CS 371 Project 4:

Photon Mapping

Figure 1: *Sponza scene with indirect illumination rendered by photon mapping.*

1 Introduction

1.1 Overview

Photon mapping [1995; 1996] is a popular global illumination algorithm with attractive mathematical properties, especially for simulating caustics. It is popular for film and game production. For example, it was used for lighting in *Halo 3* [Chen and Liu 2008] and *Alice in Wonderland* [Martinez 2010]. Henrik Wann Jensen, the primary inventor of photon mapping, received an academy award ("Oscar") in 2004 for his algorithmic contribution to film rendering.

A common problem in global illumination is that most light transport paths traced backward from the eye never reach a light source, and most paths traced forward from the light sources never reach the eye. Photon mapping is a Monte Carlo algorithm for tracing both kinds of light paths half-way and then loosely connecting them to form full transport paths. It can capture all real-world illumination phenomena, and is mathematically **consistent**, meaning that its radiance estimate converges to the true solution as the number of path samples increase.

1.2 Educational Goals

In this project you will implement a photon mapping framework that can render a photorealistic image of *any* scene for which you can provide a suitable BRDF and geometric model. You will work from resources in the computer science literature including Jensen's paper [1996] and his SIGGRAPH course notes. Along the way you will:

- Learn to read a computer science paper and implement an algorithm from it.
- Design your own structure for a complex mathematical program.
- Learn to create and perform experimental analysis without explicit guidance.
- Solve the kind of radiometry problems we've seen in lecture on your own.
- Gain experience with **Monte Carlo importance sampling**, an essential technique for graphics and other high-performance computing domains including computational biology, finance, and nuclear science.
- Gain experience with the **hash grid** spatial data structure for expected $O(1)$ insert, remove, and linear-time gather in the size of the query radius and output.
- Use the **trait** and **iterator** software design patterns employed for statically-typed polymorphic data structures in C++.

1.3 Schedule

This is a challenging, pair-programming project that builds extends the previous ray tracer project. The program will only add about 100 lines to your existing ray tracer. However, they are mathematically sophisticated lines and you will take responsibility for the program design and experiment design yourself this week instead of having them given to you in the handout.

Note that you have three extra days and one extra lab session compared to most projects because the project spans Fall reading period. If you plan to take a four-day weekend I recommend completely implementing and debugging the forward trace before you leave.

As a reference, my implementation contained 9 files, 420 statements, and 300 comment lines as reported by iCompile. I spent substantially longer debugging this program than I did for the other projects this semester.

Out:	Tuesday,	October 5	
Checkpoint 1 (Sec. 3.2):	Thursday,	October 7,	1:00 pm
Checkpoint 2 (Sec. 3.3):	Thursday,	October 14,	1:00 pm
Due:	Friday,	October 15,	11:00 pm

2 Rules/Honor Code

You are encouraged to talk to other students and share strategies and programming techniques. You should not look at any other group's code for this project or use

code from other external sources except for: Jensen's publications, the pbrt library, RTR3 and FCG textbooks, materials presented in lecture, and the G3D library (as limited below). You may look at and use anyone's code from *last week's* project, with their permission.

During this project, you may use any part of the G3D library and look at all of its source code, including sample programs, with the exception of `SuperBSDF::scatter`, which you may not invoke or read the source for.

You may share data files and can collaborate with other groups to create test and visually impressive scenes. If you share a visually impressive scene with another group, ensure that you use different camera angles or make other modifications to distinguish your image of it. On this project you may use any development style that you choose. You are not required to use pure side-by-side pair programming, although you may still find that an effective and enjoyable way of working.

