15-462 Project 4: Photon Mapping (Global Illumination)

Release Date: Friday, October 25, 2013

Due Date: Tuesday, November 12, 2013, 23:59:59

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 caustics and diffuse inter-reflection.

If you have not completed the minimum requirements for project 3, your first priority should be to finish those requirements.

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.

1 Submission Process and Handin Instructions

Failure to follow submission instructions will negatively impact your grade.

1. Your handin directory may be found at one of the two directories below, depending on which class you are enrolled in. Graduate students should be enrolled in 15-662 and undergraduates in 15-462.

   `/afs/cs.cmu.edu/academic/class/15462-s15-users/`*andrewid*`/p4/`.

   `/afs/cs.cmu.edu/academic/class/15662-s15-users/`*andrewid*`/p4/`.

   All your files should be placed here. Please make sure you have a directory and are able to write to it well before the deadline; we are not responsible if you wait until 10 minutes before the deadline and run into trouble. Also, remember that you must run `aklog cs.cmu.edu` every time you login in order to read from/write to your submission directory.

2. You should submit all files needed to build your project, as well as any textures, models, shaders, or screenshots that you used or created. Your deliverables include:

   - `src/` folder with all `.cpp` and `.hpp` files.
   - CMake build system.
   - `writeup.txt`
   - Any models/textures/shaders needed to run your code.

3. Please **do not** include:

   - Anything in the `build/` folder
   - Executable files
   - Any other binary or intermediate files generated in the build process.

4. Do not add levels of indirection when submitting. For example, your source directory should be at `.../`*andrewid*`/p4/src`, **not** `.../`*andrewid*`/p4/myproj/src` or `.../`*andrewid*`/p4/p4.tar.gz`. Please use the same arrangement as the handout.

5. We will enter your handin directory, and run `cd build && rm -rf * && cmake ../src && make`, and it should build correctly. **The code must compile and run on the GHC 5xxx cluster machines**. Be sure to check to make sure you submit all files and that it builds correctly.

6. The submission folder will be locked at the deadline. There are separate folders for late handins, one for each day. For example, if using one late day, submit to `.../`*andrewid*`/p4-late1/`. These will be locked in turn on each subsequent late day.

2 Checkpoint

For the checkpoint, you should have some sort of model/ray intersection test optimization. You should also shoot photons, but you do not need to organize them into a k-d tree, perform KNN, or do a radiance estimate. You can test the photon shooting by pressing the `y` key. This will render your photons with as points in OpenGL, with the proper color.

3 Required Tasks

The purpose of this assignment is to achieve a particular effect. We will not grade you based on how close your screen shots match the reference solution. The probabilistic nature of this assignment means that your results may appear different from the reference images. In fact, if your code is too slow or uses too much memory, you may find your results very blotchy or noisy. We recommend simulating global illumination using photon mapping, but you are free to use an advanced unbiased technique such as Bidirectional Path Tracing, or Metropolis Light Transport if you wish.

The general pipeline for simulating photon maps is as follows:

- Fire photons from each light source at the scene and store them in the photon map.
- Organize the photon map into a KD-tree.
- Perform raytracing. Replace the ambient estimate with a radiance estimate using k-nearest neighbors from the photon map.

4 Photon Mapping Algorithm

Before we begin we will summarize 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. 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 construction and lookups.

In the second pass, color at each point is computed by breaking the rendering equation into terms: direct illumination, specular reflection, 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. Indirect illumination can be calculated by computing another radiance estimate using the global photon map.

5 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. The only difference is that 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 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.

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 surface normal at the point where the photon hit a surface

You may also want to store some information about the incoming direction of the photon, especially if you want to implement non-Lambertian surfaces.

Since you be storing at least several hundred million in memory simultaneously, photons should be as compact as possible. You will store them in a contiguous array rather than allocating each separately. A simple implementation can easily get the size down to 40 bytes, but 12 bytes is possible with minimal black magic. Note that a reasonably efficient implementation will be limited by RAM, not time.

Here are a few ideas on compression:

- Eliminate unnecessary fields. For Lambertian surfaces, the incoming direction does not need to be stored. Note that this does not mean you can ignore this direction, but rather that you can perform all relevant computations before storing.
- Use compressed representations for numbers, specialized for the individual field:
  - Normal vectors can be represented by spherical coordinates, and do not need to be particularly precise.
  - It is relatively simple to determine an axis-aligned bounding box for the entire scene. Note that all position vectors will be contained in this box.
  - One color coordinate will be 1.0, and the other two will be less than 1.0.
  - If the k-d Tree is sufficiently balanced, there is no need for each node in the k-d Tree to contain left and right child pointers.
- Ensure that the structure does not contain any padding. You may want to manually unpack classes and reorganize the fields.

6 Photon Map and k-d Tree

Once we emit all our photons, we store them in a map for retrieval during our radiance estimate.

This is called the photon map. Since it's holding several hundred million or more photons, we want this to be as efficient a data structure as possible. The two operations we must support are construction and retrieving the k nearest photons to any point, for some fixed k (e.g. 100 or 1000).

6.1 k-d Tree

The data structure we would like you to implement for this assignment is a k-d tree. Recall that a k-d tree is much better than either a uniform grid or an oc-tree because in general our distribution of photons in our scene is not uniform (especially in the case for caustics). A k-d tree allows us to create a balanced tree to improve look up times. A k-d tree is a binary tree that partitions space along each dimension (in our case, 3). A more detailed description can be found at http://www.ri.cmu.edu/pub_files/pub1/moore_andrew_1991_1/moore_andrew_1991_1.pdf.

