Hardware-Accelerated Global Illumination by Image Space Photon Mapping

Morgan McGuire
Williams College

David Luebke
NVIDIA Corporation

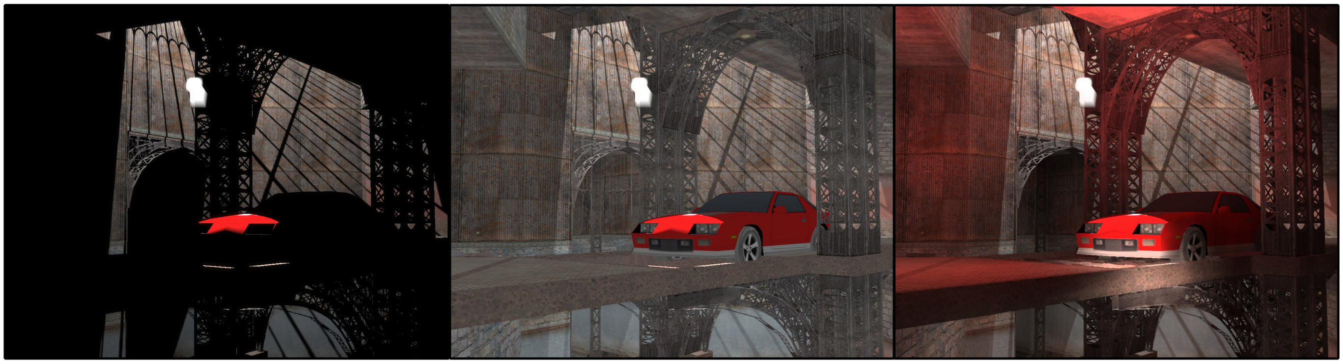


Figure 1: Image-space photon mapping can compute global illumination at interactive rates for scenes with multiple lights, caustics, shadows, and complex BSDFs. This scene renders at 26 Hz at 1920 × 1080. (Indirect and ambient intensity are amplified for comparison in this image.)

Abstract

We describe an extension to photon mapping that recasts the most expensive steps of the algorithm – the initial and final photon bounces – as image-space operations amenable to GPU acceleration. This enables global illumination for real-time applications as well as accelerating it for offline rendering.

Image Space Photon Mapping (ISPM) rasterizes a light-space *bounce map* of emitted photons surviving initial-bounce Russian roulette sampling on a GPU. It then traces photons conventionally on the CPU. Traditional photon mapping estimates final radiance by gathering photons from a *k*-d tree. ISPM instead scatters indirect illumination by rasterizing an array of *photon volumes*. Each volume bounds a filter kernel based on the a priori probability density of each photon path. These two steps exploit the fact that initial path segments from point lights and final ones into a pinhole camera each have a common center of projection. An optional step uses joint bilateral upsampling of irradiance to reduce the fill requirements of rasterizing photon volumes. ISPM preserves the accurate and physically-based nature of photon mapping, supports arbitrary BSDFs, and captures both high- and low-frequency illumination effects such as caustics and diffuse color interreflection. An implementation on a consumer GPU and 8-core CPU renders high-quality global illumination at up to 26 Hz at HD (1920×1080) resolution, for complex scenes containing moving objects and lights.

Keywords: photon mapping, global illumination, photon volumes

1 Introduction

Many important applications can benefit from accurate and efficient illumination computation, including offline rendering (e.g., film), realtime rendering (games), architecture and urban planning (lighting design and analysis), and manufacturing (design and review). Researchers have proposed many algorithms that make different tradeoffs between accuracy, simplicity, and performance for simulating surface-scattering phenomena due to global illumination. Pure geometric ray/photon tracing in the Whitted [1980]-Jensen [1996; 2001]-style provides near-ideal results, but at minutes-per-frame is too slow for dynamic illumination in real-time applications. By contrast, a series of recent global illumination methods described in Section 2 use splatting, shadow maps, and other image-space techniques to achieve interactive performance, but sacrifice accuracy and generality in the process. Speaking broadly, geometric ray-based approaches tend to be desirable properties from both correctness and software engineering perspectives, working for all scenes and illumination phenomena without special cases. Approaches that exploit rasterization hardware tend to limit generality and are more challenging to integrate into real systems, but are orders of magnitude faster. An ideal algorithm should be *consistent*–meaning that the limit of the radiance estimate converges to the true mean, e.g., for photon mapping, as the photon and bounce count approach infinity–and physically-based algorithm like photon mapping, while exploiting the speed of rasterization.

Jensen [1996] introduced **photon mapping**, which has since been expanded by several methods. All of these simulate light transport forward along rays from emitters through multiple **bounces** (i.e., scattering events) against the scene, and represent the resulting incident radiance samples as a **photon map** of stored **photons** at the bounce locations. At each bounce scattering is performed by Russian roulette sampling against the surface's bidirectional scattering distribution function (BSDF). To produce a visible image, a renderer then simulates transport backwards from the camera and estimates incident radiance at each visible point from nearby samples in the photon map. Traditional photon mapping algorithms, e.g., [Jensen 1996; Jensen 2001], simulate transport by geometric ray tracing and implement the radiance estimate by gathering nearest neighbors from a *k*-d tree photon map.

We introduce **Image Space Photon Mapping** (ISPM), a complete, consistent, and efficient solution to the rendering equation for point and directional lights with a pinhole camera. ISPM directly accelerates photon mapping using image space techniques, dramatically increasing its performance without sacrificing accuracy, elegance, and generality, through three key observations:

1. **The initial bounce is expensive**: the initial photon-tracing segment from the emitter to the first bounce must trace the most photons[1] (see Section 6.3).
2. **The final bounce is expensive**: the final segment to the camera requires the most computation because the algorithm must sum over many photons at each pixel.
3. These expensive initial and final bounces **each have a single center of projection** (assuming point emitters and pinhole camera model), and thus can be computed using rasterization.

By exploiting these observations we achieve substantial speedups over both traditional CPU-based and prior GPU-accelerated photon mapping techniques. Our experimental implementation targets game-like environments with a combination of static background geometry (e.g., game levels) and dynamic lights and objects (e.g., characters, doors, and destructible objects). We report performance on real game assets up to 26 Hz at 1920 × 1080 for scenes of up to 160k triangles in the potentially (indirectly) visible set and consider scenes with up to 1.2M triangles total.

We introduce the *bounce map*, which accelerates the initial bounce, and the *photon volume*, which accelerates the radiance estimation on the final bounce. Neither decreases the quality of the result from traditional photon mapping, thus both are appropriate for both real-time and offline cinema-quality rendering. For game scenes, rasterization of photon volumes generally proves to be the limiting factor of ISPM performance on today's GPUs. We address this by optionally upsampling sparse indirect illumination with joint bilateral filtering, which dramatically enhances performance by decreasing image quality.

1.1 Limitations and Advantages

Like traditional shadow maps, ISPM does not support area light sources or participating media, and translucent surfaces require special handling. For small radiance estimation kernels and no upsampling, ISPM results are mathematically identical to traditional photon mapping, which is consistent but biased because any non-zero kernel extent biases the estimator [Jensen 2001]. As with traditional photon mapping, the forward photon trace requires the last surface hit before the eye to have a finite BSDF sample (i.e., not refractive or mirror reflective), so mirrors and translucent surfaces require backward tracing or rasterization approximations of it. Under these limitations and for suitable kernel size and photon count, the only artifact observed is that indirect illumination fades out as the viewer comes very close (within the scatter kernel) to a surface and photon volumes are clipped by the near plane. For many scenes, the precision of indirect illumination is limited by the CPU portion of the photon trace.

ISPM extends previous hardware-accelerated and screen-space global illumination methods with:

- Unlimited indirect bounce count.
- Arbitrary BSDFs on intermediate bounces.
- Arbitrary finite BSDFs on the final bounce.
- A unified solution for all surface-scattering phenomena.

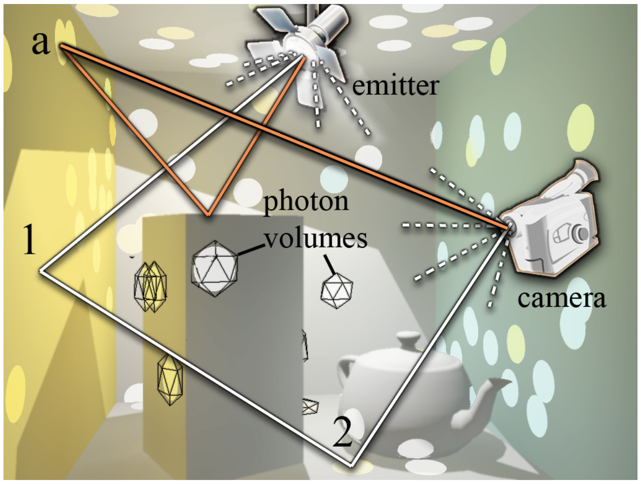


Figure 2: We highlight a few ISPM light transport paths (lines), photons (discs), and photon volumes (wireframe) in a rendered scene. Light path segments from emitters and to the camera have a common center of projection, so we can compute these with rasterization on the GPU. ISPM stores 16k photons (from 40k emitted) to generate the actual rendering, blending individual photons with a conic falloff (see Section 5.1) for this scene.

ISPM's primary advantage is algorithmic efficiency: it enables realtime illumination quality approaching that of offline photon mapping, and it can also accelerate offline rendering where the camera and light restrictions are acceptable. A second advantage is practicality. Our experimental implementation is surprisingly simple: about 600 C++ and GLSL statements inserted into a deferred-shading game engine. A single code path handles all surface-scattering phenomena, spanning high- and low-frequency illumination effects including reflected and refracted caustics, diffuse color bleeding, and contact shadows. The current alternative is combining special case methods for each phenomena. The problem with that alternative is that special cases often interact poorly with one another at both a software engineering level and in the final image. Ray tracing is a simple way to simulate complex phenomena and ISPM inherits that elegance.

1.2 Overview

Here we give an overview of computing indirect illumination using ISPM. We use "image space" to denote any 2D projected space; "screen space" is the camera's image space and "light space" is the light's. The ISPM algorithm is:

1. Rasterize the scene from the eye to create a screen space deferred-shading G-buffer [Saito and Takahashi 1990].
2. For each light, rasterize a **bounce map** in light space.
3. Advance bounced photons by world-space ray tracing.
4. Scatter the final photons in screen space, invoking illumination on the G-buffer by rasterizing **photon volumes**.

