EXTENDED PHOTON MAP IMPLEMENTATION

Matt Pharr

© 2005 Matt Pharr

Figure 1: Example scene used for photon mapping algorithm comparisons. Light is coming through a skylight in the ceiling and reflecting off of the wall in the back, indirectly lighting the rest of the scene. The implementation in this document renders this scene seven times faster than the `PhotonIntegrator`, producing a higher quality image as well.

This plugin implements solutions to exercises 16.13, 16.17 (partially), and 16.18. (And more!)

0.1 Overview

The `PhotonIntegrator` integrator in Section 16.5 of *Physically Based Rendering* (PBR) implements a basic version of the photon mapping algorithm. While that version suffices to explain the underlying ideas and translate the theory of the approach into practice, it doesn't include a number of important enhancements that help the algorithm produce the best possible results as efficiently as possible. Many of these enhancements are described in the exercises and further reading at the end of Chapter 16.

This document describes an extended photon mapping integrator with numerous improvements, `ExPhotonIntegrator`. Some parts of this integrator are very similar to or the same as the implementation in `PhotonIntegrator`; in this document we will elide code that is the same and focus on the differences between this implementation and the original one.

Figure 1 shows the scene we will use to demonstrate most of the improvements to the photon mapping algorithm implemented in this document. Note that indirect illumination is the dominant lighting effect; sunlight is shining through a skylight in the ceiling of the room, such that the room is primarily illuminated by indirect light reflecting from the wall. With the same parameter settings, the implementation here renders the example scene in 13.8% as much time as the original implementation, a 7.2 times speedup, while also producing a higher-quality image.

The improvements implemented here include:

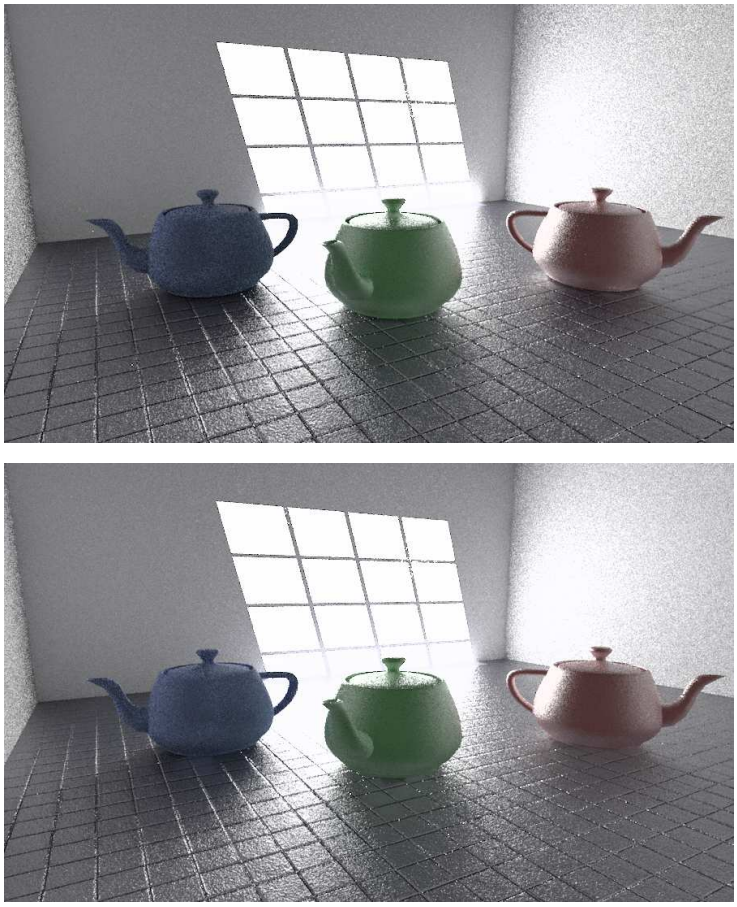

Figure 2: Top: scene rendered with original `PhotonIntegrator` from Section 16.5 of PBR. Bottom: scene rendered with photon mapping implementation described here. Note only does this implementation give fewer artifacts (note the reduction in noise on the side walls, for example), but it is over seven times faster than the original.

1. An improved photon shooting algorithm that reduces image artifacts by reducing the variation in the weights of the photons, such that the majority of them all have the same weights.

2. Much faster final gathering thanks to a separate "outgoing radiance" photon map that eliminates the need for a full shading calculation and photon map lookup at the final gather rays' intersection points.

3. An improved final gathering step that uses photons around the point being shaded to generate a distribution for importance sampling that tries to match the directional distribution of indirect lighting, tracing more rays in the directions where most of the indirect illumination is coming from.

4. Improved density estimation of photons using a non-constant kernel that gives less weight to photons farther away from the lookup point than those nearby.

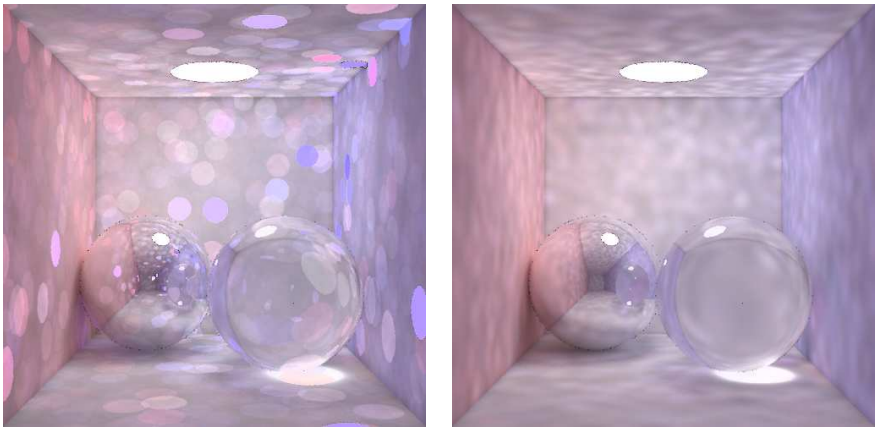

Figure 3: Left: scene rendered with original `PhotonIntegrator`. Right: scene rendered with `ExPhotonIntegrator`, which tries to have all photons have equal weights. The new implementation does not suffer from artifacts due to stray high-weight photons that make distinct bright circular blotches in the image. Photon maps are viewed directly here without final gathering to make artifacts more evident.

