Simulating Reflective and Refractive Caustics in POV-Ray Using a Photon Map

Nathan Kopp - May, 1999

Directed Study with Howard Whitston
Lawrence Technological University

Abstract

One drawback of many computer graphics rendering engines, including most ray-tracers, is the inability to correctly render reflective and refractive caustics. While many techniques have been developed to overcome this limitation, one technique stands out. The photon map, developed by Henrik Wann Jensen, provides an indirect lighting model for reflective and refractive caustics that is stored separately from the scene geometry. Here, the adaptation of the photon map for use in the popular freeware ray-tracer, POV-Ray, is explained. An adaptive range search technique is described which increases both the speed of the kd-tree searches and reduces blurring at boundaries between areas of high and low photon density. This implementation of the photon map allows the rendering of true caustics at a relatively low cost to memory and speed in a popular freeware ray tracing package.

Problem History

Ray tracing is a commonly used technique to create photo-realistic images from 3D computer models. A ray-tracer traces light rays from a virtual camera into the scene, and locates all intersections with the scene geometry. The surface normal, texture information, and location of an intersected object are used to compute the color of the pixel that corresponds to the intersecting ray. The ray tracing algorithm is inherently recursive, allowing rays to be spawned from the intersection point to create realistic reflections and refractions.

One limitation of the ray tracing algorithm is the lack of an indirect lighting model. The amount of light illuminating a point in the scene is determined by tracing *shadow rays* from that point to all of the light source points in the scene. If a shadow ray does not intersect any objects in the scene, the full intensity and color of the light source are used to illuminate the point in question. If an object is found which blocks the shadow ray, the brightness of the light is filtered by the object's material properties. Unfortunately, this model does not simulate indirect lighting caused by reflection or refraction (or any indirect lighting, for that matter). Although images of objects seen through a prism by the virtual camera will be distorted by refraction, light rays passing through a prism to illuminate other objects will be filtered, but will not change direction.

Refracted and reflected patterns of indirect light are generally referred to as caustics. Caustic patterns that are easily visible in the real world include the patterns of convergent and divergent

light on the bottom of a swimming pool, light reflected from a mirrored surface, and light refracted through a drinking glass onto a table. These patterns of light can be quite stunning and thus it is desirable that a rendering program be able to simulate them.

For this directed study, I chose to implement an algorithm for rendering specular indirect lighting (caustics) by adding this feature to POV-Ray, a freely available ray-tracer. Not only is POV-Ray freeware, but its source is also freely available, and the license allows anyone to modify the source code. POV-Ray is also supported by a community of users who can both help with development and also appreciate the finished product. Also, I have done previous work with POV-Ray, such as adding UV surface mapping for textures and implementing a spherical camera. By making these previous modifications, I learned a lot about the source code, which is (for the most part) clean but generally not well commented. I also have had contact with the program's developers, known as the POV-Ray team, while making these modifications. All of these aspects worked together to make POV-Ray a good choice for implementing the rendering of caustics.

There have been quite a few ideas presented for rendering caustic effects in a ray-tracing renderer. Much of my initial research was done by reading sections of *Advanced Animation and Rendering Techniques* by Alan Watt and Mark Watt.[1] The authors of this book explain Arvo's method for rendering caustics and also present their own method. Arvo's method involves tracing rays from the light sources through the scene. These rays are directed only at specular surfaces (surfaces which produce specular reflections or refractions). Arvo suggests storing the data from ray intersections (ignoring the top-level intersection) in 'illumination maps' attached to each object. An illumination map is a grid of data points, such as a bit-mapped image. Alan and Mark Watt suggest a method of 'light beam tracing.' This method, which has it's roots in Heckbert's "beam tracing" (described in the book by Alan and Mark Watt), traces beams instead of rays through the scene. A beam is a bundle of rays, instead of a single ray. In backwards beam tracing, each beam consists of three rays, making a triangular shaped beam. When this beam intersects the scene beyond the top-level intersection, special polygons, called 'caustic polygons,' are stored in the scene geometry. These caustic polygons are used only for shading purposes and do not change the shape of objects. Recently, Steven Collins has suggested a method which he calls splatting.[2] Splatting is an extension of Arvo's illumination maps, and basically attempts to deal with aliasing problems that affect Arvo's method. Splatting allows illumination maps to be dynamically resized based on an extensive wavefront-tracking algorithm which estimates the density of incoming rays to determine the desired level of detail in the illumination map. A recent addition to this suite of algorithms is photon mapping.[3] The photon map was introduced by Henrik Wann Jensen, who currently works at MIT. Like Arvo's method, photon mapping traces rays. However, it stores the data from the light-ray tracing step in a data structure independent from the scene geometry, which leads to some great advantages over other methods.

Unfortunately, none of these methods are perfect. None of them completely solve the problem of efficient rendering of realistic indirect illumination. Each has its positive and negative aspects. For example, while Watt's caustic triangle method allows quick rendering in scan-line and ray tracing renderers, the method requires that the entire scene be represented as triangles.

POV-Ray and other ray-tracers often use non-triangular objects, such as perfect spheres, for both accuracy and efficiency. As mentioned previously, Arvo's method suffers from aliasing problems. While these problems are somewhat alleviate by Collins' splatting approach, the use of the illumination map restricts the rendering of caustics to objects that support surface mapping. Some objects, such as metaballs, certain isosurfaces, and fractals to not support parametric surface mapping.

Jensen's photon mapping technique looked the most promising for POV-Ray. It stores data in a tree data structure which is independent of scene geometry, which aids both in ease of implementation and in the types of supported objects. Jensen's method can also be extended for use with atmospheric effects, which none of the other caustic-rendering techniques can do. The photon map can also be used to decrease the number of shadow rays used by the ray-tracer, improve the speed of Monte-Carlo rendering, and improve the speed of POV-Ray's indirect lighting (radiosity) model. Therefore, the photon map was chosen as the method of solution.

Before I explain my work, I want to point out the extent of Jensen's work. Henrik Wann Jensen has spent many years working on the photon map technique. He published his first paper on the topic in 1995.* He has spent the years since then working on various aspects of the technique for his PhD studies and also for post-PhD work at MIT. In fact, he only implemented media interaction in 1998.[4] Therefore, I only concentrated on rendering caustics for this directed study. I did do some work on diffuse indirect lighting, which was originally a feature I wanted to implement. However, I ran into some difficulties in my implementation, which will be detailed later. After doing more research on that topic, I learned just how difficult it would be to implement that aspect of the photon map, so I decided to concentrate on speed and memory optimizations of the rendering of caustics. Also, POV-Ray's current atmosphere and radiosity features need improvement before photon mapping can be added to them.

Therefore, my goal for this project was to implement rendering of reflective and refractive caustics on surfaces in POV-Ray using a photon map. Other aspects of the photon map could later be added, but I determined that they were beyond the scope of a 15 week project.

Implementation

Initial Kd-tree Work

Shortly after I began work on the project, I was contacted by Mike Gibson, who learned of my project through the POV-Ray programming newsgroup. Mr. Gibson had recently created a ray-tracer called MSGTracer.[5] In his ray-tracer, he had implemented the photon map, and although it was not completely bug-free, the source code was available. By this time, I had already created my own kd-tree code, so I only used his as a reference. His kd-tree was not much like the code I had at the time. He used a median-split kd-tree balancing technique, and I was using

---

* I'm not sure which is Jensen's first paper on the topic, but the first ones that I found were dated 1995, such as the following: Jensen, Henrik Wann; Niels Jorgen Christensen. *Efficiently Rendering Shadows Using the Photon Map.* In *Proceedings of Compugraphics '95.* pp. 285-291. 1995.

a mean-split technique. The differences between these will be explained later. As I mentioned, his implementation had some problems, such as overly-bright caustics, but it provided a good reference.

I was also given a reference to another kd-tree implementation. Ron Parker, another POV-Ray programmer, suggested that I look at a program called Ranger.[6] Mr. Parker sent me the source code to the program. The kd-tree implementation in Ranger was much more like the implementation that I was using at the time, since it used child pointers to create the tree structure.

Prior to viewing these two implementations of the kd-tree, I worked out the code for it myself. When I first tackled the problem, I found an amazing lack of information on kd-trees on the Internet. It was not until much later that I was shown a textbook that mentioned them. I was also unable to find the resource cited by Jensen which supposedly described the kd-tree in detail. (I still want to find that reference to try to further optimize my code.) Fortunately, I was able to find a Java kd-tree visualization program.[7] This program allowed me to visualize the building of a KD-tree and thus create the code myself.

