Interactive Global Illumination for Improved Lighting Design Workflow

Wojciech K. Jarosz
Senior Thesis with Prof. Michael Garland
Department of Computer Science
University of Illinois, Urbana-Champaign, 2002

January 16, 2005

Contents

Abstract

In this paper, we present a hybrid software–hardware rendering technique which can compute and visualize global illumination effects in dynamic scenes at interactive rates. Our system uses a hardware splatting technique simular to, but developed independently of, Stüerzlinger *et al*. The technique involves a progressively refined photon tracing calculation capable of simulating a wide range of BRDFs. Photons in the scene are rendered to the screen as oriented Gaussian splats. This compact representation allows for rapid rendering of scenes with thousands of photons on consumer-level hardware.

In contrast to previous methods using simular techniques, our system does not use precomputed lighting, and is capable of achieving interactive feedback to object and light manipulations. Such a feature is invaluable to lighting designers dealing with complex globally-illuminated scenes. The progressive refinement algorithm allows for rapid preview during interaction, while producing higher quality images over time. The images produced also maintain a high correlation to the appearance of full renderings using a conventional Monte Carlo ray tracer.

1 Introduction

Global illumination arises in the real world through the interaction of light with the materials it encounters. As photons travel through space and hit surfaces, any number of outcomes are possible. Photons can be absorbed by, reflected by, or transmitted through surfaces, all based on the material properties of the object. Photons trace complex paths through the scene, interacting with the surfaces that they encounter. The material properties of a surface have an influence on incident photons, and therefore through multiple bounces, can have an effect on the illumination within another part of the scene. The integration of all these photon paths is what helps determine the lighting effects within a particular environment. This complex system is known as global illumination.

One of the goals of physically accurate and photo-realistic rendering is the indistinguishability of a rendering from a photograph. In order to faithfully represent the physical world in a rendered image, global illumination must be taken into account. The importance of this phenomenon in suspending the disbelief of the audience can be inferred from the increased use of global illumination in computer generated films and special effects. In the past, these sort of effects were achieved through the use of numerous, strategically positioned lights intended to simulate the indirect illumination falling onto objects. This large collection of extra lights had to be positioned individually and manually by lighting designers. With the increase of computational power and the availability of more advanced rendering techniques, more physically accurate methods have started to replace these tedious workarounds.

1.1 Motivation

Graphics hardware today is very successful at rendering diffuse, directly illumated polygons quickly. Virtually all 3D animation packages take advantage of this and allow for fully-shaded interactive viewing of a scene. This feature is incredibly helpful for a lighting designer because it provides real-time feedback for object and light manipulations. It greatly reduces the amount of time needed for test renders in order to see the effects of a particular lighting change.

As opposed to the behavior of direct illumination, indirect illumination, especially caustics, is fairly difficult for a human observer to predict. A slight movement of a light source might change the shape and location of a caustic dramatically. Therefore, in order to make sure that lighting changes have the desired effect, many test renders must again be performed. With the increased use of global illumination in the production environment, it would be similarly advantageous to interactively preview indirect illumination, as direct illumination already is. Seeing the change in shape and location of caustics and other indirect illumination effects interactively could dramatically improve the workflow of a lighting designer.

1.2 Global Illumination Methods

Current graphics hardware does not, however, directly support advanced global illumination algorithms. Therefore, the challenge is in finding a way to cleverly take advantage of the processing power of the GPU within current software-based global illumination techniques. In order to understand how a GPU could potentially be utilized, it is important to review these global illumination methods.

1.2.1 Radiosity

Radiosity is a finite element method where the scene must be discretized into patches. Form factors are calculated for each patch, accounting for diffuse illumination from all other patches within the scene. A form factor between two patches is conventionally computed in a simular way to direct lighting with ray tracing used to account for occlusion. This collection of form factors is then used when

solving, either directly or indirectly, a large linear matrix equation [5].

Interactive globally illuminated walk-throughs have been achieved in the past using the results of a radiosity calculation. The radiosity solution can be stored in the vertex colors of a mesh. Graphics hardware can then be used to render the scene using the pre-calculated illumination in real-time. This provides a partial solution in that only diffuse reflections can be simulated, and, aside from trivial changes such as turning individual lights on and off, lighting changes cannot be previewed dynamically. View-independent modifications have been implemented, but do not allow for interactive viewing [24]. The addition of dynamic lighting and object manipulation would require the radiosity calculation to be performed individually for each affected frame, impairing the interactivity of the preview as well.

1.2.2 Ray Tracing and Path Tracing

Ray tracing and path tracing methods rely on shooting out numerous rays in order to interrogate the illumination present at various locations within the scene. The lighting incident from all directions onto a surface is calculated by integrating these values over the hemisphere of directions. Optimizations to the brute force method have been developed, such as irradiance caching [36] and bi-directional path tracing [18]. Globally illuminated scenes with arbitrary material properties can be simulated using these method; however, they are still not suitable for interactive rendering.

A pure path tracing method does not seem like a good candidate for non-programmable graphics hardware acceleration due to the lack of both temporal and spatial caching. Modifications of both ray tracing [30], and path tracing [29], which utilize cached information, have been implemented for interactive preview. However, these generally rely on large distributed systems to offset the heavy cost associated with tracing many rays per pixel. Other solutions have been implemented which utilize the programmability of modern graphics hardware to trace rays rather than rasterize polygons [3, 19] in real-time.

1.2.3 Photon Mapping

Similar to path tracing, photon mapping shoots many rays into the scene in order to simulate global illumination effects. The algorithm, however, proceeds in the opposite direction of path tracing, tracing photons from lights instead of rays from the camera. In fact, photon mapping can be thought of as a special form of bi-directional path tracing [9, 18] which uses an intermediate caching step. As opposed to standard path tracing, caching is central to the photon mapping method. Lighting information from the photon simulation is stored in a three dimensional data structure for later integration into a Monte Carlo ray tracing pass.

The caching nature of photon mapping provides more opportunity for graphics hardware utilization than pure path tracing, and allows for arbitrary material properties, unlike radiosity based methods. Interactive walk-throughs of globally illuminated glossy scenes have been achieved utilizing photon mapping and graphics hardware [25]. Intended as a replacement for diffuse-only radiosity walk-throughs, the method did not investigate the feasibility of dynamic scene changes, using a pre-computed photon map for interactive rendering instead.

For our implementation of interactive global illumination, the photon mapping method was chosen due to both its flexibility in simulating complex reflection models, and its heavy use of exploitable caching.

2 The Photon Mapping Method

The following section outlines the photon mapping method, which follows the work presented in [9, 12, 11, 8, 7, 13, 4].

In photon mapping, image creation is split up into two distinct passes. The first pass constructs the photon map through a method simular to path tracing. “Photons” carrying flux are emitted from all light sources in the scene based on the emissive characteristics of each light. As these photons hit surfaces their information is recorded in the photon map, storing flux from a specific incoming direction. Storage of a low-level quanity such as flux as opposed to irradiance allows for simulation of a greater range of BRDFs when using the photon map in the rendering stage.

The rendering stage is a modified Monte Carlo ray tracer. The photon map is used in the rendering stage to help terminate recursion when simulating global illumination. The ray tracer accesses the information in the photon map in order to add lighting from caustics as well as indirect diffuse illumination. Other enhancements include using “shadow photons” to speed up shadow ray calculations [11], and the use of irradiance caching [36] or irradiance gradients [35] at Lambertian surfaces instead of standard path tracing.

2.1 Photon Tracing Phase

The photon map is constructed by shooting out a large number of “photons” (packets of flux) into the scene. When

a photon hits an object, an algorithm determines the fate of the photon based on the material properties of the surface. If the surface is at least partially diffuse then the photon is inserted into the photon map at the intersection point. If the surface at the intersection point is completely specular, then the photon is discarded. In addition, Russian Roulette, based on the reflective properties of the material, is used to determine if a photon is reflected, refracted, or absorbed. If the photon is not absorbed, the BRDF is used to determined the new direction for the photon.

For improved range searching speed, the photons are split up into two photon maps: the global photon map, and the caustic photon map. If volumetric effects are to be considered, a third, volume photon map can be constructed. For the global photon map, the scene is showered with global photons by shooting photons at each object in the scene. When these photons hit a diffuse surface, Russian Roulette determines if the photon will be reflected, or if it will terminate. If a global photon hits a diffuse surface and did not come directly from a light source (it has been reflected diffusely at least once) it is stored in the global photon map for retrieval in the second pass.

