15-462 Project 4: Photon Mapping

Release Date: Thursday, April 9 2009

Due Date: Thursday, April 30, 2009 at 23:59:59

Starter Code: http://www.cs.cmu.edu/~15462/proj/04/p4.tar.gz

1 Overview

In the third project, you learned about global illumination models and wrote a ray tracer that was able to render certain natural phenomena that could not be rendered (at least without much effort) with OpenGL. For this project, we will be taking your ray tracer and using it as the starting point for a basic photon mapper into order to achieve a caustic effect.

If you have not completed the minimum requirements for project 3, your first priority should be to finish those requirements. As mentioned during lecture, we will return half-credit to any requirements for project 3 that you complete for project 4.

Note that you **must** have ray casting, refraction, reflection, and color computation correct in order to see any results at all. Without these working correctly, it is very unlikely you will see reasonable results. However, you can still write the code for project 4 since we grade code more than results, but do not expect to get any results.

As in project 3, this assignment is code intensive and will require you to make design decisions about how you wish to code it. Since the textbook is unfortunately a poor resource for this assignment, you will want to use the slides from lecture as well as some resources that we provide you to give you more information on the topic.

The slides from Lecture 17 contain an overview of the material as well as some insight in the steps you will need to take to structure your code. The slides from Lecture 14 might be useful when thinking about your kd-tree. We also will provide some links to useful resources and papers at the end of this document.

1.1 Submission Process

Your handin directory may be found at /afs/cs.cmu.edu/academic/class/15462-s09-users/andrewid/p4/ You should submit the entire contents of the handout, your modified files, and any screen shots in this directory.

To test your code, we will simply `cd` into the directory, then run `make clean; make`. Therefore, your code should should be placed in the root directory uncompressed (i.e. the makefile's absolute path should be `.../15462-s09-users/`*andrewid*`/p3/Makefile`.). **DO NOT** submit a compressed file or binary files. So, if you made it on Windows, delete the `Debug` and `Release` folders, and if you made it on a Unix system, run `make clean` before submitting.

In addition, you must fill out the p4.txt file in the handout, describing which features you have implemented for project 4, and which you have not. In particular, if you have missing features, be sure to explain how much you accomplished and why. If you did extra work, be sure to explain what that was and how you did it. Furthermore, you should detail anything you think we need to know to understand your code.

Regardless of what machine you use to code your assignment, it must compile and run correctly on the machines in the Wean 5336 cluster. There are a lot of students in the course and only 25 machines, so you probably don't want to wait until the last minute to test it, even more so because you will want a substantial amount of time to render your scenes.

By university policy, we can only accept assignments through Friday, May 1, 17:00. Therefore, you may **not** use late days on project 4.

2 Handout/Requirements

2.1 Required Tasks

Please note that the purpose of this assignment is to try and achieve a particular effect. We are not grading you on how close your screen shots match those from the reference shots; due to the probabilistic nature of the photon mapping algorithm, your screen shots may appear to be different. The reference shots will provide a good reference about where the caustics should be located, but you do not need to be concerned about matching them perfectly. It's more about getting a rough approximation of the effect.

- Define photons
- Implement a kd-tree as your photon map and provide functionality to properly construct it and insert photons.
- Implement nearest-neighbor search for your photon map
- Fire photons at the `Scene::caustic_generator` object in your scene and store them in your photon map.
- Properly calculate the radiance estimate and recalculate color properly.
- Support scenes 0 and 1. Note that due to complexity, we do not require you support scene 2, the pool scene. But you may explore using photon mapping on it if you wish.

2.2 Extra Credit

See section 8.1 for information about possible extra-credit ideas.

2.3 Files to Update/Edit

The handout for project 3 includes the following files:

- `staffldr.cpp`

Note: You will probably want to create a new file for you photon mapping code that will hold any classes any functions you may need for photon mapping. You will likely also modify `raytrace.cpp` to integrate the photon mapper.

3 Photon Mapping

Before we begin, here is a quick refresher on the photon mapping algorithm and how our project uses it. Photon mapping is an alternative global illumination model that attempts to provide a solution to the rendering equation. Like ray tracing, photon mapping can be considered a "point-sampling" model, that is, colors are evaluated at points as opposed to surfaces. However, photon mapping provides computational speedups over other prominent point-sampling models such as path tracing and Metropolis Light Transport.