Before starting work on the kd-tree, I considered using other data structures to store the data. For example, POV-Ray already contained code for a octree data structure. Therefore, I did a small amount of research regarding octrees, 3-dimensional r-trees, and view-point trees. Unfortunately, these trees provided no apparent advantages over a balanced kd-tree. None of these data structures were as compact as a median-balanced kd-tree, and most based on rather complicated algorithms. Finally, I made the assumption that Jensen had probably already done a significant amount of research before choosing the kd-tree to represent the photon map, so I decided to follow his lead.

Backward Ray Tracing

The first step in implementing the photon map was to incorporate a backwards ray-tracing step into the existing POV-Ray renderer. The first problem I faced was the question of how to shoot photons from the light sources. My first approach was to use pure Monte-Carlo sampling. This seemed to be what Jensen suggested in his paper. Unfortunately, the initial results were extremely splotchy and the images looked horrible. At first I thought that the splotchiness was caused by bugs in the code, so I started looking for bugs. For about a week I squashed a large number of bugs, but the splotchiness remained. Once I had decided that the existing code was relatively bug-free, I realized that the Monte-Carlo sampling technique might be the cause of the splotchiness. I switched to a uniform radial sampling and immediately saw good results. The results were so good that I was able to post some images on the POV-Ray images newsgroup. The use of uniform sampling was confirmed by an email from Mike Gibson who expressed that he had run into similar problems in his implementation. At this point, the code that aimed the photons at an object left a lot to be desired, and the direction that the photons were aimed was hard-coded into the program. But at least the caustics looked good, and this was very encouraging.

A Closer Look

Jensen's Photon Visualization Equations

Before I go too much further, I should give a detailed explanation of Jensen's photon visualization equations. These equations, detailed in Jensen's papers and only slightly modified here, describe how to use the photons stored in the kd-tree to visualize caustic patterns of light. The basic process is to search the kd-tree to find photons close to the intersection point of a standard eye-centric trace of a ray. These photons are used to estimate the incoming indirect light at that point.

The flux for each photon is determined during the light-ray tracing stage by the looking at the spacial distance between photons at a distance of one unit from the light source (ω) and the intensity of the light source (I_l), shown in equation (1):

$$\Phi = I_l \omega^2 \qquad (1)$$

Jensen's explanation of the photon visualization equation is rather complicated, but I have tried to simplify it somewhat here.

The amount of light leaving a surface from a point (x) in a direction (v) is a function of the incoming light and the surface's BRDF (bi-directional reflectance distribution function). The BRDF of a surface is a probability distribution function that describes the probability that an incoming ray will be reflected in a particular direction. Equation (2) shows the general surface rendering equation.

$$L_s(x,v) = L_e(x,v) + L_r(x,v) = L_e(x,v) + \int_{\Omega} f_r(x,v,w) L_i(x,w) dw \quad (2)$$

In equation (2), $L_e(x,v)$ is the light emitted from the surface from point x in direction v. As Jensen explains, L_e is determined by the surface's texture definition. $L_r(x,w)$, which is the light reflected (by specular and diffuse reflection), is found by integrating $L_i(x,w)$, the incoming light coming from direction w, multiplied by $f_r(x,v,w)$, the surface's BRDF, over Ω, the sphere of all w directions about x. Note that the BRDF $f_r(x,v,w)$ gives the probability that light hitting point x from direction w will be reflected in the direction of v.

Jensen explains that L_i can be split into three separate parts: $L_{i,l}$ from light sources, $L_{i,c}$ from specular reflections and refractions (caustics), and $L_{i,d}$ from diffuse reflections. Many ray-tracers, including POV-Ray, only compute light directly from light sources. $L_{i,c}$ (caustic light) and $L_{i,d}$ (diffuse light) can be computed using monte-carlo techniques, but only if the user does not mind extended render times. Rendering specular reflections and refractions to an acceptable level of detail is especially computationally expensive. POV-Ray versions 3 and above use a

cached monte-carlo sampling algorithm to effectively compute light from diffuse reflections. Therefore, this paper will focus on the second term, $L_{i,c}$.

Jensen describes how to estimate L_r using a photon map. The n photons closest to x within a specified maximum radius are retrieved and used to estimate light, as shown in equation (3).

$$L_r(x,v) = \int_{\Omega} f_r(x,v,w)\frac{d^2\Phi(x,w)}{dAdw}dw \approx \sum_{i=1}^{n} f_r(x,v,w)\frac{\Delta\Phi_p(i,x,w)}{\pi r^2} \quad (3)$$

Here, Φ is the light (or flux) from caustic illumination entering point x from direction w. In the approximation, this flux is estimated by Φ_p, the flux carried by each of the n closest photons gathered. Also, dA is approximated by πr^2, where r is the radius of the sphere required to enclose the n photons closest to x.

Light-Ray Sampling

The techniques described in Jensen's papers seem to suggest a purely random sampling method. Unfortunately, images rendered using this technique showed splotchiness, as predicted in Foley and Van Dam's book.[8] Instead, a jittered sampling technique was employed. Rays are distributed with a uniform angular density using a spiral pattern, which is jittered to remove aliasing.

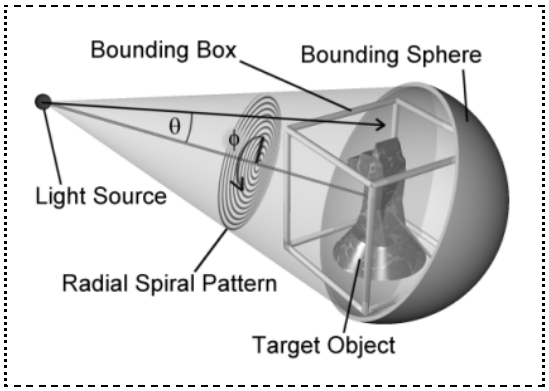


Figure 1 Visualization of the spiral pattern used for shooting photons.

To shoot rays at a particular object, the center of the object's bounding box is first found. A bounding sphere is computed which completely encloses the object's bounding box, and rays are shot at the object in a spiral pattern until the entire bounding sphere has been covered with photons (see figure 1). The angular density of the rays is chosen such that the spacial density of photons at the center of the object's bounding box will be equal to a user-specified value. Equation (4) shows the computation of the step sizes $\Delta\theta$ and $\Delta\phi$, as well as the computation of the actual photon density (ω).

$$\omega = \frac{\Delta x}{d}, \quad \Delta\theta = \Delta\phi = \tan^{-1}(\omega) \qquad (4)$$

The photon density, ω, is an estimate of the spacing between photons at a distance of one unit from the light source. This is used to compute the spacing of photons at any distance from the object in the absence of reflection and refraction. Computation for this predicted spacing is found by rearranging the first part of equation (4) above:

$$\Delta x = \omega d \qquad (5)$$

Here, d represents the distance from the light source to the intersection. This is not a Euclidian distance, but rather the actual distance that the photon has traveled (i.e. the sum of all segments of the photon's path from the light source to the intersection point).

The two values $\Delta\theta$ and $\Delta\phi$ in equation (4) are used as step sizes in two nested 'for' loops (shown in figure 2) to fire rays in a spiral pattern towards the object.

```
for(theta=mintheta; theta<maxtheta; theta+=dtheta)
{
  if (theta<EPSILON)
    dphi2=maxphi-minphi;
  else
    dphi2=dphi/sin(theta);
  minphi2 = minphi + dphi2*FRAND()*0.5;
  maxphi2 = maxphi - dphi2/2 + (minphi2-minphi);
  for(phi=minphi2; phi<maxphi2; phi+=dphi2)
  {
     /* shoot ray toward object using phi & theta */
     /* code omitted */
  }
} /* rays loop */
```

Figure 2 Ray sampling loop for one object

Other implementations of the photon mapping concept, such as Mike Gibson's MSGTracer, have distributed photons with uniform spacial density. However, as light expands from a single point, it expands in the form of a sphere, with energy distributed uniformly on the surface of the sphere. This corresponds to uniform angular density.