A much higher density of caustic photons is shot out at the specular objects (any object that has a non-zero reflection or transmission coefficient). Any photon that is transmitted or reflected onto a diffuse surface is stored in the caustic photon map. Caustics, as opposed to soft diffuse illumination, often exhibit very sharp details and therefore require a very high resolution simulation. Also, since caustics are very hard to simulate using path tracing, they are visualized directly with the photon map, which requires a much higher density of photons for acceptable results.

If volumetric effects are also considered, a separate volume photon map should be used for simulating global illumination effects in participating media. The volume photon map is also visualized directly; however, the details in the lighting are softened by the cloudy nature of the media and therefore the high density used for the caustic photon map is not neccassary here. In order to account for participating media, each photon that was shot for the global photon map is ray-marched through the medium, and deposited in the volume photon map if necessary.

2.2 Rendering Phase

Once the photon maps have been constructed, the second pass uses a modified Monte Carlo ray tracer to calculate the surface radiance of the closest object for each pixel on the screen. This can be solved using the rendering equation [16, 9]:

$$L_o(x,\vec{\omega}) = L_e(x,\vec{\omega}) + L_r(x,\vec{\omega}) \tag{1}$$

where L_o is the outgoing radiance from point x in direction $\vec{\omega}$, and is the sum of the radiance emitted by the surface, L_e, and the radiance reflected by the surface, L_r.

L_r can be described in terms of the local illumination model, and Equation 1 can be re-written as:

$$L_o(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \int_{\Omega} f_r(x,\vec{\omega}',\vec{\omega}) L_i(x,\vec{\omega}')(\vec{\omega}\cdot\vec{n})\, d\vec{\omega}' \tag{2}$$

where f_r is the BRDF of the surface, $L_i(x,\vec{\omega}')$ is the radiance incident on the surface from direction $\vec{\omega}'$, $(\vec{\omega}\cdot\vec{n})$ is the foreshortening term, and Ω represents the hemisphere of incoming directions.

$L_e(x,\vec{\omega})$ is strictly determined by the material properties at the hitpoint and can therefore be easily calculated directly. The recursive intergral, L_r, is however much more difficult to evaluate. It can be beneficial to separate L_r into several different components, each of which can be calculated more efficiently seperately, using different techniques.

Incoming radiance can be split up into a diffuse part, $L_{r,d}$, and a specular component, $L_{r,s}$.

$$L_r = L_{r,d} + L_{r,s} \tag{3}$$

Specular reflections, $L_{r,s}$, are highly directional and can therefore be evaluated efficiently with Monte Carlo integration using few samples. $L_{r,d}$, however, is not highly directionally localized and would require a large number of samples to integrate using a strictly Monte Carlo based method; therefore, a different approach would be more appropriate. Further examination shows that $L_{r,d}$ can be broken down further:

$$L_{r,d} = L_{i,l} + L_{i,c} + L_{i,d} \tag{4}$$

where $L_{i,l}$ is the light coming directly from light sources, $L_{i,c}$ are caustics coming directly off of specular surfaces, and $L_{i,d}$ is soft indirect lighting from multiple diffuse bounces.

Combining Equation 3 and Equation 4 we get:

$$L_r = L_{i,l} + L_{i,c} + L_{i,d} + L_{r,s} \tag{5}$$

Using the fact that an integral of a sum is the sum of

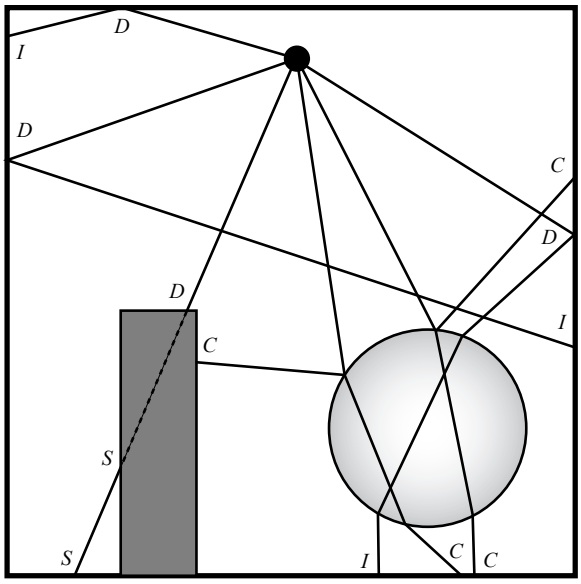


Figure 1: Various photon paths with resulting photon types. Caustic photons (C) occur after a specular bounce directly after being emitted from the light source. Direct photons (D) are deposited at diffuse surfaces on paths directly from the light source. Indirect photons (I) are stored after more than one diffuse bounce. Shadow photons (S) are stored at all intersections past the first along the initial photon path.

the integrals, we can therefore write this as:

$$\begin{aligned} L_r = & \int_\Omega f_r(x,\vec{\omega}',\vec{\omega}) L_{i,l}(x,\vec{\omega}')(\vec{\omega}\cdot\vec{n})\,d\vec{\omega}' + \\ & \int_\Omega f_{r,d}(x,\vec{\omega}',\vec{\omega}) L_{i,c}(x,\vec{\omega}')(\vec{\omega}\cdot\vec{n})\,d\vec{\omega}' + \\ & \int_\Omega f_{r,d}(x,\vec{\omega}',\vec{\omega}) L_{i,d}(x,\vec{\omega}')(\vec{\omega}\cdot\vec{n})\,d\vec{\omega}' + \\ & \int_\Omega f_{r,s}(x,\vec{\omega}',\vec{\omega})(L_{i,c}(x,\vec{\omega}') + L_{i,d}(x,\vec{\omega}'))(\vec{\omega}\cdot\vec{n})\,d\vec{\omega}' \end{aligned} \tag{6}$$

where the first component represents contribution from direct illumination, the second term simulates reflective and refractive caustics, the third term contributes illumination from multiple diffuse bounces, and the last term simulates specular and glossy reflections.

This requires the BRDF to be split into two parts:

$$f_r = f_{r,d} + f_{r,s} \tag{7}$$

where $f_{r,d}$ represents all reflection directions from Lambertian to slightly glossy, and $f_{r,s}$ incorporates slightly glossy to perfectly specular reflections.

2.2.1 Direct Illumination

The first part of Equation 6 is the local illumination. Many methods have been developed which handle this case fairly efficiently [23, 32, 38, 31]. Most of these methods rely on sending out a number of shadow rays towards points on each light in order to calculate visibility. Sending shadow rays to each light can become expensive if the number of lights in a scene is large. Various optimization techniques have been developed to handle these cases [34, 23]. In addition, it is possible to use the shadow photon information within the global photon map to optimize shadow ray calculations [11].

2.2.2 Caustics

With earlier approaches, caustics have been a very hard effect to simulate. Path tracing almost always fails except in some contrived scenes, so a Monte Carlo method does not seem optimal. Caustics can be much better handled with methods which start at the light and move out into the scene, as opposed to trying to find paths back to the lights from locations the eye sees. The information we gathered from our photon mapping simulation in the first phase is ideal for visualizing these effects; therefore, we calculate caustics by directly visualizing the caustic photon map.

2.2.3 Indirect Diffuse Illumination

The third part of Equation 6 is indirect diffuse illumination. This illumination has bounced off of a diffuse surface at least once already, and therefore changes very slowly. This component is calculated indirectly by performing a "final gather" on the photon map. This involves performing a path tracing like step where many rays are shot out from a query point to probe the radiance in the whole scene. In regular path tracing, those rays would in turn spawn more rays. However, with the photon map available, we can eliminate this recursion by looking up directly in the global photon map for the final gather rays. At completely Lambertian surface, a significant optimization can be made by utilizing irradiance caching [36] or irradiance gradients [35]. These schemes too would terminate recursion by looking up in the global photon map.

2.2.4 Specular Reflections

The last term in Equation 6 represents highly specular reflections. For this illumination, the BRDF is very localized, and Monte Carlo sampling works very well.

2.3 The Radiance Estimate

The photon map can be used to estimate the radiance leaving a point in a particular direction. The components, from Equation 6, which the radiance estimate will include depends on which photon map is used and what type of photons were inserted into the photon map. Each photon represents the amount of flux $\Delta\Phi_p$ coming in from a particular direction; therefore, we can integrate many photons over all directions using a BRDF to get the outgoing radiance.

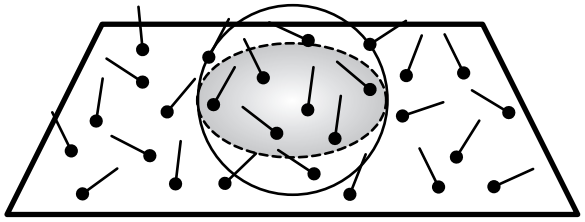


Figure 2: The radiance estimate is evaluated by locating the n nearest photons in the photon map using an encompassing sphere in range searching. The density of the photons is based on the area of the circle formed by the intersection of the sphere with the locally flat surface.