The bounce map resembles a shadow map in which BSDF scattering and Russian roulette are applied to a photon at each pixel. The bounce map stores a buffer rendered from the viewpoint of the light in which each pixel stores a position, power, and direction representing respectively the nearest surface point at that pixel, the exitant power of the photon bounced from that point. Power is zero and direction undefined for absorbed photons.

[1] ISPM and traditional photon mapping do not *store* 0- or 1-bounce photons, since they represent the same contribution as emitted and direct illumination, but they must still be simulated to reach the indirect bounces.

The intermediate world-space ray tracing step follows classic photon tracing. We currently implement this on the CPU in straightforward fashion and transfer the result back to the GPU. The final scatter operation renders a bounding volume about each photon to the screen. Within the volume, a gaussian filter kernel is applied to reconstruct the total indirect illumination at all affected pixels from the sparse samples. Note that most photons are never considered at most pixels, an important optimization enabled by the image-space nature of ISPM and the rasterization-based scatter of photon volumes. We thus avoid the *k*-d tree used by traditional gather-based photon mapping to reduce the cost of finding relevant photons.

Figure 2 illustrates photons and light transport paths in ISPM. In this figure we brighten a small number of photons for illustration purposes, causing bright discs in the neighborhoods where the photon influences the surface geometry. These neighborhoods are defined by the photon volumes, which we implement as oriented and scaled icosohedrons.

ISPM uses *path density estimation* [Herzog et al. 2007] to maintain constant sampling variance across the scene, scaling each photon volume's kernel radius inversely with the a priori likelihood of sampling that path under Russian roulette. For example, this will result in small filter kernels in caustic regions, better capturing the resulting fine details.

2 Related Work

2.1 Photon Tracing

The concept of tracing photons forward from the light source into the scene was introduced to the graphics community by Appel [1968]. A seminal paper by Jensen [1996] introduced photon mapping, which decouples illumination from visibility by tracing photons forward from lights and storing them in a *k*-d tree, and then produces an image by tracing eye rays backwards and gathering nearby photons to approximate illumination where each ray intersects the scene. In practice, photon maps are used only for indirect illumination and direct illumination is computed explicitly to reduce variance. We summarize conventional photon mapping in Appendix A. Many variations of photon mapping have been proposed; for example Ma and McCool [2002] use a hashgrid rather than a *k*-d tree to store the photons. Purcell et al. [2003] produced the first GPU implementation of photon mapping, using a GPU ray tracer [Purcell et al. 2002] to trace photons which they scatter into a regular grid using a technique called *stencil routing*. Larsen and Christensen [2004] augmented a similar tracer with hemi-cube final gathering, similar to hardware accelerated radiosity methods.

Jensen [2001] introduced projection maps, which are essentially shadow maps employed to avoid emitting photons into empty space. These are distinct from our bounce maps, which store the result of a full first-bounce scattering simulation on the GPU, including importance sampling the BSDF on the GPU. The bounce map leverages both the efficiency of rasterization for perspective projection and visible surface determination (to avoid rays into empty space and find the intersection) and the massive parallelism of GPU architectures (to sample the BSDF at many pixels simultaneously).

Note that while it can simulate caustic paths with infinite peaks in the BSDF along a transport path, photon mapping requires the final bounce before the eye to have a finite BSDF sample (i.e., not perfect refraction or mirror reflection). ISPM inherits this limitation. Both traditional and image-space photon mapping depend on another algorithm for rendering perfect reflection and refraction. For example, our result figures use a combination of dynamic and static environment maps.

2.2 Virtual Point Lights

Keller [1997] introduced a family of methods that trace *virtual point lights* (VPLs) into the scene, potentially bouncing off surfaces according to material reflectance properties, then render the scene lit by each VPL. This can be seen as tracing a very small number of photons (e.g., 100-1000 VPLs for interactive use) and trying to cover the undersampling with a final-gather step (equivalent to local illumination by the VPLs, which fit well into GPU pipelines).

Since a major cost in this process is rendering the shadow map for each VPL, significant subsequent work has gone into optimizing that process. Ritschel et al. [2008b] use precomputed shadow maps for rigid objects; Laine et al. [2007] amortize the VPL shadow map creation over multiple frames and by sharing maps between VPLs. Dachsbacher et al. [2007] and Dong et al. [2007] use implicit visibility computations to avoid the shadow maps altogether. Ritschel et al. [2008a] use splatting and pull-push reconstruction to compute extremely low level of detail shadow maps, and packs them into a single texture for memory coherence.

Note that photon map photons are *not* VPLs. Virtual point light sources represent samples of *exitant* illumination. The photons in photon mapping are a set of *incident* illumination samples. During deferred shading for indirect illumination, a surface point is only affected by nearby photons, whereas *all* VPLs must be considered. Photons thus scale better for large numbers of samples, and the major limitation of VPLs is undersampling: scenes containing many thin parts or holes, and illumination effects such as caustics, cannot be adequately captured with small numbers of VPLs.

2.3 Reflective Shadow Map (one bounce)

Reflective shadow map (RSM) techniques compute single-bounce global illumination via deferred gather [Dachsbacher and Stamminger 2005] or splatting [Dachsbacher and Stamminger 2006]. They avoid the shadow map cost of instant radiosity by simply ignoring occlusion for indirect illumination (often compensating by incorporating an ambient occlusion term). Nichols and Wyman [2009] use a multiresolution RSM to compute low-frequency components at low resolution, conserving fill-rate.

2.4 Interactive Ray Tracing

Fast geometric ray tracing has received considerable attention in the past few years, beginning with seminal papers by Parker et al. [1999] and Wald et al. [2001]. Researchers have introduced many clever techniques to speed up construction of acceleration structures and the traversal of rays through an acceleration structure. Wald et al. [2007] give a relatively recent survey of this fast-moving field. Researchers have also demonstrated fast ray tracing traversal and construction techniques on GPU architectures [Zhou et al. 2008; Horn et al. 2007; Popov et al. 2007; Robison 2009].

Notably we do not currently use any advanced ray tracing techniques in our current implementation, since by shifting the expensive initial and final photon path segments to the GPU we no longer need a highly optimized ray tracer to achieve the performance results reported in Section 6. The initial and final segments also represent the most coherent ray workloads, and many modern ray traversal techniques such as packet tracing [Wald and Slusallek 2001] and frustum tracing [Reshetov et al. 2005] would likely be of limited use on the remaining incoherent rays. Our straightforward (200 lines of code) multithreaded CPU ray tracer uses a median-split *k*-d tree and approximates deforming objects with bounding sphere hierarchies.

2.5 Splatting Global Illumination

Splatting is a point-based rendering method that projects a sample onto the screen, often using rasterization. Sturzlinger and Bastos [1997] introduced global illumination splatting for offline rendering, with the surface BSDF assumed constant across the splat and the stencil buffer used to identify surfaces within the kernel extent. More recent related methods [Lavignotte and Paulin 2003; Krüger et al. 2006] limit the final bounce to a Lambertian BSDF;

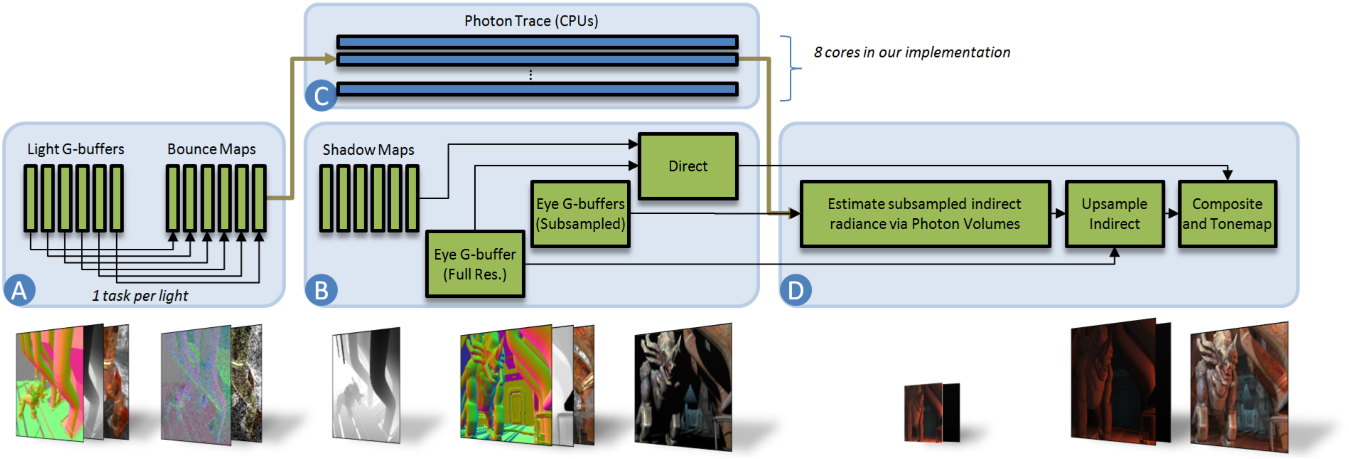


Figure 3: ISPM algorithm overview and task scheduling to address architecture constraints: GPUs have high latency, so a task should not use its immediate upstream neighbor's output; CPU photon tracing and GPU radiance estimation tasks require the most time; thick arrows are barrier synchronization points. Images represent the buffers for the NS2 scene computed by each box and widths are proportional to runtimes.

this allows fast 2D accumulation of photons by painting intensity over a diffuse texture map instead of evaluating the rendering equation at each pixel. Lavignotte and Paulin [2003] were the first to specifically target GPU architectures for scalable parallel evaluation. They introduced a series of approximations like a constant kernel, grouping similar photons, and simplifying the radiance estimate to fit real hardware and avoid data dependence. Their primary contribution is a method for correcting boundary bias resulting from the constant kernel. ISPM extends their work to full photon-mapping quality through the bounce map and photon volumes, the latter avoiding boundary bias more efficiently than [Lavignotte and Paulin 2003].