Brightness

Light Attenuation

Even before testing the brightness of the photon rendering, I knew that I would have to work on the light attenuation model. As rays get farther from light source, they spread out. The energy decreases proportional to d^2, where d is the distance from the light source to the point in where the energy is being measured. In reality, this occurs because the wave-front of the light is spread out over a larger spherical surface area. Because of this, the decrease in visualized energy due to the spreading out of photons in the photon-mapping algorithm is realistic. Unfortunately, POV-Ray already has a light attenuation model which is not exactly realistic. By default, light energy does not decrease as the distance from the light source increases. The user can specify a somewhat realistic light attenuation by setting the "fade_power" option to the value "2".

Equation (6a) is the physically accurate light attenuation equation, while equation (6b) represents POV-Ray's estimation:

$$I \propto \frac{I_l}{d^2} \quad (6a) \qquad I \propto \frac{I_l}{1 + \left(\frac{d}{fade\_distance} \right)^{fade\_power}} \quad (6b)$$

Under the assumption that a consistent lighting model is more important than a physically accurate lighting model, I decided to compensate for the automatic, physically accurate light attenuation and use POV's light-fading model. Thus, the new equation for a single photon's flux, modified from equation (1), is:

$$\Phi = I_l \, \omega^2 \, d^2 \, \lambda(d) \qquad (7)$$

where $\lambda(d)$ is the POV light-attenuation approximation function.

The Fudge Factors

To determine if my code was producing caustics of the correct brightness, as predicted by the equations above, I rendered a scene with a clear object that had an IOR of 1.0. This would allow photons to continue a straight line which should lead to the spacing predicted by equation (5). If the photon flux and gathering equations were correct, the light rendered using photons should be of the same intensity as light rendered when not using photons. Unfortunately, this was not the case. The scene rendered quite a bit darker than it should have. After searching the code for bugs and finding none, I introduced a 'fudge factor' into the code to correct the problem. Unfortunately, later when I was rendering another scene, I noticed that things looked too bright, so I introduced another fudge factor to correct that problem.

Not liking fudge factors in my code, I set out to find the true cause of this problem. I re-worked equations (1) and (4), which were the locations of the fudge factors, and kept coming up with the same equations (which did not include any kind of constant multipliers). After re-working the equations, I was convinced that the fudge factors were not supposed to be there, so I removed them. Again, the scenes were rendering very dark.

The Culprit: POV's BRDF

Finally, I realized what the problem was. The spacial density of photons on a plane is affected by the angle that the photons hit the surface (see figure 3). As θ, the incident angle, increases, Δx also increases. I then created a 'brightness' test scene that contained concentric rings at varying distances from the light source and camera that could be rotated with respect to the light source, changing the incoming angle of the photons. Using this test scene, I verified this idea. The automatic way that photon spacing increases as the incident

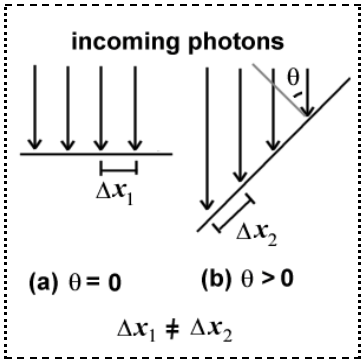


Figure 3 Incoming angle affects photon spacing.

angle increases is a true-to-life phenomenon. Unfortunately, this phenomenon is already estimated by the diffuse term in POV-Ray's BRDF. Therefore, the true-to-life decrease in brightness caused by the spacing-out of photons must be compensated so that POV's estimation can be used. Again I assumed that, for backwards-compatibility reasons, a consistent lighting model was more important than a physically accurate lighting model.

Equation (8) shows how equation (7) is extended to incorporate this compensation (θ is the angle between the incoming photon and the surface normal).

$$\Phi = I_l\left(\frac{\omega^2 d^2 \lambda(d)}{\cos\theta}\right) \qquad (8)$$

Radiance: A Disappointment

As I mentioned earlier, part of my original goal was to implement a new 'radiosity' technique to render indirect diffuse lighting. I named this technique radiance, and at the time I did not fully understand the limitations of the photon map visualization algorithm shown in equation (3). The concept was as follows. First, photons are shot randomly into the scene. At each intersection with an object, the ray is bounced randomly based on the BRDF, very similar to Monte-Carlo ray-tracing. Ray intersections below the top-level intersection are stored in a global photon map and are visualized using the same visualization technique used for visualizing caustic photons.

Unfortunately, this concept was flawed. Images rendered using this technique not only looked very splotchy, but took a very long time to render. For example, a simple Cornell-box scene took between 7 and 8 minutes to render at a resolution of 320x200 on my P266MMX with 32MB of RAM running Windows 95. POV's existing radiosity technique could render the same scene in only 55 seconds and produce a much higher quality image. I decided that the splotchiness is a negative side effect of the visualization equation's ability to render detail, which is, unfortunately, undesirable in this situation.

At this point I researched another paper by Jensen on this topic, *Global Illumination using Photon Maps*.[9] In this paper he explains that to use the photon map for diffuse illumination, it is not visualized directly. Instead, the data from the photon map is used to direct the importance-sampling step of a cached Monte-Carlo ray-tracing technique. This cached Monte-Carlo technique is already used by POV-Ray to render diffuse indirect lighting, so this adaptation of the photon map could be applied to POV-Ray. Unfortunately, to implement this I realized that I would need to learn exactly how POV's radiosity feature works. Unfortunately, POV's radiosity code is quite sloppy and not well commented, so I realized that this would take longer than the remainder of the semester to complete.

Speed Optimizations

With diffuse lighting out of the picture (literally), I decided to focus on speed and memory optimizations. While Jensen asserts that rendering using the photon map is fast, some implementations, such as the 1996 independent study by Shu Chiun Cheah[10] have reported relatively high render times. One area of the algorithm that is executed many times during the rendering process is the kd-tree search. Speeding up the kd-tree search could lead to a significant improvement in render time. Searching a kd-tree is usually done using a range search. A maximum range is specified before starting the search, so that values outside of this range can be easily ignored. Failure to specify a range, or the specification of too large of a range, can cause the algorithm to have to wade through many extra nodes before it narrows in on the ones it is looking for. Nodes that are found to be within the range are added to a priority queue. When the queue fills up, the maximum node is removed and the range is decreased based on the new maximum in the queue. Thus, no specifying a range almost guarantees that the priority queue will fill up.

Before I get into too much detail about speedups relating to the kd-tree search, I should mention one other optimization. For this implementation, photons are shot only at specular objects (i.e. objects that will produce either reflective or refractive caustics). This prevents extra work caused by shooting photons into nothingness or into areas of the scene that will not produce caustics. The user is also given the ability to specify which objects receive photons and which do not. With that said, I will now explain two techniques that I used in an attempt to speed up the kd-tree search.

Multiple Kd-Trees

If all photons in the tree have similar spacing between them (i.e. similar density) then we are able to intelligently choose a good range which will contain almost exactly the number of photons we want. If too small of a search radius is chosen, the render will look splotchy because too few photons will be found. However, render time increases significantly when the search radius is increased. This is especially true of scenes which have some areas where the photons are tightly packed and some areas where they are very spread out. In such scenes, if a small search radius is chosen, the scene renders quickly, but shows aliasing and splotchiness in areas of low photon density. On the other hand, if a large search radius is chosen, the areas of low density are smoothed out, but areas of high density are blurred and lose their high-contrast features. Also, the kd-tree searching routine is bogged down in high-density areas since it has to wade through a large number of photons in order to find the *n* closest.

My first attempt to alleviate this problem was to use multiple kd-trees. Each kd tree had a minimum and maximum photon spacing. Photons would be placed into trees based on an estimation of photon density. To estimate photon density, I used a wavefront tracking approach similar to the one described by Collins.[11] Instead of using a paraxial ray (i.e. a ray very close to the regular ray used to measure divergence or convergence), I simply compared the current ray with the previous and next rays. Of course, rays had to be cached to do this. I was able to use the previous and next rays because the rays are shot in a spiral pattern, with consecutive rays

spaced at a predictable angular distance. A ray-intersection tree was used to make sure that proper intersections were compared to each-other.

While the wavefront tracking technique worked, the multiple-tree concept did not. Dark boundaries appeared in the image at borders between the various kd-trees. These boundaries were unacceptably noticeable and could not be removed. Also, some scenes actually rendered more slowly using multiple maps. At this time I do not have any exact results, since I did not seriously pursue this method after I decided that it would ultimately fail. Overall, this technique produced slower render speeds and poor image quality, so the concept was discarded.