0.2 Constant Weight Photons

Recall from Section 16.5.1 on page 769 of PBR that it is both the distribution and the weights of the collection of particles in a scene that together represent the equilibrium radiance distribution. For example, the light on a surface with a relatively large amount of illumination arriving at it could be represented by a small number of particles each with large weights, a large number of particles with smaller weights, or something between these two extremes. As far as the theory underlying the method, any of these representations is an equivalently reasonable representation of the scene's radiance distribution.

In practice, the photon mapping algorithm works best if all of the photons have the same (or similar) weights, and if it is only their varying density throughout the scene that represents the variation in illumination. If the photon weights have substantial variation, there can be unpleasant image artifacts, as shown in Figure 3. If one photon takes on a much larger weight than the others, it is easy to see the region of the scene where that photon contributes to density estimation: there is a bright circular artifact in that area.

A subset of the photons may end up with large weights like this for a variety of reasons: for example, if the PDF used for sampling the BSDF when a photon scatters from a surface doesn't match its distribution perfectly, sometimes an outgoing direction may be sampled where the PDF's value is relatively small but the BSDF's value is relatively large. Then, the ratio $f(x)/p(x)$ will be large, and the scattered photon's weight α will also be large. Another case is a scene with surfaces with reflectance close to one. `PhotonIntegrator` terminates paths with probability $1/2$ after a few bounces and so for those photons that survive termination where the reflectance is large, the Russian roulette weighting term effectively causes their weights to double each time they aren't terminated. A few steps of this can give rise to photons with very large weights.

Two parts of the `ExPhotonIntegrator::Preprocess()` method must be modified so that the photons all have similar weights. The first part occurs as photons are shot from light sources and the second is related to how photons scatter from surfaces.

0.2.1 Sampling Lights by Power

Recall that in the `PhotonIntegrator::Preprocess()` method, each light is sampled with probability equal to one divided by the number of lights—there is equal probability of shooting a photon from all lights. If one light is much brighter than another, then with this approach the photons from that light will have much higher weights than photons shot from the others.

Instead, we'd like to shoot more photons from the brighter lights, giving them lower weights. If the number of photons shot from each light is proportional to its power, then the weights of photons leaving all lights will generally be the same. In this case, it will purely be a greater number of photons from the brighter lights that accounts for their greater contributions to illumination in the scene.

In order to implement this approach, we define a discrete PDF for choosing lights to shoot particles from based on each light's power. Before photon shooting starts, the `ComputeStep1dCDF()` function defined on page 643 of PBR is used to compute the discrete PDF.

⟨*Compute light power CDF for photon shooting*⟩≡ used: none

```
int nLights = int(scene->lights.size());
float *lightPower = (float *)alloca(nLights * sizeof(float));
float *lightCDF = (float *)alloca((nLights+1) * sizeof(float));
for (int i = 0; i < nLights; ++i)
    lightPower[i] = scene->lights[i]->Power(scene).y();
float totalPower;
ComputeStep1dCDF(lightPower, nLights, &totalPower, lightCDF);
```

The following fragment replaces the fragment of the same name defined on page 777 of PBR. The `SampleStep1d()` function chooses a light according to the PDF computed above and sets `lightPdf` to the value of the PDF for selecting the light.

⟨*Choose light to shoot photon from*⟩≡ used: none

```
float lightPdf;
float uln = RadicalInverse((int)nshot+1, 11);
int lightNum = Floor2Int(SampleStep1d(lightPower, lightCDF,
        totalPower, nLights, uln, &lightPdf) * nLights);
lightNum = min(lightNum, nLights-1);
const Light *light = scene->lights[lightNum];
```

After the light has been chosen, its `Sample_L()` method is used to sample an outgoing ray. This fragment below that does this is the same as in `PhotonIntegrator`, but is included here for reference. Note that it isn't possible to guarantee that all of the rays sampled by a light will have the same weights—some variation in photon weights may remain. For example, an area light with a texture map defining an illumination distribution might choose points uniformly over the light's surface rather than according to a PDF that matched the texture map's variation. In such a case, the weights for the rays from the lights would still be varying. Thus, the

`ExPhotonIntegrator` does a better job than the `PhotonIntegrator` at creating photons with equal weights, but it's not possible to guarantee this property by only making changes to the integrator here.