3 Specification

1. Implement the specific variant of the photon mapping [Jensen 1996] algorithm described in Section 3.1.

2. Create a graphical user interface backed by functionality for:

   (a) Selecting resolution

   (b) Real-time preview with camera control

   (c) Rendering the view currently observed in the real-time preview

   (d) Setting `RenderSettings::numEmittedPhotons`

   (e) Setting `RenderSettings::maxForwardBounces` and `RenderSettings::maxBackwardBounces`

   (f) Enabling explicit direct illumination (vs. handling direct illumination via the photon map)

   (g) Enabling shadow rays when direct illumination is enabled

   (h) Enabling real-time visualization of the stored photons over the wireframe (see Section 6.1).

   (i) Setting the photon gather radius r, `RenderSettings::photonRadius`

   (j) Displaying the number of triangles in the scene

   (k) Displaying the forward and backward trace times

3. Document the interfaces of your source code using Doxygen formatting.

4. Devise and render one visually impressive scene of your own creation, as described in Section 3.4.

5. Devise and render as many custom scenes as needed to demonstrate correctness or explore errors and performance, as described in Section 3.4.

6. Produce the reports described in Sections 3.2-3.4 as a cumulative Doxygen mainpage.

3.1 Details

Photon mapping algorithm was introduced by Jensen and Christensen [1995] and refined by Jensen [1996]. Use those primary sources as your guide to the algorithm. Additional information about the algorithm and implementation is available in Jensen's book [2001] and course notes [2007]. The notes are almost identical to the book and are available online. Section 7 of this project document summarizes mathematical aspects covered in class that are ambiguous in the original papers.

Make the following changes relative to the algorithm described in the 1996 paper. These increase the number of photons needed for convergence by a constant factor but greatly simplify the implementation. The algorithm remains mathematically consistent.

1. Do *not* implement:
   (a) The **projection map**; it is a useful optimization for irregular scenes but is hard to implement.
   (b) The **illumination maps** described in the 1995 paper. Pure photon maps are today the preferred implementation [Jensen 1996].
   (c) The **caustic map**—just store a single **global photon map** and do not distinguish between path types.
   (d) **Shadow photons**; they are an important optimization for area light sources but are unnecessary for point emitters.
2. Use `G3D::PointHashGrid` instead of a kd-tree to implement the photon map data structure. This gives superior performance [Ma and McCool 2002].
3. Ignore the 1995 implementation of the photon class. Instead use a variant on the 1996 implementation: store position, incident direction, and power for each photon. Both papers recommend compressing photons. Use a natural floating point representation instead of a compressed one. Do not store surface normals in the photons.
4. Use a constant-radius gather sphere (as described on page 7 of the 1996 paper). When you have that working, you may optionally implement the ellipsoid gathering method or the k-nearest-neighbor method and compare performance and image quality.
5. Implement the algorithm that Jensen calls"direct visualization of the photon map"[1] rather than final gathering. This means that you should not make multiple ray casts from a point when estimating indirect illumination, but instead use the radiance estimate from the photon map directly.

Beware that the description of the trace from the eye sounds more complicated than it actually is. You're just going to extend your existing ray tracer by adding indirect light to the direct illumination at every point.

[1] Jensen refers to the kind of gathering you're implementing as "visualizing the photon map." This document calls that simply "gathering" and the "radiance estimate" and uses "visualization" to refer to debugging tools.

3.2 Checkpoint 1 Report (due Thursday, Oct. 7, 1:00 pm)

1. Implement `Photon` class (it need not have any methods!)

2. Give pseudo-code for the entire photon mapping algorithm in your Doxygen mainpage.

   (a) Almost all of the information that you need is in this handout, but is distributed so that you have to think and read instead of copying it. Be sure to read everything in here.

   (b) Use a hierarchical list, with the four major steps at the top level and the details inside. Your full pseudo-code, including some LaTeX for the equations should be around 40 lines long.

   (c) I recommend using either nested HTML lists (`<ol><li>...</ol>`) or simple preformatted HTML (`<pre>...</pre>`) inside a Doxygen `\htmlonly....\endhtmlonly` block.

   (d) The details you need are in the implementation section (7), Jensen's notes [2007], and McGuire and Luebke's appendix [2009].