Krüger et al. [2006] introduced a method for rendering caustics that operates entirely in screen space on a GPU. We extend this idea by changing the emission screen-space to the light's frame and drop back to world-space geometry for intermediate bounces to avoid their limitation of missing bounces at surfaces culled by the view frame.

Ritschel et al. [2008a] use splatting for illumination differently, as level-of-detail method for rendering coarse shadow maps, and often combined with additive alpha blending as an inexpensive gather filter (e.g., for rendering caustic maps from the light's viewpoint [Shah and Konttinen 2007; Wyman and Dachsbacher 2008; Wyman 2008; Wyman and Nichols 2009].

Dachsbacher and Stamminger [2006] use screen-space splatting to accelerate final gathering by directly splatting light onto the scene in an RSM algorithm instead of treating the VPLs as local sources; Nichols and Wyman [2009] vary splat size for efficiency in the context of that algorithm.

Our photon volumes most closely resemble the photon splats of Herzog et al [2007]. They use screen-space splatting for the final gather step of an offline photon mapping ray tracer that renders in about one minute per frame. Their "splats" are real 3D volumes with spatially-varying kernels like ours, unlike previous 2D splatting methods (e.g., [Krüger et al. 2006]). We introduce the new term "photon volumes" to emphasize the 3D nature, and because ISPM photon volumes extend Herzog et al.'s 3D splats by conforming more tightly to surfaces and avoiding their expensive cone estimation step. We also extend their idea of relating the filter kernel support to the inverse probability of the given photon path. For example, the kernel should be large for low-probability paths, such as multi-bounce indirect diffuse lighting, and small for high-probability paths, such as caustics. They observe that accurate path density is impractical and unnecessary; following their conclusions, we introduce an even more aggressive approximation to increase efficiency.

2.6 Relationship to GPU Shadowing Methods

The bounce map resembles a shadow map, and inherits its limitations. Photon volumes resemble shadow volumes, and receive the benefits of GPU architectural features intended for them, like fast depth tests, while sharing their drawbacks of near-plane clipping artifacts and high fill rate. However, while shadow volumes expend many pixels of their high fill rate on empty space that they pass through. Photon volumes conform tightly to surfaces and are only rendered within a few pixels of a visible surface. The overdraw of photon volumes is therefore more usefully applied to reducing variance.

3 Algorithm

Pseudocode We now describe the steps of the ISPM algorithm (depicted in Figure 3). Except where indicated, all steps take place on the GPU.

1. For each emitter:
   (a) Render shadow map.
   (b) Render G-buffer from the emitter's view.
   (c) Emit photons, bounce once, and store in a bounce map.
   (d) **(CPU)** Continue tracing bounce map photons through the scene until absorption, storing them before each bounce in the photon map.
2. Render G-buffer from the eye's view.
3. Compute direct illumination using shadow maps and deferred shading of the eye G-buffer.
4. Render indirect illumination by scattering photon volumes.
5. Render translucent surfaces back-to-front with direct, mirror, and refracted illumination only.

The original bounce map photons are not duplicated in the photon map and are are never converted to photon volumes. Reflections can be implemented by either environment maps and rasterization during the direct illumination step, or by backwards recursive ray tracing on the CPU, with recursive global illumination approximated by an ambient model or a traditional photon map gather. For surfaces seen *through* translucent surfaces there are three options: (1) CPU backwards ray tracing of refracted contribution; (2) no refraction for visible rays, although photons do refract, creating caustics and colored illumination; or (3) environment maps and other GPU methods for distorting the background image. Backwards ray tracing gives physically correct results but is very expensive. Most games settle for environment maps today.

Pipeline scheduling ISPM is designed for parallel (and potentially long) hardware pipelines found in GPUs, DSPs, SPUs, and multi-core CPUs. Tasks should avoid reading buffers immediately after rendering to avoid data hazard stalls, and must not stall processors on each other at data transfer. We present the following scheduling specifically for CPUs and a GPU working in concert, which, differs from the logical order just presented. Figure 3 shows our task scheduling plan. Because GPU tasks are asynchronously initiated by the CPU, the CPU command issue order is:

1. Box A commands
2. Begin asynchronous GPU→CPU DMA bounce map (e.g., `glReadPixels` to a Pixel Buffer Object)
3. Box B commands
4. Box C CPU-side trace (includes `glMapBuffer`, which now blocks if the DMA from step (2) has not completed)
5. Begin asynchronous CPU→GPU DMA photon volumes (e.g., `glTexImage` from a PBO)
6. Box D commands (the GPU may still be executing box B)

4 Photon Tracing

4.1 Data Structures

Photons contain the fields in Table 1, which extend Jensen's photons with the index of refraction of the current material, Herzog et al.'s path density, and our normal to the surface the photon most recently hit. The $\vec{n}_p$ and ρ_p values are used to shape the radiance estimation kernel at each photon, which biases radiance estimation away from certain artifacts when there are few photons in the scene but has no impact on the limiting case of large numbers of photons. The bounce map, live photons during the CPU trace, and photon map are all simply arrays of **photons**.

Geometry buffers (**G-buffers**) [Saito and Takahashi 1990] store deferred shading information from the eye or an emitter's viewpoint. This includes the BSDF parameters, world-space position, world-space surface normal, and depth value. An emitter's G-buffer depth buffer is a shadow map, although ISPM renders separate shadow maps for direct illumination so that the resolution can be controlled independently.

Bounce maps store *outgoing* photons immediately after the first, direct-illumination bounce, which have the same energy as *incoming* photons immediately before the second bounce, but are located at the other end of that path segment. This is in contrast to the photon map, which stores *incoming* photons immediately before each indirect bounce (in fact, the bounce map is equivalent to very high-density VPL field computed on the GPU). Bounce map photons are not used for the radiance estimate. Instead, they store the intermediate result of photon tracing so that we can interrupt the computation and move it from the GPU to the CPU at the point where the algorithm's structure changes the preferred processor architecture. Because of this, bounce map data are similar but not identical to photon map data (Table 1). The bounce map does not require surface normal $\vec{n}_p$ because that is only used in the radiance estimate. It does require the index of refraction η_i, which will be needed at a later bounce when a photon exits a transmissive material. Not all photons scatter, so we encode those that are absorbed as $\Phi_i = 0\,\mathrm{W}$.

Symbol	Definition	Bounce Map	Photon Map
Φ_o or Φ_i	Power (W)	float[3]	float[3]
ρ_p	Path density estimate	(as $\lvert\vec{\omega}_o\rvert - 1$)	(as $\lvert\vec{\omega}_i\rvert - 1$)
η_i	Index of refraction	float[1]	—
$\vec{x}$	Photon position	float[1] (depth)	float[3]
$\vec{n}_p$	Normal to surface hit	—	float[3]
$\vec{\omega}_o$ or $\vec{\omega}_i$	Incident light vector	float[3]	float[3]
	floats required	8	12

Table 1: Photon notation and storage for bounce and photon maps. The storage scheme minimizes bandwidth during rasterization and CPU-GPU transfers. We pack unit-length $\vec{\omega}$ with non-negative ρ_p as $\vec{\omega} * (\rho_p + 1)$ (the +1 avoids dividing by zero when unpacking); bounce map $\vec{x}$ is the light's G-buffer depth value.

Precomputed random numbers enable a seedable random number generator (RNG) to ensure frame coherence for static portions of the scene, where temporal variation is very noticeable. We seed the RNG for scattering in the bounce map by a hash of the light position and pixel within the light's bounce map. For CPU ray tracing scattering, we re-seed the RNG for each photon path with a hash of its world space in the bounce map .

4.2 Initial Bounce

The bounce maps represent the initial bounce of photons from each light source, efficiently computed using rasterization. We create one bounce map for each emitter by rasterizing a rectangle over the emitter's G-buffer, which contains a description of the surfaces and their BSDFs. A fragment shader scatters the photon traveling through each pixel center by importance sampling the BSDF. We render omnidirectional lights using a six-view "bounce cubemap", but could also use parabolic projections following e.g. Ritschel et al. [2008a]. At each pixel, we must correct Φ_i to account for two factors: texture size roundoff and angular distortion. First, the bounce map aspect ratio is determined by the light's field of view. The total number of pixels it contains is proportional to Φ_L/Φ_T, the ratio of the light's power to the total power of all lights. The indicated size from those two constraints is not necessarily realizable as the integer area of a rectangle, so we round up to the nearest realizable size and then scale Φ_i within each bounce map by the remaining factor. Second, the solid angle of each pixel decreases with distance from the center of the bounce map. We cancel this by scaling Φ_i by an inverse cosine (as is done in hemicube radiosity).

4.3 Remaining Bounces

After the initial bounce computed on the GPU, ISPM traces surviving photons through the scene on a CPU with geometric ray tracing just as in traditional photon map tracing [Jensen 2001], but stores more information in the photon map. In addition to Φ_i, $\vec{\omega}_i$, and $\vec{x}$, the ISPM photon map stores the unit normal $\hat{n}_p$ to the surface hit by a photon and a coarse estimate ρ_p of the a priori probability density of this photon having scattered this many times. It uses these to shape the photon volume and filter kernel as described in sections 5–5.1. Path density ρ_p is the product of the BSDF samples at each scattering event, but because the BSDF sample is on the range zero to infinity, that gives a poor kernel size bound [Herzog et al. 2007]. After experimenting with more sophisticated methods for estimating ρ_p based on Herzog et al. [2007], we found that we could obtain equally useful estimates with the more efficient approximation of setting ρ_p to the product of an ad hoc constant based on the importance sampling at each bounce. We used $\rho_L/1000$ when importance sampling took the Lambertian scattering code path, $\rho_S/10$ for glossy scattering, and ρ_S for perfect mirrors and refraction.

In traditional photon mapping, the cost of the radiance estimate typically subsumes the cost of building a ray tracing acceleration structure, such as a *k*-d tree of surfaces, for photons to traverse.

Our radiance estimate is sufficiently fast that constructing an acceleration structure per frame can become a bottleneck. (Note that this structure is not needed for creating the bounce maps, since those are rendered from the same data structures used for visible surface rasterization.) The rapid construction and traversal of ray tracing acceleration structures is well studied (see Section 2.4). However, since this lies outside the core contributions of our algorithm, we use simple techniques appropriate to our example application domain of video games.

