Toward Practical Real-Time Photon Mapping: Efficient GPU Density Estimation

Michael Mara
NVIDIA & Williams College

David Luebke
NVIDIA

Morgan McGuire
NVIDIA & Williams College

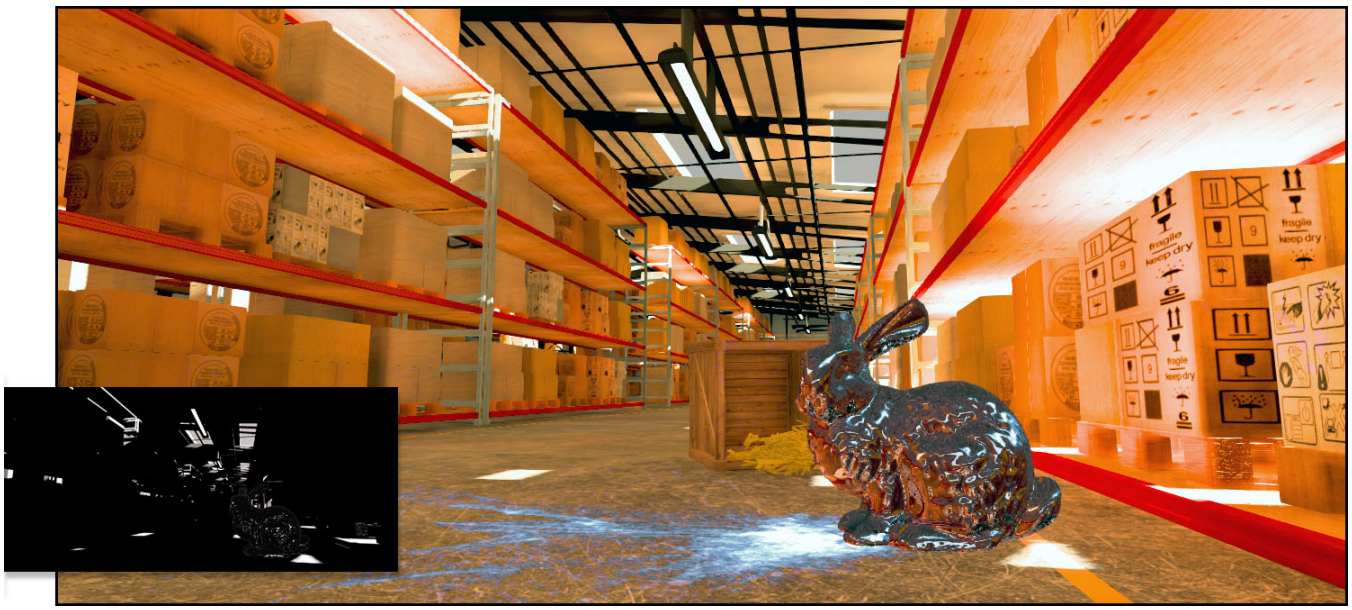


Figure 1: *A warehouse in which small gaps create a complex, indirect paths for late afternoon sunlight from windows on the far right. The Tiled algorithm computes scattered indirect illumination from traced photons for this scene in 28ms at 1920×1080 on the mid-range NVIDIA GeForce 670 GPU. The inset image with only direct illumination shows the importance of rendering indirect light for this scene.*

Abstract

We describe the design space for real-time photon density estimation, the key step of rendering global illumination (GI) via photon mapping. We then detail and analyze efficient GPU implementations of four best-of-breed algorithms. All produce reasonable results on NVIDIA GeForce 670 at 1920×1080 for complex scenes with multiple-bounce diffuse effects, caustics, and glossy reflection in real-time. Across the designs we conclude that tiled, deferred photon gathering in a compute shader gives the best combination of performance and quality.

CR Categories: I.3.3 [Picture/Image Generation]: Display Algorithms; I.3.7 [Three-Dimensional Graphics and Realism]: Color, shading, shadowing, and texture

Keywords: photon map, global illumination, distribution

1 Introduction

Jensen introduced photon mapping in 1996 as a consistent estimator for indirect illumination in offline rendering. It not only excels at sampling caustic paths that converge slowly for many other rendering methods, but also estimates lower-frequency glossy and diffuse interreflection well and is surprisingly simple to implement.

Many variations on photon mapping intended for eventual real-time rendering of complex scenes have since been proposed and even demonstrated on limited scenes at interactive rates. Combined with this algorithmic progress, GPUs are now powerful enough that photon mapping might soon be practical in real-time rendering systems. The challenge is maintaining quality while scaling efficiently on a modern GPU.

Photon mapping contains two steps: tracing photons along rays from light sources and estimating radiance due to those photons scattering off visible surfaces (i.e., "shading"). Efficient parallel ray tracing hardware and software systems, such as OptiX [Parker et al. 2010], can trace hundreds of millions of rays per second and the process can be amortized over multiple frames. Thus existing systems meet the performance needed for photon *tracing*. Net performance thus hinges on efficient photon *shading*. In this paper, we explore the design space of architecture-aware optimization of photon shading for parallel vector architectures, using a current NVIDIA GPU as a concrete, motivating case.

We describe the algorithm design space defined by a tree of design decisions. For these, we contribute key details for efficient execution on vector processors in both graphics and compute APIs, including a new approach for distribution sampling during GPU photon gathering. We then analyze the four best algorithms in this space in detail on complex, realistic scenes.

Why photon mapping? Photon mapping is a good candidate for robust, real-time rendering. It naturally simulates a range of GI phenomena, including: diffuse interreflection, caustics, glossy reflection, area (especially sky) sources, and indirect shadowing ("ambient occlusion" at all scales) for fully dynamic scenes. Most important for a real-time application, photon mapping converges to the

correct result as more computational power is made available and degrades gracefully (by blurring lighting) on lower-end devices. Other fast GI algorithms [e.g., VPLs, Direct-to-Indirect, Irradiance Volumes] do not provide all of these properties simultaneously.

A limitation of photon mapping is that does not capture transport paths that terminate in a series of perfectly-specular scattering events (i.e., eye reflections and refractions). Those must be ray traced or approximated with screen space techniques.