   (e) Use actual (and anticipated actual) names of `RayTracer` methods and members in the code so that they will be linked correctly when you really implement them. With careful use of `\copydoc` and `\copybrief` you can avoid typing the documentation twice.

   (f) Give *all* details for the implementation of `RayTracer::emitPhoton` (discussed Wednesday in class).

3. Prove (in your Doxygen mainpage) that the emitted photons collectively represent the total emitted power of the lights,

$$\sum_{P \in photons} P.\Phi = \sum_{E \in lights} L.\Phi. \tag{1}$$

4. Prove that L_o in Equation 5 has radiance units in your Doxygen mainpage.

5. Show a screenshot of the specified user interface in your Doxygen mainpage.

Tip: Parts of the photon mapping algorithm will be presented in lecture the Wednesday *after* the project starts, however you can begin now because the handout and papers contain all of that information and more.

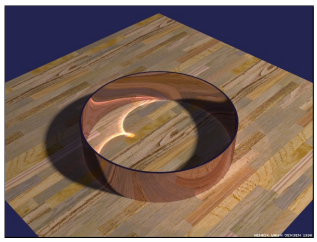
We will set up the photon map data structure together in the scheduled lab using the **trait** and **iterator** design patterns, and you will then implement and begin debugging your forward trace.

3.3 Checkpoint 2 Report (due Thursday, Oct. 14, 1:00 pm)

Complete a preliminary version of your report and hypothetical results, so that the formatting is done.

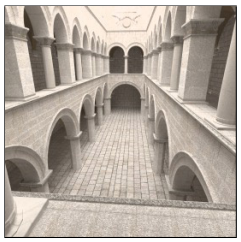
Answer the BSDF Diagrams questions from the final report. You can change your answer later, but try to get it right the first time.

Tip: You should have written the code for the entire program at this point, but it need not be working correctly yet. This way I can help you work through bugs during the scheduled lab without having downtime while you generate lots of new code. Your



(a) Cardiod caustic from a metal ring rendered by Henrik Wann Jensen using photon mapping.

(b) Cornell box photographic reference image by Francois Sillion



(c) Sponza atrium rendered by Matt Pharr and Greg Humphreys using photon mapping.

Figure 2: *Classic global illumination test scenes.*

forward trace should be working at this point, so you'll probably spend most of your time in the scheduled lab debugging the radiance estimate and performing experiments for your final report.

3.4 Final Report

Extend the mainpage of your Doxygen-generated documentation to include a brief, motivating description of your design (including where each major part of the algorithm is implemented), any known bugs and what you know about them, and the following elements. Retain the information from the checkpoint reports.

1. Render the following classic global illumination test scenes, **attempting to match the reference images as closely as possible**:

   Tip: `ifs/ring.ifs`

   (a) Metal ring from by Jensen [2001] shown in Figure 2(a), `http://graphics.ucsd.edu/~henrik/images/caustics.html`

   (b) Cornell box photographic reference by Francois Sillion shown in Figure 2(b), `http://www.graphics.cornell.edu/online/box/compare.html`

   (c) Sponza atrium from Pharr's and Humphreys's book shown in Figure 2(c), `http://www.pbrt.org/scenes_images/sponza-phomap.jpg`

2. Show one "teaser" image of a custom scene intended to impress the viewer at the top of your report. Plan to spend at least three person-hours just creating the scene for this image. You may collaborate with other groups on this step.

3. Show other images as needed of custom scenes to demonstrate correctness in specific cases and to support your discussion.

4. **BSDF Diagrams.** Create a set of BSDF diagrams as 2D schematics of the 3D shape for a fixed angle of incidence, as we commonly draw in lecture. Show the BSDF for red, green, and blue wavelengths (in the appropriate colors) either side-by-side or overlaid on the same image. To save time, I

recommend that you draw them by hand on paper or a black/whiteboard and just include the images. Show BSDFs for the following materials. You may wish to include a photograph of the material to support your diagrams.