First, so we create a k-d tree for static scenery in an offline preprocess. For articulated rigid bodies, we precompute the k-d tree for each part and update each frame a bounding sphere hierarchy over all parts, and transform each ray into object space when it intersects a part's bounding sphere. For fully dynamic objects (e.g., skinned characters or physically simulated deformation), we use a bounding sphere hierarchy as a proxy for the actual geometry, refined such that all spheres are within the maximum filter kernel radius. This guarantees that photons deposited on the proxy surface will still illuminate the full-detail rasterized model. Since direct illumination and first indirect bounce use the full-detail model, the use of this proxy only affects second- and higher- bounce indirect illumination–which is often so diffuse that the approximation is imperceptible. This can introduce visible artifacts when a caustic casts a shadow. Such artifacts could be avoided by the use of an acceleration structure on the full dynamic model, rebuilt each frame using the techniques surveyed by Wald et al. [2007].

5 Radiance Estimate by Scattering

Traditional photon mapping gathers: each pixel at world space $\vec{s}$ gathers radiance from nearby photon hits at $\vec{x}$, where the photon's power is weighted by a filter kernel κ. ISPM instead scatters radiance from photons to nearby pixels, using the same weighting. This is mathematically identical, but better suited for rasterization. In effect we switch the inner and outer loops of the radiance estimate (see Appendix A), so that our outer iteration is over photons instead of surface points.

For each photon, ISPM synthesizes and renders in eye space a *photon volume* that bounds the non-zero portion of the kernel. ISPM then invokes the illumination computation on all visible surfaces within this volume. The filter kernel is radially symmetric, although it is compressed to an ellipsoid as described in the following subsection. The fragments generated by rasterizing the photon volume will conservatively include any pixels that might be affected by the photon, with the (previously rendered) eye-space depth buffer culling surfaces not visible in the final image (note that hierarchical depth-culling GPU hardware makes this elimination especially efficient).

For each rasterized photon volume pixel, ISPM reads the deferred shading parameters stored in the eye-space G-buffer and computes the photon's incremental contribution to that pixel,

$$\Delta L_o(\vec{s},\vec{\omega}_o) = f(\vec{s},\vec{\omega}_i,\vec{\omega}_o) * \max(0,\vec{\omega}_i\cdot\vec{n}) * \Phi_i * \kappa(\vec{x}-\vec{s},\vec{n}_p) \tag{1}$$

This is the BSDF scaled by projected area and κ for the photon-to-surface distance at that pixel. In OpenGL, these per-pixel operations can be implemented as a fragment shader and the resulting contribution additively blended with the direct illumination image.

Any sufficiently large bounding volume shape will produce correct results, however overly conservative volumes will create many unecessary fragments and can cause the application to be fill-bound. Since the nonzero volume of the kernel is an oriented ellipsoid, we use instanced icosohedrons to provide a tighter bound than bounding boxes while remaining relatively inexpensive to render.

We note that unlike most illumination "splatting" approaches, which actually accumulate the values of 2D splats where they pass the depth test, we splat a 3D shape in world-space and then invoke the illumination computation on the surfaces that lie within it. That allows use of the true BSDF for anisotropic, glossy reflective, and glossy transmissive scattering at the final bounce. In many ways this approach more closely resembles shadow- and fog-volume rendering than traditional splatting.

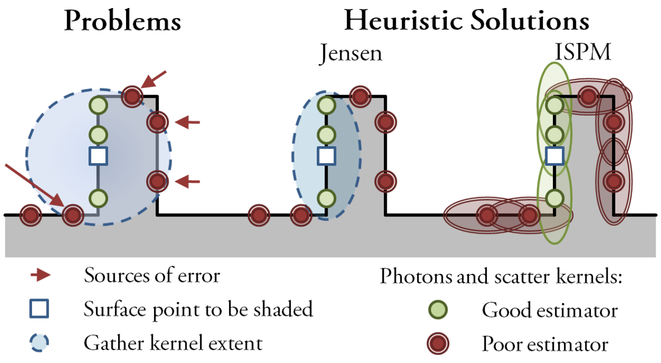


Figure 4: At corners and on thin features, photon-surface point pairs with different illumination conditions that lie within the radiance estimation kernel are a source of error. Compressing the kernel along the normal to the shading surface $\vec{n}$ [Jensen 96] or the normal to the surface hit by each photon $\vec{n}_p$ [ISPM] are mitigating heuristics.

5.1 Choosing Filter Kernel κ

A filter kernel should ideally invert the sampling function to achieve perfect reconstruction of illumination. This ideal is unachievable for photon mapping because the sampling function is too complex: it is effectively the entire photon tracing system [Herzog et al. 2007]. One reasonable kernel is a ball of constant density with radius r chosen such that g photons are likely to fall within it. This is the kernel used for "direct visualization" of the photon map by Jensen [1996]. This produces a visible discontinuity at the ball's boundary, so a 3D gaussian density falloff function within a finite sphere is typically used instead to produce smooth illumination.

The radius of the kernel affects variance. Large radii decrease variance (i.e., noise) in the indirect illumination at the expense of aliasing (i.e., blurring). Aliasing is particularly problematic at geometric corners (Figure 4 left), where illumination conditions are often discontinuous, and at nearby parallel surfaces, where the radiance estimate changes quickly along the surface normal direction.

The conventional solution to this problem is to use different radii for different directions [Jensen 1996], as shown in Figure 4 (center). Compressing the kernel function to lie closer to the plane of the surface being illuminated reduces variance along the surface, where illumination is likely more smooth, while avoiding aliasing along the normal direction, which is likely less smooth. This heuristic works extremely well on planes, but on curved surfaces only limited compression can be applied or the curve will escape the kernel.

Since we scatter rather than gather photons, we can not efficiently change the kernel radius per surface point. Instead we apply a different but equivalent heuristic: we compress the kernel along the normal to the surface hit by the photon instead of the normal to the surface being illuminated, as shown in Figure 4 (right).

We represent the κ scatter filter kernel as a 1D falloff function on the unit interval, mapped symmetrically to a 3D ellipsoid support region. The ellipsoid has major radii both of length r_{xy} for the axes in the plane of the surface and minor radius of $r_z \leq r_{xy}$ along the axis corresponding to the normal $\vec{n}_p$ to the surface hit by the photon. An efficient implementation of the scatter filter is vital since it is evaluated for every visible pixel of every photon volume. Fortunately, it can be implemented with a single fetch from a 1D texture representing κ as a function of radial distance within the ellipsoid and a few arithmetic operations to compute that distance as a texture coordinate. Let texture coordinate t be the radial distance between the photon and the surface point scaled with respect to the ellipsoid's inhomogenous axes to the unit interval. It is given by

$$\begin{aligned} t &= \frac{|\vec{x}-\vec{s}|}{r_{xy}}\left(1-\left|\frac{(\vec{x}-\vec{s})}{|\vec{x}-\vec{s}|}\cdot\vec{n}_p\right|\frac{r_{xy}-r_z}{r_z}\right) && (2)\\ \kappa(\vec{x}-\vec{s},\vec{n}_p) &= \text{kernel1D}[t] && (3)\end{aligned}$$

where $\vec{x}$ is the location of the photon and $\vec{s}$ is the location of the surface point stored in the G-buffer at this pixel. When $t > 1$, point $\vec{s}$ is outside the extent of the kernel, so $\kappa = 0$.

5.2 Subsampling Radiance

When fill-limited by rendering photon volumes, we can optionally subsample radiance from the photons and then use geometry-aware filtering to upsample the resulting screen space radiance estimate to the final image resolution.

Note that we do not perform shading and then interpolate pixel color; instead, the upsampled radiance applied to the high-resolution BSDF parameters stored in the G-buffer produce the final pixel colors. We perform geometry-aware upsampling by applying a joint bilateral filter with a box filter and weights based on 2D bilinear interpolation, normals, and depth between the low- and high-resolution eye G-buffers.

While this works quite well in practice (see Figure 12), we should note that interpolating low-resolution radiance onto a high-resolution image can not only produce aliasing (blurring) but will introduce statistical inconsistency, so the final image is no longer guaranteed to converge to the correct result. This optional upsampling step is the first point in the algorithm where we diverge from consistent rendering, and should be omitted for applications that demand accuracy at the expense of peak performance, such as architectural simulation and cinematic rendering. For applications such as games, the upsampling technique brings a large benefit in performance for almost imperceptible visual cost.

6 Results

6.1 Implementation Details

Platform. All results use a NVIDIA GeForce GTX 280 GPU for all rasterization steps, using OpenGL and the GLSL shading language. We implement photon volumes as instanced icosohedrons, use GL framebuffer objects for the render-to-texture steps, and GL pixel buffer objects for GPU↔CPU transfers. We use 3.2 GHz dual-processor quad-core Intel Extreme CPU with 2 GB memory running Microsoft Windows Vista 32-bit, and use all 8 cores for CPU tracing. To avoid synchronizing the photon map we maintain 8 separate arrays and concatenate them when tracing completes. When reporting timing results, we time GPU tasks with the `GL_TIMER_QUERY` extension and our CPU tasks with Windows `QueryPerformanceCounter`.

BSDF. We use an energy conserving, physically plausible isotropic "über-BSDF" from [G3D 2009], which combines Lambertian, glossy specular, Fresnel, mirror reflection, and refractive transmissive terms. This full BSDF is used for photon tracing, including alpha masking and Fresnel terms, as well as for deferred shading during rasterization steps.

Random numbers. To reduce the run-time cost of generating random numbers according to various distributions on CPU and GPU, we pre-compute 2M samples from scalar uniform, sphere uniform, and sphere cosine distributions. The random number seed is the index from which to start reading in these arrays.

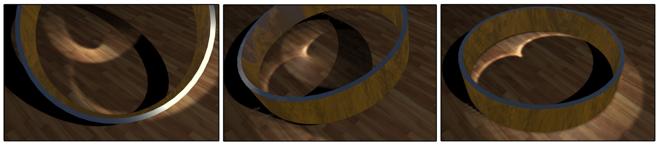


Figure 5: A reflective ring creates a cardiod caustic. This scene renders at 24 Hz at 1920 × 1080 with 101k emitted photons.