Why fast? Offline global illumination takes minutes to hours to render with current tools. Bringing this time down by two orders of magnitude, to around 0.2-1.0s, will be transformative for design applications (i.e., CAD and DCC). It will change workflow from render-and-wait to interactive editing of lighting, materials, and geometry. This will enable efficient workflow for architects, industrial designers, film lighting technicians, and game developers creating design spaces and lighting.

Another order of magnitude increase in throughput, to around 0.01s/frame, will make dynamic GI viable for video games. This will increase visual fidelity and realism, and enable new gameplay (e.g., by observing the bounced light from a muzzle flash around a corner). True real-time GI also reduces the amount of pre-baked lighting data that must be managed throughout development and installed on the consumer's machine, which are significant production and workflow issues today.

1.1 Photon Mapping and Related Work

Photon mapping [Jensen 2001] consists of three steps: emit photons, trace photons, and compute the scattered radiance by estimating the density of photons (which for brevity we call "shading"). Photon emission and trace steps are similar to path tracing. They produce a series of scattering points along transport paths. Tracing stores an incident "photon" in a *photon map* at each scattering point. The choice of data structure for the map depends on the shading algorithm employed later.

Photon emission and tracing have the same computational profile as ray tracing, so methods from GPU ray tracing literature apply directly. See the OptiX [Parker et al. 2010] article for recent evaluation of GPU photon tracing in particular (an even faster approximation performs the trace in image space, against the depth buffer [Yao et al. 2010]). Because ray tracing has been explored significantly more than shading, we focus solely on shading in this paper. Hachisuka et al.'s report [2012] provides a comprehensive survey and history of photon shading methods; the remainder of this section describes selected GPU methods.

The shading step integrates at each visible scene point X the contribution of a set $\mathbf{Q}(X)$ of nearby photons from the photon map. Each photon P in that set is a sample of radiant flux Φ_P at a position Y_P propagating in direction $\hat{\omega}_P$. The scattered indirect light is the sum of that flux modulated by the bidirectional scattering function f, normalized by the area of the disk,

$$L_{\mathrm{s}}(X, \hat{\omega}_{\mathrm{o}}) \approx \sum_{P \in \mathbf{Q}(X)} \frac{\Phi_P \;\; f_X(\hat{\omega}_P, \hat{\omega}_{\mathrm{o}}) \;\; \kappa(||Y_P - X||)}{\int_0^{2\pi}\int_0^{r_P} \kappa(s) d\theta ds} \tag{1}$$

where kernel $\kappa(s)$ is a density falloff function that weights the contribution of each photon by its proximity to the shaded point X.

A key algorithmic choice in building a GPU photon mapping system is how to find the set $\mathbf{Q}(X)$ of nearby photons. Several systems [Zhou et al. 2008; Wang et al. 2009; Fabianowski and Dingliana 2009] *gather* the k-nearest neighbor (kNN) photons using a k-d tree (Li et al. [2012] optimized the tree building process). Ma and McCool [2002] gather from a hash grid instead of a tree.

Scatter techniques [Szirmay-kalos et al. 2005; Nichols and Wyman 2009; McGuire and Luebke 2009; Jakob et al. 2011] rasterize bounding geometry around each photon to discover scene points affected by it, in much the same was as light volumes are applied in deferred shading. These methods use rasterization to efficiently find the pixels near a photon in screen-space xy, where the depth test culls in the $-z$ direction and the pixel shader explicitly tests depth in the $+z$ direction. We evaluate the implications of Ma and McCool's and McGuire and Luebke's approaches (others are slower or more limited in practice), investigate four additional new alternatives, and show that two of these new methods can be significantly more efficient at the same quality level. Some of these ideas for efficient GPU photon scattering were first explored by Pantaleoni and demonstrated in early form but not fully explained in Figure 5 of the VoxelPipe paper [2011], using a realtime voxel-plus-photon renderer.

A number of other methods use the GPU and splatting in interesting ways (e.g., [Jarosz et al. 2011; Papaioannou 2011; Maletz and Wang 2011]). These are not directly relevant to our shading investigation but are complementary; our observations about efficiency may also apply.

Offline path tracers often perform final gathering as a variance reduction technique, essentially tracing two scattering events backward from the eye before evaluating photons. That process is inherently slow and tied to ray casting. Fast photon methods like those in this paper that estimate photon radiance at primary visible points must reduce variance with wider kernels and more photons.

Temporal coherence is a concern in all sampling algorithms. We note that this issue has been addressed in previous work with varying levels of success. Our scope is narrower: how to estimate radiance in a system where the temporal aspect of the photon trace has already been addressed satisfactorily for the application.

Shading equation 1 has several tunable scalar and function parameters. These affect the image quality and performance of all shading algorithms in approximately the same way. While *some* values are necessary for our experiments, exploring the space of those parameters is orthogonal to investigating performance. We simply adopt McGuire and Luebke's choices as described below. The falloff function is a cone clamped to $c = 1/5$ of its height

$$\kappa(s) = \min\left(c, 1 - s/r_P\right) \tag{2}$$

$$\int_0^{2\pi}\int_0^{r_P} \kappa(s) d\theta ds = \frac{r_P^2}{\pi c(\frac{1}{3}c^2 - c + 1)} \tag{3}$$

We choose the radius for photon P as

$$r_P = r_{\min} + (r_{\max} - r_{\min})\sqrt{1 - \rho_P} \tag{4}$$

(the square root appears because $r \propto \sqrt{\mathrm{area}}$). The path probability ρ_P of photon P is the product over each previous bounce b of the *a priori* scattering probability:

$$\rho = \prod_b \int_\Gamma f_b(\hat{\omega}_b, \hat{\omega}_{\mathrm{o}}) |\vec{n} \cdot \hat{\omega}_b| d\hat{\omega}_{\mathrm{o}} \tag{5}$$

computed for a solid angle of measure $||\Gamma|| = 10^{-3}\mathrm{sr}$.

McGuire and Luebke vary the radius of scattered photons according to the a priori probability of the scattering events along the photons path. Since this variable photon radius affects more pixels in low-density areas, and fewer pixels in high-density areas, it serves to

improve density estimation in much the same way as kNN queries. We apply this technique in all tested examples for consistent quality.