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.

6.2 Construction

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 Use quickselect on L to find the median element along the current axis. Note that quickselect places elements smaller than the median before the median and elements larger than the median after the median.

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 ($x \to y, y \to z, z \to x$). You may wish to consider a more careful selection of splitting axis.

5 Recurse on the right child using the right half of the list as L and the next axis.

For quickselect, look at `math/quickselect.hpp`. The `PhotonAxis` class in `p3/photon.hpp` may be useful as a comparator.

6.3 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 collection of the n photons nearest to p. The algorithm is as follows:

1 Create an empty collection[1] 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. 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.

[1] Use the appropriate data structure, not a sorted array.

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.

7 Photon Tracing

Now that we have photons and a map to store them in, we need to actually fire them. Photon tracing 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.

7.1 Emission from lights

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 non-deterministic 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 spherical lights, so in order to pick a direction our photons should travel from a random point on the surface of a sphere in a random direction. Recall that to pick p, a random point on the surface of a unit sphere we let p be defined as:

$$p = \frac{1}{\sqrt{x^2 + y^2 + z^2}} \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \text{ where } x, y, z \sim \mathcal{N}(0, 1)$$

Select a random start position on the surface of the sphere, and then choose another random direction to shoot the ray.

7.2 Intersections

We must now define how photons are traced through the scene. Photons are very similar to rays and so you will be able to use the your trace function with some modification. The difficulty is that we want to ensure that each photon has the same intensity (in the maximum color component). So after each bounce, we use a Russian Roulette method to ensure this invariant. We divide the photon's color by the maximum component, and then continue bouncing with probability equal to the maximum component.

There are two places where we must modify the code for intersections. The first is when hitting diffuse objects (objects with a diffuse term greater than 0). Instead of simply stopping when we hit a diffuse point and returning the direct illumination component like you did in Project 3, we must continue to trace the photon through the scene. Since we consider a diffuse surface to be perfectly diffuse we will reflect the photon in a random direction along the sphere defined at the normal. One simple way to do this is as follows:

```
Vector3 uniformSampleSphere(const Vector3& normal){
  return normalize(uniformPointOnSphere()+normal);
}
```

The color of the photon must also be modified. To do this, simply multiply the color of the photon by the direct component at the surface.

We must also modify the code for dielectrics for the sake of efficiency. We want to emit photons and reflect/refract them off of refractive objects objects. So if the photon hits a refractive 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 uniform 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.

Photons hitting specular objects have their color changed by the specular color and texture.

One more concern is when to stop following a photon. You have a few options. One option is to add the photon to the photon map every time you hit a diffuse object and simply stop tracing once you hit the max recursion depth. This is sufficient for this assignment.

The other option is to trace the photon and every time it intersects a diffuse object you stop tracing the photon with probability p, which you can define to be a constant or a probability based on direct lighting component at that point. Do not store photons which do not intersect any objects.

8 Color Computation

8.1 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 (which we will get from the second pass where we ray trace and intersect an object at a point), 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 k-d 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 negated direction of the photon 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

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

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.

8.2 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 global component.

The global component is just the radiance estimate from the global 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.

8.3 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.

9 Extra Credit

There is much scope for extra credit in this project and you are encouraged to experiment by adding support for more advanced surfaces, more advanced rendering techniques etc. Below are some ideas:

- **Progressive Photon Mapping** PPM is a technique that gets around some of the problems of photon mapping, namely the need to store all photons.

  A description: http://cs.au.dk/~toshiya/ppm.pdf

- **Metropolis Light Transport** Instead of tracing photons through the scene, we can instead extend the idea of ray tracing to sample more possible light paths. This is a nice idea in theory, but the light integral we are trying to approximate is very high dimensional and so it takes a long time to get a low variance image. Metropolis Light Transport is a technique for more intelligently sampling all of the possible light rays in a scene.

  A description: https://graphics.stanford.edu/papers/metro/metro.pdf

- **Subsurface scattering**. Implement subsurface scattering or volume caustics.

10 Words of Advice

10.1 Programming Hints

Since raytracing takes a lot of time, paying attention to writing efficient code is important. Of course, efficiency is most certainly not the most important consideration. Correctness, maintainability and good code organization are your most important concerns. However, you should avoid writing obviously unnecessarily slow code. Here are a few hints:

- **Do not** allocate memory in performance sensitive areas. Memory allocation is really, really slow. Do any necessarily allocations in an initialization step, or, even better, use the stack or add members to already-allocated structs or classes to avoid additional allocations at all.

- Avoid trigonometric and square root functions when you can do without them, as they are rather expensive. Note that a lot of vector operations such as normalization, magnitude, and distance use square root, so use squared magnitude and squared distance where possible, and avoid normalizing vectors unnecessarily. Of course, a lot of algorithms require unit-length vectors, so only avoid it when possible.

- Avoid virtual functions if a non-virtual function will suffice, since virtual functions are more expensive to call. Note that this does *not* mean to use switch statements or casting instead of virtual functions, but rather, don't make a function virtual if you can leave it non-virtual.

Some more general programming hints:

- We provide a lot of useful starter code for you, so you don't have to bother writing a lot of basic routines. Take a look at the headers, for if you need some basic vector or matrix operation, it is likely already there.
- If you use Windows to implement the project, be sure to test on the Linux machines. The compilers are not quite the same, and certain things that compile with MSVC do not compile or behave differently with GCC.
- `abs` is a function for integer types. You almost always want `fabs`, which is for floating point types.