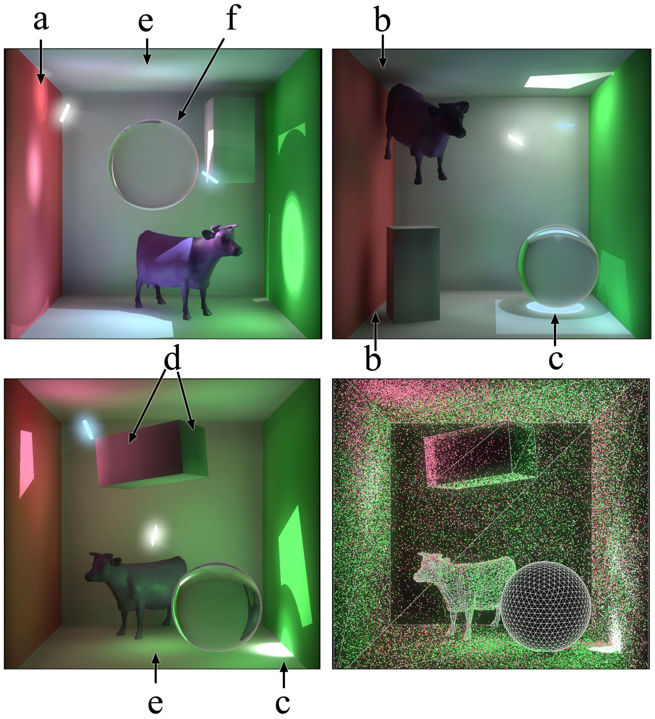


Figure 6: Multi-bounce global illumination effects: (a) reflective caustic, (b) contact shadows, (c) refractive caustics, (d) diffuse interreflection, (e) refractive caustic from indirect illumination, and (f) Fresnel reflection.

No visibility culling. We do not perform any sophisticated potentially visible set (PVS) computations: except for one experiment on the NS2 scene and one on the Ironworks scene, we render all geometry and perform all lighting for an entire scene (e.g., a whole game level) every frame. Real applications such as games would likely use PVS computations, or simply omit indirect illumination from distant lights. The NS2 experiment (Figure 9 / top rows of Table 2) shows that accurate culling can yield a 10× performance gain on a typical game scene, so this is an important area for analysis in future work.

6.2 Illustrative Results

Figure 5 demonstrates a cardoid caustic, which is a standard test for indirect illumination. Note the sharpness of the caustic, and the absence of noise as the rotating ring causes the caustic to expand.

Figure 6 shows a Cornell box containing a glossy purple cow, a glass sphere (η = 1.3, slightly reflective, low Lambertian term), a diffuse white box, and white and blue spot lights. The images demonstrate several complex multiple-bounce global illumination phenomena, from color bleeding by diffuse interreflection to a refractive caustic from *indirect* illumination. The bottom right subfigure shows the wireframe and photon hit locations for that frame.

Figures 7 and 8 show caustics in complex scenes. The water in Figure 8 is modeled as a mesh on the GPU and a bump-mapped

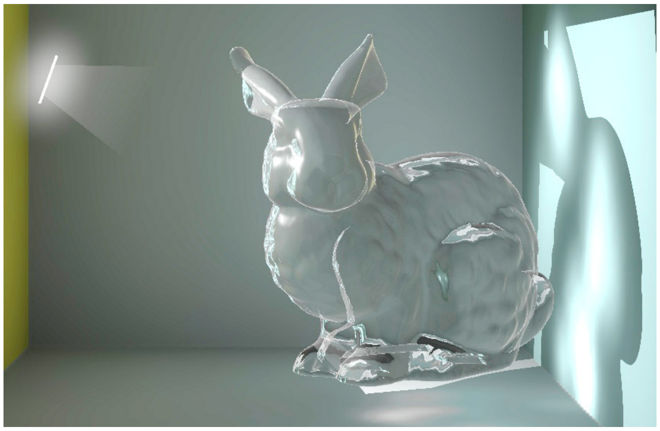


Figure 7: Caustics from a complex object. Eye-ray refraction is computed for front faces using a dynamic environment map.

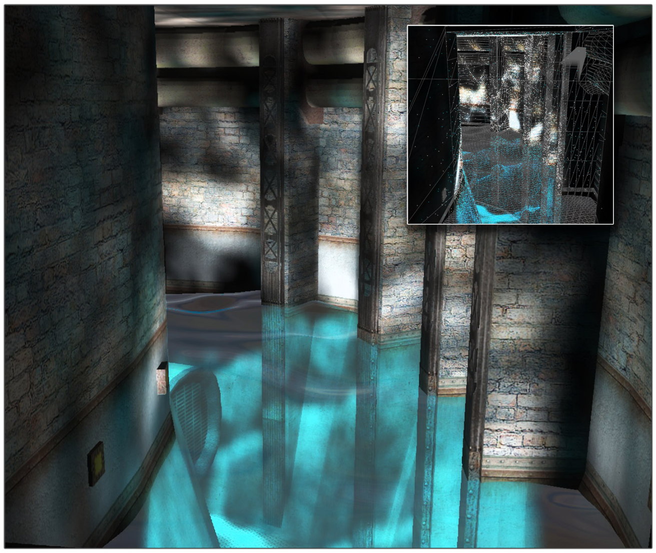


Figure 8: Reflected and refracted caustics from water on the *Quake Live* Ironworks map (lightmaps and artifical caustic textures from the original map were removed). *Inset:* Wireframe and photon map.

plane on the CPU. The light shines directly on the water, so all wall illumination is due to reflective and refractive caustics.

The NS2 scene contains 21 movable rigid instances of the Onos alien character model from the *Natural Selection 2* game [Unknown Worlds 2009] and static level geometry from the Nexus 6 map of the *Tremulous* game [Tre 2009]. All lights in this scene are white or blue: the red and yellow illumination is from white lights reflecting off colored walls.

6.3 Analysis

Photon attrition. Our use of the bounce map exploits the fact that the initial bounce (direct illumination from emitter to surface) comprises the most photons, and subsequent steps involve significantly fewer photons because of Russian roulette sampling. Figure 10 analyzes photon attrition on our example scenes and supports the importance of this improvement. For scenes in which a majority of photons do not survive the first bounce, optimizing that bounce can yield performance improvement of up to 300% for the forward simulation of 1-bounce indirect illumination. We observe nearly asymptotic behavior in practice because bounce map computation is less than 1 ms for all scenes in our experiments.

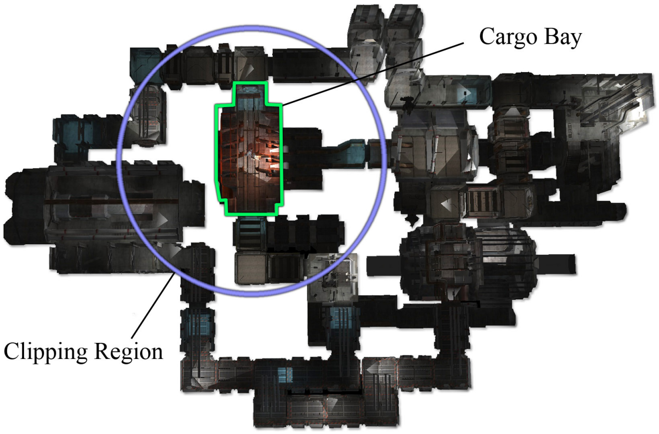


Figure 9: Top view of the NS2 scene. The central room is the Cargo Bay, seen from first-person in Figure 13. The outer circle marks the conservative PVS distance at which lights and objects are culled for a viewer in the center of the room. The scene contains 1.2M polygons, of which 310k are on the 21 dynamic objects, and 30 lights emit a total of 98k photons that are simulated for 3 indirect bounces. All lights have full global illumination computed every frame. At 1920×1080 it renders at 2.0 Hz from this viewpoint, where everything is in the visible set. For comparison, rendering with direct illumination and shadow maps only achieves 6.4 Hz from this viewpoint. When inside the map, culling to the outer circle (404k polys) renders at 15 Hz, and culling to the exact set of Cargo Bay (73k polys) renders at 21 Hz.

Artifacts. As the camera comes very close to a surface, the photon volumes on that surface are clipped by the near plane before the surface itself is. The surface therefore incorrectly loses illumination and slowly darkens. This artifact resembles a contact shadow from the viewer's approaching head, which may be unobjectionable for a first-person rendering but is inappropriate for a third-person view. We explored two corrections: converting partially clipped photon volumes to full-screen rectangles, and rendering them as backfaces with depth testing disabled. Both corrects the artifact at a tremendous fill rate cost. This is analogous to the problems with shadow volume near-plane clipping and the inefficiency of all known solutions. A reasonable alternative (beyond simply accepting it for first-person rendering) is to simply restrict the viewpoint from approaching the surface closer than the photon volume minor radius, which is typically 0.1m for the human-scale scenes in our experiments. We note that many applications already impose a similar restriction to avoid clipping nearby geometry or penetrating it with

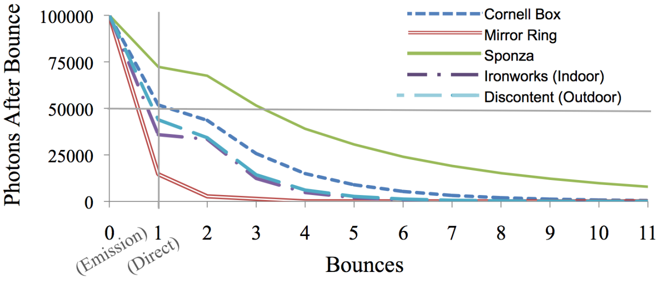


Figure 10: Photon attrition due to Russian roulette sampling for the example scenes. The results support our claim that rasterizing first-bounce photons via the bounce map significantly reduces the number of photons to be simulated; in most scenes, by at least 50%.