Adaptive Range Search

Fortunately, my next approach to speeding up the kd-tree search was a success. This approach was an adaptive tree range search. The adaptive search is similar to techniques used by Cheah[12], but is intended for use as a speed optimization. The user is allowed to specify n_{min}, the minimum number of photons to gather, r, the initial radius of the range search, and r_{step}, the step size for expanding the radius. If less than n_{min} photons are found in the original search, the radius of the searching range is expanded by r_{step} and the kd-tree is searched again. This process is stopped when at least n_{min} photons are located or after the kd-tree has been searched a set number of times (k_{max}). The values of r_{step}, n_{min} and k_{max} are provided by the user. Good values have been found (through many tests in five different scenes) to be:

$$r_{step} = r, \quad n_{min} = 20, \quad k_{max} = 2.$$

The adaptive tree search size both increased the render speed, and increased the contrast of the photon map in areas of high-density photons.

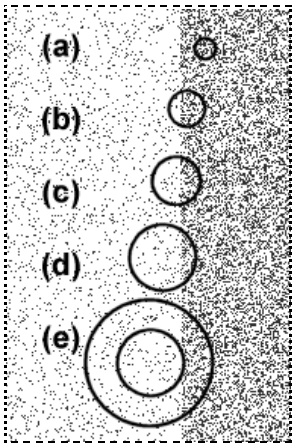


Figure 4 Adaptive search problem.

Artifacts and the Adaptive Search

One problem needed to be eliminated before the adaptive search could work properly. Figure 4 shows an area with two distinct photon densities: very dense (right side), where more than n photons exist in an area with radius of r, and very sparse (left side), where less than n_{min} photons exist in the same area. When photons are gathered with search radius r from the high density area (a), the n required photons are located in an area with a radius less than r. As the search for photons moves out of the high-density area (b, c, d), the size of the radius required to gather n photons increases, and thus the estimated intensity of the light gathered also decreases. However, once the initial search radius does not contain any photons from the high-density area (e), the required number of photons (n_{min}) is not met, so the search radius is expanded. Unfortunately, this search radius now includes a large portion of the high-density area, leading to a greater intensity of light gathered than a point slightly closer which did not require a search-radius-

expansion. See figure 5b, a close-up of the caustic from the cylinder in the Sphere and Cylinder scene, for an example.

Elimination of this problem requires detection of a boundary between areas of high and low photon density. The heuristic used here is based on two assumptions. First, it is assumed that increasing the search radius will not create a very large increase in the density of photons. If a large change does occur, it is assumed that the search is being performed near a boundary and such a change will result in density-boundary artifacts.

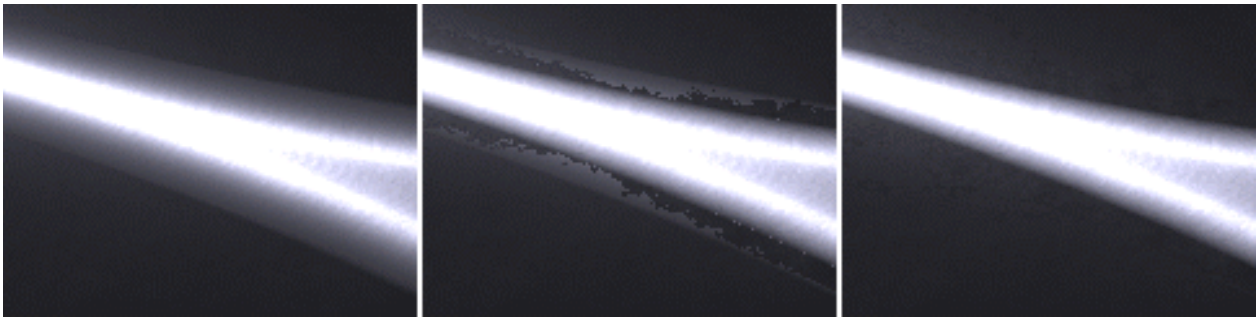
Addressing this first assumption requires introduction of E_{max}, which is the minimum percentage by which the expanded search is allowed to increase the photon density. If the original photon density is found to be d_{photon}, the results of the larger search radius is only used if the new density is not greater than $(d_{photon})(1+E_{max})$.

The second assumption is that a reasonable estimation for lighting cannot be made with too few photons. Previously, n_{min} was introduced as the threshold photon count for expansion of the search radius. Another threshold, n_{exp}, is used to determine if the larger search radius should be kept. If at least n_{min} photons are required in the small search radius, it should be reasonable to assume that more photons should be allowed in the larger search radius without regarding the initial density. If too few photons are located, a good estimate of the incoming flux cannot be made. Also, this threshold should be greater than n_{min}, since a larger search radius is being used. Search results from the expanded search will be used if the number of photons gathered in the expanded search is less than n_{exp} regardless of d_{photon} and E_{max}.

The values of n_{exp} and E_{max} are specified by the user. Reasonable values are:

$$E_{max} = 20\%, \quad n_{exp} = 2\, n_{min}.$$

These values were found by testing five different scenes and trying various combinations until I had something that looked good and rendered quickly. These values also conceptually made sense to me. I should point out that these values do not work well for every scene, however, which is why the user is able to specify different values.



(a) (b) (c)

Figure 5 These three images show a close up of the Sphere and Cylinder scene, showing the (a) non-adaptive search, (b) adaptive search, and (c) adaptive search with artifacts removed. Notice the sharper edges in (c) compared to the non-adaptive search.

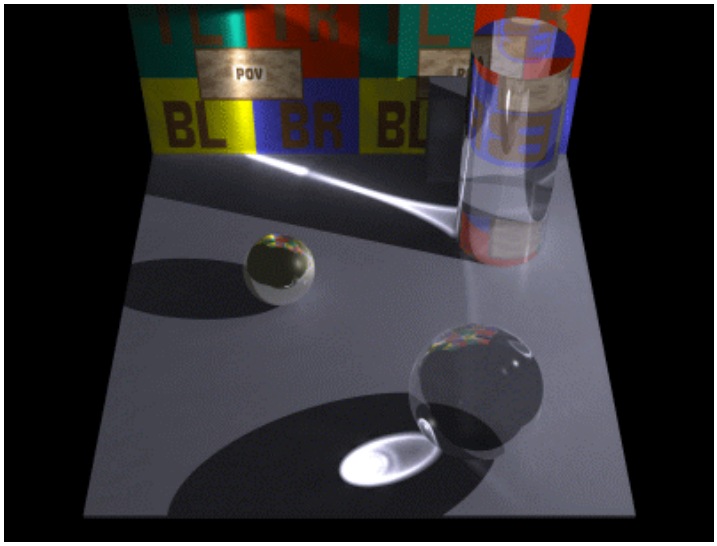


Figure 6 This is the Sphere and Cylinder test scene that was used extensively during the testing process.

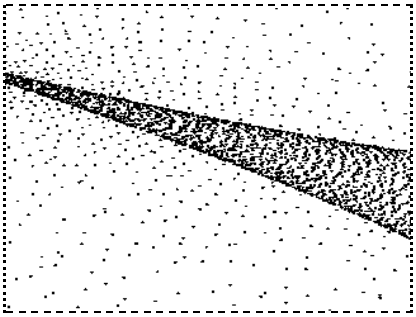


Figure 7 A visualization of the photon density used in figure 5.

Results of the Adaptive Search

The adaptive search technique, with the density-boundary compensation described above, has shown to be quite effective at decreasing render times. As previously mentioned, it has the added benefit of increasing the sharpness of the boundaries between areas of high photon density and areas of low photon density. Figure 5 shows a close-up of the Sphere and Cylinder scene, where the increased sharpness is visible. Figure 7 shows the density of photons used in figure 5.

Table 1 shows the results of a test of the adaptive searching technique. The scene is the Sphere and Cylinder Scene, shown in figure 6. For the adaptive search, the search radius was expanded if less than 20 photons were gathered. The scene was rendered at a resolution of 800x600 with an anti-aliasing threshold set to 0.3 on a P266MMX with 32MB of RAM running Windows 95. In the table, "Gather Count" refers to the number of times the function which gathers photons was called. "Radius Expansion Count" refers to the number of times the search radius was expanded by the adaptive search algorithm. This scene shows a drop in render time of almost 35%. It could be argued that the smaller search radius is used predominantly, and that this is the cause of the render speedup. However, the table also shows that the search radius was expanded for the majority (almost 64%) of gathering searches. This demonstrates that the smaller search radius was only used in areas of high photon density, and that the adaptiveness of the algorithm is functioning properly. The differences in "Gather Count" between the two renderings are caused by POV's adaptive anti-aliasing super-sampling algorithm. This test was run using the same compile of the program by specifying different options in the script file (see appendix A).