The radiance, L_r, for which we are trying to solve can be expressed as:

$$L_r(x,\vec{\omega}) = \int_{\Omega} f_r(x,\vec{\omega}',\vec{\omega}) L_i(x,\vec{\omega}')(\vec{\omega}\cdot\vec{n})\,d\vec{\omega}' \quad (8)$$

In order to approximate the L_r using the photon map, we will consider the M closest photons to x. These photons can be acquired using an efficient range searching algorithm which locates photons within a specified bounding volume. As long as many photons are used, and the local density of the photons at x is high, this should yield a reasonable approximation. We can rewrite Equation 8 in terms of flux, which can be approximated using the photon map:

$$\begin{aligned} L_r(x,\vec{\omega}) &= \int_{\Omega} f_r(x,\vec{\omega}',\vec{\omega}) \frac{d^2\Phi_i(x,\vec{\omega}')}{dA_i} \\ &\approx \sum_{p=1}^{M} f_r(x,\vec{\omega}_p,\vec{\omega}) \frac{\Delta\Phi_p(x,\vec{\omega}_p)}{\pi r^2} \end{aligned} \quad (9)$$

where r is the radius of the sphere which is expanded to contain the M photons.

The above technique uses a box filter, which gives the same amount of weight to all photons within the gathered sphere. In order to reduce blurring, a more advanced kernel can be used. In these cases, each photon would have a weight based on its distance from the query point, and the area by which the sum is divided would change based on the kernel. Some common kernels are discussed in [9].

2.4 Storing Photons

Since the photon map could possibly contain millions of photons which must be searched, two major concerns arise. The representation must be compact so that memory usage is reasonable, and the photons must be kept in some sort of structure which allows for fast range searches.

2.4.1 Photon Structure

In this context, photons represent flux hitting a surface from a given direction. This indicates that we need to store the energy (color) of the photon, the worldspace location of the photon, and its incoming direction. We can also add in an extra flag variable which will distinguish between global and caustic photons. All these values could be represented as floats; however, some of these do not need this precision. The power of the photon must be a high dynamic range color value since it is trying to simulate real world intensity values. In order to compact this, we can use Greg Ward's Real Pixels [33] format and

```
class Photon
{
    char power[4];             // Power of the photon.
    float pos[3];              // Position of the photon.
    unsigned char theta, phi;  // Incoming direction.
    char flags;                // Some extra flags.
}
```

Figure 3: A photon structure occupying only 19 bytes of memory.

pack the power into four bytes. Since the incoming direction also does not need to be very exact; it can be stored in longitude/latitude as only two bytes. Precision in the location of the photons cannot be sacrificed however, so this must remain represented as a vector of three floats. The resulting structure is shown in Figure 3. With this structure, each photon will only take up 19 bytes.

2.4.2 Balanced Kd-Tree

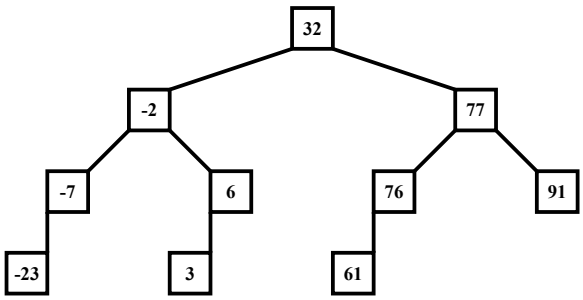
Photons are stored in a balanced kd-tree because of its fast range search capabilities [9, 7, 12]. Since we know the total number of photons we wish to store in the photon map in advance, we can create a static array which will represent the kd-tree. This heap-like structure eliminates the need to store extra child pointers, reducing the memory requirements for the photons by over 40%.

Initially, the collection of photons is kept as a flat, unstructured array. As photons are shot, they are simply inserted into the array sequentially. This keeps the photon shooting phase very quick. Once the photon map is full, the unstructured array is reorderd to correctly represent a kd-tree.

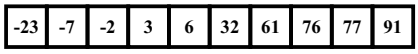
A proper heuristic must be determined to uniquely define a balanced binary tree as a flat array. One such ordering is a heap structure; however, we developed a modified representation in which the array is simply a flattened version of the original tree. This is equivalent to an inorder traversal of the tree. Figure 4 shows an example array for a binary tree using this encoding technique.

The flat array must be rearranged to fit this ordering. At each step in the algorithm, we must find the median photon – this is the root of the (sub)tree. We then partition the photons into two sets – all photons with a location less than or equal to the median, and all photons with a location greater than the median. This algorithm is then repeated on each of these subsets individually.

The `partition_list` algorithm in Figure 5 needs to find the median element and partition the list about this element. A perfect implementation of this sort of algorithm is `nth_element` found in the STL [15]. `nth_element`



(a)



(b)

Figure 4: An example one-dimensional left-balanced kd-tree. (a) shows the tree representation, while (b) shows the flattened encoding of the tree. The whole tree is flattened vertically, with the root being the median element. It is interesting to note that in the one-dimensional case, the flattened tree is the sorted list of elements.

is linear in time complexity, making the overall time complexity of `BuildTree` $O(n \lg n)$.

In order to expand this algorithm to more than one dimension, we need to partition the photon list about all three dimensions during our building process. One method of doing this is to split each node about alternating axes based on depth: the root node about the x-axis, nodes at the second level about the y-axis, nodes in the third level about the z-axis, and so on. Though this would correctly create a balanced kd-tree of three dimensions, it is not the optimal strategy. This method may split along an axis which is very small already, while splitting along an axis with a wider range would more quickly narrow a range search. A better technique would be to split about the axis with the greatest extent. This would partition the space more uniformly and can be done by maintaining a bounding box for the photons while balancing. Since there is no

```
1  BuildTree (lo, hi)
2  {
3      partition_list (from index lo to index hi);
4
5      // the root of this tree is now the median
6      // element, everything to the left of the median
7      // is the unordered left subtree, and everything
8      // to the right the unordered right subtree.
9
10     median_index = lo + (hi-lo)/2;
11
12     // Repeat on both subtrees
13     BuildTree (lo, median_index - 1);
14     BuildTree (median_index + 1, hi);
15 }
```

Figure 5: The algorithm which turns the unstructured array into a balanced kd-tree representation.

explicit formula for the splitting dimension of a node anymore, the splitting-plane axis needs to be stored with each photon. This can be easily accomodated by using a few bits of the `flags` member.

2.5 Range Searching

The range search is performed at every pixel, and in the case of final gather for the global photon map, many times per pixel; therefore, it plays a key roll in the efficiency of the photon mapping algorithm and must be optimized extensively in order to attain the shortest render times.

The `BuildTree` algortihm in the previous section creates a balanced tree from the photons. The balanced tree can guarantee an efficient $O(M \cdot \lg N)$ search time for M photons within a tree containing N photons total. If the tree were heavily skewed, however, then the search time could be significantly longer.

Range searching is performed in a recursive algorithm. First, a searching radius is determined. Once this is done, each step looks at the root of the current (sub)tree and determines if that photon is within the search radius. If it is, then the photon is accepted; otherwise, it is rejected. The algorithm then determines if the search radius intersects the bounding boxes of the left or right subtrees. If a subtree potentially intersects the search range then the procedure is repeated on it.

As it is searching, the algorithm keeps a found-heap of already located photons. The user specifies how many photons each search should locate, and the heap is fixed to that size. As the found-heap fills up, the furthest photon is kept at the head of the heap. Once the heap is full, we have found our desired number of photons. However, the found-heap could fill up before we have considered all of the photons in the photon map, and we would therefore retrieve photons that are not closest to the query point. Therefore, once the found-heap is full, we test each photon searched against the head of the found-heap. If the photon is closer to the query point than the head is, the head is removed from the found-heap and the current photon is added in its place. Figure 6 outlines the `rangeSearch` procedure that could be used for a one-dimensional kd-tree. The algorithm can easily scale to arbitrary dimensions. In order to expand the search algorithm for three-dimensional kd-trees, `range` needs to be modified to represent the one-dimensional distance on the axis specified by the photon's splitting plane.

2.6 Shooting Photons

A common area for error is in the implementation of the photon shooting algorithm. Since the final rendered image will be combining lighting from two completely different methods (ray traced direct lighting, illumination gathered from the photon maps), it is very important that the photons are shot with the proper power distribution, and in the proper directional distribution. In order to match the output produced with standard ray traced lights, we will assume that they are implemented to adhere to physical laws such as inverse squared falloff. If this is not the case, the photon shooting algorithms will need to be appropriately modified in order for the two methods to match.