1: Scene	2: Figure	3: Polygons	4: Static Polys	5: Dynamic Polys	6: Dynamic Objects	7: Emitters	8: Emitted Photons	9: Indirect Bounces	10: Resolution	11: Subsampling	12: Total Photon Hits	13: Stored Photons	14: GPU Direct + Shadow Maps + Mirror + Transparency (ms)	15: GPU Bounce Maps (ms)	16: All CPU <-> GPU Copies (ms)	17: CPU Trace (ms)	18: GPU Radiance Estimate (ms)	19: Direct Only FPS	20: Full Global FPS
Ironworks (Car)	**1**	**161,612**	**141,492**	**20,120**	**20**	**2**	**8,448**	**1**	**1920x1080**	**4x4**	**7,162**	**3,450**	**14.2**	**0.1**	**6.2**	**17.2**	**8.1**	**121.8**	**26.5**
		161,612	141,492	20,120	20	2	8,448	3	1920x1080	4x4	8,846	5,134	14.2	0.1	6.3	24.8	10.2	121.8	20.6
Ironworks (Water)	**8**	**121,962**	**66,734**	**55,228**	**1**	**1**	**26,896**	**1**	**1920x1080**	**4x4**	**37,806**	**18,852**	**8.9**	**0.1**	**13.5**	**49**	**8.2**	**159.9**	**13.4**
		121,962	66,734	55,228	1	1	40,000	1	1920x1080	2x2	56,689	28,273	9	0.1	19.7	72.7	16.9	159.9	8.7
NS2 (Cargo Bay PVS)	**13**	**73,081**	**58,319**	**14,762**	**1**	**4**	**20,356**	**3**	**1920x1080**	**6x6**	**12,228**	**7,143**	**12.5**	**0.1**	**8**	**21.6**	**9.6**	**138.9**	**21.4**
(Cargo Bay PVS)		73,081	58,319	14,762	1	4	20356	1	1920x1080	6x6	10,004	4,919	12.6	0.1	7.6	16.1	8.8	138.9	26.7
(Circled PVS)		404,773	257,153	147,620	10	4	20,356	3	1920x1080	6x6	12,578	7,489	17.2	0.1	8.3	29.9	21.0	95.7	15.2
(Full Map)	**9**	**1,190,405**	**880,403**	**310,002**	**21**	**30**	**95,948**	**3**	**1920x1080**	**6x6**	**97,982**	**61,000**	**150.2**	**0.7**	**67**	**284.4**	**14.5**	**6.5**	**2.0**
Glass Sphere	**11**	**8,034**	**4,022**	**4,012**	**2**	**5**	**26,640**	**2**	**512x512**	**6x6**	**19,343**	**10,941**	**2.2**	**0.6**	**9.8**	**8.9**	**3.8**	**199.0**	**38.8**
		8,034	4,022	4,012	2	5	137,780	3	512x512	3x3	108,569	66,447	2.5	0.8	36.1	31.0	21.4	199.0	10.3
		8,034	4,022	4,012	2	5	18,000	2	1920x1080	6x6	12,935	7,319	9.0	0.3	7.6	4.0	11.3	163.0	34.1
Objects in Box	**6**	**19,642**	**9,826**	**9,816**	**3**	**2**	**27,620**	**3**	**1920x1080**	**8x8**	**58,965**	**37,433**	**8.2**	**0.1**	**16.6**	**22.8**	**40.1**	**177.6**	**11.8**
		19,642	9,826	9,816	3	2	27,620	2	1920x1080	10x10	52,047	30,515	8.3	0.1	16.1	17.2	23.2	177.6	15.8
Sponza Atrium	**11**	**67,414**	**66,934**	**480**	**1**	**1**	**13,688**	**1**	**1920x1600**	**4x4**	**23,244**	**11,308**	**8.8**	**0.1**	**8.4**	**56.7**	**12.9**	**195.2**	**11.9**
		67,414	66,934	480	1	1	6,240	1	1920x1080	4x4	10,573	5,136	8.8	0.1	5.4	28.2	7.2	195.2	22.4
Glass Bunny	**7**	**277,814**	**138,912**	**138,902**	**1**	**1**	**29,928**	**2**	**1920x1080**	**6x6**	**55,139**	**32,767**	**9.8**	**0.1**	**15.1**	**45.4**	**35.3**	**137.0**	**9.6**
		277,814	138,912	138,902	1	1	29,928	2	1920x1080	16x16	55,139	32,767	9.4	0.1	15.1	45.3	10.1		12.8
Caustic Ring	**11**	**812**	**412**	**400**	**1**	**1**	**101,124**	**1**	**512x384**	**1x1**	**27,917**	**5,020**	**1.4**	**0.1**	**21.3**	**6.4**	**3.4**	**200**	**28**
		812	412	400	1	1	324,900	1	512x384	1x1	89,919	16,030	1.4	0.3	63.5	20.1	9.6	200	10.9
	5	**812**	**412**	**400**	**1**	**1**	**101,124**	**1**	**1920x1080**	**1x1**	**27917**	**5020**	**7.6**	**0.1**	**22.1**	**7.3**	**14.3**	**197.5**	**24.3**

Key: **Bold: Corresponds to a figure** Highlight: Changed from first row Highlight: Bottleneck

Table 2: Detailed performance statistics for scenes from the paper and accompanying video. Several experimental results are reported for each scene: highlighted parameters were varied from the first row for that scene and highlighted times are the bottlenecks.

a view model.

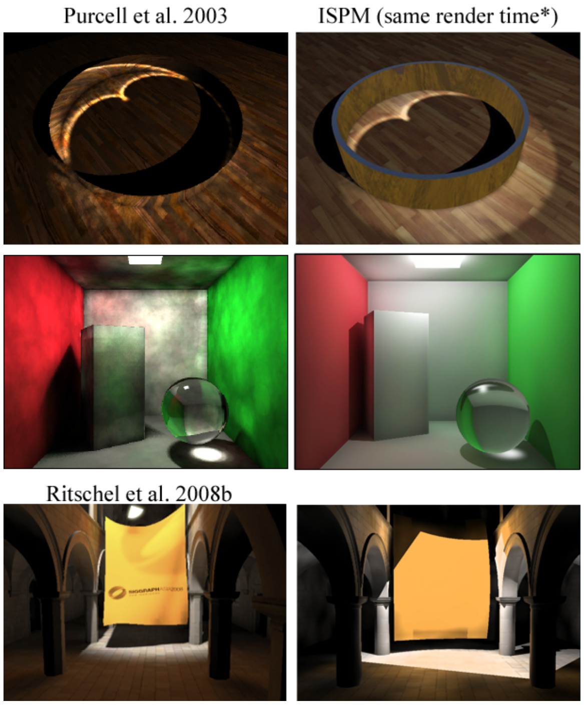
Performance. Table 2 reports performance for a number of experiments. Highlighted parameters were varied from the first row reported for that scene. We adjusted the photon count for each scene until we did not observe variance in diffuse reflected illumination under moving lights and objects.

Column 14, "GPU Direct ..." reports the GPU time for rendering the scene without indirect illumination, as in a typical game rendering engine. The corresponding frame rate is given in column 19, "Direct Only FPS"; that is the end-to-end frame rate measured on the CPU, including the buffer-flip, tone mapping, GUI rendering, and OS overhead, so it is not exactly the inverse of column 14. These gives a baseline for performance of the host engine against which to compare the ISPM frame rate in column 20.

Columns 14-18 report the computationally significant parts of ISPM on the CPU and GPU. Bottlenecks are highlighted in red. Geometrically-dense scenes tend to be CPU trace limited, and those with high photon counts and diffuse surfaces tend to be GPU fill-rate limited. For a few scenes with high photon counts but relatively low geometry and small photon volumes, the data transfer cost dominates. We observed a perfectly linear speedup of the CPU trace in the number of CPU cores, and linear slowdown in GPU radiance estimation with the surface area of the photon volumes on visible surfaces. We speculate that CPU efficiency could be improved by a factor of two because according to the Windows Vista task manager, the CPU load was 55-75% for all scenes.

Comparison to other GPU methods. The Purcell et al. [2003] GPU photon mapping algorithm targets the same illumination simulation as ISPM, but with different architectural features. It assumes a stencil buffer for sorting, but limited per-pixel programmability. The Ritschel et al. [2008b] imperfect shadow map algorithm targets a different problem with identical architecture to ISPM; it computes shadowed 1-bounce VPL illumination only. While no objective formal comparison to these is possible, we performed some informal comparisons in the course of our research. At the request of an anonymous reviewer, we report these in Figure 11 and Table 3.

The columns of Figure 11 compare previously-published image with ISPM results at the same resolution, rendered in the same (adjusted) time. Because published results were measured on different GPUs, we adjusted run time by the ratio of fill rate between GPUs. This combines rasterization and memory performance. Given that implementation, compiler, OS, and driver efficiency always affects


Purcell et al. 2003

ISPM (same render time*)

Ritschel et al. 2008b

Figure 11: Quality comparison against GPU photon mapping and imperfect shadow maps (* penalizing ISPM time for higher GPU fill rate.)

Scene, resolution and source	Previously reported time adjusted for GPU fill rate	ISPM time → speedup
Ring 512×384 [Purcell03]	$\frac{8100\text{ ms} * 3.8\text{ Gtex/s}}{48.2\text{ Gtex/s}} = 639\text{ ms}$	36 ms → 18×
Box 512×512 [Purcell03]	$\frac{47210\text{ ms} * 3.8\text{ Gtex/s}}{48.2\text{ Gtex/s}} = 3721\text{ ms}$	26 ms → 144×
Sponza 1920×1600 [Ritschel08b]	$\frac{217\text{ ms} * 36.8\text{ Gtex/s}}{48.2\text{ Gtex/s}} = 166\text{ ms}$	84 ms → 2×

Table 3: Adjusted render time for similar image quality between ISPM, [Purcell03] and [Ritschel08b]. The algorithms have different restrictions and target architectures, so this is a coarse comparison.