	Single Search	*Adaptive Search*
Caustic Photons	61,858	61,858
Search Radius	0.4472	0.2236
Extended Search Radius	-	0.4472
Gather Count	630,496	653,235
Radius Expansion Count	0	417,663
Light-Ray Trace Time	0 min, 14 sec	0 min, 14 sec
Eye-Ray Trace Time	8 min, 55 sec	5 min, 44 sec

Table 1 Render times for the Sphere and Cylinder Scene (800x300 AA 0.3) comparing adaptive search versus a single search of the kd-tree. The source for this scene is in appendix A.

Priority Queue Types

Searching the kd-tree utilizes a range-search technique which uses a priority queue to keep only the closest photons within the given range. Three different priority queues were tested to

determine their speed. This analysis was done because the "insert" operation is used substantially more than the "remove largest" operation in the priority queue. The three implementations of the priority queue that were tested were the unordered array (insert $O(1)$, remove $O(n)$), the ordered array (insert $O(n)$, remove $O(1)$), and the heap (insert $O(\log_2 n)$, remove $O(\log_2 n)$). Table 2 shows the results from testing these with three test scenes. Three scenes instead of only one were used since different scenes will have different characteristics for photon gathering. The three scenes used, one of which is shown in figure 5, provide a variety of environments for testing. Although the number of inserts of the priority queue, as shown in table 2, was generally an order of magnitude higher than the number of removes, the choice of priority queue implementation did not greatly affect the render time. However, in general, the heap implementation was the fastest, so it was used.

These results were actually unexpected. I had expected the unordered array implementation to be the fastest, due to the high number of inserts compared to removes. Surprisingly, in two of the three test scenes, the unordered array was the slowest. In one scene, the "Three Lenses" scene, the heap implementation was not the fastest. This is probably caused by a different proportions of insert and remove in this scene (only about two times the number of inserts as removes) compared to other scenes (which had close to ten times more inserts than removes).

Scene	Three Lenses	Pool Underwater	Sphere and Cylinder
number of photons	45,694	107,114	61,852
priority queue inserts	11,691,538	24,842,023	4,871,963
priority queue remove	5,220,772	2,522,424	310,966
parse time	12 sec	30 sec	17 sec
trace time (heap)	3 min, 01 sec	6 min, 15 sec	1 min, 13 sec
trace time (ordered array)	2 min, 54 sec	6 min, 31 sec	1 min, 15 sec
trace time (unordered array)	3 min, 33 sec	6 min, 27 sec	1 min, 16 sec

Table 2 Trace times with the various priority queue implementations for three test scenes. The Sphere and Cylinder scene is shown in figure 5. Again, this was rendered on a P266MMX with 32MB RAM running Windows 95.

Memory Optimizations

For photon mapping in POV-Ray, memory considerations are very important. POV-Ray users will likely be using this feature on low-end computers with limited memory. Also, when a scene uses thousands (or even hundreds of thousands) of photons, the size of the photon data structure is very important.

Shrinking the Photon

A photon contains four parts: flux, location, direction, and an info byte. The location is stored as a vector of three 'float' values. This, due to necessities in precision, cannot be compressed. The flux and direction of a photon, on the other hand, can be stored in a very small amount of space.

Flux Storage

The flux of a photon can be stored as only four bytes, instead of the 12 bytes needed to store three floating-point values (one for each of the red, green, and blue color channels). There are two techniques used for storing a color in only four bytes. Jensen uses a technique known as *rgbe* (for red-green-blue-exponent), which was developed by Greg Ward. Unfortunately, I was unable to find a reference to this method at first. Later, I learned that the *rgbe* format is described by Ward under the heading "Real Pixels" in the book *Graphics Gems II*. Instead, I developed a method that I call *rgbi*, which stands for red-green-blue-intensity. Equation (9) shows the computation of *r, g, b,* and *i*, which each represent one byte in the *rgbi* data structure. The variables *R, B,* and *B* represent the red, green, and blue values from the initial three-component color. The value of α, the range selector, can be set by the user, but the default value of 2048 allows for a large range of values.

$$
\begin{aligned}
i &= \left\lfloor \frac{256}{\max(R,G,B)\alpha} \right\rfloor + 1, \text{ (clip } i \text{ to 0..255 range)} \\
r &= \lfloor R\alpha(i+1) \rfloor \\
g &= \lfloor G\alpha(i+1) \rfloor \\
b &= \lfloor B\alpha(i+1) \rfloor
\end{aligned}
\qquad (9)
$$

To convert from *rgbi* format back to a three-component vector, the following formulas are used:

$$
R = \frac{r}{(i+1)\alpha}, \quad G = \frac{g}{(i+1)\alpha}, \quad B = \frac{b}{(i+1)\alpha} \qquad (10)
$$

Direction Storage

Direction of the incoming flux for a photon can also be stored using very little space. In fact, this can be stored using only two bytes, one for each of the two angles needed to specify a direction in 3D space. This gives a resolution of approximately 1.4 degrees in the worst case, which is adequate for a lighting approximation. Equation (11) shows how to convert the origin and intersection point of a ray into a θ-ϕ pair. These angles are then converted to a -127..+127 range and stored in signed bytes.

$$\vec{d} = \|Origin - Point\|, \quad \delta = \sqrt{d.x^2 + d.z^2}$$
$$\phi = \begin{cases} \cos^{-1}\left(\frac{d.x}{\delta}\right), & \text{if } d.z \geq 0 \\ -\cos^{-1}\left(\frac{d.x}{\delta}\right), & \text{if } d.z < 0 \end{cases} \qquad (11)$$
$$\theta = \begin{cases} \cos^{-1}(\delta), & \text{if } d.y \geq 0 \\ -\cos^{-1}(\delta), & \text{if } d.y < 0 \end{cases}$$

Equation (12) demonstrates how a directional vector *d* is recovered from a θ-ϕ pair.

$$y = \sin(\theta), \quad y = \cos(\theta), \quad z = 0$$
$$z' = x\sin(\theta), \quad x' = x\cos(\theta), \quad y' = y \qquad (12)$$
$$\vec{d} = \|\{x', y', z'\}\|$$

These two compression techniques decrease the size of the photon data structure by 18 bytes, which has a significant impact on total memory usage.

Median vs. Mean Split for KD-Tree

Two approaches were tried for building a balanced kd-tree to store the photons. The first approach used was a mean-split. With this approach, the node closest to the mean of all nodes in a sub-tree was chosen as the root for that sub-tree. This approach gave the advantage of a computational time of $O(n \log_2 n)$ for balancing the tree, and required a minimum amount of data movement for the balancing. Unfortunately, it also required two pointers per node with an additional array of pointers to each of the nodes for a total of three pointers for each node.

Because of this, a median-split approach was used instead. This approach used much more data movement and a computational time of $O(n \log_2 n \log_2 n)$ to balance the tree. However, it eliminated all need for pointers in the data structure, saving 12 bytes per photon. The trade-off of sort time, which is only a matter of seconds for most renderings, in exchange for such a

significant memory reduction seemed acceptable, so the median-split balancing approach was kept.

Using median-split cleaned up the code, too. The elimination of child pointers in the data structure increased code readability. The memory-allocation strategy that I use requires a somewhat confusing array mapping function, however. Memory for photons is allocated in blocks. An array of pointers points to these blocks. This basically creates a two-dimensional array of photons which dynamically grows in one direction, while the other dimension stays constant. A special array mapping function converts a one-dimensional index and into a two-dimensional address consisting of a block index and an offset within that block. Note that, while this data structure allows easy allocation of photons, it does not easily permit deallocation of photons. Once photons are placed into the photon map, they are not removed.

Figure 8 shows a visual representation of the two-dimensional photon memory allocation scheme. The base array contains pointers to blocks of photons. Each block is an array of fixed size. The 'C' code below is used to determine the two-dimensional address. In this code, *index* is the one-dimensional index of the photon, *base* is the index in the base array, and *offset* is the index in the photon block.