⟨*Generate* `photonRay` *from light source and initialize* `alpha`⟩≡ used: none

```
RayDifferential photonRay;
float pdf;
Spectrum alpha = light->Sample_L(scene, u[0], u[1], u[2], u[3],
    &photonRay, &pdf);
if (pdf == 0.f || alpha.Black()) continue;
alpha /= pdf * lightPdf;
```

0.2.2 Russian Roulette at Scattering Events

The other part of the implementation to be adjusted to keep photon weights as similar as possible is in the photon scattering step, when a photon interacts with a surface. In order to have a distribution of photons with equal weights, one approach suggested by Jensen (Jensen 2001, Section 5.2) is the following: at each intersection, compute the surface's reflectance. Then, randomly decide whether or not to continue the photon's path with probability proportional to the reflectance. Determine the photon's outgoing direction by sampling from the BSDF's distribution, and continue with its weight unchanged modulo adjusting the spectral distribution based on the surface's color. Thus, a surface that reflects very little light will reflect few of the photons that reach it, but those that are scattered will continue on with unchanged contributions and so forth.

This particular approach isn't possible in `pbrt` due to a subtle implementation detail (a similar issue held for light source sampling previously as well): in `pbrt`, the `BxDF` interfaces are written so that the distribution used for importance sampling BSDFs doesn't necessarily have to perfectly match the actual distribution of the function being sampled. It is all the better if it does, but for many complex BSDFs, exactly sampling from its distribution is difficult or impossible.

Therefore, here we will use an approach that generally leads to a similar result: at each intersection point, an outgoing direction is sampled with the BSDF's sampling distribution and the photon's updated weight α' is computed in the same manner as in `pbrt`'s original photon mapping integrator. Then, the ratio of the luminance of α' to the luminance of the photon's old weight α is used to set the probability of continuing the path after applying Russian roulette.

The probability is set so that if the new photon has a very low weight, the termination probability will be high and if the photon has a high weight, the termination probability is low. In particular, the termination probability is chosen in a way such that if the photon continues, after its weight has been adjusted for the possibility of termination, its luminance will be the same as it was before scattering. It is easy to verify this property from the fragment below, which replaces the fragment ⟨*Sample new photon ray direction*⟩ on page 781 and ⟨*Possibly terminate photon path*⟩ on page 782. (This property actually doesn't hold for the case where $\alpha' > \alpha$, as can happen when the ratio of the BSDF's value and the PDF is greater than one.)

⟨*Compute new photon weight and possibly terminate with RR*⟩≡ used: none

```
Spectrum fr = photonBSDF->Sample_f(wo, &wi, u1, u2, u3,
                                   &pdf, BSDF_ALL, &flags);
if (fr.Black() || pdf == 0.f)
    break;
Spectrum anew = alpha * fr *
    AbsDot(wi, photonBSDF->dgShading.nn) / pdf;
float continueProb = min(1.f, anew.y() / alpha.y());
if (RandomFloat() > continueProb || nIntersections > 10)
    break;
alpha = anew / continueProb;
```

One disadvantage of this photon shooting approach in comparison to the original one in `PhotonIntegrator` is that photon shooting generally takes longer with the implementation here. (In particular, the less light the scene surfaces reflect, the more paths will be terminated early with Russian roulette and the more photons will need to be shot from lights to populate the photon maps with the desired number of photons.) However, since photon shooting is usually a small fraction of total rendering time, and since artifacts are reduced with the improved approach here, this is normally a price worth paying.

0.3 Precomputed Radiance Values

An important performance optimization for photon mapping is based on an idea described in a paper by Christensen (Christensen 1999): much of the time taken up in photon mapping implementations is in photon map lookups at the points where final gather rays intersect the scene geometry. For example, if 64 final gather rays are traced from a point being shaded, it is necessary to do 64 traversals of the kd-tree to find the nearby photons at each of their intersection points in order to compute their contributions to incident radiance at the point being shaded.

Christensen observed that the outgoing radiance values computed with the photon map at the final gather ray intersection points don't necessarily need to be as accurate as radiance values computed when the photon maps are used for shading directly visible points. He then suggested choosing a fraction of the photons in the scene and precomputing the incident irradiance at each of them. Then, when a final gather ray intersects a point in the scene, only the surface's reflectance at the intersection point and the single closest photon with a precomputed irradiance value are needed to compute an outgoing radiance value for the ray, rather than tens or hundreds of photons as usual.

At a small cost in pre-processing, this approach substantially improves the efficiency of final gathering by reducing the time spent in kd-tree traversal and eliminating the need for maintaining a heap of photons. In `pbrt`, implementing this approach sped up rendering of the example scene in Figure 1 by a factor of over seven, even including the cost of precomputing the radiance values, which was roughly equal to the time spent shooting photons. (Because the example scene is a simple one, this speedup is something of a best case. For a more complex scene, more time will be spent finding intersection points of the final gather rays so that photon lookups aren't the only bottleneck as they are here. However, this optimization is still quite effective for complex scenes as well.)

Figure 4: Scene rendered directly using radiance values from radiance photons to shade visible surfaces. The radiance photons do a reasonable job of capturing an approximation of the outgoing radiance from surfaces in the scene. When used to compute radiance along the final gathering rays, the error they introduce usually isn't detectable and the final gathering process is substantially accelerated.

Our implementation here uses this basic idea introduced by Christensen but instead stores a constant exitant radiance value based on the surface's reflectance and the incident radiance distribution at the photon's position.[1] In contrast, the approach in Christensen's paper is to compute the irradiance at the photons, find the BSDF at the final gather ray's intersection point and then use the incident irradiance to compute the outgoing radiance along the ray. Our approach saves the step of finding the BSDF at final gather ray intersection points, though at the cost of not capturing BSDF spatial variation. Figure 4 shows an image of the scene rendered using the radiance photons, showing the scene "seen" by the final gathering rays.