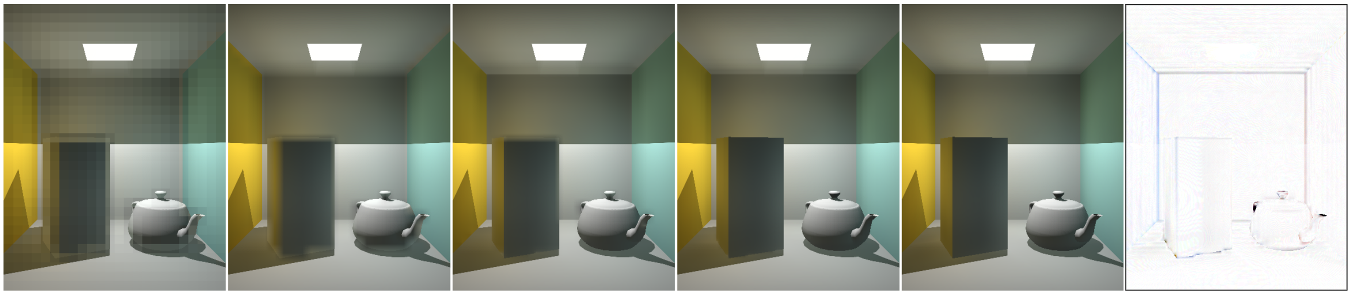

a) 16x16 Subsampling b) Bilinear weight c) × Normal weight d) × Depth weight e) 1x1 Sampling f) Error: 10 * |(d) - (e)|

Figure 12: Reconstructing subsampled radiance estimates in image space. (a) Worst-case 16×16 subsampled radiance with naïve nearest-neighbor upsampling. Multiplying by additional weights improves: (b) smoothness, (c) corners, (d) and edges. (e) Compare to indirect rendered at full resolution; (f) residual error is primarily at edges and on glancing surfaces, where L_o changes rapidly in screen space.

experimental software results, this at least gives a plausible estimate of how the algorithms might compare on identical hardware. Table 3 reports the complementary experiment, comparing the time to render images of similar overall quality. However, even 10× lower photon counts we were able to produce much lower variance than Purcell et al., and we were unable to produce contact shadows behind the columns of the quality of Ritschel et al., even with 1M photons. This is expected because ISPM has an improved kernel and Ritschel et al.'s algorithm is primarily for rendering contact shadows (at the loss of multiple bounces), but we stress that overall quality does not reflect specific phenomena.

ISPM's bottleneck is data transfer in the ring scene and photon-volume fill rate in the box scene. Purcell et al.'s photon mapping's bottleneck is *k*-d tree gathering for both. For the Sponza scene (bottom row), ISPM's bottleneck is the CPU photon trace against the highly-tessellated arches, while the Ritschel et al. imperfect shadow map bottleneck is shadow map fill rate.

Subsampling radiance. Figure 12 illustrates the application and artifacts of subsampling the radiance estimate as described in Section 5.2. Compare the decreased error when subsampled radiance is upsampled with a bilateral filter incorporating (progressively) 2D distance, surface normal disparity, and depth disparity.

7 Conclusion

Limitations. ISPM builds on shadow mapping and deferred shading, and shares their limiting assumptions. We assume rasterizable scene geometry such as triangle meshes. Note that triangles may be produced procedurally on the GPU, but in our current implementation this would require transferring triangles to the CPU or duplicating triangle generation on the CPU. Performing the photon tracing on the GPU would lift this restriction. To rasterize the final bounce we assume a pinhole camera model. Of perhaps more importance, rasterizing the initial bounce assumes point light sources. These can reasonably approximate small area lights (e.g., fluorescent banks) for indirect illumination, but are poor for large area lights (e.g., the sky). In future work, we would like to explore conservative rasterization and techniques from the soft shadow literature to extend ISPM to handle true area lights.

Finally, while ISPM can trace photons correctly for translucent or refractive surfaces, we cannot compute the radiance estimate for more than one point per pixel without a prohibitively expensive depth peeling step. In the case of a translucent foreground object with an opaque background, we could choose to estimate radiance on the background and use only direct illumination on the foreground, as in Figure 6, or to estimate radiance on the foreground and ignore indirect contributions on background pixels seen through the foreground object. The best choice depends on the material and opacity of the foreground object.

Multiple processors. We currently perform photon tracing on the CPU, which fully exploits all processors in the system but introduces unfortunate synchronization constraints and readback penalties. It also requires duplicating code paths, such as the BSDF evaluation. Several researchers have explored GPU ray tracers [Purcell et al. 2002; Horn et al. 2007; Popov et al. 2007] and commercial products are starting to appear [Robison 2009]. We plan to moving photon tracing to the GPU, which will simplify the system and reduce synchronization demands. On the other hand, multiple-GPU systems are common among video game enthusiasts and high-end commercial applications, and standard alternate-frame-rendering approaches built into the driver should transparently increase the effective fill rate. We also plan to explore partitioning the ISPM computation among multiple GPUs in pipelined fashion, which will increase utilization but re-introduce more complexity.

Discussion. ISPM possesses two main advantages over prior work. The first advantage is accuracy: because it is based on a consistent global illumination method, the algorithm will converge to the correct image and capture important effects in the limit. While approximations (such as our geometry-aware upsampling described in Section 5.2) may be necessary to achieve a particular performance goal, begining with an accurate algorithm reduces the complexity of interactions between techniques and increases the robustness and predictability of the rendering system.

The second, perhaps surprising, advantage of ISPM is simplicty: By combining ray tracing and rasterization, we can use the algorithm that maps naturally to each different type of transport path. Our total prototype implementation adds about 600 C++ and GLSL statements to an existing real-time rendering engine (about 200 for the ray tracer, the balance for the OpenGL upsampling and hybrid rendering machinery) and has proven considerably simpler to implement than a conventional software-only photon map. In particular, eliminating the photon *k*-d tree and final gather stages reduces the actual photon mapping process to straightforward ray tracing and BSDF evaluation. We also consider the final system more elegant than many prior techniques complicated by e.g. multi-layer data structures, cached per-frame data, techniques specific to particular graphics architectures, and special-case extensions such as separate caustic maps.

Summary. By recasting expensive steps of the photon mapping algorithm as image-space problems, we are able to simplify the overall algorithm (for example by eliminating the *k*-d tree on photons) and exploit the GPU for tremendous speedups. Our final system is able to render global illumination effects on scenes approaching the complexity of modern video games, achieving interactive rates at 2-megapixel resolution. With increasing computational power and more sophisticated ray tracing techniques, we

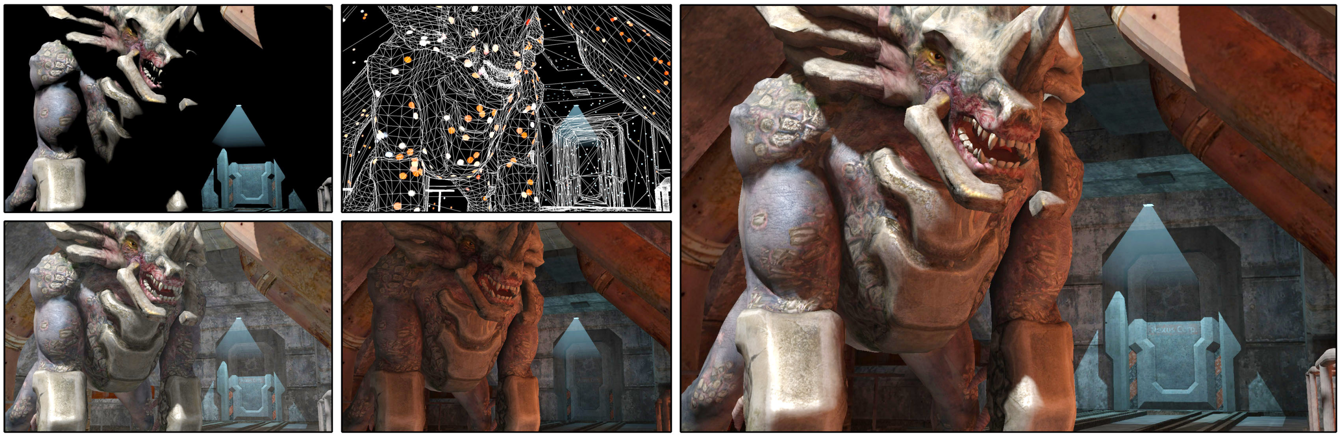


Figure 13: ISPM applied to the NS2 scene. *Left top:* direct illumination only. *Left bottom:* direct plus constant "ambient" term. *Middle top:* photons drawn as large spots on wireframe to illustrate the sampling and geometry. *Middle bottom:* ISPM's 3-bounce indirect illumination radiance estimate. *Right:* composite direct+indirect image rendered by ISPM; compare to the direct+ambient on the lower left. The potentially visible portion of the scene (Figure 9, Cargo Bay only) contains 58k polys on static objects and 15k on dynamic ones, and 4 lights that emit 20k photons total. We import all assets directly from games. This view renders at 21.4 Hz at 1920×1080.

believe this technique will soon prove practical for realtime applications.

Acknowledgments We thank the anonymous reviewers and Evan Hart for substantial suggestions, Unknown Worlds Entertainment for the the pre-release Onos model from *Natural Selection 2*, Dan Fast for 3D modeling, and Qi Mo and Kefei Lei for proofreading. Model sources: Nexus 6 from *Tremulous*, Ironworks from *Quake Live* (id Software), bunny from the Stanford 3D Scanning Repository, and Sponza Atrium by Marko Dabrovic.

References

APPEL, A. 1968. Some techniques for shading machine renderings of solids. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, ACM, New York, NY, USA, 37–45.

DACHSBACHER, C., AND STAMMINGER, M. 2005. Reflective shadow maps. In *Proc. of ACM SI3D '05*, ACM, New York, NY, USA, 203–231.

DACHSBACHER, C., AND STAMMINGER, M. 2006. Splatting indirect illumination. In *Proc. of ACM SI3D '06*, ACM, New York, NY, USA, 93–100.

DACHSBACHER, C., STAMMINGER, M., DRETTAKIS, G., AND DURAND, F. 2007. Implicit visibility and antiradiance for interactive global illumination. *ACM Trans. Graph. 26*, 3, 61.

DONG, Z., KAUTZ, J., THEOBALT, C., AND SEIDEL, H.-P. 2007. Interactive global illumination using implicit visibility. In *PG '07: Proceedings of the 15th Pacific Conference on Computer Graphics and Applications*, IEEE Computer Society, Washington, DC, USA, 77–86.

G3D, 2009. G3D Engine 8.0. http://g3d-cpp.sf.net.

HERZOG, R., HAVRAN, V., KINUWAKI, S., MYSZKOWSKI, K., AND SEIDEL, H.-P. 2007. Global illumination using photon ray splatting. In *Eurographics 2007*, vol. 26, 503–513.