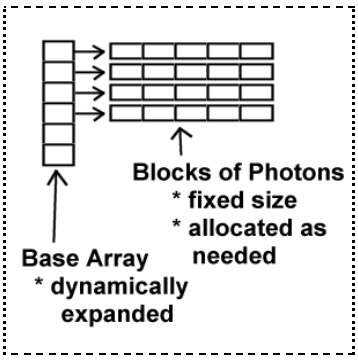


Figure 8 Photon memory allocation.

```
offset=(index & PHOTON_BLOCK_MASK);
base=(index >> (PHOTON_BLOCK_POWER));
```

The constant *PHOTON_BLOCK_SIZE* is the number of photons stored in one block. The constant *PHOTON_BLOCK_POWER* is equal to $log_2(PHOTON\_BLOCK\_SIZE)$. The final constant, *PHOTON_BLOCK_MASK* is set to *PHOTON_BLOCK_SIZE - 1* and represents the mask used to find the index.

Other Details

The Parser

Because of keyword changes and changes to the techniques used, the parser for photon mapping went through many revisions. For example, when I implemented multiple photon maps (multiple trees), the parser needed to read various options. When I later added adaptive searching and removed the multiple maps, I needed to revise the parser again to read a different set of options. Fortunately, the design of POV-Ray's parser made it relatively trivial to add and remove keywords. Unfortunately, I do not feel that I could do justice in explaining details about the parser here, except for the statement that making small changes is rather easy.

Keywords for the parser were suggested by people in the POV-Ray *unofficial-patches* newsgroup after I solicited suggestions. For example, in order to remove to remove some

confusion, the keyword “density” was changed to “separation” as per suggestion of a member of the group.

This implementation of the photon map in POV-Ray includes specific rules that allow the user to specify which objects receive photons, which objects do not, and which objects photons are allowed to pass through. These rules were created to minimize the number of photons used by a scene to improve memory usage. Implementing these rules required a change to the Trace() function in the file render.c, so that it would stop the ray tracing process if the proper conditions were not met.

Since no global photon map is used, photons are shot into the scene on a per-object basis. Photons that are shot toward an object and do not hit it (because they miss it or hit another object instead) are discarded. The user is able to specify the density of the photons for each object using the “separation” keyword. Users can also turn on photon reflection and refraction individually for each object as well as for each light. By default, no photons are used. If photons are turned off for a light, no photons will be shot from the light to any objects. If photons are turned off for an object, no photons will be shot to the object from any lights. On the other hand, to turn photons on, refraction or reflection need only to be specified in either the light or the object, not both. Appendix B contains part of the photon mapping documentation which further explains the syntax.

Photon storing and gathering can be completely turned off for an object as well through the use of the “ignore_photons” keyword. Using the “ignore_photons” keyword on glass objects can significantly speed up a render without much of a noticeable difference in the image quality. Similarly, some objects can be flagged as pass-through objects using the “photons_pass_through” keyword. Photons on their way to a target object are allowed to go through a pass-through object as if it was not there. An example of a pass-through object is a plate-glass window, where the target object is a glass of water inside the window and the light source is the sun outside the window. In such a situation, we do not want to use photons for the window, but we do want to use photons which hit the glass after passing through the window.

Photon gathering options are specified in the “global settings” section of a POV-Ray scene script file. These options are “radius,” which controls the size of the gathering radius, “gather,” which controls the number of photons to gather, and “jitter,” which controls a random jittering effect. Another option, “autostop,” allows the user to specify an the “auto-stop angle.” To minimize the number of rays shot in a spiral pattern at an object’s bounding box, the system will stop shooting rays if one revolution of the spiral is completed with no rays hitting the object, making the assumption that the entire object has already been covered with photons. For some objects, such as a metal ring, this will cause the system to stop prematurely. Therefore, the user can specify an angle, such that the auto-stopping feature is disabled until the spiral pattern has reached this angle (angle theta in figure 1). The user can also specify “max_trace_level” and “adc_bailout” to tell the rendering engine how many levels of recursion are to be used.

Dispersion

One feature that I though would really make this implementation of the photon map produce beautiful pictures is the phenomenon known as dispersion. Dispersion is the result of the splitting of wavelengths of light as they pass through a boundary between two materials. The refraction effect bends different wavelengths of light to different degrees, causing white light to separate into a beautiful spectrum of color.

Prior to this photon-mapping work, Daren Scot Wilson produced a modified version of POV-Ray, called DSW-Pov, that supported dispersion of refracted images.[13] Mr. Wilson's implementation works by dividing the color spectrum into discrete wavelengths and iterating through this discretized color spectrum. Each part of the color spectrum is reflected based on the frequency of the light being traced and the material's IOR. These various refracted rays are recombined to produce the total color. Mr. Wilson's modification works with eye-rays only, so my goal was to use a similar concept with light-rays.

The other ray-tracer that I had looked at which supported photon mapping, MSGTracer, also included the dispersion effect. Mike Gibson included a prism demonstration scene with his ray-tracer which demonstrated dispersion in action. In MSGTracer, each time a photon hits an object that has the dispersion option enabled, the ray-tracer chooses a random wavelength for the photon and uses this wavelength to determine the refraction angle and the color of the photon. This approach, unfortunately, produces splotchy results similar to the problems caused by pure Monte-Carlo sampling.

My implementation is a mixture of MSGTracer and DSW-Pov. I actually used code from both implementations (with permission) in my work. When a photon hits a surface that will cause dispersion, it is split into multiple photons. The colors of these spawned photons are determined by a color spectrum that is defined for the light source that emitted the photon. This process actually creates *more* photons (unlike Mr. Gibson's implementation), which leads to slower rendering times. However, the results look good and do no suffer from splotchiness like MSGTracer. I utilized code from MSGTracer to convert the RGB color value into a hue. The hue is then used to estimate wavelength. This estimation is also done both in MSGTracer and DSW-Pov. Code from DSW-Pov was employed to determine the refraction angle. Because all of the code for this section comes from either Mike Gibson or Daren Scot Wilson, I am not including the equations here, since I did not derive them. (In fact, I do not claim to fully understand all of the equations that are used.)

Error Analysis

Throughout this process, I basically ignored most error analysis. There is a good reason for this. First, let me explain what I mean by error analysis. Many lighting models are analyzed for error by comparing rendered scenes to real-life scenes. Various radiosity algorithms, for example, have been tested using the now famous "Cornell box" scene. However, the POV-Ray philosophy has always been, "if it looks good, it is right." This means that very few of the existing features of POV-Ray have been analyzed. In fact, POV-Ray contains a lot of

approximations and heuristics, making true error testing next to impossible. I did maintain the spirit of POV-Ray by doing a number of visual tests using a few example scenes (described below) to see if things 'looked right.' This 'looks right' philosophy was used for determining the brightness fudge factors explained previously, and for determining that the problem could be fixed without fudge factors.

Demo Scenes

The following demo scenes were created for testing photon mapping in POV-Ray.

Sphere and Cylinder
This scene was modeled after a scene I had scene on the web page of another renderer that rendered caustics. It consists of a transparent cylinder and sphere (IOR of 1.2 for each), a small reflective sphere, and two boxes to collect the refracted light and show the caustics. This scene was the most-used scene for testing and thus I have found great rendering options to make this scene render fast with high quality. The POV script source is provided in appendix A.

Underwater Pool
The underwater pool scene was used to test brightness as well as test caustic effects of water. I rendered the scene with and without photons with an IOR of 1.0 to test brightness. The scene contains three objects suspended under the water in a pool. The camera is also under the surface of the water. The first object is a wooden super-quadratic ellipsoid, the second is a glass CSG object, and the third is a Bezier patch object.

Three Lenses
This scene was created to demonstrate the effects of turning photons on and off for light sources and objects. It contains three lenses and three light sources.

Brightness
The brightness scene was used for testing brightness, as described previously. It contains three rings which are lit half by photons and half by direct lighting. The rings can be rotated to change the incoming angle of the light.

Spherical Camera
This is a simple scene consisting of a light source and camera at the origin and a sphere centered on the origin. The camera takes a full 360^{o} picture of the entire scene. I created the spherical camera earlier for creating images for sky-spheres.

Crystal Dreams
This was my IRTC (Internet Ray Tracing Competition) for the January-February 1999 contest. It contains a crystal bird which creates caustic patterns on the surface of the ground.