(a) Aluminum foil
(b) Opaque purple plastic
(c) Green cloth
(d) Green glass

5. **Analyze your results.** Below are the kinds of questions that you should be asking yourself and discussing in your report. Don't answer these questions specifically–instead, pose and answer your own questions, which may overlap with these. Begin each part of the discussion with either an explicit question in boldface or an appropriate section title, such as "performance." Aspire to the kinds of result analysis presented in scientific papers. See McGuire and Luebke [McGuire and Luebke 2009] for examples of exploring parameter space, comparing, and presenting dense data.

   Sample questions:

   - How do your results compare, quantitatively and qualitatively with the input and output of previously published images in the papers that we've read?
   - What illumination effects are visible in each result? Conclude that the images confirm correctness or specific errors.
   - Are errors due to approximations in the algorithm, the data, or errors in your implementation?
   - How do the input parameters affect the quality and performance of results?
   - How much time is spent in forward and backward trace steps?
   - How expensive is the radiance estimate compared to the trace time?

Tip: Your analysis is the most important part of your report. Render a lot of different images, create plots and tables of data, and take the time to explore the behavior of the photon mapping algorithm.

4 Evaluation Metrics

I will evaluate this project in line with the metrics used for previous projects. However, because your program is significantly more complicated than in previous weeks, the mathematical correctness and program design areas will require more attention to receive high scores.

As is always the case, I am more concerned with your process than the result of the process. Put your effort into clean program and report structure and understanding the algorithm. It is less important whether your program produces correct results. A nearly-correct program may still produce completely incorrect images!

5 Getting Started

Start by exporting the previous week's code to new Subversion project. See the Tools handout from last week or refer to the Subversion manual for information

about how to do this. Remember that you can use *anybody's* code from the previous week with their permission, so if you aren't happy with your own project as a starting point, just ask around. You're responsible for everything in your program, so if you inherit bugs and lousy documentation from the code that you import, then you need to clean those up.

The Subversion command to check out your project this week is:

```
svn co svn://graphics-svn.cs.williams.edu/4-PhotonMap/photon-<group>
```

Replace `<group>` with your group name.

6 Debugging Advice

6.1 Visualization

Early on, implement a method for visualizing the photon locations over your wire-frame rendering, because that is an essential tool for debugging the forward trace.

Visualize small (< 1000) numbers of photons as arrows pointing along their $\hat{\omega}_i$ directions. For large numbers of photons, visualize them as points. In each case, set the color based on the photon power. Because the power of each photon is fairly small, normalize the power so the largest component is 1.0.

Use `G3D::Draw::arrow` to render arrows. To render points, use something like:

```
rd->setPointSize(5);
rd->beginPrimitive(PrimitiveType::POINTS);
for (PhotonMap::Iterator it = m_photonMap.begin(); it.hasMore(); ++it) {
    rd->setColor(it->power / it->power.max());
    rd->sendVertex(it->location);
}
rd->endPrimitive();
```

6.2 Program Trace

You will probably also need to instrument your forward trace to print information after each scattering event and trace a small number of photons in order to verify that the importance sampling is working. Use `G3D::debugPrintf` to output to the OS X console. If your program is crashing, `G3D::debugPrintf` may not actually be printing the last output before the crash. In that case, use `G3D::logPrintf` to write to `log.txt`, which is guaranteed to complete before the function returns. Of course, gdb is often the best tool for debugging a program that is crashing.

6.3 GuiTextureBox

The `G3D::GuiTextureBox` interactive inspector allows you to see the radiance values that your program wrote to the image. Remember that you can zoom in and out and change the exposure of this box. If you see unexpected black or white areas, hold the mouse over them to see their floating-point values. A value of `nan` or `inf` means that somewhere in your code you divided by zero or performed another undefined mathematical operation. You can create a `GuiTextureBox` explicitly or use the one that `GApp::show` produces.