The general photon mapping approach is a two-pass algorithm. In the first pass, or photon tracing step, photons are emitted from all lights in the scene and are bounced around until they are eventually absorbed somewhere in the scene. Information about each photon is then stored in a map, the photon map, which is generally a spatial data structure optimized for fast insertions and lookups. Normally, there are two photon maps created: one for overall global illumination and one specifically for caustics.

In the second pass, color at each point is computed by breaking the rendering equation into terms: direct illumination, specular reflection, caustics, and indirect illumination. The summation of these terms returns the color at each point. The first two terms can be solved using just ray tracing. Caustics are handled by computing an estimate of the radiance at a given point using the caustics photon map. Likewise, indirect illumination can be calculated by computing another radiance estimate using the global photon map.

We are not requiring you to implement the full photon mapping algorithm. You are only implementing a rough simplication of it. We will be taking your ray tracer from project 3 and extending it to compute caustics for a single specular object per scene. Therefore, instead of two photon maps, you will be constructing only one photon map for caustics.

4 Photons

Photons are very similar to rays in behavior in that we will be firing them into our scene in the same manner we fired rays in the ray tracer. However, they carry different information. Whereas the point of eye rays are to return a color from the destination point, the point of photons are to carry a color to the destination point to later be used in a radiance estimate.

A photon is basically a unit of light at a given position. It is born at the position of the light and is fired into the scene where it goes through repeated bounces off any number of objects until some termination criteria is reached. For this project, you will be firing photons at specular objects and terminating them once they hit a diffuse surface.

We impose no restrictions on how you represent photons, though there are a few required elements and a few considerations to bear in mind. The need for each element will be explained in later sections. You will, at a minimum, need to store:

- The position of the photon (at its final destination point).
- The intensity of the photon as an RGB color.
- The incidence angle of the photon. It is the direction of the ray when the photon hits its destination surface.

Since you be storing at least several thousand in memory simultaneously, photons should be as compact as possible. You may also wish to consider storing them in a contiguous array rather than allocating each separately.

5 The Photon Map and Kd-Tree

5.1 Photon Map

Once we emit all our photons (see section 6), we store them in a map for retrieval during our radiance estimate (see section 7.1). This is called the photon map. Since it's holding several thousand or more photons, we want this to be as efficient a data structure as possible. The two operations we must support are insertion and retrieving the n nearest photons to any point, for any n.

5.2 The Kd-tree

The data structure we would like you to implement for this assignment is a kd-tree. Recall that a kd-tree is much better than either a grid or an octree because in general, our distribution of photons in our scene is not uniform (especially in the case for caustics). A kd-tree allows us to create a balanced tree to improve look up times.

A kd-tree is a binary tree that partitions space along each dimension (in our case, 3). A more detailed description can be found on Wikipedia at `http://en.wikipedia.org/wiki/Kd-tree`. The tree splits along the x, then y, then

z axes as you descend. Each node defines a splitting plane. At each node, all photons in the left child are to one side of the splitting plane, and all photons in the right child on the other. Since we split along the major axes, it means that, for example, all photons in the left child of the root have a x position less than the splitting value, and all children in the right child have x position greater than or equal to the splitting value.

5.3 Insertion and Balance

We want our tree to be as balanced as possible, meaning that the number of photons in each child are the same for all nodes (give or take 1 if the total is even). This minimizes the depth of the tree, improving search times and condensing the number of nodes needed. No matter how your tree construction works, we require that it be balanced.

Since we don't have to do a single lookup until all photons have been emitted, we can actually insert them all at the same time to make balancing simpler.

We can construct a balanced tree from all photons as follows. Beginning at the uncreated root node with the list of all photons L and the current axis as x, do the following recursion:

1. If L is empty, there is nothing to do, so return without making a node.
2. Sort L by position of the current axis (either x, y, or z). Find the median element of L.
3. The median element becomes the current node, with its value on the current axis as the splitting value.
4. Recurse on the left child using the left half of the list as L and the next axis ($x \to y$, $y \to z$, $z \to x$).
5. Recurse on the right child using the right half of the list as L and the next axis.

For sorting, look at `std::sort`. You can create a comparator and pass it into the `sort` function. Online documentation of the STL should be able to help.

5.4 Nearest-Neighbor Search

The lookup function we need is called nearest-neighbor search. Given a point p and $n \in \mathbb{N}$, it returns a list of the n photons nearest to p. The algorithm is as follows:

1. Create an empty list L of the n "current best." Whenever we encounter a photon, we can add it to L if L is not yet full or if the photon's position is closer to p then at least one photon in L. In that case, we remove the farthest photon from p to keep the list length n.
2. Starting with the root, move down the tree recursively in the same way as inserting a single point. That is, go left if you are less than the split value and right otherwise.