POV-Ray Community

This report would not be complete without another mention of the POV-Ray community. The community gave suggestions for the syntax, as mentioned before. Members also suggested features, such as adding max_trace_level and adc_bailout options that are specific to the photon pre-processing step. Someone in one of the newsgroups also suggested adding an option to specify if a CSG union should be split. Previously, I had wanted to test to see if it was more efficient to split up unions before shooting photons at them or if it was better to keep them together. After this suggestion and some further thought, I realized that neither option was necessarily better for all scenes. Therefore, I decided to use the suggestion and implemented the optional splitting of unions. The POV-Ray community also helped me find bugs, such as a hard-coded array length and a memory deallocation problem. They also utilized the new features, posting over twenty creative pictures using photon mapping to an images newsgroup.

Conclusions

The photon mapping technique works very well for rendering caustics in POV-Ray. It is not too bad in terms of memory consumption, as it was developed and tested on a P266 with only 32 MB of RAM running Windows 95. Testing was done with MS Visual C++ (which is very memory-hungry) running in the background and most test scenes did not swap to disk at all. The caustic effects are very pleasing to the eye and can add a lot to the realism of scenes that contain transparent objects. If options are set correctly, the photon map also does not greatly increase render time of most images, making it feasible for use even on low-end systems. It is not as fast as some other techniques (such as the illumination maps used in Collins' work), however. The biggest advantage that the photon map has over other techniques when implemented in POV-Ray is its detachment from the scene geometry.

Another conclusion that I regret to report is that directly visualizing global photon map for indirect diffuse illumination is not a good idea. Results look very splotchy and render times are very high. The technique may sound good at first (in fact, someone else even suggested it in the POV newsgroup before I reported my findings), but in actuality it does not work well at all.

Finally, this photon map implementation can and should be extended as Jensen has done with his ray-tracer (MIRO). A global photon map should be implemented which will be used to decrease shadow rays and aid in the rendering of global illumination. Adding these features will require further study of POV-Ray's features, but will be greatly beneficial to the future of the software.

Glossary

caustics: patterns of light created by indirect light refracted through or reflected from an object. Refracted and reflected light often converges or diverges based on variations in the surface normal of the object.

radiosity: a technique which renders indirect diffuse illumination by solving a large system of linear equations. The term radiosity is sometimes used to refer to any algorithm which renders indirect diffuse illumination, such as the cached-Monte-Carlo scheme used by POV-Ray.

Monte-Carlo (MC) ray tracing: a process which uses multiple rays statistically chosen for each pixel in the image. MC ray tracing, also known as distributed ray tracing, is used to render blurred reflections and refractions, as well as indirect lighting.

media: in the POV-Ray world, media (or "participating media") refers to atmospheric effects simulated by ray-marching algorithms. Media rendering is sometimes called volumetric rendering.

reflection: specular reflected light or images that are reflected clearly from an object (as opposed to diffuse reflection).

refraction: the bending of light as it passes from a material with one index of refraction (IOR) to a material of another IOR.

diffuse: in POV-Ray, the term of the rendering equation that represents light being scattered from the surface. Diffuse interaction refers to indirect lighting created when this scattered light contributes to the light hitting other surfaces in the scene.

specular: in POV-Ray, the term of the rendering equation that represents specular reflection of the light source. This is very similar to *phong*, but uses a different BRDF. Specular also refers to specular reflections (see reflection).

phong: in POV-Ray, the term of the rendering equation that represents specular reflection of the light source. This is very similar to *specular*, but uses the usual phong BRDF.

BRDF: Bi-directional Reflectance Distribution Function. This function specifies the probability that a light ray hitting the object in a specific direction will be reflected at some other direction. It is used to specify the 'finish' of a surface to determine how shiny or dull the surface looks.

CSG: Constructive solid geometry. In POV-Ray, this consists of union, intersection, difference, and merge, which work similarly to mathematical set operations.

Appendix A - Sphere and Cylinder scene POV Source

```
/*
  Sphere and Cylinder Scene
  for testing photon mapping

  Nathan Kopp
  Spring, 1999
*/

#declare refl=on;
#declare phd=2;

global_settings {
  max_trace_level 4
  ambient_light 1.3

  photons {
  #if(1)
    // use adaptive search radius
    gather 20, 100
    radius .05*phd*sqrt(20)/2, 2, 0
    autostop 0
    jitter .2
  #else
    // use non-adaptive search radius
    gather 20, 100
    radius .05*phd*sqrt(20), 1
    autostop 0
  #end
  }

}

camera {  //  Camera StdCam
  location  <      0.000,     -31.875,      35.630>
  direction <        0.0,         0.0,      2.1178> // Aperture is 0.46 degrees
//  angle 10
  sky       <    0.00000,     0.00000,     1.00000> // Use right handed-system
  up        <        0.0,         0.0,         1.0> // Where Z is up
  right     <    1.33333,         0.0,         0.0> // Aspect ratio
  look_at   <     -0.000,       1.019,       1.166>
//  look_at <0,1.019,5>
}

//
// *******  L I G H T S *******
//

light_source {   // Light1
  <17.904, -1.056, 11.705>
  color rgb <1.000, 1.000, 1.000>*.7
  fade_distance 0
  fade_power 0
}

//
// *******  T E X T U R E S *******
//

#declare Glass =
   material  // Glass
   {
      texture
      {
         pigment
         {
            color rgbf <1.0, 1.0, 1.0, 0.7>
         }
         finish
         {
            ambient 0.0
            diffuse 0.0
            specular 1.0
```

```
            roughness 0.001
            reflection 0.5
         }
      }
      interior
      {
         ior 1.2
      }
   }


#declare Chrome_Metal =
   material  // Chrome_Metal
   {
      texture
      {
         pigment
         {
            color rgb <0.729167, 0.718733, 0.552067>
         }
         finish
         {
            ambient 0.1007
            diffuse 0.884867
            brilliance 8.0
            phong 0.330933
            phong_size 52.316667
            specular 0.129467
            roughness 0.022533
            //reflection 0.575533
            reflection .9
         }
      }
   }


#declare PlexiGlass =
   material  // PlexiGlass
   {
      texture
      {
         pigment
         {
            color rgbf <1.0, 1.0, 1.0, 0.791667>
         }
         finish
         {
            ambient 0.0
            diffuse 0.0
            specular 1.0
            roughness 0.001
            reflection 0.5
         }
      }
      interior
      {
         ior 1.2
         // uncomment these lines and add a spectrum to the light source to
         // get dispersion
         //dispersion 1.05
         //disp_nelems 10
      }
   }


#declare GroundTex =
   material  // GroundTex
   {
      texture
      {
         pigment
         {
            color rgb <0.552067, 0.552067, 0.645833>
         }
         finish
         {
```

```
            ambient 0.1
            diffuse 0.913667
         }
      }
   }


#declare WallTex =
   material  // WallTex
   {
      texture
      {
         pigment
         {
            image_map
            {
               // change this file location
               gif   "C:\PROGRAM FILES\MORAY FOR WINDOWS\Maps\povmap.gif"
            }
         }
         finish
         {
            ambient 0.1007
            diffuse 0.906467
         }
      }
   }

//
// ********  OBJECTS  *******
//

box { // box by the wall behind the cylinder
  <-1, -1, -1>, <1, 1, 1>
  material {
    GroundTex
  }
  scale <1.0, 0.748224, 3.014206>
  translate  <4.142054, 8.593331, 2.014206>
}


sphere { // ClearSphere
  <0,0,0>,1
  material {
    Glass
  }
  scale 2.5
  translate  <4.312204, -6.330586, 2.6>
  photons {
    separation 0.02*phd
    refraction on
#if(refl)
    reflection on
#end
    ignore_photons
  }
}

cylinder { // clear cylinder
  <0,0,1>, <0,0,0>, 1
  material {
    PlexiGlass
  }
  scale <2.0, 2.0, 10.0>
  translate  <6.305907, 4.386718, 0.10001>
  photons {
    separation 0.02*phd//*1.5
    refraction on
#if(refl)
    reflection on
#end
    ignore_photons
  }
}
```

```
box { // floor
  <-1, -1, -1>, <1, 1, 1>
  material {
    GroundTex
  }
  scale <10.0, 10.0, 0.1>
}




box { // wall
  <-1, -1, -1>, <1, 1, 1>
  material {
    WallTex
    rotate 90.0*x
  }
  scale <10.0, 0.1, 10.0>
  translate  <0.0, 10.0, 10.0>
  //photons { ignore_photons }
}


sphere { // Reflecting Sphere
  <0,0,0>,1
  material {
    Chrome_Metal
  }
  scale 1.5
  translate  <-3.364552, 0.797831, 1.6>
  photons {
    separation 0.02*phd
#if(refl)
    reflection on
#end
    ignore_photons
  }
}
```

Appendix B - Documentation Excerpt

Overview

Photon mapping is a technique which uses a backwards ray-tracing pre-processing step to render refractive and reflective caustics realistically. This means that mirrors can reflect light rays and lenses can focus light.