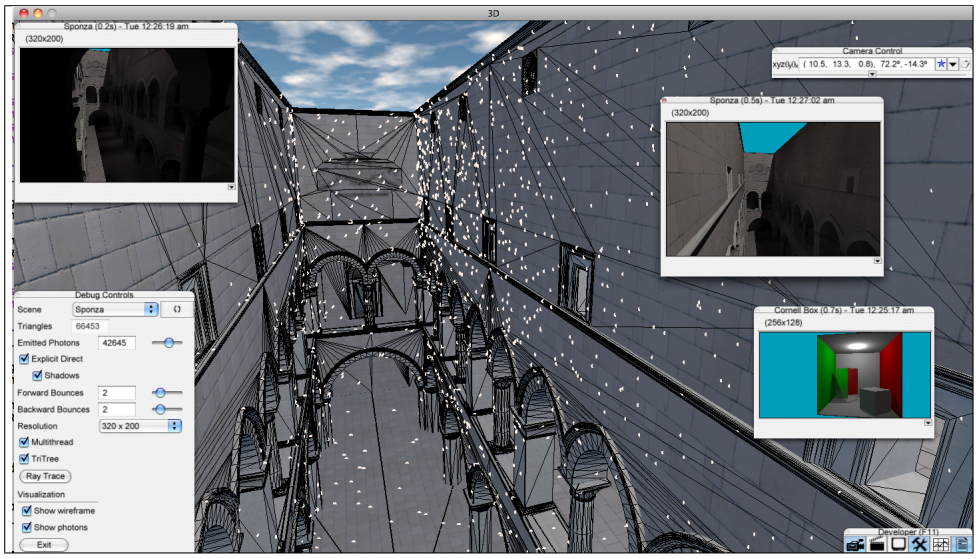


Figure 3: *User interface and debugging visualization for large numbers of photons.*

7 Implementation Advice

The following sections expand McGuire and Luebke's [2009] concise definition of global illumination by direct visualization of a global photon map. It contains design choices and advice designed to minimize implementation complexity at the cost of decreased convergence rate. I use the following notation:

- To reduce the number of distinct variables and subscripts, I use object-oriented notation to group the properties of a single object. E.g., Emitter E has power $E.\Phi$ and position $E.X$.
- Roman, non-italic subscripts are names (not array indices). E.g., in $\hat{\omega}_{\mathrm{i}}$, the "i" is a name (here, an abbreviation of "incident").
- All directions point outward from a location. Thus the "incident direction" of a photon is *opposite* the direction of propagation and corresponds to the light vector in a local illumination model.
- The average of a quantity c over all wavelengths is $\bar{c}$.

Because we consider the steady state, a single emitted photon may create multiple stored photons. Those represent points along the same path at different times. This is not "double counting" the same photon. Each stored P has a position $P.X$ in meters, an incident direction $P.\hat{\omega}_{\mathrm{i}}$, and incident power (a.k.a. radiant flux) $P.\Phi$ in Watts.

The algorithm consists of two phases: a **forward photon trace** outward from the light sources and a **backward ray trace** from the eye. These are linked by a **radiance estimate** performed where the paths nearly meet that is implemented using a hash grid of photons called a **photon map**.

7.1 Forward Trace

The forward photon trace computes the photon map, scattering ("bouncing") each photon a limited number of times. Each emitted photon may produce multiple stored photons. Repeat the following $numEmitted$ times:

1. Select an emitter $E \in emiters$ with probability proportionate to its relative power, averaged over wavelengths:

$$\bar{\rho}_{\mathrm{e}}(E) = \frac{E.\bar{\Phi}}{\sum_{F \in emitters} F.\bar{\Phi}}. \tag{2}$$

2. Let E be the selected emitter. Create a new photon P with power[2]

$$P.\Phi \leftarrow \frac{E.\Phi}{numEmitted \cdot \bar{\rho}_{\mathrm{e}}(E)}, \tag{3}$$

   and initial position $P.X \leftarrow E.X$ at the emitter. For an omnidirectional point emitter, choose direction $P.\hat{\omega}_{\mathrm{i}}$ uniformly at random on the sphere. For a spot light, use rejection sampling against the cone of the spot light to ensure that the emitted direction is within the cone.