In order to preserve the power distribution of the lights in the scene, photons should be shot with probability based on the brightness of the lightsource – more photons should be shot out from brighter light sources than from dimmer ones. This should be used, whenever possible, instead of scaling the power of each light's photon energy because it creates photons with simular intensities, providing better results when averaging during the radiance estimate [9].

```
void rangeSearch(ctr, rr, nphotons, heap, lo, hi)
{
    if (hi-lo >= 0)
    {
        median = (lo+hi)/2;
        dist = distance form this photon to ctr;

        if (dist < rr)
        {
            if (found_heap isn't full)
                add this photon to the found-heap
            else
                remove the heap of the found-heap, and
                add this photon to the found-heap
        }

        range = 1D distance from ctr to this photon;

        // range only crosses one side
        if (range > rr)
        {
            if (range <= 0)
                rangeSearch left subtree
            else
                rangeSearch right subtree
        }

        // range crosses both sides, search both sides
        else
            rangeSearch left and right subtrees
    }
}
```

Figure 6: The range searching algorithm.

2.6.1 Point Lights

Though they are completely non-physical, point lights are perhaps the most commonly and easily implemented light sources. Point lights emit light equally in all directions; hence, the photon shooting algorithm will shoot photons out in all directions with equal probability. We present two techniques which can be used to generate these outgoing directions.

The first method employes simple rejection sampling of points generated within a unit cube. Only points which are also within the unit sphere are accepted, and must then be normalized for use as a direction vector. Since the difference in volume of a unit cube and a unit sphere is small, a large percentage of sample are accepted. This allows for the rejection sampling method to perform efficiently. Pseudo-code which achieves this is presented in Figure 7.

An alternate approach is to transform two uniform random variables into polar space such that they are uniformally distributed on the surface of a unit sphere. The following transformation can be used to generate random points on the surface of a unit sphere:

$$\begin{aligned} \theta &= arccos(1-2r_1) \\ \phi &= 2\pi r_2 \end{aligned} \tag{10}$$

where r_1 and r_2 are two uniform random variables, and (θ, ϕ) are the polar coordinates of a point on the sphere [21]. Using this method, no rejection sampling needs to be performed – all generated samples are valid. However, our tests have shown that rejection sampling tends to be more efficient in this case because a large portion of samples are accepted, and there is no need to call trigonometric functions.

2.6.2 Spot Lights

Spot lights limit the emited light to a user specified cone: let us call this angle α. In order to shoot photons out in this distribution, we could use rejection sampling like with the point light; however, this could be very inefficient for small α. Instead, the second method for generating points on a unit sphere, expressed in Equation 10, can be used while limiting the range of r_2 to $[0, \alpha/2\pi]$. This point

```
point pointLightDirection()
{
    point d;
    do
    {
        d = point(random(),random(),random());

    } while(d.length() > 1.0);

    return d;
}
```

Figure 7: Rejection sampling algorithm which calculates uniformly distributed random points on a unit sphere.

must then be converted to the polar space defined by the spot light, which can be done using an ortho-normal basis constructed from the vector defining the direction of the spot light. See Section (4.2) and Figure 11 for more details about constructing ortho-normal bases.

2.6.3 Area Lights

True light sources are not infinitesimally small but have a finite area. Adding area to the light sources slightly increases the complexity of the photon shooting algorithm. In order to shoot photons out from area light sources, not only do we need to choose a correct outgoing direction according to the light's emission characteristics, but we must first choose a random location on the light's surface as the photon origin. Methods have been developed which generate random points on commonly used 2D and 3D geometry [21].

In order to choose a point v uniformally distributed on the surface of a triangular luminaire, the following transformation can be applied to two uniform random variables r_1 and r_2:

$$v = v_0 + s(v_1 - v_0) + t(v_2 - v_0) \tag{11}$$

where

$$s = 1 - \sqrt{1 - r_1}$$
$$t = (1 - s)r_2$$

and v_0, v_1, and v_2 are the vertices of the triangle.

In the real world, lights are often covered with shades used to diffuse the harshness of the outgoing light. A common shape of such shades can be simulated as a spherical light source. In order to simulate spherical lights, the outgoing directions and origins of photons can be calculated using the two methods discussed for point lights in Section (2.6.1).

Disks can also be used as a shape for light sources. Generation of points on the surface of a disk can be accomplished using the concentric map [22]. The translation for the first region of the map is:

$$r = r_1 R$$
$$\theta = \frac{\pi}{4}\frac{r_2}{r_1} \tag{12}$$

where (r,θ) are the polar coordinates of a point on a disk with radius R. Equations for the other regions of the map have simular formulations [22].

3 Interactive Photon Mapping

3.1 Speed Issues In Photon Mapping

Though photon mapping is currently one of the fastest methods for computing complex global illumination, in its standard implementation it does not perform at interactive rates. In order to attain the goal of interactive photon mapping, it is important to pinpoint the performance bottlenecks in traditional photon mapping. Once these areas are identified, they can be modified in order to perform more quickly, either by using a better algorithm, or by trading accuracy for performance.

3.1.1 Balancing

Both our balancing algorithm and the one developed by Jensen operate in $O(n \lg n)$ [10]. When included in a Monte Carlo ray tracer, which in itself is a very time consuming algorithm, the time needed to balance the photon map is neglegible. Table 1 shows that balancing $100,000$ photons takes less than a second on a Pentium II 400 MHz machine, which is equivalent to about 1% of the total render time. Interactive rates suggest that a full image must be displayed about ten times per second. Waiting a few

second to balance a million photons before they are even displayed on the screen is therefore unacceptable. However, these statistics show that other portions of the photon mapping algorithm should also be investigated for speed benefits.

3.1.2 Range Searching

Ranging searching is performed at least once per rendered pixel. In our test scene, the range searching accounted for about 52% of the total render time for images with caustics alone. For images which simulate diffuse indirect illumination, the range search is performed numerous times per pixel during final gather, so the total time performing range searches increases. However, the total render time also significantly increases due to the cost of shooting rays in the final gather step. We employ an optimization technique which precalculates irradiance values in the global photon map [4]. Using this optimization, in scenes with diffuse indirect illumination calculated using a final gather on the global photon map, range searches take about 47% of total rendering time. Range searching is certainly an ideal candidate for optimization within the standard photon mapping algorithm.

3.1.3 Ray Tracing

The Monte Carlo ray tracing stage of the photon mapping pipeline is certainly not interactive. Using a full Monte Carlo simulation, a rendered image could take minutes, or even hours, whereas a simplified rendering of the scene could be generated using a less flexible method on consumer level graphics hardware in real-time.

3.2 Modifying the Photon Mapping Method

One of the goals is to be able to implement an interactive form of photon mapping in the workflow of already existent animation packages. For this to be useful, the method must firstly be fast enough to preview global illumination interactively, providing the animator or lighting designer with real-time feedback for lighting changes. It would also be advantageous for the method to be simple enough for easy incorporation into current software. Lastly, for the preview to be useful in eliminating test renders, it must have a high corrleation with the final look of the rendered image.

The ray tracing stage of the photon mapping method can certainly be replaced by a Z-buffer render in hardware using a graphics API such as OpenGL. Such implementations already exist for previewing local illumination in commercial packages. However, in eliminating ray tracing, and moving from software into hardware, a great deal of flexibility is lost and the challenge becomes incorporating the photon mapping simulation into a fixed graphics pipeline.

Two methods come to mind for integrating the photon map into an OpenGL environment. The first method is a direct implementation of Jim Arvo's backward ray tracing [2] using hardware texturing capabilities. The second method involves representing each individual photon in the photon map as a geometric entity which can be rendered in hardware. The two methods would still implement the photon shooting phase of conventional photon mapping in software, but would represent those photons differently in the hardware accelerated rendering pass.

3.2.1 Backwards Ray Tracing

In backwards ray tracing, or light tracing, light rays are shot out into the scene in a manner virtually identical to photon mapping. The distinguishing characteristic of the backward ray tracing method is the way in which these photon hits are stored for later retrieval.

As rays of light are bounced around in the scene, energy packets are stored in textures on each diffusely reflecting surface. When a light ray hits a diffuse surface, instead of inserting a photon into the photon map, a small packet of energy is added to the corresponding pixel of the surface's illumination map. This method requires that all surfaces have pre-generated texture parametrazations for the mapping of the illumination map onto the object. Using the same function that looks up texture values for locations in worldspace, we can calculate into which pixels to add energy. Arvo performed bilinear interpolation and distributed the ray's energy into the four neighboring pixels of the hitpoint.