We choose a typical [Akenine-Möller et al. 2008] energy-conserving bidirectional scattering distribution function inspired by the Blinn-Phong model:

$$F = F_0 + (1 - F_0)\,(1 - \max(0, \hat{\omega}_\mathrm{h} \cdot \hat{\omega}_\mathrm{i}))^5 \quad (6)$$

$$\hat{\omega}_\mathrm{h} = (\hat{\omega}_\mathrm{i} + \hat{\omega}_\mathrm{o})/||\hat{\omega}_\mathrm{i} + \hat{\omega}_\mathrm{o}|| \quad (7)$$

$$f_\mathrm{R}(\hat{\omega}_\mathrm{i}, \hat{\omega}_\mathrm{o}) = \frac{\rho_\mathrm{L}}{\pi} + \frac{(8+s)F}{8\pi}\max(0, \hat{\omega}_\mathrm{h} \cdot \vec{n})^s \quad (8)$$

for glossy and Lambertian reflection, and impulses for specular reflection and transmission. Of course, one might tune the BSDF for performance instead of quality in a specific application. To provide a lower bound on execution time for such cases, we conducted a separate set of experiments using a Lambertian-only BSDF.

2 Algorithms

We evaluate a hierarchy of designs represented in figure 2. The nodes of that tree are key design choices and the leaves are eight specific algorithms. We implemented all of these, named the four best–"*3D Bounds*", "*2.5D Bounds*", "*HashGrid*", and "*Tiled*"–and then evaluated them as reported in section 3. This section describes the GPU implementation of components of the algorithms. Note that if one holds the radius of effect (r_P) constant for all P and disables subsampling, then all of these algorithms give identical results. Furthermore, they are all mathematically *consistent*, converging to the same, correct result as the photon count increases [and the radii decrease], even with subsampling and varying radii.

2.1 Choosing the Major Iterator

Because the shading algorithm evaluates combinations of photons and pixels, it can iterate in either photon-major or pixel-major order. This is analogous to some classic graphics iteration choices, such as rasterization vs. ray tracing (triangle- or pixel-major) or forward vs. deferred shading (triangle- or light-major). The choice of photon- or pixel-major order carries similar implications for efficient concurrent evaluation and options in the design subtree.

2.2 Photon-Major Iteration (Scattering)

Listing 1 shows the structure of a shading algorithm whose outer loop is over photons, i.e. one that *scatters* photons onto pixels. This implements equation 1 for all pixels; the inner-loop's per-photon contribution is the summand in equation 1. Photon-major methods execute in graphics mode (e.g., OpenGL, DirectX) on a GPU. They render *bounding* geometry about each photon by issuing a draw call for the outer loop. As with shadow volume and screen-space decal [Kim 2012] rendering, this leverages the rasterizer as a power-efficient generalized iterator for the inner loop. The pixel stage then evaluates the body of the inner loop. The bounding geometry may be true 3D or 2.5D (billboards).

```
for each photon P: # Draw call
    for each pixel (x, y) with visible point X near P: # Rasterizer
        image[x, y] += contribution of P at X # Pixel shader
```

Listing 1: *Photon-major Iteration*

2.2.1 Bounding Geometry

3D Polyhedral Bounds A *photon volume* is a polyhedron circumscribed about a photon's sphere of effect [McGuire and Luebke 2009]. This conservatively bounds the photon from all viewpoints. Subfigure 3c shows icosahedron bounds for a few photons (in a real scene, they would completely cover every surface many times).

2.5D Polygonal Bounds *Splatting* methods (e.g., [Nichols and Wyman 2009]) rasterize small point primitives for photons with small radii relative to the screen resolution. We generalize for arbitrary 2D screen-space bounds-with-depth by computing the tightest polygon circumscribing the silhouette of the photon's sphere of effect [Mara and McGuire 2012]. This allows us to efficiently rasterize bounds for radii varying from very large (e.g., $\approx$ 1m, arising from multiple diffuse reflections) to very small (e.g., $\approx$ 1mm for caustics from perfect specularly scattering). Subfigure 3d shows a side view of these polygonal billboards. For the same triangle count as a 3D bounding shape, they present a slightly looser z bound but much tighter xy bound.

2.2.2 Transformation Stage

Vertex Stage (VS) One method to render n topologically identical bounding shapes is to submit n instances of a single bounding shape (e.g. with `glDrawArraysInstanced`) and then transform the shapes individually in the vertex stage [McGuire and Luebke 2009]. The vertex shader reads the position and radius of each photon from a texture map based on the instance ID. For the large amount of geometry needed for 3D bounds, this proved efficient. However, transforming each vertex independently was inefficient for 2.5D bounds because the projected-bounds algorithm naturally computes pairs of vertices at once and cannot use both in the vertex shader.

Geometry Stage (GS) Another method for rendering n bounding shapes is to submit n point primitives to the GPU and then amplify each into a triangle strip in the geometry stage. For 3D cubes this is efficient–but the bounds themselves are not efficient. For 3D icosahedra the level of amplification overloads the geometry shader and performance degrades so rapidly that our GPU driver timed out on all but the simplest scenes during testing. However, we found that computing the 2.5D polygonal bounds was faster in the geometry stage than in the vertex stage.

2.3 Pixel-Major Iteration (Gathering)

Listing 1 shows the basic structure of our gathering algorithms. These use a compute mode shader (our implementaion is in CUDA) for the entire iteration process. A thread launch over pixels replaces the outer loop, while each threads kernel performs the inner loop.

```
for each pixel (x, y) with visible point X:
    for each photon P near X:
        image[x, y] += contribution of P at X
```

Listing 2: *Pixel-major Iteration*

2.3.1 Gather Space

3D: HashGrid A hash grid is a hash table of sparse grid cells, each of which contains an array of photons. The renderer iterates over all pixels on the screen and for each one gathers photons by indexing the 3D location of a shaded point X into the containing grid cell [Ma and McCool 2002]. The hash grid is viewer-independent, so it can be used over multiple frames without modification when the lighting does not change, and can be updated incrementally when lighting does change. Subfigure 3e shows the non-empty hash grid cells for our sample scene. Efficient GPU hash grid access must minimize pointer-chasing and maximize coherence for simultaneous access. To achieve this, we use an open-addressing scheme with quadratic probing and a power-of-two number of table slots. We pack the values for each cell into a single array of photons, each cell corresponding to a continuous block, and store the start and end indices from each cell. It is also essential that the hash