3. Once you reach a leaf node, store that node in L.
4. Unwind the recursion, doing the following at each node on the way up:
   (a) Add the current node to L if possible.
   (b) Check if the distance from p to the nearest point on the splitting plane is closer than the farthest photon in L. If this is case (or if L is not yet full), we must check the other branch. Otherwise, we can just skip the other branch and return. To check the other branch, you follow the same algorithm as the entire search by starting at step 2 with the current node as the "root."
5. Once we return from the unwinding of the root node, we return L.

Note that to avoid taking square roots, the algorithm generally uses squared distances, which are much more efficent to compute.

5.5 Suggestions and Alternative Approaches

This isn't the only way to make the kd-tree. As long as it is balanced and both operations run in a low complexity, you are free to make changes to this algorithm. Below are a few optional things you can do.

If you don't want to construct it all at the end, you can insert as you go, balancing the tree in-place periodically.

If you use node objects that are separate from the photons, you may wish to consider, as with photons, storing them in a contiguous array. The balanced nature of the tree allows you to do this efficiently, without even storing child pointers. You can use the heap structure, where the children of the node at index i are at $2i + 1$ and $2i + 2$.

6 Photon Tracing

Now that we have photons and a map to store them in, we need to actually fire them. Photon tracing is is the first pass in our photon mapping algorithm is done prior to ray tracing the scene. Photon tracing emits photons from all of the lights and bounces them around the scene, storing all the hits in the photon map.

6.1 Emission From Lights

6.1.1 General Case

The first step is determining how many photons to fire, and what the direction and color should be for each. In the most general case, this involves firing some large number of photons, say n, from each light in the scene.

Photon mapping is nondeterministic in that we fire off n photons in random directions from the light, spreading the light's intensity over each photon. We only deal with point lights, which means a photon starts at the light and travels

in any direction outward. So, given a light's color c, the color c_p of an individual photon will be

$$c_p = \frac{c}{n}. \tag{1}$$

6.1.2 A More Efficient Approach

However, there are many optimizations and simplifications we can make, especially for our more simplistic model. For your assignment, we only require you to fire and store a very specific subset of possible photon paths since we only care about caustics. Therefore, you should only fire the photons in your scene at a particular specular object, which we specify for you in each of the scene via the class member `Scene::caustic_generator`. You may fire at all the objects if you wish, but you will have to come up with a different approach then described here.

The goal is to fire as few photons as possible, but to fire enough so that we can get a good caustic effect. This means that we want to have as few wasted photons as possible. Obviously firing photons randomly around the scene is extremely wasteful, and so we want a better approach.

There are many different ways of going about this, but we describe a relatively simple approach that is sufficient for the purposes of this assignment. Note that it only works since we are firing at one object. You may another approach if you choose.

It involves projecting the bounding sphere of our object, O. A projection map is a two-dimensional map of the scene as seen from the eye source; think of it as the image plane as seen from the camera if the camera were at our light source. We can draw a bounding sphere around O. Then, we can project this sphere onto a plane based on the position of the light. We can then fire photons in the direction of our object by simply firing them at the circle which is the two-dimensional projection of our bounding sphere.

6.1.3 Computing Photon Direction

We need to compute the direction and color of each photon. First we consider the direction, $\vec{d}$. Let $\vec{c}$ be the center of the bounding sphere of radius r, and $\vec{p}$ the position of the light. So $\vec{c} - \vec{p}$ is the direction pointing right at the center of the bounding sphere.

The projection of the bounding sphere looks like a circle to the light, so we want to fire rays randomly within that circle. To do this, we want two orthogonal vectors $\vec{a}, \vec{b}$ that lie on this circle of length r. Then, any linear combination $\alpha\vec{a} + \beta\vec{b}$ where $0 \leq \alpha^2 + \beta^2 \leq 1$ will lie in the projection of the circle. So, given such $\vec{a}, \vec{b}, \alpha, \beta$, the direction of the photon would be

$$\vec{d} = \vec{c} + \alpha\vec{a} + \beta\vec{b} - \vec{p} \tag{2}$$

α and β should just be selected randomly to fire the rays randomly. But we also need $\vec{a}$ and $\vec{b}$. We note that both $\vec{a}$ and $\vec{b}$ are orthogonal to $\vec{c} - \vec{p}$, and so we just need to find one such vector, and then cross to get the second.