One drawback to this method is that the resolution of the illumination map must be chosen carefully to correspond with the amount of photons which are shot out. If few photons are shot out, the result will look "speckly." In order to alleviate this problem, a modified approach can be taken.

Instead of, or in addition to, distributing the light energy into four neighboring pixels, once the light tracing stage is complete, the whole illumination map can be blurred. This step can be performed efficiently in $O(n \cdot m)$ where n is the radius of the blurring kernel, and m is the size of the illumination map [6]. Further speed improvement can be realized by using SIMD instructions on most modern CPUs. The blurring will spread each photon's influence over a larger area, simulating an effect comparable to the radiance estimate in photon mapping. The blur radius can

	Final Gather	Cautics Photons			Global Photons			Range search	Total
		Count	Tracing	Balancing	Count	Tracing	Balancing		
a	OFF	0	0 sec	0 sec	0	0 sec	0 sec	0 sec	16.15 sec
b	ON	0	0 sec	0 sec	0	0 sec	0 sec	0 sec	115 sec
c	OFF	100,000	11.796 sec	0.632 sec	0	0 sec	0 sec	31.4 sec	60 sec
d	ON	100,000	11.575 sec	0.673 sec	50,000	2.527 sec	0.342 sec	114.4 sec	244.5 sec

Table 1: Render time statistics for a conventional photon mapping implementation on a Cornell box scene with various global illumination settings: standard local illumination ray tracing (a); Monte Carlo ray tracing with a final gather performed using an irradiance cache (b); ray tracing plus the caustic photon map (c); and (d), full global illumination simulation with final gather, global photon map, and caustic photon map.

be increased as the photon search radius is increased in the actual renderer, maintaining a loose correspondence between the final results.

By using an illumination map, and completely eliminating the three-dimensional photon map, the costly range searching and balancing can be completely avoided. The illumination maps can simply be rendered on top of each diffuse surface in the same way as normal texture maps. However, the replacement of the photon map with a two-dimensional texture introduces some new problems. Firstly, the nature of the algorithm ties the illumination to the geometric representation of the scene, a quality which was specifically avoided with the conventional photon mapping method. Specifically, each geometric entity needs to be polygonized and parameterized. Objects represented as implicit surfaces or fractals must be converted to polygonal representations, which can often be hard to achieve. However, since this method is being integrated into already existing animation packages, we can assume that all objects which the renderer supports are also supported in the real-time workflow. Another drawback stems from the fact that different techniques are being used for the final render and the interactive preview. With such differing display techniques, it can be difficult to make the preview correspond highly to the final rendered image.

3.2.2 Photon Splatting

The second method works with the photon map representation more directly. If a very rough idea of what the global illumination looks like is sufficient, then each photon can simply be rendered to the screen as a point. This would give a result simular to Figure 9b. From this image it is clear where the concentrations of photons reside and is especially good at depicting the location and general shape of caustics. Since *each* photon is non-discriminately drawn to the screen, the structure of the kd-tree is no longer required. This allows us to store the photons in a completely unstructured array, eliminating the balancing costs and, more importantly, the cost of range searching. In addition, since we are no longer performing a radiance estimate over the hemisphere of directions, the incoming direction of the photon is no longer needed, reducing the memory costs of the realtime photon map to 17 bytes.

Using non-physical geometry such as points to render the photons prevents us from being able to accurately specify the intensity of the photons. If more accuracy is needed in the intensity of global illumination effects, then a different approach can be taken. Instead of using points, we can use worldspace geometry to display our photons. This will allow for proper calculation of photon intensity, and should correspond to the final rendered image much more closely.

First we note that Equation 9 using a fixed width search radius r, corresponds exactly to splatting each photon onto the framebuffer as a disc of radius r. If a weighting function is used in the radiance estimate, we can replace each simple disc, with a textured disc. The texture would contain a greyscale raster representation of the weighting kernel. Each photon disc can be drawn to the screen as a textured quad; however, a quad is generally rendered to the screen as two triangles. For increased performance one triangle can just as easily be used to represent the photon splats, making the triangle cost per photon 1 to 1.

Stürzlinger, *et al.* point out that glossy BRDFs can be simulated using this method if the incoming direction is stored in the photon [25]. The intensity of each photon splat can be modulated by the evaluation of a Phong-like glossy BRDF based on the surface normal, the viewing angle, and photon's incoming direction. This allows for caustics and soft indirect illumination on non-lambertian, semi-glossy surfaces.

4 Implementation

Because of the requirement to parameterize all geometry in the scene using a backwards ray tracing approach, and the increased flexability and accuracy of using photon splatting, we decided to implement the photon splatting

method.

4.1 Photon Tracing Phase

In order for the technique to produce interactive feedback, not only does the cost of the render pass in photon mapping need to be improved, but the cost of casting out photons must also be kept to a minimum. For scenes with only a few simple objects, tracing a million rays can be done fairly efficiently. However, for more complex scenes with thousands or millions of triangles this operation becomes prohibitive without any optimization techniques.

4.1.1 Projection Maps

One optimization which is performed for caustic photons is restricting the outgoing directions of the photons from the light towards only specular objects. Since caustics can only be formed if a photon first encounters a specular surface, and subsequently get deposited at a diffuse object, this restriction should improve performance while maintaining the accuracy of the simulation.

One way of implementing this would be to create raster projection maps at each light source [9]. A bounding sphere representation of each specular object could then be rasterized in the projection maps using ray tracing or scan conversion. Any pixel which a bounding sphere projects to would be white, and all other pixels would be black. This provides a conservative estimate to the directions in which the photons must be distributed. The photon shooting algorithm must then be appropriately modified to only shoot photons in directions represented by white pixels in the projection maps [9]. Care must be taken to preserve the proper probability distribution of photons emitted from each light source.

Another method can be implemented which avoids the need for storing a raster in each light [17]. Instead of rasterizing the projection of bounding spheres onto the light, this information can be kept as a few geometric constants from which the cone of directions can be accurately reconstructed. For explanatory purposes assume that the light source is a point or spot light. Generalizations to this assumption will be derived later. In order to conservatively shoot photons at specular objects, photons should be emitted within the cone defined by the light location and the radius of the bounding sphere. Geometric relationships of these structures are shown in Figure 8. If calculation and retrieval of the bounding sphere for objects is efficient, this cone of directions can be calculated directly for each photon emitted. However, if the bounding sphere algorithms are not cached, and therefore the operation takes considerably more time, these values should

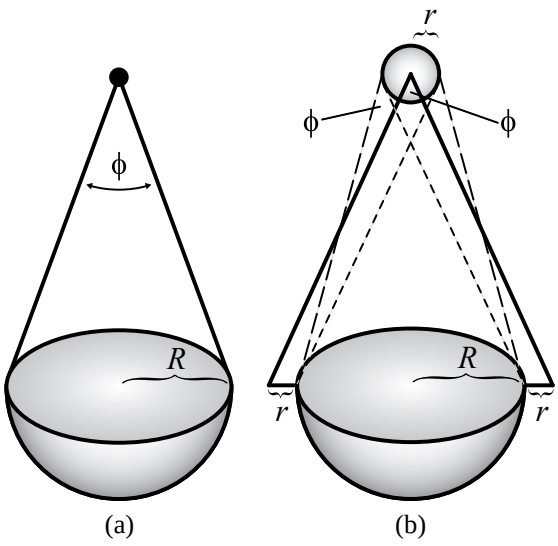


Figure 8: Determining the cone within which to emit photons for (a), point light sources and (b), area light sources. The radius for the bounding sphere of the light must be added to the bounding sphere of the object in order to achieve a conservative estimate for the cone of directions.

be stored at each light for each specular object. In order to reconstruct the cone, a vector towards the center of the bounding sphere should be stored, along with a apex angle. For area light sources, the cone of directions can be similarly calculated; however, different locations on the light source would produce different valid ranges for directions. In order to account for the different cones of directions, a single cone, which encloses any possible valid outgoing direction can be constructed. It is sufficient to simply add the radius of the bounding sphere of the light to the bounding sphere of the object and use this new bounding sphere for the projection [10]. This creates a conservative bound for outgoing directions for photons emanating from any location on the area light source.