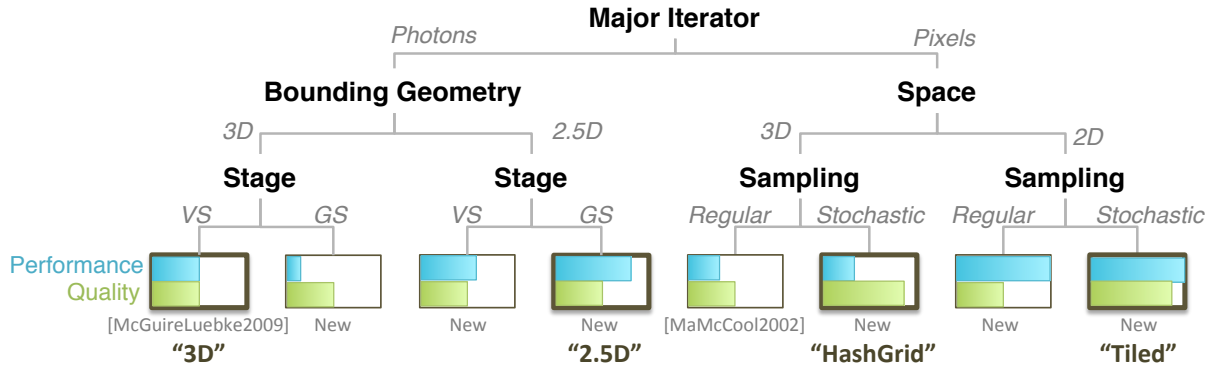
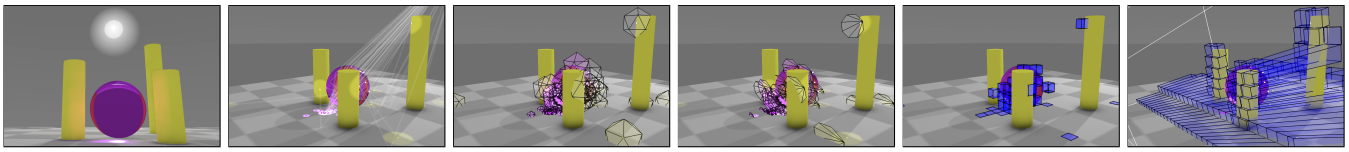


Figure 2: *Hierarchy of major design choices, with resulting shading methods along the bottom rated coarsely by performance (pixel throughput) and subjective image quality. We present comparative experimental analysis of the best in each class (named in quotes) in section 3. The two cited methods are our modern extensions of ones originally designed for more limited GPUs.*



(a) *Front view* **(b)** *Side view* **(c)** *3D Bounds* **(d)** *2.5D Bounds* **(e)** *HashGrid Cells* **(f)** *Tile Frusta*

Figure 3: *Visualizations of shading algorithms showing selected photons, paths, and bounding geometry. Note the transport paths converging, displaying how the glass sphere creates a caustic.*

function produces few collisions. This is especially important on a SIMD architecture because all lanes share an instruction pointer and thus take time equal to the highest number of probe iterations encountered across the vector when accessing different cells. We tried many hash functions and found that the hash6432shift function by Wang [2007] (see listing 4) was the fastest to compute and gave the fewest collisions across all of our test scenes. For example, in the Sponza scene the table contains 2^{16} slots, 3/4 of which are empty. The average number of probe iterations is 1.08 and the maximum is 6.

```
for each pixel (x, y) with visible surface X:

    # Iterate through all cells around X
    r⃗ = (r_max, r_max, r_max)
    for cell = ⌊(1/s)(X − r⃗)⌋ to ⌊(1/s)(X + r⃗)⌋:
        b = hash(cell) & (numBuckets − 1)
        d = 0

        # Quadratic probe for the bucket of photons
        while (bucket[b].cell != cell) and not bucket[b].empty:
            ++d; b = (b + ((d + d^2) >> 1)) & (numBuckets − 1)

        # Iterate through bucket contents
        i = bucket[b].start + step(k^2) − 1
        count = 0; sum = 0
        while i < bucket[b].length:
            ++count
            P = photon[i]
            if ||Y_P − X_P||^2 < r_P^2:
                sum += contribution of P at X
            i += step(k^2) # See section 2.3.2
        image[x, y] += L * bucket[b].length / count;
```

Listing 3: *Gathering from a hash grid with cells of diameter s*

Adjacent pixels are likely to read from the same cells in memory. However, amortizing those DRAM fetches by leveraging shared memory proves difficult under the hash grid algorithm, because the grid cells are cubes in 3D (which do not have a simple mapping under perspective projection to screen space). Fortunately, memory caches in modern GPUs capture this reuse implicitly and adequately. The 2D gather space in the following section addresses reuse explicitly.

For consistency across algorithms, our hash grid implementation assigns radii to photons following McGuire and Luebke [2009]. An alternative is to perform a k-nearest neighbor search about X [Ma and McCool 2002]. Doing so might affect shading slightly, but should not impact performance: this search requires gathering the same conservative size initially, since going back to the table for more cells at a pixel if k is not reached in the initial iteration is expensive on a vector architecture.

```
uint32_t hash(Point4int16 cell):
    key = *(uint64_t*)&cell;
    key = (~key) + (key << 18);
    key = key ^ (key >> 31);
    key = key * 21;
    key = key ^ (key >> 11);
    key = key + (key << 6);
    return uint32_t(key) ^ uint32_t(key >> 22);
```

Listing 4: *Wang's hash6432shift algorithm.*

2D: Tiles A tiled algorithm inserts copies of each photons in buckets corresponding to the screen-space tiles it might affect. This allowed a second pass to shade all pixels within a tile from a common subset of photons that fit within shared memory for a compute shader. This yields a significant DRAM bandwidth savings compared to the other methods we've described, which load each pixel or photon multiple times. In the case where there are more photons in a tile than fit in memory, we make multiple passes over a tile, processing a sizable chunks of the photons in the tile in each pass.