We don't care what vector we get, so need any vector $\vec{a}$ such that $\vec{a}\cdot(\vec{c}-\vec{p}) = 0$. So let us choose any $\vec{v} \neq \vec{c}-\vec{p}$. Consider $\vec{v} - x(\vec{c}-\vec{p})$, where $x \in \mathbb{R}$. For what values of x is this orthogonal to $\vec{c}-\vec{p}$? If you recall homework 1, the answer is

$$x = \frac{\vec{v}\cdot(\vec{c}-\vec{p})}{\|\vec{c}-\vec{p}\|^2}$$

and therefore, for any arbitrary $\vec{v}$, we can find our desired vector via

$$\vec{a} = \vec{v} - \frac{\vec{v}\cdot(\vec{c}-\vec{p})}{\|\vec{c}-\vec{p}\|^2}(\vec{c}-\vec{p}).$$

Consider why we need $\vec{v} \neq \vec{c}-\vec{p}$ for this to work correctly. Then, we can easily get $\vec{b}$ via

$$\vec{b} = \vec{a} \times (\vec{c}-\vec{p}).$$

Note both $\vec{a}$ and $\vec{b}$ must be of length r, so you should normalize them and scale by r. Then you can use equation (2) with randomly chosen α, β to compute directions for all your photons.

6.1.4 Computing Photon Color

Next we consider the color. Since we are only firing photons at a small subset of the region of the light, we must scale the intensities of our photons accordingly. We can approximate the flux that should go towards our object and divide it by the total flux to get this scaling factor.

We do this by considering a closed volume around the light, such as the unit sphere. We know the surface area is $4\pi r^2$, which is this case is just 4π. The area taken up by the projection of O can be approximated by computing the area of the circle of the projection. Using equation (2) to compute directions for the center point and a point on the edge, we can normalize and take the distance of the two vectors to get the radius r_1 of the bounding sphere projected onto our unit sphere. The area is of course πr_1^2, which means that our scaling factor f should be

$$f = \frac{r_1^2}{4}.$$

Due to our model's simplicity, we also must introduce a non-realistic intensity factor i for each light in order for caustics to be visible. The member is `Light::intensity`. The color should be scaled by i. So modifying equation (1), our final photon color is

$$c_p = \frac{cir_1^2}{4n}.$$

Note that since we care more about the method than the results, you may fiddle with the intensity when attempting to get results.

6.2 Russian Roulette

6.2.1 General Algorithm

We can actually reuse much of our ray casting functionality here. Firing a photon is no different from firing a ray with the same starting position and direction. You will ultimately want to find the closest point of intersection on the object or terminate if no intersection is found. You should be using the same ray casting, reflection, and refraction functionality that you used in project 3.

When a photon hits an object, one of several things can happen to it based on the material properties of the object. In the simplest case, it can either be reflected, transmitted, or absorbed. We model this iteration using a probabilistic method known as Russian Roulette.

Instead of modifying the intensity (color) or the photon as it bounces, we instead keep it the same, using probability to decide whether a photon should be reflected, transmitted, or absorbed. Russian Roulette provides a good way for us to reduce computation by firing fewer photons with higher intensities.

6.2.2 For Caustics

We will use a slightly dampened version of Russian Roulette, since we are only really concerned with firing photons at our caustic generating objects, all of which are dielectrics (at least the ones we have specified for you) or fully specular materials.

The caustics photon map is a special case for Russian Roulette. We want to emit photons and reflect/refract them off of specular objects until they hit a diffuse object. If we hit a diffuse object, we want to terminate and store (absorb) the photon in the photon map. If it hits the maximum recursion depth or leaves the scene without hitting a diffuse object, we just ignore the photon. As you can see, this ignores all diffuse reflections.

So if the photon hits a specular object, we will either reflect or refract our photon. As always, we use the Fresnel equations (or rather, the Schlick approximation) to compute the ratio of reflected rays, which we use as the probability. So given the Fresnel coefficient R of the surface, the probability that we reflect is R and the probability that we refract is $1 - R$. Therefore, if we generate some random number $x \in [0, 1]$, we reflect the ray if $x \in [0, R)$, else refract when $x \in [R, 1]$. Note that we do **not** modify the color of the photon. Of course, if our surface is opaque, or there is total internal reflection, we reflect with 100% probability

If the photon hits a diffuse surface, we take the intersection point as its position, the incoming direction as its incidence angle and we store it in the photon map. Note that you may assume surfaces are either completely diffuse or completely specular for this lab, though that is not true in general and the algorithm would need to be modified to take care of those cases.