The bounding sphere construction of the cone of directions provides an excellent analytic alternative to the rasterized projection maps; however, care must be taken to avoid errors in the simulation if this method is employed. Figure 10 shows a situation which would produce inaccurate results if a naïve implementation of this technique is used. Regions 1 and 3 would recieve the proper amount of photons; however, region 2 falls into the cone of directions of both objects, and therefore would recieve twice as many photons. Photons shot towards either object would contribute to the caustic illumination within that region, thereby producing an inaccurately high probability of photons. We developed a simple test which can be

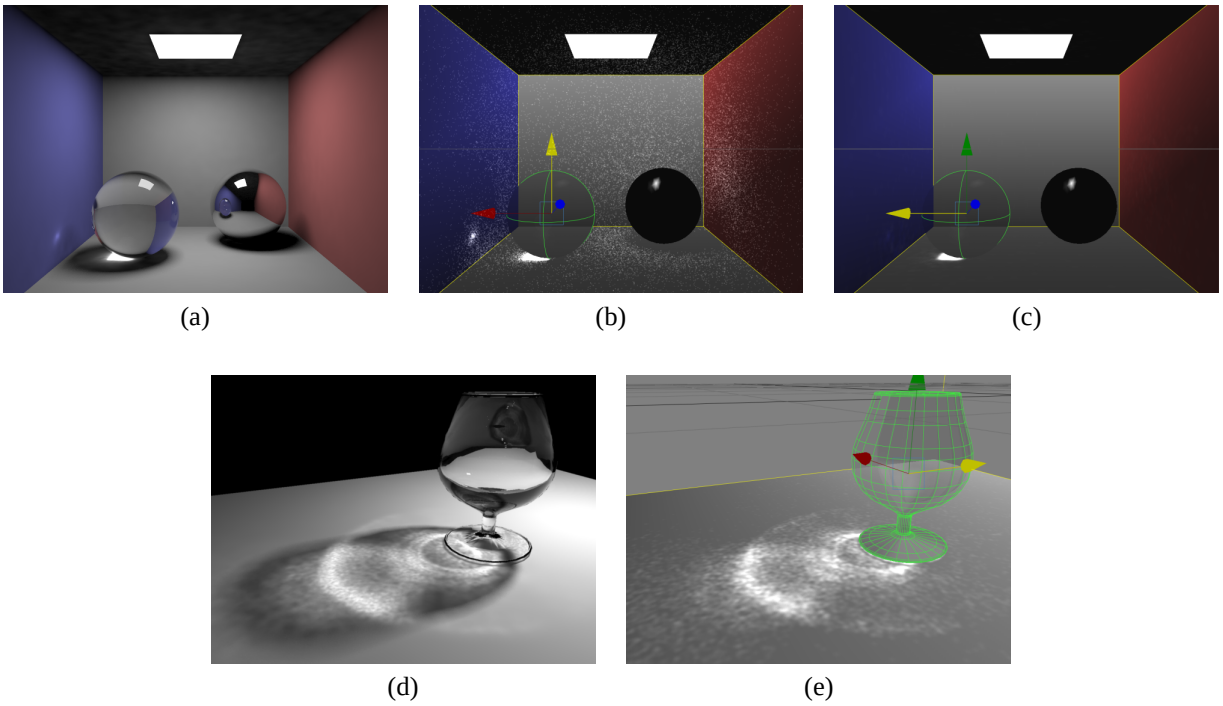


Figure 9: Rendering comparison of caustics using a conventional Monte Carlo photon mapping approach, (a) & (d); our interactive photon mapping method using points, (b); and interactive photon mapping using textured triangles, (c) & (e). The area light sources are approximated for local illumination in OpenGL as single point lights. This, along with the use of Gouraud interpolation for shading explains all the major differences in the local illumination of the images. The global illumination however maintains a high level of correspondence between the interactive previews and the full renderings.

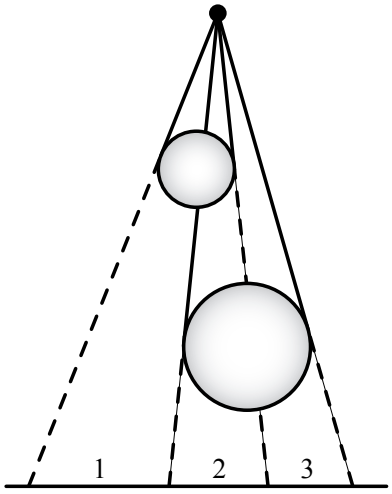


Figure 10: An example scene which would cause inaccurate results in a naïve implementation of the cone of directions. Region 2 would recieve twice as many photons as regions 1 and 3.

administered in order to circumvent this problem. When a caustic photon is sent out from a light towards a specific specular object, Object_c, the first hit along the path of the photon must be Object_c in order for the photon to be valid. If the photon encounters any other object before it hits Object_c, the photon should terminate. This ensures the proper power distribution in all three regions.

4.1.2 Spatial Subdivision

Another important optimization which should be implemented for use in complex scenes is some form of spatial subdivision. Various methods have been proposed for the way in which space is discritized, including BSP trees [26], octrees [20], and uniform grids [22]. The method chosen for our implementation uses octrees and a parametric approach to ray traversal of the tree.

The first step in a spatial subdivision scheme is to divide the scene into discrete cells. For grids and octrees, the bounding box of each object is calculated, and the object is added to any cell which overlaps with this bounding

volume. This creates a conservative estimate for which cells each object occupies. A three-dimensional version of a scan converter could be implemented instead to provide a tighter estimate; however, the overhead in such a technique may outweight its marginal benefits [22]. Once the data structure has been created, ray tracing proceeds by traversing the structure along each ray. Our method implements a parameteric ray traversal algorithm described in [20].

4.1.3 Frame Coherent Photons

Another important issue when shooting photons for interactive use is frame coherence. A distracting artifact present in the standard implementation of photon mapping is caused by the fact that each frame is independent of the other. Photons shot from lights in one frame will have the same overall distribution as the photons in the next frame, but each individual photon is emitted within this distribution randomly. In animations, and also in interactive light and object manipulations, this causes a distracting flickering appearance. In order to alleviate this, a trick can be utilized which will produce inter-frame coherence [14]. When shooting out photons from the lights, the pseudo-random number generator responsible for choosing reflection directions for photons is seeded with the photon's number. This way, the random number sequence for each individual photon will remain constant from frame to frame.

4.1.4 Progressive Refinement

Generating a photon map of a million photons takes a matter of a few seconds on a Pentium II 400 MHz machine; however, waiting a few seconds does not allow for an interactive preview. The user should be able to move lights and objects and interactively see the effects these actions have on global illumination. To allow for interactive use, a simple progressive refinement technique can be used to incrementally increase the quality of the global illumination simulation over time.

This technique requires the implementation of an idle function which continually shoots out photons into the scene and stores them in the photon maps. For systems with multiple processors it may be advantageous to replace the idle function with an idle thread. Our test system only had one CPU, so multi-threading was not implemented. Another function continually updates the OpenGL screen if any change is made. If the camera is moved the photon simulation proceeds as normal, since the effects are view-independent. The photon splatting method supports view-dependent glossy BRDFs; however, the photons are stored with their incoming direction, which allows us to change the viewpoint without having to shoot out the photons from scratch. If an object or light is moved, the photon maps must be cleared, and the simulation starts over. This produces an effect that accumulates accuracy when the scene is static, but also provides a quick, refining view during light and object manipulations.

In our implementation, photons are shot for a quantum of time, τ, after which the results are displayed to the screen. The parameter, τ, controls the responsiveness of the interactive progressive refinement method. Choosing a small value for τ would produce more frequent updates to the screen, and a more fluid responsiveness of the system. However, updating the screen often while shooting photons can increase the time needed to acquire a high quality preview; therefore an appropriate value of τ should be chosen to properly compromize between the responsiveness to user input and the speed of the simulation. In our tests, values of τ between 0.1–0.5 seconds produced acceptable results.

4.2 Photon Splatting Phase

Once a photon mapping framework is implemented, modifying it to use this method is straightforward. After all other geometry is rendered to the screen using OpenGL, a new `render()` function of the photon map class is called which iterates through the photon array and spits out a `glPoint` (for the rough preview) or a `glTriangle` (for the accurate preview) at the location of each photon [37]. Each photon's location is precisely on a surface, which can cause flickering due to numerical imprecision in the Z-buffer. Therefore, a small offset towards the camera should be added to each photon in order to reduce numerical error. In OpenGL, this can be accomplished using the `glOffset` command [37].

Though the location and shape of the photon map is represented very well using points, since we are using the location of each photon directly, the intensity of the global illumination effects are more difficult to display correctly. In OpenGL, a `glPoint` does not have a specifiable worldspace size [37]. Instead, all points are rendered to the screen with the same screenspace dimensions, no matter how far or close to the camera they are. This can make choosing the correct color for each photon very difficult. The photon's intensity should coorespond to its displayed radius in worldspace. If a photon is shown as a point of radius 1 in worldspace, its intensity should be divided by the area of the point, π. Determining the worldspace dimensions of an arbitrary two-dimensional `glPoint` can be complicated and unnecessary.

```
void orthoBasis(vector3d N, vector3d X, vector3d Y)
{
    min_index = index of smallest component of N;

    if (min_index = 0)
        X = vector3d(0,-N[2],N[1]);

    else if (min_index == 1)
        X = vector3d(-N[2],0,N[0]);

    else if (min_index == 2)
        X = vector3d(-N[1],N[0],0);

    Y = cross(X,N);

    normalize(X);
    normalize(Y);
}
```

Figure 11: Numerically stable algorithm which generates an orthogonal basis constained such that one axis is the vector N.