During the process of categorizing the photons into tiles, a tiled algorithm can cull photons that do not intersect the depth range of

the scene within a tile. In contrast, each 3D or 2.5D photon volume used in rasterization-based methods only affects pixels within a narrow depth range, yet the single-sided hardware depth test requires that those algrithims launch shading threads for either all pixels in front of or all pixels behind a certain depth. In scenes with high depth variance, that may launch many shading threads that immediately terminate, reducing vector lane occupancy without providing useful computation.

Listing 5 outlines the tile insertion pass. Our actual implementation further partitions tiles into chunks each containing a bounded photon count; this is a tradeoff of simplicity for flexibility (e.g., new photons can be directly appended to partial chunks). Each photon consumes little memory, so we duplicate and embed them directly in the chunks instead of chasing pointers. This also uses the cache efficiently and enables coalesced parallel memory fetches. Subfigure 3f shows the bounding frusta for tiles in a simple scene; all photons accessed by pixels in a tile intersect its frustum.

```
let m = number of tiles
let photonCount[m], nextPhotonIndex[m] = arrays of 0s

for each photon P: # counting pass
    for each tile T with frustum intersecting the effect sphere of P:
        photonCount[T] += 1

for each tile T: # allocation pass
    allocate photonCount[T] photons in tilePhotons[T]

for each photon P: # copy pass
    for each tile T with frustum intersecting the effect sphere of P:
        s = atomicGetAndIncrement(nextPhotonIndex[T])
        tilePhotons[T][s] = P
```

Listing 5: *Tile insertion.*

```
for each n×n tile T: # Thread launch

    let photon[ ] = shared memory array
    for each pixel (x, y) in T:
        load 1/n^2 of tile contents into photon[ ]

    for each pixel (x, y) in T with visible surface X:

        # Iterate through tile contents
        i = step(k^2) − 1 # See section 2.3.2
        count = 0; sum = 0
        while i < photon.length:
            ++count
            P = photon[i]
            sum += contribution of P at X
            i += step(k^2)

        image[x, y] += sum ∗ photon.length / count
```

Listing 6: *Tile gathering.*

2.3.2 Sampling Method

Regular As is a common practice to for slowly varying effects such as indirect illumination, ambient occlusion, and glossy reflections, one can estimate photon density at reduced resolution on regularly subsampled geometry buffers, and then bilaterally upsample the result. The drawback of doing so is blurring and under-sampling of thin features and edges. For example, 4×4 subsampling blurs caustics over 16 pixels and shading flickers objects that cover fewer than 4 pixels.

There are many more sophisticated upsampling filters and ways of integrating them specifically for photons [Dammertz et al. 2010; Yao et al. 2010]; we leave full analysis of upsampling as future work to focus on the dominant cost of shading in this paper.

Regular sampling is imperative for scatter methods because the rasterizer can only iterate over regular patterns. For gathering, one has a choice. The listings in the previous subsection refer to a function step(x), which would simply return 1 if one wanted to implement regular sampling of gathered illumination. However, there is a better sampling strategy for gather algorithms that we now describe.

Stochastic Because we can iterate over photons at a pixel (instead of pixels for a photon), we can stochastically select a subset of photons independently at each pixel. This allows us to simultaneously sample over space and photons, preserving resolution. step(x) is a geometrically distributed random number generator representing samples of the number of trials until the first success a Bernoulli random variable with success probability $1/x$. We increment by this value when iterating through the photon array to randomly sample $1/x$ photons on average.

2.4 Photon Representation

Encoding all properties of a photon at full precision can make all shading algorithms bandwidth limited on a modern GPU [Ma and McCool 2002]. To avoid this and measure the true computation and BSDF bandwidth limits of the shading algorithms, we store photons in three texture maps of varying precision:

Field	Format	Bytes
Position Y_P	`GL_RGB32F`	12
Power Φ_P	`GL_RGB[A]8`	4
Direction and prob. $\hat{\omega}_P, \rho_P$	`GL_RGBA8`	4
	TOTAL	**20**

This compares favorably to previous representations of 24 [Ma and McCool 2002] and 48 bytes [McGuire and Luebke 2009] with no observable loss in quality. At this size the algorithms were not bandwidth-limited by photons, so we stopped optimizing. However, we note that two further space reduction methods are possible: encoding directions in two bytes instead of three [Meyer et al. 2010], and encoding positions for the Tiled algorithm and HashGrid algorithms relative to the tile or grid cell. This can reduce the size of each photon to 12 bytes.

3 Experimental Analysis

We experimented with all eight algorithms described in the previous section. We report detailed analysis of the 3D Bounds, 2.5D Bounds, HashGrid, and Tiled algorithms which each exhibited the best performance and quality in their class.

3.1 Test Scenes

Shading performance is scene dependent, so we investigate their behavior under a variety of scenes that exhibit interesting properties with respect to photon mapping (figure 4). Much previous work on real-time global illumination is concerned with rendering quality and artifacts, and thus uses minimal scenes such as a high-polygon statue in a Cornell box for testing. We're starting with established methods and primarily targeting quality for complex video game, CAD, and DCC scenes so we test directly at that level of complexity.

