graph. 26*, 3 (2007), 26. 1, 2, 5

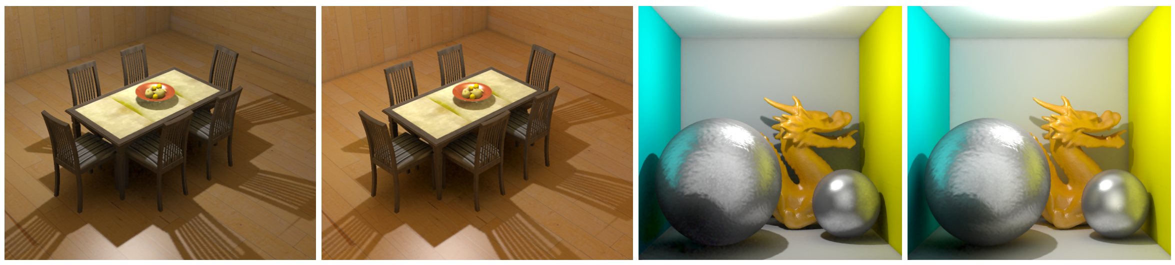

Figure 13: *The dining room scene and the cornell box scene. Here we compare global illumination with one and two bounces of indirect lighting. Note how the additional bounce adds more color bleeding and brightens up the indirect shadow regions.*

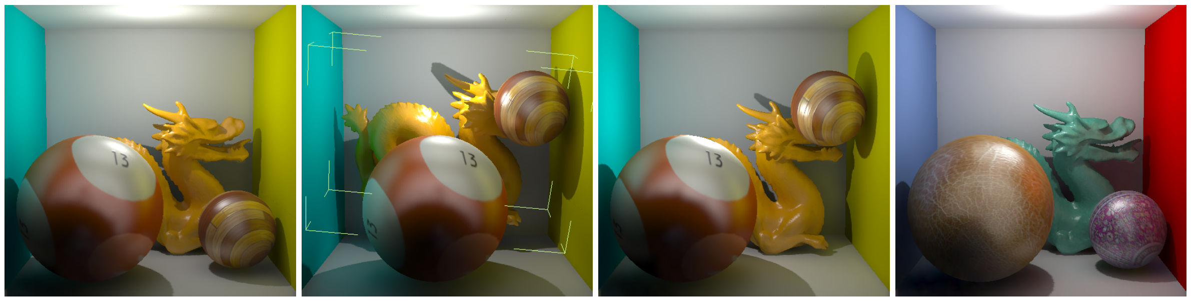

Figure 14: *his example shows online scene manipulation, including object, lighting, and material changes.*

[Jen95] JENSEN H. W.: Importance driven path tracing using the photon map. In *Proc. of Eurographics Rendering Workshop* (1995), pp. 326–335. 2

[Jen01] JENSEN H. W.: *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd., 2001. 2

[KD10] KAPLANYAN A., DACHSBACHER C.: Cascaded light propagation volumes for real-time indirect illumination. In *Proceedings of I3D* (2010), pp. 99–107. 2

[Kel97] KELLER A.: Instant radiosity. In *Proc. of SIGGRAPH '97* (1997), pp. 49–56. 1, 2

[LSK*07] LAINE S., SARANSAARI H., KONTKANEN J., LEHTINEN J., AILA T.: Incremental instant radiosity for real-time indirect illumination. In *Proceedings of Eurographics Symposium on Rendering* (2007), pp. 277–286. 2

[ML09] MCGUIRE M., LUEBKE D.: Hardware-accelerated global illumination by image space photon mapping. In *Proc. of HPG '09* (2009), pp. 77–89. 2

[NPW10] NICHOLS G., PENMATSA R., WYMAN C.: Interactive, multiresolution image-space rendering for dynamic area lighting. *Computer Graphics Forum 29*, 4 (2010), 1279–1288. 2

[PDC*03] PURCELL T. J., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *Proc. of Graphics Hardware* (2003), pp. 41–50. 2

[REG*09] RITSCHEL T., ENGELHARDT T., GROSCH T., SEIDEL H.-P., KAUTZ J., DACHSBACHER C.: Micro-rendering for scalable, parallel final gathering. *ACM Trans. Graph. 28*, 5 (2009), 1–8. 1, 2, 5, 9

[RGK*08] RITSCHEL T., GROSCH T., KIM M. H., SEIDEL H.-P., DACHSBACHER C., KAUTZ J.: Imperfect shadow maps for efficient computation of indirect illumination. *ACM Trans. Graph. 27*, 5 (2008), 1–8. 2

[SGNS07] SLOAN P.-P., GOVINDARAJU N. K., NOWROUZEZAHRAI D., SNYDER J.: Image-based proxy accumulation for real-time soft global illumination. In *Proc. of Pacific Graphics 07* (2007), pp. 97–105. 3, 5

[SKS02] SLOAN P.-P., KAUTZ J., SNYDER J.: Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Trans. Graph. 21*, 3 (2002), 527–536. 2

[TPWG02] TOLE P., PELLACINI F., WALTER B., GREENBERG D. P.: Interactive global illumination in dynamic scenes. *ACM Trans. Graph. 21*, 3 (2002), 537–546. 3

[WÅ09] WANG R., ÅKERLUND O.: Bidirectional importance sampling for unstructured direct illumination. *Comput. Graph. Forum 28*, 2 (2009), 269–278. 3

[WFA*05] WALTER B., FERNANDEZ S., ARBREE A., BALA K., DONIKIAN M., GREENBERG D. P.: Lightcuts: a scalable approach to illumination. *ACM Trans. Graph. 24*, 3 (2005), 1098–1107. 1, 2

[WNLH06] WANG R., NG R., LUEBKE D., HUMPHREYS G.: Efficient wavelet rotation for environmentmap rendering. In *Proc. of Eurographics Symposium on Rendering* (2006), pp. 173–182. 3

[WRC88] WARD G. J., RUBINSTEIN F. M., CLEAR R. D.: A ray tracing solution for diffuse interreflection. In *Proc. SIGGRAPH '88* (1988), pp. 85–92. 3

[WWZ*09] WANG R., WANG R., ZHOU K., PAN M., BAO H.: An efficient gpu-based approach for interactive global illumination. *ACM Trans. Graph. 28*, 3 (2009), 1–8. 2, 3

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph. 27*, 5 (2008), 1–11. 2