0.3.1 Radiance Photon Creation

In order to implement this approach, are a few modifications to be made to the `Preprocess()` method to create the radiance photons. The following fragment replaces the fragment of the same name on page 779 of PBR. As the photon maps are being populated, a random subset of photons is selected to store these precomputed illumination values. Note that it's only necessary to do this if final gathering has been enabled; otherwise radiance photons won't ever be used.

[1] Christensen suggested this variant in a presentation; see "Photon Map Tricks" in the course slides from the SIGGRAPH 2001 course number 38.

⟨*Deposit photon at surface*⟩≡ used: none

```
Photon photon(photonIsect.dg.p, alpha, wo);
if (nIntersections == 1) {
    ⟨Deposit direct photon def?⟩
}
else {
    ⟨Deposit either caustic or indirect photon def?⟩
}
if (finalGather && RandomFloat() < .125f) {
    ⟨Store data for radiance photon 8⟩
}
```

The radiance photons only store the outgoing radiance value for the hemisphere of directions on the side of the surface the photon arrived from. It is important to distinguish between the two sides of the surface, since typically the outgoing radiance on different sides of a surface will be quite different–distinguishing between the two hemispheres here reduces "light leak" image artifacts. Therefore, when deciding to create a radiance photon, the implementation here checks to see if the surface is reflective, transmissive, or both at the photon's intersection point. It then creates a placeholder radiance photon, storing the surface normal (oriented to lie in the hemisphere the incident ray arrived from) and the hemispherical-hemispherical reflectance and transmittance of the surface.[2] Later, after all of the photons have been traced, the final outgoing radiance value is computed using the irradiance at the point and the reflectance values stored here.

⟨*Store data for radiance photon*⟩≡ used: 8

```
Normal n = photonIsect.dg.nn;
if (Dot(n, photonRay.d) > 0.f) n = -n;
radiancePhotons.push_back(RadiancePhoton(photonIsect.dg.p, n));
Spectrum rho_r = photonBSDF->rho(BSDF_ALL_REFLECTION);
rpReflectances.push_back(rho_r);
Spectrum rho_t = photonBSDF->rho(BSDF_ALL_TRANSMISSION);
rpTransmittances.push_back(rho_t);
```

Because the reflectance and transmittance only need to be stored temporarily during the preprocessing step between the time the `RadiancePhoton` is first created and the time that its outgoing radiance value is computed using the fully-populated photon maps after photon shooting is done, these values are stored in temporary arrays declared at the top of the `Preprocess()` photon shooting function.

⟨*Declare radiance photon reflectance arrays*⟩≡ used: none

```
vector<Spectrum> rpReflectances, rpTransmittances;
```

[2] Most of the `BxDF`s in `pbrt` use the default `BxDF::rho()` method to compute their radiance values. The default implementation of this method uses Monte Carlo integration. This estimation quickly becomes a bottleneck in the implementation of radiance precomputation here since those methods are called many times. We have found that by modifying the various `BxDF::rho()` method definitions in the files `core/reflection.h` and `core/reflection.cpp` to instead directly return approximate reflectance values computed directly, the time for this stage is substantially reduced. For example, for the Oren–Nayar model, the value of `OrenNayar::R` is returned for the reflectance. While the actual reflectance will in general be slightly different, this is a close enough approximation for most purposes.

⟨*Local Declarations*⟩+≡ used: none

```
struct RadiancePhoton {
    RadiancePhoton(const Point &pp, const Normal &nn)
        : p(pp), n(nn), Lo(0.f) {
    }
    Point p;
    Normal n;
    Spectrum Lo;
};
```

After the photon maps maps are all filled, exitant radiance can be computed at the radiance photons. One detail that is different from the `PhotonIntegrator` is that the `ExPhotonIntegrator` always computes direct illumination by tracing shadow rays and never using a direct lighting photon map. Therefore, a temporary direct lighting photon map needs to be created here in order to find the total incident radiance at the radiance photons.

⟨*Precompute radiance at a subset of the photons*⟩≡ used: none

```
KdTree<Photon, PhotonProcess> directMap(directPhotons);
int nDirectPaths = nshot;
if (finalGather) {
    for (u_int i = 0; i < radiancePhotons.size(); ++i) {
        ⟨Compute radiance for radiance photon i 10⟩
    }
    radianceMap = new KdTree<RadiancePhoton,
        RadiancePhotonProcess>(radiancePhotons);
}
```

The product of the reflectance and the irradiance at the point, as computed from the photon maps with the `estimateE()` method, can be used to find an approximate outgoing radiance value for this point, ignoring directional variation in both the incident radiance distribution and the BSDF.

$$
\begin{aligned}
L_{\mathrm{o}}(\mathrm{p},\omega_o) &= \int_{\mathcal{H}^2(\mathbf{n})} f_{\mathrm{r}}(\mathrm{p},\omega_o,\omega_i) L_{\mathrm{i}}(\mathrm{p},\omega_i) |\cos\theta_i| \, \mathrm{d}\omega_i \\
&\approx \frac{1}{\pi}\rho_{\mathrm{hh}}(\mathrm{p}) \int_{\mathcal{H}^2(\mathbf{n})} L_{\mathrm{i}}(\mathrm{p},\omega_i) |\cos\theta_i| \, \mathrm{d}\omega_i \\
&= \frac{1}{\pi}\rho_{\mathrm{hh}}(\mathrm{p}) E(\mathrm{p},\mathbf{n})
\end{aligned}
$$

This value is used to set the `RadiancePhoton::Lo` member variable.