3. Repeat at most $maxForwardBounces$ times:

   (a) Let Y be the first intersection of ray $(P.X, -P.\hat{\omega}_{\mathrm{i}})$ with the scene. If there is no such intersection, abort processing of this photon by immediately exiting this loop.

   (b) Update the photon with $P.X \leftarrow Y$.

   (c) Store a copy of photon P in the photon map.

   (d) Scatter the photon. If this photon is absorbed instead of scattering, abort processing it by immediately exiting this loop (this is **Russian roulette** sampling of the scattering function). At a surface whose reflectivity is ρ_{s} that varies with wavelength, let the scattering probability be $\bar{\rho}_{\mathrm{s}}$. If the photon scatters, let its power be updated by $P.\Phi \leftarrow P.\Phi \cdot \rho_{\mathrm{s}}/\bar{\rho}_{\mathrm{s}}$.

If explicit direct illumination is enabled in the GUI, do not *store* photons on their first bounce, but do still scatter them.

Some advice for the photon forward trace and the photon radiance estimate:

- A spot light has `G3D::GLight::spotHalfAngle` $\leq \pi/2$

[2] Note that the power of an individual photon is small when either $numEmitted$ or $|emitters|$ is large. Jensen chooses to scale power such that the average photon has power 1.0 at each wavelength, claiming that this avoids underflow and improves floating point accuracy [Jensen 2001]. I believe this is a misunderstanding of the IEEE floating point format and have not experimentally observed any loss of precision even when using millions of photons.

- `G3D::GLight::color` is the power that a spot light would have if it were a point light. The actual emitted power of the light taking the cone into account is `G3D::GLight::power`, which is what you should use. This interface makes it so that lights don't appear to get "darker" at points already within the cone when you adjust the cone angle.

- Be really careful with the photon direction convention. It is confusing for two reasons. First, the propagation direction is different than the incident direction that you eventually want to store. Second, the photon travels between two points during each iteration, and the vectors describing the direction between them point in opposite directions at each end. So your convention will be backwards for half of the iteration no matter what you do. Use assertions heavily and run targeted experiments with small numbers of photon and small numbers of bounces to ensure that you have this correct. There are several places that you will have to negate the photon direction.

- Just as with backwards tracing, you have to bump photons a small amount along the geometric normal of the last surface hit to avoid getting "stuck" on surfaces. If you see strange diagonal patterns of photons or the photons generally have the same color as the surface that they are on, you aren't bumping (or are bumping in the wrong direction!)

- Photons store *incident* illumination–before a bounce.

7.1.1 Scattering

Our BSDF from the previous projects contained a diffuse Lambertian lobe and a glossy Blinn-Phong lobe. The Blinn-Phong lobe could have an infinite exponent on the cosine term, representing a mirror reflection.

Scattering from a finite Blinn-Phong lobe is a little tricky[3]. Therefore, to simplify the implementation you will will approximate the approximate the BSDF during forward scattering with only a diffuse Lambertian lobe and a mirror-reflection impulse.

To implement the scattering, first query the intersection for the impulse(s) and the Lambertian coefficient. We ignore the glossy coefficient (consider the kinds of errors this creates, and what it means for the images). Choose a number t uniformly at random on [0, 1]. The coefficient of an impulse is the probability that a photon scatters along that direction. So, for each impulse, decrement t by the impulse's coefficient averaged over its wavelengths. If $t < 0$ after a decrement, the photon scattered in that direction, so update the photon's information and return it.