Photon mapping works by shooting packets of light (photons) from light sources into the scene. The photons are directed towards specific objects. When a photon hits an object after passing through (or bouncing off of) the target object, the ray intersection is stored in memory. This data is later used to estimate the amount of light contributed by reflective and refractive caustics.

Limitations

Photon mapping can require a significant amount of memory.
Photon mapping can slow down the render (but not much more than an additional light source).
Photon mapping does not work with atmosphere *yet*.

Using Photon Mapping in Your Scene

To use photon mapping in your scene, you need to provide POV with two pieces of information. First, you need to specify details about photon gathering and storage. Second, you need to specify which objects receive photons, which lights shoot photons, and how close the photons should be spaced.

Global Settings

To specify photon gathering and storage options you need to add a **photons** block to the **global_settings** section of your scene. Here's an example:

```
#declare phd=1;
global_settings{
  photons{
    gather 20, 100
    radius 0.1*phd, 2, 2, 0.1*phd
    autostop 0
    jitter .4
    expand_thresholds 0.2, 40
  }
}
```

```
global_photon_block :==
photons {
  gather <min_gather>, <max_gather>
  radius <gather_radius>, <gather_count> [, <expand_step>]
  [autostop <autostop_angle>]
  [jitter <jitter_amount>]
  [expand_thresholds <percent_increase>, <expand_min>]
  [max_trace_level <photon_trace_level>]
  [adc_bailout <photon_adc_bailout>]
}
```

The keyword **gather** allows you to specify how many photons are gathered at each point during the regular rendering step. The first number (20 in this example) is the minimum number to gather, while the second number (100 here) is the maximum number to gather. These are good values and you should only use different ones if you know what you're doing. See the advanced options section for more information.

The keyword **radius** specifies the search radius for photons. When searching for photons, the system finds all photons within this search radius and has to sort through them to find only the closest ones. Thus, if the radius is too large, the render will be very slow, since the system has to sift through lots of photons to get the ones it needs. However, if the radius is to small, the system will not find enough photons and the image will look splotchy. The variable "phd" is used here to allow the user to quickly change the overall quality of the photon mapping used. More on this later. The other values following **radius** are explained later.

jitter specifies the amount of jitter used in the sampling of light rays in the pre-processing step. The default value is good and usually does not need to be changed. Both **autostop** and **expand_thresholds** will be explained later.

Shooting Photons at an Object

To shoot photons at an object, you need to tell POV that the object receives photons. To do this, create a **photons** block within the object. Here is an example:

```
object{
  MyObject
  photons {
    separation 0.02*phd
    refraction on
    reflection on
    ignore_photons
  }
}
```

```
object_photon_block :==
photons{
  [separation <separation_distance>]
  [refraction on|off]
  [reflection on|off]
  [ignore_photons]
}
```

In this example, the object both reflects and refracts photons. Either of these options could be turned off (by specifying **reflection off**, for example). By using this, you can have an object with a reflective finish which does not reflect photons for speed and memory reasons.

The density of the photons is specified using the **separation** keyword. The number after **separation** is the spacial separation of photons at the center of the object's bounding box. The photons are actually shot using a spiral pattern with uniform angular spacing, but this angular spacing is difficult to estimate, and therefore is computed internally from the spacial spacing specified by **separation**.

The "phd" variable is used here again. As you can guess, the separation of photons and gather radius are directly related. As long as both are kept in proper proportion, they can be scaled up and down to quickly change the total number of photons used. This is very useful for creating test renders and also for determining the correct ratio of separation and gather radius.

Note: Photons will not be shot at an object unless you specify a positive **separation** distance. Simply turning refraction on will not suffice.

Photons and Light Sources

Sometimes, you want photons to be shot from one light source and not another. In that case, you can turn photons on for an object, but specify "photons {reflection off refraction off}" in the light source's definition. You can also turn off only reflection or only refraction for any light source.

FAQ

My scene takes forever to render.
When POV-Ray builds the photon maps, it continually displays in the status bar the number of photons that have been shot. Is POV-Ray stuck in this step and does it keep shooting lots and lots of photons?
yes

- If you are shooting photons at an infinite object (like a plane), then you should expect this. Either be patient or do not shoot photons at infinite objects.
- Are you shooting objects at a CSG difference? Sometimes POV-Ray does a bad job creating bounding boxes for these objects. And since photons are shot at the bounding

box, you could get bad results. Try manually bounding the object. You can also try the **autostop** feature (try "autostop 0"). See the docs for more info on autostop.

no

- Does your scene have lots of glass? Glass is slow and you need to be patient.
- Does the render slow down at in places that have lots of photons (bright caustics)? If so, you should decrease gather radius and/or use adaptive search radius.

My scene has polka dots but renders really quickly.

You need to increase the density of the photons by decreasing the **separation** between them. If you already have plenty of photons (more than 50,000), try increasing the gather radius.

Adding photons slowed down my scene a lot, and I see polka dots.

This is usually caused by having both high- and low- density photons in the same scene. The low density ones cause polka dots, while the high density ones slow down the scene. It is usually best if the all photons are on the same order of magnitude for spacing and brightness. Be careful if you are shooting photons objects close to and far from a light source. The **separation** keyword specifies the separation at the *target object*, but the gathering step is concerned with separation at the photons' final destinations. The target object usually focuses or diverges the photons, so their separation will change depending on distances in the scene.

I added photons, but I don't see any caustics.

When POV-Ray builds the photon maps, it continually displays in the status bar the number of photons that have been shot. Where any photons shot?

no

- If your object has a hole in the middle, do not use the **autostop** feature.
- If your object does not have a hole in the middle, you can also try not using **autostop**, or you can bound your object manually.
- Try decreasing the **separation** of photons.

yes

Where any photons stored (the number in parentheses in the status bar as POV shoots photons)?

no

- Maybe they are getting shot

yes

- The photons may be diverging more than you expect. They are probably there, but you can't see them since they are spread out too much

The base of my glass object is really bright.

Use **ignore_photons** with that object.

References

1. Watt, Alan; Watt, Mark. *Advanced Animation and Rendering Techniques, Theory and Practice*. ACM Press. Addison-Wesley. New York. 1992.

2. Collins, Steven. *Adaptive Splatting for Specular to Diffuse Light Transport*. In proceedings of 5. Eurographics Workshop on Rendering, pp. 119-135. Darmstadt, 1994. (downloaded from http://vangogh.cs.tcd.ie/scollins/scollins.html)

3. Jensen, Henrik Wann. *Rendering Caustics on Non-Lambertian Surfaces*. In Proceedings of Graphics Interface '96. pp. 116-121. Toronto. May, 1996.

4. Jensen, Henrik Wann; Christensen, Per H. *Efficient Simulation of Light Transport in Scenes with Participating Media using Photon Maps*. In Proceedings of SIGGRAPH'98, pp. 311-320. July 1998.

5. Gibson, Mike. *MSGTracer*. http://members.prestige.net/~mikegi/msgtracer/msgtracer.htm. January, 1999.

6. Murphy, Michael; Dr. Steve Skiena. *A study of data structures for orthogonal range and nearest neighbor queries in high dimensional spaces*. Master's Project. Department of Computer Science, State University of New York at Stony Brook.

7. Ng, Andrew Java Kd-trees Demo www.ai.mit.edu/~ayn/. January,1999

8. Foley, Van Dam; Feiner, Huges. *Computer Graphics: Principles and Practice Second Edition in C*. Addison-Wesley. 1995.

9. Jensen, Henrik Wann. *Global Illumination using Photon Maps*. In "Rendering Techniques '96". Eds. X. Pueyo and P. Schröder. Springer-Verlag, pp. 21-30. 1996.

10. Cheah, Shu Chiun. *An Implementation Of A Recursive Ray Tracer That Renders Caustic Lighting Effects*. Unpublished independent study with Prof. David Mount, Dept. of Computer Science, University of Maryland. 1996.

11. Collins, Steven. *op. cit.*

12. Cheah, Shu Chiun. *op. cit.*

13. Wilson, Daren Scot. *Adding Dispersion to POV-Ray*. http://www.newcolor.com/darenw/dswpov/disp.html. April, 1999.