⟨*Compute radiance for radiance photon* `i`⟩≡ used: 9

```
RadiancePhoton &rp = radiancePhotons[i];
const Spectrum &rho_r = rpReflectances[i];
const Spectrum &rho_t = rpTransmittances[i];
Spectrum E;
Point p = rp.p;
Normal n = rp.n;
if (!rho_r.Black()) {
    E = estimateE(&directMap,  nDirectPaths,   p, n) +
        estimateE(indirectMap, nIndirectPaths, p, n) +
        estimateE(causticMap,  nCausticPaths,  p, n);
    rp.Lo += E * INV_PI * rho_r;
}
if (!rho_t.Black()) {
    E = estimateE(&directMap,  nDirectPaths,   p, -n) +
        estimateE(indirectMap, nIndirectPaths, p, -n) +
        estimateE(causticMap,  nCausticPaths,  p, -n);
    rp.Lo += E * INV_PI * rho_t;
}
```

Most of the work in this step happens in the `estimateE()` method, which estimates the incident irradiance at the given point about the hemisphere with the given normal, using the provided photon map.

⟨*ExPhotonIntegrator Method Definitions*⟩≡ used: none

```
Spectrum ExPhotonIntegrator::estimateE(
    KdTree<Photon, PhotonProcess> *map, int count,
    const Point &p, const Normal &n) const {
    if (!map) return 0.f;
    ⟨Lookup nearby photons at irradiance computation point 10⟩
    ⟨Accumulate irradiance value from nearby photons 11⟩
}
```

This method first performs the usual photon map lookup, finding the given number of photons around the lookup point.

⟨*Lookup nearby photons at irradiance computation point*⟩≡ used: 10

```
PhotonProcess proc(nLookup, p);
proc.photons = (ClosePhoton *)alloca(nLookup *
    sizeof(ClosePhoton));
float md2 = maxDistSquared;
map->Lookup(p, proc, md2);
```

Given these photons, density estimation is performed, this time to estimate the incident irradiance at the point. How to compute the value of this estimate can be determined by defining a measurement function to compute irradiance at a point p with normal $\mathbf{n}$:

$$W_e(p', \omega) = \delta(p' - p) \begin{cases} 1 & (\mathbf{n} \cdot \omega) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

and following the approach used in Section 16.5.5 of PBR for deriving how to use photons to compute reflected radiance using density estimation. Note that photons

in the opposite hemisphere from the given surface normal are ignored, as they don't contribute to the outgoing radiance in the hemisphere of interest.

⟨*Accumulate irradiance value from nearby photons*⟩≡ used: 10

```
ClosePhoton *photons = proc.photons;
Spectrum E(0.);
for (u_int i = 0; i < proc.foundPhotons; ++i)
    if (Dot(n, photons[i].photon->wi) > 0.)
        E += photons[i].photon->alpha;
return E / (float(count) * md2 * M_PI);
```

0.3.2 Using Radiance Photons for Final Gathering

After these radiance photons have been created, the final gathering method needs to use them at the intersection points of final gather rays. The fragments below are part of the ⟨*Compute exitant radiance at final gather intersection*⟩ fragment that replaces the original one on page 788 of PBR.

⟨*Compute exitant radiance using precomputed irradiance*⟩≡ used: 15,17

```
Spectrum Lindir = 0.f;
Normal n = gatherIsect.dg.nn;
if (Dot(n, bounceRay.d) > 0) n = -n;
RadiancePhotonProcess proc(gatherIsect.dg.p, n);
float md2 = INFINITY;
radianceMap->Lookup(gatherIsect.dg.p, proc, md2);
if (proc.photon)
    Lindir = proc.photon->Lo;
```

The `RadiancePhotonProcess` structure only needs to record the single closest radiance photon to the lookup point.

⟨*Local Declarations*⟩+≡ used: none

```
struct RadiancePhotonProcess {
    ⟨RadiancePhotonProcess Methods def?⟩
    const Point &p;
    const Normal &n;
    mutable const RadiancePhoton *photon;
};
```

Along the lines of `PhotonProcess`, defined on page 790 of PBR, `operator()` of the `RadiancePhotonProcess` structure is called by the kd-traversal method for each photon within the search radius. Because it is only necessary to record the single closest photon to the lookup point, the implementation here is particularly straightforward. In looks for the single closest radiance photon with normal pointing in the same hemisphere as the normal at the point the final gather ray intersected a surface, storing a pointer to it and reducing the maximum search radius when it finds one. (Recall from page 871 of PBR that `KdTree::Lookup()` doesn't call this method for any items outside the search radius, so it's not necessary to check that the given `RadiancePhoton` is the closest one here.)

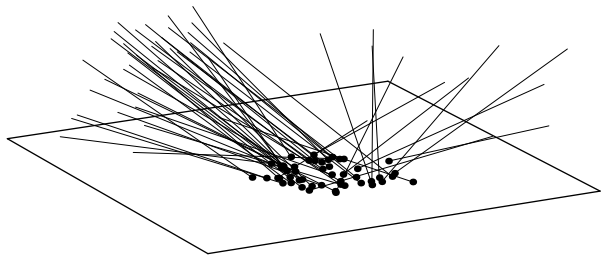

Figure 5: Incident directions of fifty photons around a point on the floor of the example scene. Many of the photons are arriving from a cluster of nearby directions, representing light reflected from the bright area on the back wall of the room. These directions can be used to define a distribution for importance sampling that approximates the distribution of indirect illumination at the point.