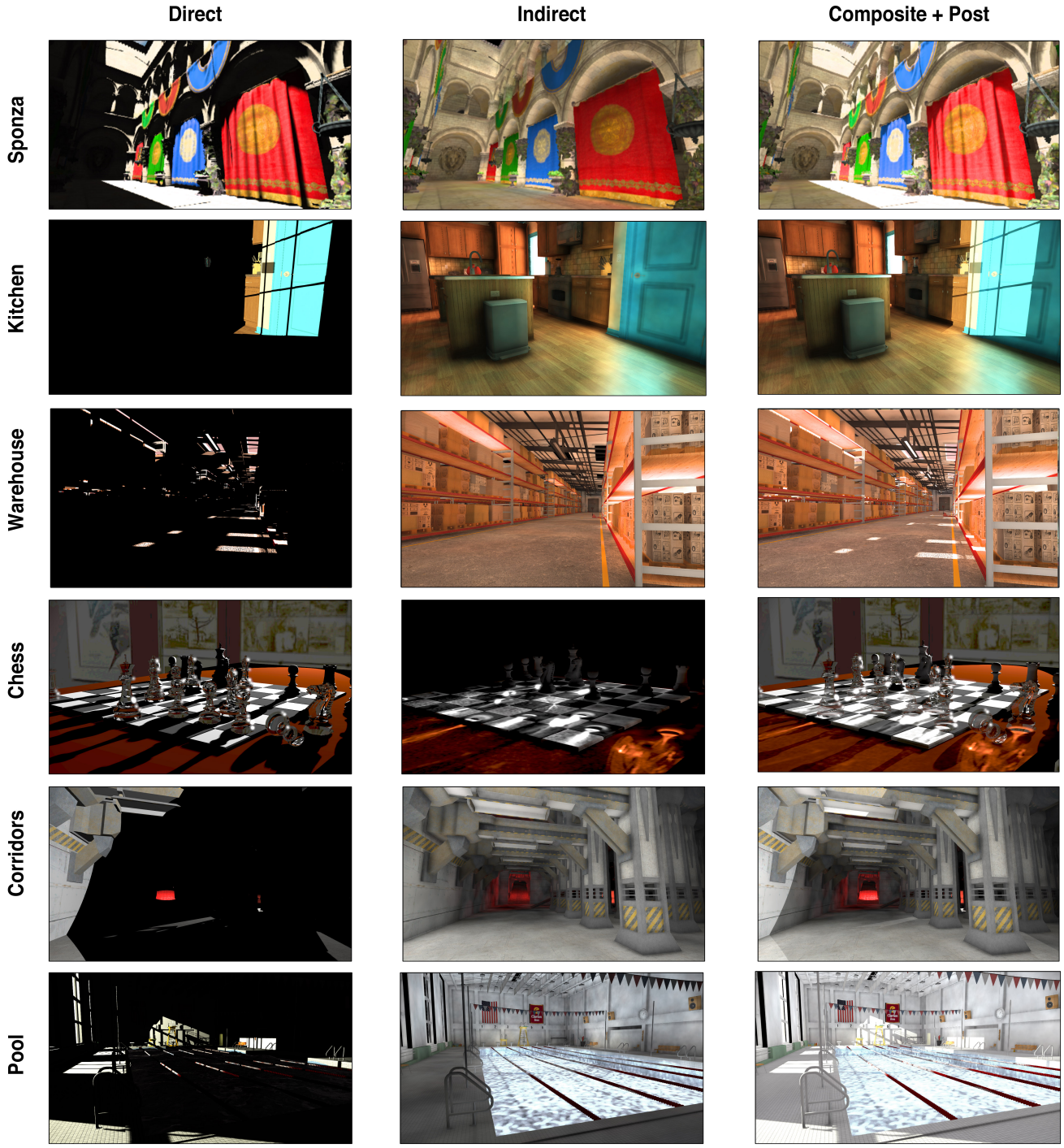


Figure 4: *Test scenes covering a variety of indirect illumination phenomena that represent typical use cases for real-time applications.*

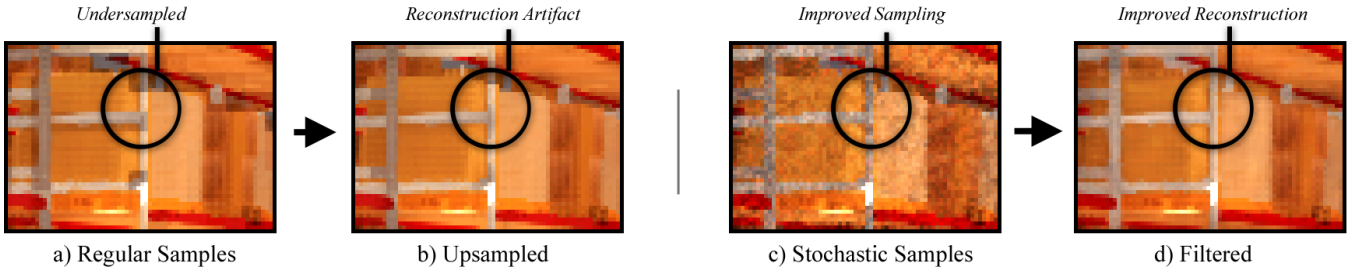


Figure 5: *Detail of Warehouse image. Left: Regular samples, bilateral upsampled by a factor of $k = 5$ in each direction. Large surfaces like the boxes are reconstructed well but small features like the gray struts are undersampled and appear splotchy. Right: Stochastic samples of $1/k^2$ of eligible photons at each pixel and bilateral-filtered result; distributing over space and direction better-samples both.*

	Input Parameters			*Full BSDF Execution Time*				*Lambertian-Only Execution Time*			
Scene	**Tris**	**Stored Photons**	**2.5D Verts**	**3D Bounds** VS + PS	**2.5D Bounds** GS + PS	**HashGrid** Compute	**Tiled** Compute	**3D Bounds** VS + PS	**2.5D Bounds** GS + PS	**HashGrid** Compute	**Tiled** Compute
Sponza	262k	448k	18	29.3 ms	24.8 ms	92.1 ms	25.7 ms	25.4 ms	20.8 ms	45.7 ms	11.1 ms
Kitchen	386k	520k	10	15.0	11.6	56.2	21.0	13.1	9.3	36.0	7.2
Warehouse	438k	1607k	10	57.4	42.3	137.8	72.8	47.8	33.6	77.1	27.5
Chess	66k	295k	8	18.5	15.2	174.7	38.0	14.1	11.2	79.7	12.8
Corridors	782k	596k	6	17.1	4.8	11.1	4.1	16.5	3.7	4.8	1.3
Pool	815k	586k	8	25.7	21.3	72.0	24.4	16.1	12.1	46.1	19.9
				Regular Sampling*		Stochastic		Regular Sampling*		Stochastic	

* 3D and 2.5D Bounds are inherently limited to lower visual quality on scenes with thin objects (see figure 5)

Table 1: *Indirect illumination times for figure 4 scenes at 1920×1080, after reconstruction filtering with k=4. The 2.5D algorithm exhibits good peak performance without compute shaders. The Tiled algorithm's quality scales best to high resolutions/thin objects because of stochastic sampling, and under an optimized the BSDF implementation it can also provide the best performance on complex scenes.*

Our scenes are:

- **Sponza**: A common benchmark featured in previous work
- **Kitchen**: Glossy floor and counter and diffuse walls
- **Warehouse**: Thin features and narrow gaps, big depth range
- **Chess**: Complex multi-bounce caustics
- **Corridors**: High depth complexity; only 8% of all photons affect surfaces visible in the shot
- **Pool**: Millions of diffuse and caustics photons

We produced all results at 1920×1080 on the mid-range NVIDIA GeForce 670 GPU, and all images were rendered with the Tiled algorithm and $k = 4$ unless otherwise noted.