Using textured triangles to represent photons automatically provides proper perspective effects for the splats. In addition, the intensity of each splat can be directly computed to match a radiance estimate, providing a much closer correspondence to the final output. Using a box filter, each photon's power would need to be divided by the area of the kernel disc. When more complex filters are used, the power needs to also be divided by a constant determined from the kernel:

$$color_p = \frac{\Delta\Phi_p}{C \cdot \pi r^2} \tag{13}$$

where C is the average weight in the rasterized kernel.

Each photon splat is drawn at its hitpoint oriented using a ortho-normal basis derived from the photon's normal vector. This requires each photon to store the normal vector at the location of the surface to which it belongs. This information can be encoded in the photon structure in the same way as the incoming direction; or, if glossy effects are not desired, it could be stored instead of the incoming direction. An ortho-normal basis can be created from a single vector by generating a random vector and taking the cross product. Generating numerous random numbers just to create a basis is highly inefficient, and in fact numerically unstable. Another option would be to use Gram–Schmidt orthogonalization [1]. We instead developed our own technique for this process, pseudo-code for which is presented in Figure 11.

Using simple geometry, an equilateral triangle is then constructed in the plane perpendicular to the normal vector (defined by the vectors X and Y). The triangle is scaled to enscribe a circle with radius equal to the radiance estimate's search radius. A precomputed kernel texture is applied to the triangle such that the support area of the kernel fits exactly into the triangle. The kernel texture is used as a transparency map over the triangle, fading out the calculated triangle intensity from Equation 13.

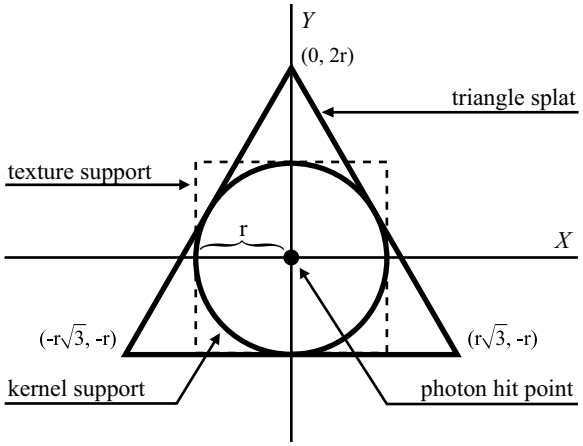


Figure 12: The geometric relationship between the ortho-normal basis, the triangle splat, and the texture.

5 Results and Discussion

The described algorithm was implemented on a 400 MHz Pentium II machine running Windows and Red Hat Linux with an NVidia GeForce 256 graphics card. Our progressive refinement method allowed for an almost instantanious low-quality preview, while a higher quality image

could be realized within a matter of a few seconds. The graphics hardware portion of the algorithm is fill limited due to the enormous amount of textured triangles which are generated per frame. Ray tracing the photons was also a major bottleneck.

5.1 Numerical Imprecision

Throughout development, we encountered problems due to the limited color precision in OpenGL provided by the graphics card. The problem involved the use of thousands or even millions of photons, which combined add significant brightness to the scene, but individually are all very low in intensity. This created noticeable precision errors. In fact, without any modification, once a certain amount of photons was shot, none of the photons would display onscreen. This is due to the fact that each photon's power is divided by the total number of photons shot out from a particular light source. Once this number becomes very large, numerical imprecision rounds the overall power of the photon to 0. As more hardware starts supporting full floating-point precision colors, as is the current trend, this problem should disappear completely. However, in order to eliminate this problem using currently available hardware, two methods were tested.

The first method was inspired by error diffusion in digital imaging [28]. Since each photon's power will be rounded to a discrete intensity value, each one introduces some error to the final output. Some values will increase the overall power, while others will decrease it. In order to maintain a correct overall average intensity, we can add the round-off error from the currently rasterized photon to the subsequent photon. This eliminated the problem of the photon map completely disappearing once a large number of photons have been shot. However, at the point where the photon map would completely disappear, no further detail was added to the rendered image by using error diffusion, only noise. Any additional photons are essentially useless, and just add to the total render time of the frame.

The second method was based on the observation that adding noise may prevent the photon map from disappearing, but it does not add any detail. Instead, a counter keeps track of how much error is introduced during each frame refresh. As the number of photons grow larger, this error counter should decrease, since many photons will start losing energy during round-off. The photon shooting algorithm is stopped if the value of the counter reaches some empirically determined value. When chosen correctly, this prevents the photon map from disappearing due to numerical imprecision.

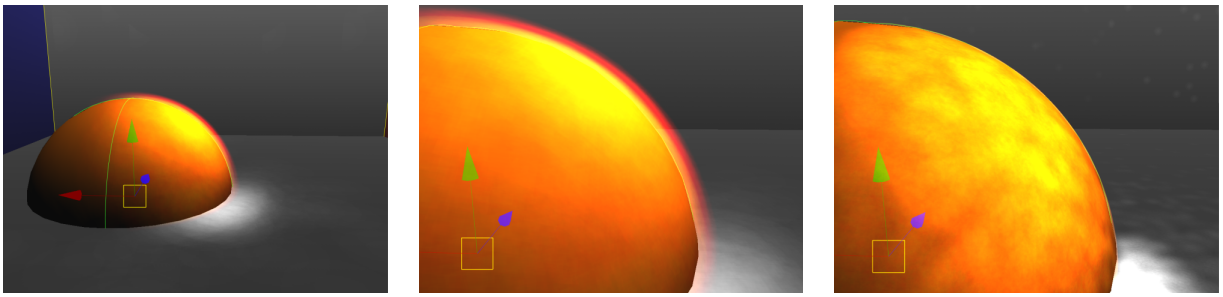
5.2 Photon "Haze"

When using the triangle splatting method, photons are drawn into the scene using textured triangles oriented according to the normal vector of the surface at the hit-point. Approximating the appearance of the range search using a single flat, textured triangle produces acceptable results when photons are distributed on locally flat surfaces. However, when this method is used for scenes with more complex geometry, visible artifacts appear due to this simplified representation. On highly curved surfaces a photon splat will deviate from the contour of the object, producing a visible gap between the photon and the surface. When viewed from the side, this appears as though the photons hover above the surface as a cloudy haze, instead of being directly on the surface.

This problem can be alleviated in several ways. One method would be to try to calculate the local curvature of the surface, and incorporate this information when drawing the splats. Each splat could be represented using a higher order primitive, such as a tessilated ellipsoid, instead of a simple triangle. The curvature of the textured ellipsoid would be chosen to correspond to the local curvature of the surface, allowing the splat to hug the surface more closely. This method, however, suffers from several drawbacks. Calculating the local curvature of a surface for each photon hit-point would be a very costly operation, decreasing the performance of the preview. Also, many more triangles would need to be rendered using this representation, further increasing the run time of the algorithm.

A method suggested by [25] maintains the use of a single triangle per photon and does not introduce any new highly costly computations. Instead of rendering all the photons to the screen after all the objects have already been drawn, photons are assigned to the objects on which they lie. After an object is drawn, all the photons belonging to it are rendered. A mask is created when drawing the object such that the rendering of the photon splats is limited to the pixels which are covered by the object. This ensures that photons only influence the displayed brightness of objects which they hit.

In our tests, however, the effect of this artifact in degrading the quality of the preview was minimal. A small search radius ensures that the artifacts are insignificant unless the camera is zoomed in very close to a particular surface. At this point, a higher density, and smaller search radius, for the photons would be needed in order to preview the global illumination effectively. See Figure 13 for a comparison.



(a) (b) (c)

Figure 13: An example scene (a) with a caustic concentrated partially onto a curved surface and exhibiting "photon haze." (b) Closeup of the artifact. (c) Reduction of the artifact by using a smaller photon splat size (search radius).