⟨*RadiancePhotonProcess Methods*⟩+≡ used: 11

```
void operator()(const RadiancePhoton &rp,
        float distSquared, float &maxDistSquared) const {
    if (Dot(rp.n, n) > 0) {
        photon = &rp;
        maxDistSquared = distSquared;
    }
}
```

0.4 Improved Final Gathering

The photons around a point in the scene are not just useful for estimating the incident radiance distribution at the point in order to compute the outgoing reflected radiance: in their incident directions, they also carry useful information about the directional distribution of illumination at the point. This information can be put to use in the final gathering step in order to define a sampling distribution for the final gathering rays that tries to match the distribution of indirect illumination, tracing more rays in directions where most of the illumination is coming from. This idea was first introduced by Jensen (Jensen 1996); improvements have since been proposed by Hey and Purgathofer (Hey and Purgathofer 2002). Figure 5 shows a plot of the incident directions of fifty photons around a point on the floor of the example scene; note that most of them have incident directions along a relatively small cluster of directions.

When choosing ray directions for final gathering from a point, in some cases, sampling the BSDF will be the most effective strategy (for example, where the BSDF is very glossy and the amount of indirect illumination is similar in all directions); in others, sampling the indirect illumination will be a better approach (for example, when the indirect illumination is only from a small subset of directions, and the BSDF is diffuse.) In general, this situation is similar to the case we

are faced with when sampling direct illumination from area light sources: sometimes it is most effective to sample the BSDF, and sometimes it is most effective to sample points on the surface of the light source. In both of these cases, allocating some samples to each approach works well in general, particularly when multiple importance sampling is used when evaluating the Monte Carlo estimator.

Therefore, in the implementation to follow, final gathering is done by sampling the BSDF to find directions for half of the final gather rays and sampling the indirect illumination distribution, as represented by the nearby photons' directions, for the other half. The `ExPhotonIntegrator::RequestSamples()` method therefore needs to request two sets of samples for final gathering; one will be used for the BSDF samples and the other for the samples taken using the nearby photons to define a sampling distribution.

⟨*ExPhotonIntegrator Private Data*⟩≡ used: none

```
int gatherSampleOffset[2], gatherComponentOffset[2];
```

⟨*Request samples for final gathering*⟩≡ used: none

```
if (finalGather) {
    gatherSamples = scene->sampler->RoundSize(max(1,
        gatherSamples/2));
    gatherSampleOffset[0] = sample->Add2D(gatherSamples);
    gatherSampleOffset[1] = sample->Add2D(gatherSamples);
    gatherComponentOffset[0] = sample->Add1D(gatherSamples);
    gatherComponentOffset[1] = sample->Add1D(gatherSamples);
}
```

Final gathering is handled by the fragment ⟨*Do one-bounce final gather for photon map*⟩ below, which replaces the fragment of same name on page 786 of PBR.

⟨*Do one-bounce final gather for photon map*⟩≡ used: none

```
BxDFType nonSpecular = BxDFType(BSDF_REFLECTION |
    BSDF_TRANSMISSION | BSDF_DIFFUSE | BSDF_GLOSSY);
if (bsdf->NumComponents(nonSpecular) > 0) {
    ⟨Find indirect photons around point for importance sampling 14⟩
    ⟨Copy photon directions to local array 14⟩
    ⟨Use BSDF to do final gathering 14⟩
    ⟨Use nearby photons to do final gathering 15⟩
}
```

First, a fixed number of photons are searched for around the point being shaded by performing a lookup in the indirect photon map. (Recall that the squared distance parameter passed to the `KdTree::Lookup()` method may be modified during the traversal process. Thus, it's necessary to store the current search radius in a separate variable here, `searchDist2`.) The photon search radius is progressively expanded until the desired number of photons is found.

⟨*Find indirect photons around point for importance sampling*⟩≡ used: 13

```
u_int nIndirSamplePhotons = 50;
PhotonProcess proc(nIndirSamplePhotons, p);
proc.photons = (ClosePhoton *)alloca(nIndirSamplePhotons *
    sizeof(ClosePhoton));
float searchDist2 = maxDistSquared;
while (proc.foundPhotons < nIndirSamplePhotons) {
    float md2 = searchDist2;
    proc.foundPhotons = 0;
    indirectMap->Lookup(p, proc, md2);
    searchDist2 *= 2.f;
}
```

Once the desired number of photons has been found, their directions are copied to a local array. This extra step improves performance for the rest of the final gathering work by ensuring that all of the photons' directions can be found without incurring excessive cache misses. If instead the original `Photon` structures pointed to by `prog.photons[].photon` were used, the pattern of memory accesses would be relatively incoherent and unused extra data from the `Photon` structure would potentially be repeatedly loaded into the cache.

⟨*Copy photon directions to local array*⟩≡ used: 13

```
Vector *photonDirs = (Vector *)alloca(nIndirSamplePhotons *
    sizeof(Vector));
for (u_int i = 0; i < nIndirSamplePhotons; ++i)
    photonDirs[i] = proc.photons[i].photon->wi;
```

The overall structure of the code for doing BSDF-based final gathering is the same as in `PhotonIntegrator`.

⟨*Use BSDF to do final gathering*⟩≡ used: 13

```
Spectrum Li = 0.;
for (int i = 0; i < gatherSamples; ++i) {
    ⟨Sample random direction from BSDF for final gather ray 14⟩
    ⟨Trace BSDF final gather ray and accumulate radiance 15⟩
}
L += Li / gatherSamples;
```

As before, specular components of the BSDF are ignored here, since they are handled separately with recursive ray tracing.