Our test harness precomputed the photon emission and trace for each scene and then injected the same photons into each shading algorithm. It then composited the indirect illumination modulated by screen-space ambient occlusion over explicit direct illumination and performed some typical post-processing steps (e.g., color grading, bloom, gamma correction, depth of field, and screen-space antialiasing). The final images thus predict what one could expect in a few years from a production real-time renderer using one of these algorithms for the shading step.

3.2 Quality

Phenomena Figure 4 demonstrates the breadth of indirect illumination pheonomena produced by photon mapping, rendered in real-time. Starting at the top of the figure, and focusing on the central column showing the indirect illumination only: all of the light on the left side of Sponza and in the balconies is due to **multiple diffuse** bounces. On the floor near the banners, **color bleeding** is visible from diffuse interreflection.

Kitchen is lit by the morning sun through a window and indirect light entering from the next room. The **glossy reflection** of the blue door is visible in the floor. **Fine details** like the molding ridges on the cabinets and tiles in the back are sampled well. Warehouse is lit primarily through windows on the right; nearly all illumination must **pass through small gaps** between the shelves and crates.

Chess demonstrates **complex caustics** due to refraction in the glass chess pieces (we render the refraction with screen space sampling). Corridors shows multiple diffuse bounces from many sources and **ambient occlusion** effects from the photons being absorbed among the pillars on the right. Pool demonstrates extensive, large-scale caustics on the bottom of the pool, including **shadows in caustics** where the floating lane dividers block incoming light. The walls are lit entirely by glossy reflection of photons off the tiles.

Subsampling All of the methods employ some form of subsampling and reconstruction. This is critical because none is fast enough to process in real-time *all* eligible photons at each pixel, at full resolution. Figure 5 shows a detail of Warehouse that is hard to render because it contains thin features and sharp edges.

Subfigures (a) and (b) show regular (spatial) sampling with bilateral upsampling employed by the 3D and 2.5D Bounds methods, which can undersample thin objects. When this occurs, the thin objects are too dark in the final image and may flicker under animation. At some performance cost, these artifacts could be ameliorated by interleaved sampling's rotated grid.

A better solution is distributed stochastic sampling with bilateral filtering, which is employed by the Hashgrid and Tiled methods (subfigures (c) and (d)). This is available to gathering methods because they can select photons independently at each pixel in the inner loop. Note that the artifacts in figure 5c resemble those of path tracing or distribution ray tracing; that is because like our method, those also distribute samples over a subset of points (X) and directions ($\hat{\omega}_P$).

3.3 Culling

Many of our scenes are entirely visible from the chosen viewpoint, which is representative of modeling but not a video game. The Corridors scene demonstrates plausible topology for a video game to stress photon culling performance.

Figure 6 shows its top view with the view frustum overlaid in orange. All of the algorithms perform well here. 2.5D Bounds performs much better than 3D Bounds because, by operating in the Geometry stage, the 2.5D algorithm can frustum cull photons before they are amplified into bounding geometry. Tiled performs exceptionally because it can eliminate almost all photons during tile insertion and therefore never has to consider them during gathering.

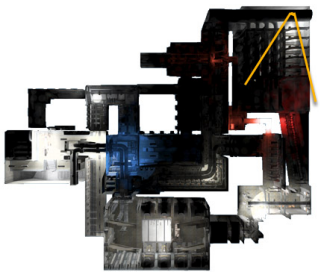


Figure 6: *Top view*

3.4 Performance

Table 1 reports the shading time for each algorithm, for each scene. The center columns describe a typical BSDF, which we did not substantially optimize for performance. The right columns show the limiting performance that one could possibly achieve by optimizing

the BSDF, which we measured by dropping all but the Lambertian term. Within each group of columns, the fastest result is highlighted on each row. The 2.5D Bounds algorithm is sensitive to the number of vertices in the polygonal bound, so for each scene we exhaustively searched for the optimal number of vertices and reported it in the 2.5D Verts column.

Note that the Lambertian BSDF *will* capture specular and glossy caustics; it just won't capture glossy reflections of *other* surfaces. Most of the phenomena in our scenes are still present, except for the reflection of the blue door in the Kitchen scene and some (barely perceptible) reflections of the walls in the Pool tiles.

4 Conclusions and Future Work

We demonstrated sufficient performance to immediately enable photon mapping for interactive design applications, provided the photon trace is amortized over multiple frames and also performed efficiently. For limited BSDFs and specific scenes, this could perhaps be optimized enough to deploy on current high-end hardware in games. Yet, a more important goal is robust and accurate lighting in the near future *without* such limitations and simplifications. We cast our work as a step toward that goal. We note that rendering bounding volumes over screen-space buffers is an increasingly popular technique for quantities other than indirect light (e.g., [Laine and Karras 2010; Maletz and Wang 2011; Kim 2012]). The approach demonstrated by our efficient 2.5D bounds and Tiled methods may be directly applicable to these.

2.5D scattering is easy to implement and fast in practice. However, both 3D and 2.5D scattering are quality limited by their inability to (efficiently) sample stochastically during shading and thus will not scale well to higher resolutions or fine detail. The Tiled gather algorithm supports stochastic sampling and already performs well; it seems the most likely candidate moving forward. Profiling revealed several opportunities for low-level optimization with its structure. The most significant may be reducing the required 40 registers for the full BSDF, which limits the number of concurrent threads and thus performance. We also observed memory bank conflicts and fairly weak coalescing across global memory read operations that may be addressable with more careful operation scheduling.