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-d tree GPU raytracing. In *Proc. of ACM SI3D 2007*, ACM, New York, NY, USA, 167–174.

JENSEN, H. W., AND CHRISTENSEN, P. 2007. High quality rendering using ray tracing and photon mapping. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, ACM, New York, NY, USA, 1.

JENSEN, H. W. 1996. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, Springer-Verlag, London, UK, 21–30.

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

KELLER, A. 1997. Instant radiosity. In *SIGGRAPH '97: Proc. of the 24th annual conference on comp. graph. and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 49–56.

KRÜGER, J., BÜRGER, K., AND WESTERMANN, R. 2006. Interactive screen-space accurate photon tracing on GPUs. In *Rendering Techniques (Eurographics Symposium on Rendering - EGSR)*, 319–329.

LAINE, S., SARANSAARI, H., KONTKANEN, J., LEHTINEN, J., AND AILA, T. 2007. Incremental instant radiosity for real-time indirect illumination. In *Proceedings of Eurographics Symposium on Rendering 2007*, Eurographics Association.

LARSEN, B. D., AND CHRISTENSEN, N. 2004. Simulating photon mapping for real-time applications. In *Eurographics Symposium on Rendering*, A. K. Henrik Wann Jensen, Ed.

LAVIGNOTTE, F., AND PAULIN, M. 2003. Scalable photon splatting for global illumination. In *GRAPHITE '03: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, ACM, New York, NY, USA, 203–ff.

MA, V. C. H., AND MCCOOL, M. D. 2002. Low latency photon mapping using block hashing. In *HWWS '02: Proc. of the ACM SIGGRAPH/EUROGRAPHICS conf. on Graph. hardware*, Eurographics Association, Aire-la-Ville, Switzerland, 89–99.

NICHOLS, G., AND WYMAN, C. 2009. Multiresolution splatting for indirect illumination. *Proc. of ACM SI3D 2009* (Feb).

PARKER, S., MARTIN, W., SLOAN, P.-P., SHIRLEY, P., SMITS, B., AND HANSEN, C. 1999. Interactive ray tracing. In *Symposium on Interactive 3D Graphics: Interactive 3D*, 119–126.

POPOV, S., GÜNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum 26*, 3 (Sept.), 415–424.

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. *ACM Trans. Graph. 21*, 3, 703–712.

PURCELL, T. J., DONNER, C., CAMMARANO, M., JENSEN, H. W., AND HANRAHAN, P. 2003. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, Eurographics Association, 41–50.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. *ACM Trans. Graph. 24*, 3 (Aug.), 1176–1185.

RITSCHEL, T., GROSCH, T., KIM, M., SEIDEL, H.-P., DACHSBACHER, C., AND KAUTZ, J. 2008. Imperfect shadow maps for efficient computation of indirect illumination. *ACM Trans. Graph. 27*, 5, 1–8.

RITSCHEL, T., GROSCH, T., KAUTZ, J., AND SEIDEL, H.-P. 2008. Interactive global illumination based on coherent surface shadow maps. In *GI '08: Proceedings of graphics interface 2008*, Canadian Information Processing Society, Toronto, Ont., Canada, Canada, 185–192.

ROBISON, A., 2009. Interactive Ray Tracing on the GPU and NVIRT Overview. `http://www.nvidia.com/research`.

SAITO, T., AND TAKAHASHI, T. 1990. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph. 24*, 4, 197–206.

SHAH, M. A., AND KONTTINEN, J. 2007. Caustics mapping: An image-space technique for real-time caustics. *IEEE Trans. on Visualization and Computer Graph. 13*, 2, 272–280. Member-Sumanta Pattanaik.

STÜRZLINGER, W., AND BASTOS, R. 1997. Interactive rendering of globally illuminated glossy scenes. In *Proceedings of the Eurographics Workshop on Rendering Techniques '97*, Springer-Verlag, London, UK, 93–102.

2009. Tremulous. http://www.tremulous.net/.

UNKNOWN WORLDS, 2009. Natural Selection 2. http://www.unknownworlds.com/ns2/.

WALD, I., AND SLUSALLEK, P. 2001. State of the art in interactive ray tracing. In *State of the Art Reports, EUROGRAPHICS 2001*. EUROGRAPHICS, Manchester, United Kingdom, 21–42.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3, 153–164.

WALD, I., MARK, W. R., GÜNTHER, J., BOULOS, S., IZE, T., HUNT, W., PARKER, S. G., AND SHIRLEY, P. 2007. State of the art in ray tracing animated scenes. In *STAR Proceedings of Eurographics 2007*, Eurographics Association, D. Schmalstieg and J. Bittner, Eds., 89–116.

WHITTED, T. 1980. An improved illumination model for shaded display. *Commun. ACM 23*, 6, 343–349.

WYMAN, C., AND DACHSBACHER, C. 2008. Reducing noise in image-space caustics with variable-sized splatting. *jgt 13*, 1, 1–17.

WYMAN, C., AND NICHOLS, G. 2009. Adaptive caustic maps using deferred shading. *Computer Graphics Forum 28*, 2.

WYMAN, C. 2008. Hierarchical caustic maps. In *Proc. of ACM SI3D 2008*, ACM, New York, NY, USA, 163–171.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time kd-tree construction on graphics hardware. *ACM Trans. on Graph. 27*, 5 (Dec.), 126:1–126:11.

A Traditional Photon Mapping Summary

This section describes the traditional photon mapping algorithm by Jensen at al. [1996; 2001; 2007], on which ISPM is based.

We describe the pure photon mapping without final gathering. Final gathering is a variance reduction technique that can extend any global illumination algorithm. It adds one level of backwards tracing by sampling the incoming illumination at a visible point from other, randomly sampled points. Those other points are where the pure technique is applied. Final gathering is not used in ISPM or most efficient global illumination solutions.

Many quantities involved vary with wavelength. These can be represented as vectors of samples, e.g., as RGB vectors. We denote the sum over all wavelengths of a quantity c as $\sum_\lambda c$ and the mean over all wavelengths as $\bar{c}$.

A **photon** is described by world-space position $\vec{x}$ in meters; incident power, Φ_i, in Watts, which corresponds to the light color in a local illumination model; and world-space incident direction, $\vec{\omega}_i$. By convention, this is $\vec{\omega}_i$ opposite the propagation direction and corresponds to the light vector in a local illumination model.

The scene contains light sources and surfaces. Let Φ_L be the emissive power of a single light source L and $\Phi_T = \sum_L \Phi_L$ be the total power of all light sources in the scene. Let each surface have a spatially-varying bidirectional scattering distribution function (BSDF) $f(\vec{\omega}_i, \vec{\omega}_o)$ that is the differential probability of an incident photon from $\vec{\omega}_i$ scattering in direction $\vec{\omega}_o$. The BSDF may vary with wavelength.

The algorithm consists of two phases: photon simulation by forward ray tracing and radiance estimation by backwards ray tracing. These are linked by k-d tree of photons called a *photon map*.

Photon tracing computes the photon map, bouncing each photon a limited number of times. Each of N emitted photons may produce multiple stored photons. Repeat the following N times:

1. Emit one photon. This is a uniform stochastic sampling process, so light L emits with probability $\rho_e = \sum_\lambda \Phi_L / \sum_\lambda \Phi_T$. That photon then has power $\Phi_i \leftarrow \Phi_L / (N * \rho_e)$.

2. Repeat at most maxBounces times:

   (a) Trace the photon to its collision (i.e., bounce) $\vec{y}$ the scene. Update the photon with $\vec{x} \leftarrow \vec{y}$ and store the photon the photon map.

   (b) Let scatter probability $\rho_s = \int_\Omega f(\vec{\omega}_i, \vec{\omega}_o)(\vec{\omega}_o \cdot \vec{n}) d\vec{\omega}_o$, where f is the BSDF at the collision point.

   (c) With probability $1 - \rho_s$, simulate photon absorption by immediately exiting this loop (this is **Russian roulette** sampling).

   (d) Otherwise, sample the outgoing direction $\vec{\omega}_o$ with respect to f, and update the photon with $\vec{\omega}_i \leftarrow -\vec{\omega}_o$, $\Phi_i \leftarrow \Phi_i * \rho_s / \bar{\rho}_s$

Radiance estimates are used for shading at eye-visible points determined by rasterization or backwards ray-tracing. First, balance the photon map k-d tree once before any radiance estimates are performed. Then, at each point $\vec{s}$ with normal $\vec{n}$ and unit vector $\vec{\omega}_o$ towards the viewer (or towards the previous point, for recursive rays) to be shaded:

1. Let $L_o \leftarrow 0\,\mathrm{W}/(\mathrm{m}^2\,\mathrm{sr})$ be the current estimate of radiance reflected towards the viewer

2. Gather the photons nearest $\vec{s}$ from the photon map. Two methods for doing this are growing the gather radius until a constant number of photons have been sampled, and simply gathering from a constant radius. Regardless of the method chosen, let g be the number of photons and r be the gather radius.

3. For each photon at $\vec{x}$ within the radius:

   (a) Let $L_o \leftarrow L_o + f(\vec{s}, \vec{\omega}_i, \vec{\omega}_o) * \Phi_i * \max(0, \vec{\omega}_i \cdot \vec{n}) * \kappa(\vec{x} - \vec{s})$

The process is reconstructing illumination from sparse points. $\kappa(\vec{d})$ is the reconstruction filter kernel. The kernel chosen trades aliasing against noise. It must also normalize for the cross-sectional area of the filter. The simplest kernel is a disk: $\kappa(\vec{d}) = 1/(\pi r^2)$. A better kernel is a 3D gaussian, with distance from the mean $|\vec{d}|$.

Note that L_o measures radiance; it is constructed from the BSDF sample (sr^{-1} units), the photon power, and the normalized filter kernel sample which provides the inverse area. Step 3(a) is a discrete approximation of the rendering equation. Jensen [2001, page 79] proves that the estimate is consistent, and that increasing photon count and decreasing r can make variance arbitrarily small.

Because direct illumination can be efficiently computed by a local shading model and shadow rays or shadow maps, photon maps are typically used to estimate only the indirect (i.e., multi-bounce) radiance. In this case, ignore photons that have only made a single bounce during the radiance estimate.