⟨*Sample random direction from BSDF for final gather ray*⟩≡ used: 14

```
Vector wi;
float u1 = sample->twoD[gatherSampleOffset[0]][2*i];
float u2 = sample->twoD[gatherSampleOffset[0]][2*i+1];
float u3 = sample->oneD[gatherComponentOffset[0]][i];
float pdf;
Spectrum fr = bsdf->Sample_f(wo, &wi, u1, u2, u3,
    &pdf, BxDFType(BSDF_ALL & (~BSDF_SPECULAR)));
if (fr.Black() || pdf == 0.f) continue;
```

Once difference here is that the radiance photons are used at the intersection point to compute exitant radiance there. Furthermore, in computing a final gather

ray's contribution, multiple importance sampling is used to re-weight the ray. Since both the BSDF and the indirect lighting distribution approximated by the nearby photons may match important components of the integrand, MIS improves the quality of the final results in cases where one or the other of the distributions matches the integrand much better than the other one.

⟨*Trace BSDF final gather ray and accumulate radiance*⟩≡ used: 14

```
RayDifferential bounceRay(p, wi);
Intersection gatherIsect;
if (scene->Intersect(bounceRay, &gatherIsect)) {
    ⟨Compute exitant radiance using precomputed irradiance 11⟩
    Lindir *= scene->Transmittance(bounceRay);
    ⟨Compute MIS weight for BSDF-sampled gather ray 15⟩
    Li += fr * Lindir * AbsDot(wi, n) * wt / pdf;
}
```

The fragment that computes the PDF for sampling a given direction from the photons' distribution, ⟨*Compute PDF for photon-sampling of direction* `wi`⟩, will be defined shortly, after the particular sampling approach used is explained.

⟨*Compute MIS weight for BSDF-sampled gather ray*⟩≡ used: 15

```
⟨Compute PDF for photon-sampling of direction wi 16⟩
float wt = PowerHeuristic(gatherSamples, pdf,
    gatherSamples, photonPdf);
```

Given the photons around the point being shaded, the sampling approach here defines a cone of directions centered about each photon's incident direction. The sampling method then chooses one of the photons at random and randomly samples a direction inside its cone to select a final gather ray.

Note that this sampling approach may not ever generate samples for some directions ω_i where the amount of indirect illumination is non-zero. Therefore, this wouldn't be an acceptable sampling technique if it was the only method being used with the standard Monte Carlo estimator. However, with multiple importance sampling, it is only necessary that *one* of the sampling methods have non-zero probability of sampling directions where the integrand is non-zero. Because it is required that this be true for the BSDF sampling methods, this shortcoming of the photon-based sampling method we have chosen isn't a problem here.

⟨*Use nearby photons to do final gathering*⟩≡ used: 13

```
Li = 0.;
for (int i = 0; i < gatherSamples; ++i) {
    ⟨Sample random direction using photons for final gather ray 16⟩
    ⟨Trace photon-sampled final gather ray and accumulate radiance 17⟩
}
L += Li / gatherSamples;
```

One of the photons is chosen with uniform probability from the collection of nearby ones.

⟨*Sample random direction using photons for final gather ray*⟩≡ used: 15

```
float u1 = sample->oneD[gatherComponentOffset[1]][i];
float u2 = sample->twoD[gatherSampleOffset[1]][2*i];
float u3 = sample->twoD[gatherSampleOffset[1]][2*i+1];
int photonNum = min((int)nIndirSamplePhotons - 1,
    Floor2Int(u1 * nIndirSamplePhotons));
⟨Sample gather ray direction from photonNum 16⟩
```

Given that photon, a direction is sampled from a cone about its incident direction with uniform probability. This is easily done with the `UniformSampleCone()` routine, defined on page 698 of PBR. The member variable `cosGatherAngle` is initialized in the `ExPhotonIntegrator` constructor based on a parameter that can be set in the input file. By default, rays are distributed in a cone that subtends an angle of ten degrees.

⟨*Sample gather ray direction from* `photonNum`⟩≡ used: 16

```
Vector vx, vy;
CoordinateSystem(photonDirs[photonNum], &vx, &vy);
Vector wi = UniformSampleCone(u2, u3, cosGatherAngle, vx, vy,
    photonDirs[photonNum]);
```

It's easy to compute the PDF for sampling a given direction with the nearby photons: it's the average of the PDFs for each of the photons to sample the direction. For each photon, a dot product indicates if the given direction is within the cone of possible directions for that photon; if the direction is inside the cone, the constant PDF for sampling a direction in a cone of the given angle is added to the PDF sum.

A small fudge-factor is used in the test for whether a given direction is within a given cone in order to handle the case where `UniformSampleCone()` selected a direction at the very edge of the cone. In that case, the straightforward dot product test here might indicate that the direction wasn't in the cone it was sampled from, leading to `photonPdf` being zero and thence to not-a-number or infinite values in the final image.

⟨*Compute PDF for photon-sampling of direction* `wi`⟩≡ used: 15,17

```
float photonPdf = 0.f;
float conePdf = UniformConePdf(cosGatherAngle);
for (u_int j = 0; j < nIndirSamplePhotons; ++j)
    if (Dot(photonDirs[j], wi) > .999f * cosGatherAngle)
        photonPdf += conePdf;
photonPdf /= nIndirSamplePhotons;
```

Given these approaches for sampling ray directions and computing their PDFs the same process is used for computing the contributions of final gather rays as is used for rays found by sampling the BSDF.

⟨*Trace photon-sampled final gather ray and accumulate radiance*⟩≡ used: 15

```
Spectrum fr = bsdf->f(wo, wi);
if (fr.Black()) continue;
⟨Compute PDF for photon-sampling of direction wi 16⟩
RayDifferential bounceRay(p, wi);
Intersection gatherIsect;
if (scene->Intersect(bounceRay, &gatherIsect)) {
    ⟨Compute exitant radiance using precomputed irradiance 11⟩
    Lindir *= scene->Transmittance(bounceRay);
    ⟨Compute MIS weight for photon-sampled gather ray 17⟩
    Li += fr * Lindir * AbsDot(wi, n) * wt / photonPdf;
}
```

Here, in order to find the MIS weight, the BSDF's PDF for choosing the direction must be computed.