7 Color Computation

7.1 The Radiance Estimate

The purpose of photon mapping is to be able to estimate the radiance at a point. For a given point in our scene, the radiance estimate is the estimate of how much light is at that point based on the information from our photon map.

Calculating the radiance estimate makes use of the nearest-neighbor search algorithm we wrote for our kd-tree. For the radiance estimate, we will look up the n nearest neighbors to our point for some fixed n. These constitute the closest photons of the point and will be used to estimate the radiance at our point.

We want to calculate the distance, r, to our furthest away of these photons. r is the radius for a sphere that encompasses all of our n nearest photons. The radiance estimate is now the sum of the contributions of all of our n photons divided by the area that they are encompassed in, πr^2. Note that this assumes that the photons lie on a circle rather than a sphere. This is because we are making the assumption that all the photons come from one surface. This is of course incorrect, and there are ways to deal with this, but they are not required for this project.

We must also take into account the incidence angle, much as we do for other computations involving diffuse surfaces. The contribution from a single photon p with color c_p and incidence angle ω_p is

$$c_p(n \cdot \omega_p)$$

where n is the normal of the surface. So the final radiance estimate c over a set of n photons P with radius r is

$$c = \frac{1}{\pi r^2} \sum_{p \in P} c_p \max\{(n \cdot \omega_p), 0\}$$

One question is what values of n are good to use in our radiance estimate. The answer is that it really depends on the scene, but a good heuristic to follow is that you should scale your radiance estimate based on the number of photons you fire. That is, the more photons you fire, the higher the number used in the radiance estimate can be. The higher both of these are, the more accurate our image becomes.

7.2 The Color Components

We are dealing only with diffuse surfaces, since caustics don't affect the specular surfaces. We inherit two of our components, direct illumination and reflection, directly from the ray tracer. Those components should be unchanged. The only addition is the new caustics component.

For a given point in our scene, the contribution from caustics is just the radiance estimate from our caustics photon map. So we look up the radiance

estimate and add it to our color, first multiplying it by the diffuse material and texture color.

Note that if you also wanted to do indirect illumination, it would be added in much the same way as caustics.

7.3 Fine Tweaking

Even with proper color computation, much of the quality of your final images is heavily dependent on the number of photons that you emit and the number of photons that are used in the radiance estimate. You will probably have to tweak these numbers until your image produces the desired effects. One thing to note is that the number of photons in the radiance estimate should scale with the overall number of photons you emit from your light sources.

The reference shots, used 10000 photons with 100 in the radiance estimate.

8 Before You Start/Words of Advice

If you thoroughly read this writeup, then you will notice that we are not asking you to write the full photon mapping algorithm, or even a very physically accurate one.

The reason for this is that this project is much like project 3 in that we are asking you to not only code, but think about the design and structure of your implementation. There is a little less math to think through, but quite a bit of logic behind making your implementation run well. Debugging is also a part of this assignment and you will need to allow time for this. So please remember to try and start early on this assignment to give yourself time to think through it!

Rendering scenes will certainly take much longer than before, since we are both constructing a very large data structure and making a large number of queries on this data structure in addition to our ray tracer computation from before. As a reference, the staff solution takes on the order of 10 to 100 times longer to render than the project 3 solution code. There are more students than computers, so you will want to schedule your render time carefully.

Lastly, please remember to comment your code thoroughly. We are interested in your final results, but we are more interested in your actual implementation of them. Style and code organization is important and commenting your code well is critical.

8.1 Extra Credit

Any improvements/optimizations to the photon mapper above the minimum requirements can be cause for extra credit. Some possibilities are:

- Add an additional photon map (the global photon map) for global illumination in order to handle indirect illumination (notably color bleeding).
- Create sharper, cleaner caustics by filtering them on the radiance estimate using a cone or Gaussian filter.

- Implement subsurface scattering or volume caustics (note: this is way beyond the scope of this project and will probably require you to very heavily modify your code structure).

8.2 Additional Resources

Some good resources to check out would be the following list of publications by Dr. Henrik Wann Jensen, the father of photon mapping, which can be found at http://graphics.ucsd.edu/~henrik/papers/. We recommend "Global Illumination Using Photon Maps" and "A Practical Guide to Global Illumination using Photon Mapping." Information on how to implement the extra credit suggestions is also contained among his many papers.