5.3 Image Quality and Speed

Our tests have shown that the photon splatting method for previewing the caustic photon map was very effective and efficient. The output of the interactive algorithm for the caustic photon map matched the output of a full Monte Carlo ray tracer with photon mapping very well. Though the interactive global photon map was implemented in a similar manner to the caustic map, the usefulness of the preview was less significant. Firstly, in a full Monte Carlo photon mapping simulation, the global photon map is not directly visualized; instead, a final gather step is used. While the photon splatting did relay useful information in the color bleeding in the scene, the results were much more "splotchy" than in a final render. Also, since generally the search radius for a global photon map is much higher, and the photons are distributed more evenly in the scene, the fill rate for the photon splats became more of an issue. The frame rate performance went down drastically when the global photon map was considered. This leads us to believe that while photon splatting is a very good method to preview caustics, diffuse indirect lighting might be better implemented using a method such as backwards ray tracing with hardware texturing.

6 Conclusion and Further Work

We have presented a hybrid hardware-software based implementation of global illumination which can be visualized interactively on consumer level PCs. The method can easily be embedded in pre-existing animation packages with little overall modification. Such a tool could provide increased productivity when lighting 3D scenes with global illumination by provide realtime feedback for lighting and object manipulations.

Though not presented in this paper, it seems that this method could be extended to simulate volumetric global and local illumination in hardware. The photons in the photon map would be deposited anywhere in space, as opposed to directly on surfaces. The triangle splats could be set to always face the camera and would use a modified kernel.

Another issue that could improve the overall quality of the rendered image is automatically shooting photons in areas of high importance. The importance map [27] can be utilized for this effect in non-interactive rendering; however, the process is currently too time consuming for realtime applications.

Acknowledgments We would like to thank Prof. Michael Garland for continuous guidence throughout the project and for proof-reading. Futhermore, we would like to thank Kelly Thomas for additional proof-reading of the text, and Don Schmidt for providing valuable comments and discussion.

References

[1] George B. Arfken. *Mathematical Methods for Physicists*, chapter 9.3 - Gram-Schmidt Orthogonalization, pages 516–520. Academic Press, Orlando, FL, 3^{rd} edition, 1985.

[2] James Arvo. Backward ray tracing. In *Developments in Ray Tracing, SIGGRAPH '86 Seminar Notes*, volume 12. ACM, August 1986.

[3] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In Thomas Ertl, Wolfgang Heidrich, and Michael Doggett, editors, *Proc. Graphics Hardware 2002*, pages 1–10. Springer, 2002.

[4] Per H. Christensen. Faster photon map global illumination. *Journal of Graphics Tools*, 4(3):1–10, 1999.

[5] James D. Folley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. The Systems Programming Series. Addison-Wesley, 2^{nd} edition, 1997.

[6] Wojciech Jarosz. Fast image convolutions. Notes from Workshop for the Student ACM SIGGRAPH Chapter @ UIUC. http://www.acm.uiuc.edu/siggraph/workshops/, October 21 2001.

[7] Henrik Wann Jensen. Global illumination using photon maps. In X. Pueyo and P. Schrder, editors, *Rendering Techniques '96*, pages 21–30. Springer-Verlag, 1996.

[8] Henrik Wann Jensen. Rendering caustics on non-lambertian surfaces. In *Proceedings of Graphics Interface '96*, pages 116–121. Springer-Verlag, 1996.

[9] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters LTD., 2001.

[10] Henrik Wann Jensen, October 2002. Personal communication.

[11] Henrik Wann Jensen and Niels Jrgen Christensen. Efficiently rendering shadows using the photon map. In *Proceedings of Compugraphics '95*, pages 285–291. Alvor, 1995.

[12] Henrik Wann Jensen and Niels Jrgen Christensen. Photon maps in bidirectional monte carlo ray tracing of complex objects. In *Computers & Graphics*, volume 19:2, pages 215–224, March 1995.

[13] Henrik Wann Jensen and Per H. Christensen. Efficient simulation of light transport in scenes with participating media using photon maps. In *Proceedings of SIGGRAPH '98*, Computer Graphics Proceedings, Annual Conference Series, pages 311–320. ACM, ACM Press / ACM SIGGRAPH, July 1998.

[14] Henrik Wann Jensen, Frank Suykens, and Per H. Christensen. A practical guide to global illumination using photon mapping. ACM, August 2001. SIGGRAPH 2001 Course Note 38.

[15] Nicolai M. Josuttis. *The C++ Standard Libary: A Tutorial and Reference*. Addison-Wesley, New York, 1999.

[16] J. T. Kajiya. The rendering equation. In David C. Evans and Rusell J. Athay, editors, *Proceedings of SIGGRAPH '86*, volume 20 of *Computer Graphics*, pages 143–150, August 1986.

[17] Nathan Kopp. Simulating reflective and refractive caustics in pov-ray using a photon map. Direct Study with Howard Whitston, May 1999.

[18] Eric P. Lafortune and Yves D. Willems. Bidirectional path tracing. In *Proc. 3^{rd} International Conference on Computational Graphics and Visualization Techniques (Compugraphics)*, pages 145–153, August 1993.

[19] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *Proceedings of SIGGRAPH 2002*, volume 21 of *acm Transactions on Graphics*, pages 703–712. ACM, ACM Press / ACM SIGGRAPH, July 2002.

[20] J. Revelles, C. Urena, and M. Lastra. An efficient parametric algorithm for octree traversal. In *Proceedings of the 8^{th} International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media*, volume 8(2), pages 212–219. WSCG, 2002.

[21] Peter Shirley. Nonuniform random point sets via warping. In David Kirk, editor, *Graphics Gems III*, pages 80–83. Academic Press, San Diego, 1992.

[22] Peter Shirley. *Realistic Ray Tracing*. A. K. Peters LTD., 2000.

[23] Peter Shirley, Changyaw Wang, and Kurt Zimmerman. Monte carlo methods for direct lighting calculations. *ACM Transactions on Graphics*, January 1996.

[24] Marc Stamminger, Annette Scheel, Xavier Granier, Frederic Perez-Cazorla, George Drettakis, and François Sillion. Efficient glossy global illumination with interactive viewing. In *Graphics Interface (GI'99) Proceedings*, pages 50–57, June 1999.

[25] Wolfgang Stürzlinger and Rui Bastos. Interactive rendering of globally illuminated glossy scenes. In Dorsey and Slusallek, editors, *Proc. 8^{th} Eurographics Workshop on Rendering*, Rendering Techniques '97, pages 93–102. Springer-Verlag, June 1997.

[26] Kelvin Sung and Peter Shirley. Ray tracing with the bsp tree. In David Kirk, editor, *Graphics Gems III*, pages 271–274. Academic Press, San Diego, 1992.

[27] Frank Suykens and Yves D. Willems. Density control for photon maps. In *Proceedings of the 11th Eurographics Workshop on Rendering*. Springer-Verlag, June 26–28 2000.

[28] Spencer W. Thomas and Rod G. Bogart. Color dithering. In James Arvo, editor, *Graphics Gems II*, pages 72–77. Academic Press, San Diego, 1991.

[29] Parag Tole, Fabio Pellacini, Bruce Walter, and Donald P. Greenberg. Interactive global illumination in dynamic scenes. In *Proceedings of SIGGRAPH 2002*, volume 21 of *acm Transactions on Graphics*, pages 537–546. ACM, ACM Press / ACM SIGGRAPH, July 2002.

[30] Ingo Wald, Thomas Kollig, Carsten Benthin, ALexander Keller, and Philipp Slusallek. Interactive global illumination using fast ray tracing. In *Proc. 13th Eurographics Workshop on Rendering*, Rendering Techniques 2002. Springer, 2002.

[31] Changyaw Wang. Direct lighting models for ray tracing with cylindrical lamps. In David Kirk, editor, *Graphics Gems III*, pages 307–313. Academic Press, San Diego, 1992.

[32] Changyaw Wang. *The Direct Lighting calculation in Global Illumination Methods*. PhD thesis, Indiana University, 1993.

[33] Greg Ward. Real pixels. In James Arvo, editor, *Graphics Gems II*, pages 80–83. Academic Press, San Diego, 1991.

[34] Gregory J. Ward. Adaptive shadow testing for ray tracing. In *Proceedings of the Second Annual Eurographics Workshop on Rendering*. Springer-Verlag, 1991.

[35] Gregory J. Ward and Paul Heckbert. Irradiance gradients. In *Proceedings of the 3rd Annual Eurographics Workshop on Rendering*, pages 85–98. Springer-Verlag, May 1992.

[36] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. A ray tracing solution for diffuse interreflection. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, pages 85–92. ACM Press, 1988.

[37] Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner, and OpenGL Architecture Review Board. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley, 3rd edition, August 1999.

[38] Kurt Zimmerman. Direct lighting models for ray tracing with cylindrical lamps. In Alan Paeth, editor, *Graphics Gems V*, pages 285–289. Academic Press, San Diego, 1995.