⟨*Compute MIS weight for photon-sampled gather ray*⟩≡ used: 17

```
float bsdfPdf = bsdf->Pdf(wo, wi);
float wt = PowerHeuristic(gatherSamples, photonPdf,
        gatherSamples, bsdfPdf);
```

0.5 Improved Density Estimation Kernel

The last change in this photon mapping implementation is that it uses a different kernel function for the density estimation step. The implementation in the `PhotonIntegrator` uses a constant kernel, giving all of the photons used equal weight. Intuitively, it might make sense to use a kernel that gives greater weight to photons near the lookup point and less weight to photons further away. Doing so is generally a good idea, since the far-away photons give less useful information about illumination at a point than the nearby ones, and it also smoothes out the illumination estimates computed by the photon map, a generally desirable property (sharp caustics aside).

Figure 6 shows the example scene rendered using the new kernel implemented in this section as well as rendered using the old constant kernel (with no final gathering, so that the differences are more evident.) In practice, a non-constant kernel helps most in scenes with photons with widely-varying weights. Since the `ExPhotonIntegrator` doesn't suffer from that problem to the same degree as the original `PhotonIntegrator`, the new kernel's effect is relatively minor. Note also that when radiance photons are used for final gathering and rays are traced for direct illumination, regular photon map lookups are only done for caustics so this kernel will see little use in this implementation as well.

Below is a minor modification to the `LPhoton()` method that uses Simpson's kernel. Recall that the constant kernel used in `PhotonIntegrator` is

$$k(x) = \begin{array}{ccl} \frac{1}{\pi} & : & |x| < 1 \\ 0 & : & \text{otherwise}. \end{array}$$

Simpson's kernel is

$$k(x) = \begin{array}{ccl} \frac{3}{\pi}\left(1 - \left(\frac{d_i(x)}{d_n(x)}\right)^2\right)^2 & : & |x| < 1 \\ 0 & : & \text{otherwise}, \end{array}$$

Figure 6: Top: scene rendered using constant kernel for density estimation, as in the standard `PhotonIntegrator`, Bottom: scene rendered using Simpson's kernel, as implemented in this section. Note that the circular artifacts on the floor are less evident with the new kernel, though the walls are more blotchy.

where $d_i(x)$ is the distance to the ith point used in the density estimate. This kernel was first used for density estimation in graphics by Shirley et al (Shirley, Wade, Hubbard, Zareski, Walter, and Greenberg 1995).

⟨*Local Declarations*⟩+≡ used: none

```
inline float kernel(const Photon *photon, const Point &p,
        float md2) {
    float s = (1.f - DistanceSquared(photon->p, p) / md2);
    return 3.f / (md2 * M_PI) * s * s;
}
```

Given this kernel, it's straightforward to modify the fragments that accumulate photon contributions to call this kernel to compute a weight for each photon. This fragment replaces the fragment of the same name defined on page 794. The modified version of ⟨*Compute exitant radiance from photons for diffuse surface*⟩ is analogous and isn't shown here.

⟨*Compute exitant radiance from photons for glossy surface*⟩≡ used: none

```
for (int i = 0; i < nFound; ++i) {
    const Photon *p = photons[i].photon;
    BxDFType flag = Dot(Nf, p->wi) > 0.f ?
        BSDF_ALL_REFLECTION : BSDF_ALL_TRANSMISSION;
    float k = kernel(p, isect.dg.p, maxDistSquared);
    L += (k / nPaths) * bsdf->f(wo, p->wi, flag) * p->alpha;
}
```

Exercises

0.1 Another approach for shooting photons that can *guarantee* constant weights is to use rejection sampling. Show how to use rejection sampling for shooting photons from lights and scattering photons from surfaces when the sampling distribution doesn't exactly match the function being sampled in a way that guarantees constant weight photons. In what ways do the light source and BSDF sampling methods need to be modified for this approach?

0.2 Try other approaches for using photons for sampling directions for final gather rays–for example, are there cases where for example Jensen's original approach or Hey and Purgathofer's approach work much better than the one implemented here?

0.3 Modify the final gathering code to use Russian roulette to terminate rays with small contributions. Compute a tentative contribution for each ray before tracing it based on the value of the BSDF, $\cos\theta$, the PDF and multiple importance sampling weight terms. If this value is below a threshold, randomly terminate the ray. How much can this approach improve efficiency?

Bibliography

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

Hey, H. and W. Purgathofer (2002). Importance sampling with hemispherical particle footprints. In A. Chalmers (Ed.), *Proceedings of the 18th Spring Conference on Computer Graphics*.

Jensen, H. W. (1996, June). Global illumination using photon maps. In X. Pueyo and P. Schröder (Eds.), *Eurographics Rendering Workshop 1996*, New York City, NY, pp. 21–30. Eurographics: Springer Wien. ISBN 3-211-82883-4.

Jensen, H. W. (2001). *Realistic Image Synthesis Using Photon Mapping*. Natick, MA: A. K. Peters, Ltd.

Shirley, P., B. Wade, P. Hubbard, D. Zareski, B. Walter, and D. P. Greenberg (1995, June). Global illumination via density estimation. In *Eurographics Rendering Workshop 1995*, pp. 219–231.