For stochastic sampling during shading, we used statistically-independent random numbers at each pixel. Cooperative sampling methods have been shown to dramatically increase convergence by coordinating samples at adjacent pixels. A more sophisticated reconstruction filter may also reduce the number of samples required to achieve acceptable convergence.

Historical trends indicate that two GPU generations will yield approximately a $6\times$ increase parallelism. Even without further optimization, that would enable real-time applications of photon mapping in production applications using the methods described in this paper. However, we hope that algorithmic and hardware advances will enable this even sooner and at higher quality.

Acknowledgements

We thank our colleagues Jacopo Pantaleoni at NVIDIA and Guedis Cardenas at Williams College for their contributions to this project.

This version of the paper differs from the one originally published by: a correction of a section reference in a code comment, page breaks, and correction of `p += ...` *to* `i += ...` *in Listing 6 (thanks to Koguchi Gussan).*

References

AKENINE-MÖLLER, T., HAINES, E., AND HOFFMAN, N. 2008. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA.

ANDERSSON, J., 2011. DirectX 11 rendering in Battlefield 3, March. Talk at the Game Developer's Conference.

DAMMERTZ, H., SEWTZ, D., HANIKA, J., AND LENSCH, H. P. A. 2010. Edge-avoiding a-trous wavelet transform for fast global illumination filtering. In *Proc. of HPG*, Eurographics Association, Aire-la-Ville, Switzerland, HPG '10, 67–75.

EBERLY, D. 1999. Perspective projection of an ellipsoid.

FABIANOWSKI, B., AND DINGLIANA, J. 2009. Interactive global photon mapping. In *Proc. of EGSR*, EGSR'09, 1151–1159.

HACHISUKA, T., JAROSZ, W., BOUCHARD, G., CHRISTENSEN, P., FRISVAD, J. R., JAKOB, W., JENSEN, H. W., KASCHALK, M., KNAUS, C., SELLE, A., AND SPENCER, B. 2012. State of the art in photon density estimation. ACM, NY, NY, USA, SIGGRAPH '12 Courses, 6:1–6:469.

JAKOB, W., REGG, C., AND JAROSZ, W. 2011. Progressive expectation-maximization for hierarchical volumetric photon mapping. vol. 30, 1287–1297.

JAROSZ, W., NOWROUZEZAHRAI, D., THOMAS, R., SLOAN, P.-P., AND ZWICKER, M. 2011. Progressive photon beams. In *SIGGRAPH Asia 2011*, ACM, NY, NY, USA, 181:1–181:12.

JENSEN, H. W. 1996. Global illumination using photon maps. In *Rendering techniques*, Springer-Verlag, London, UK, 21–30.

JENSEN, H. W. 2001. *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd., Natick, MA, USA.

KIM, P. 2012. Screen space decals in Warhammer 40,000: Space Marine. In *ACM SIGGRAPH 2012 Talks*, ACM, New York, NY, USA, SIGGRAPH '12, 6:1–6:1.

LAINE, S., AND KARRAS, T. 2010. Two methods for fast ray-cast ambient occlusion. In *Proc. of EGSR*, Eurographics Association, Aire-la-Ville, Switzerland, EGSR'10, 1325–1333.

LI, S., SIMONS, L., PAKARAVOOR, J. B., ABBASINEJAD, F., OWENS, J. D., AND AMENTA, N. 2012. kANN on the GPU with shifted sorting. In *Proc. of HPG '12*, Eurographics Association, Aire-la-Ville, Switzerland, 39–47.

MA, V. C. H., AND MCCOOL, M. D. 2002. Low latency photon mapping using block hashing. In *Proc. of Graph. Hard.*, Eurographics Association, Aire-la-Ville, Switzerland, 89–99.

MALETZ, D., AND WANG, R. 2011. Importance point projection for gpu-based final gathering. In *Proc. of EGSR*, Eurographics Association, Aire-la-Ville, Switzerland, EGSR'11, 1327–1336.

MARA, M., AND MCGUIRE, M. 2012. 2D polyhedral bounds of a clipped, perspective-projected 3D sphere. *JCGT*. *in submission*.

MCGUIRE, M., AND LUEBKE, D. 2009. Hardware-accelerated global illumination by image space photon mapping. In *Proc. of HPG*, ACM, New York, NY, USA, HPG '09, 77–89.

MEYER, Q., SÜSSMUTH, J., SUSSNER, G., STAMMINGER, M., AND GREINER, G. 2010. On floating-point normal vectors. *Comput. Graph. Forum 29*, 4, 1405–1409.

NICHOLS, G., AND WYMAN, C. 2009. Multiresolution splatting for indirect illumination. In *Proc. of I3D*, ACM, New York, NY, USA, I3D '09, 83–90.

PANTALEONI, J. 2011. Voxelpipe: a programmable pipeline for 3d voxelization. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ACM, New York, NY, USA, HPG '11, 99–106.

PAPAIOANNOU, G. 2011. Real-time diffuse global illumination using radiance hints. In *Proc. of HPG*, ACM, New York, NY, USA, HPG '11, 15–24.

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. Optix: a general purpose ray tracing engine. *ACM Trans. Graph. 29*, 4 (July), 66:1–66:13.

SZIRMAY-KALOS, L., ASZDI, B., LAZNYI, I., AND PREMECZ, M. 2005. Approximate ray-tracing on the GPU with distance impostors. *Computer Graphics Forum (Eurographics 05)*.

WANG, R., WANG, R., ZHOU, K., PAN, M., AND BAO, H. 2009. An efficient GPU-based approach for interactive global illumination. In *ACM SIGGRAPH 2009 papers*, ACM, New York, NY, USA, SIGGRAPH '09, 91:1–91:8.

WANG, T., 2007. Integer hash function, March. http://www.concentric.net/ ttwang/tech/inthash.htm.

YAO, C., WANG, B., CHAN, B., YONG, J., AND PAUL, J.-C. 2010. Multi-image based photon tracing for interactive global illumination of dynamic scenes. In *Proc. of EGSR*, Eurographics Association, Aire-la-Ville, Switzerland, EGSR'10, 1315–1324.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time kd-tree construction on graphics hardware. ACM, NY, NY, USA, SIGGRAPH Asia '08, 126:1–126:11.