The probability of the photon performing Lambertian scattering is the Lambertian coefficient scaled by one minus the sum of the impulse coefficients. This has nothing to do with the radiometry or the mathematics–it is just the way that `SuperBSDF` is defined. It is defined that way so that it is easy to ensure energy conservation. If you increase the impulse probability in the scene file, it is understood that the sampling code (which in this case, you're the one writing!) will

[3]Two options are rejection sampling, which can be slow, and an approximate analytic form that is complex [Ashikhmin and Shirley 2000].

scale the Lambertian coefficient down appropriately. If none of the impulses scattered the photon, decrement t by the average of the updated Lambertian coefficient across wavelengths. If $t < 0$ after the decrement, the photon scatters diffusely. Choose the outgoing direction to be cosine distributed about the shading normal of the intersection and update the photon appropriately.

Some advice for scattering:

- Look at `G3D::Vector3::cosHemiRandom` for implementing the diffuse scattering.
- Implement a function "`bool scatter(const SurfaceSample& s, Photon& P)`" that returns false if the photon is absorbed and updates the photon if it is scattered.
- `G3D::PointHashGrid` is not threadsafe, and photon tracing is usually the fastest part of the program. So keep it single-threaded.
- Be really careful about which way the direction vector in the `Photon` is facing.

7.2 Backward Trace

The backward trace is exactly the same as for your previous ray tracing project, except shading now contains a new **indirect illumination** component in addition to the previous **direct illumination** and **specular illumination** components. If explicit direct illumination is disabled in the GUI, do not compute the direct contribution from light sources (but do still compute the specular component).

The indirect component is also called a **radiance estimate**. Consider a point Y with normal $\hat{n}$ and direction $\hat{\omega}_o$ to the eye (or previous intersection), that we reached by backwards tracing. If there were many photons stored at Y in the photon map, that would tell us the incident radiance. We could apply the BSDF and know the outgoing radiance to the eye.

However, it is extremely unlikely that any previously-traced forward path will terminate exactly *at* Y. So we estimate the incident flux from photons *near* Y in the photon map. As we make our definition of "near" more liberal, the indirect illumination will become blurrier. As we make it more conservative, the indirect illumination will be sharper–but also noisier.

The reflected (outgoing) radiance estimate is given by:

1. Let $L_o \leftarrow 0\ \mathrm{W}/(\mathrm{m}^2\mathrm{sr})$ be the initial estimate of radiance reflected towards the viewer.
2. Gather the photons nearest X from the photon map. Two methods for doing this are growing the gather radius until a constant number of photons have been sampled, and simply gathering from a constant radius. Regardless of the method chosen, let r be the gather radius in meters.
3. For each photon P within radius r of Y:

$$\text{Let } I_{\mathrm{i}} = \frac{P.\Phi}{\int_0^r \int_0^{2\pi} \kappa(s) \cdot s \, d\theta \, ds} \cdot \kappa(||P.X - Y||) \tag{4}$$

$$L_{\mathrm{o}} \leftarrow L_{\mathrm{o}} + I_{\mathrm{i}} \cdot f_Y(P.\hat{\omega}_{\mathrm{i}}, \hat{\omega}_{\mathrm{o}}) \cdot \max(0, P.\hat{\omega}_{\mathrm{i}} \cdot \hat{n}) \tag{5}$$

Compute the double integral in Equation 4 by hand; the denominator should be a constant over the loop.

Function $\kappa()$ is the 2D falloff for which Jensen recommends a cone [Jensen 1996] filter, $\kappa(x) = 1 - x/r$. When debugging, it is easier to first choose to have no falloff within the gather sphere, i.e., $\kappa(x) = 1$. In that case, the double integral is equal to the area of the largest cross-section of the sphere, πr^2. To help build some intuition for this, recall one derivation of the area of a disk of radius r. The area of a small rectangular patch on a disk at radius s is $(s \cdot d\theta) \cdot ds$; the length of this patch along the radial axis is clearly ds and, in the limit as $d\theta \to 0$, the length perpendicular to the radial axis is the length of the arc at radius s: $sd\theta$. The integral of the patch area expression over the full circle and disk radius is

$$\int_0^r \int_0^{2\pi} s \, d\theta \, ds = \frac{1}{2} s^2 \cdot 2\pi \Big]_0^r = \pi r^2. \tag{6}$$

The more general expression given in Equation 4 is the "area" of disk of variable density, where the density at distance s is $\kappa(s)$.

Tip: When you disable the explicit direct illumination, the noise and blurriness should increase, but the intensity of all surfaces should be the same. Test this with 1-bounce photons so there is no "indirect" light to confuse the issue.

8 What's Next

This section describes directions that you may be interested in exploring on your own, and possibly as a future midterm or final project. I do not expect you to implement these as part of this project.

Elliptical gathering: Compressing the gather sphere into an ellipsoid along the normal of each intersection helps avoid gathering photons that may not describe the flux at that intersection. Jensen describes this in detail in the notes and 1996 paper.

Final gathering: Only gather from caustic photons the way that we have been. For photons with some diffuse scattering event in their path, cast a large number of rays from each shading point and gather at *those* locations. This increases rendering time but dramatically improves both smoothness and sharpness in the rendered image.

Transmission: Transmissive surfaces pose some interesting design problems. You must track the medium from which the ray is exiting as well as the one that it is entering, implement Snel's law, and attenuate the photon energy based on the distance that it traveled through the transmissive medium. Refractive caustics are one of the effects for which photon mapping is extremely well suited. A related

problem is rendering **participating media** such as fog. This is typically done by marching the ray through the medium, periodically scattering as if hitting sparse particles.

Full BSDF scattering: We simplified the BSDF to make the scattering importance sampling algorithm relatively efficient and straightforward. Processing the true BSDF increases accuracy.

8.1 G3D::PointHashGrid

`G3D::PointHashGrid` uses the C++ **trait** design pattern. This is useful for cases where a data structure must work with a class that may not have been designed to work with it, and therefore does not have the right interface. The idea of this design pattern is that adapter classes can describe the *traits* of other classes.

`G3D::PointHashGrid<T>` requires a helper class that tells it how to get the position of an instance of the `T` class; in this case, `T` is `Photon`. The helper class must have a static method called `getPosition`. `PointHashGrid` also requires either an `operator==` method on the `Photon` class or another helper that can test for equality. See the documentation, which offers examples on this point.

References

ASHIKHMIN, M., AND SHIRLEY, P. 2000. An anisotropic phong brdf model. *J. Graph. Tools 5*, 2, 25–32. 11

CHEN, H., AND LIU, X. 2008. Lighting and material of halo 3. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, ACM, New York, NY, USA, 1–22. 1

JENSEN, H. W., AND CHRISTENSEN, N. J. 1995. Photon maps in bidirectional Monte Carlo ray tracing of complex objects. *Computers & Graphics 19*, 2, 215–224. 1, 4

JENSEN, H. W., AND CHRISTENSEN, P. 2007. High quality rendering using ray tracing and photon mapping. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, ACM, New York, NY, USA, 1. 4, 5

JENSEN, H. W. 1996. Global illumination using photon maps. In *Rendering Techniques*, 21–30. 1, 2, 3, 4, 13

JENSEN, H. W. 2001. *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd., Natick, MA, USA. 4, 6, 10

MA, V. C. H., AND MCCOOL, M. D. 2002. Low latency photon mapping using block hashing. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 89–99. 4

MARTINEZ, A., 2010. Faster photorealism in wonderland: Physically based shading and lighting at sony pictures imageworks, August. in Physically Based Shading Models in Film and Game Production SIGGRAPH 2010 Course Notes. 1

MCGUIRE, M., AND LUEBKE, D. 2009. Hardware-accelerated global illumination by image space photon mapping. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, ACM, New York, NY, USA, 77–89. 5, 7, 9

Index