15-462 Project 4: Global Illumination

Release Date: Monday, October 27, 2014

Due Date: Monday, November 17, 2014, 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 fist priority should be to finish those requirements. Note: If you would prefer you may contact us and we will give you solution code for P3 that you may use as a starting point.

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-f14-users/`*andrewid*`/p4/`.

   `/afs/cs.cmu.edu/academic/class/15662-f14-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 install`, 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 Required Tasks

The purpose of this assignment is to achieve a particular effect, namely *caustics* and *color bleeding*. A portion of your grade will be determined by whether or not you achieve these affects; the rest is based on the quality of your implementation and the resulting images. However, your images do not need to exactly match the reference shots, since the algorithms involved are probabilistic in nature. 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 which we expect you to implement for photon mapping includes:

- Defining photons and dividing light between emitted photons
- Implementing a KD-tree to store absorbed photons
- Implementing nearest-neighbor search for your photon map
- Fire photons from each light source, and collect absorbed photons into two maps, one for caustics and one for global illumination.
- Raytrace the scene, using your photon maps to compute the radiance estimate

3 Photon Mapping Algorithm

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 *photon map*, which is generally a spatial data structure optimized for fast insertions and lookups. Normally, there are two photon maps: one for overall global illumination and one specifically for caustics. We separate the two because we make different assumptions about the properties of caustics, versus diffuse inter-reflection.

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.

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. 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 incidence angle of the photon, or the direction the photon is traveling when it hits the destination surface

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

5 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 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.

5.1 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 the distribution of photons in the scene is not uniform (especially in the case for caustics). A k-d tree is a binary tree that partitions space along each dimension (in our case, 3). By carefully partitioning each dimension, we can create a *balanced* tree, which improves lookup times. 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. 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.

5.2 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 collect all photons in a list, and then create a balanced tree by the following recursive process. Beginning at the root node of a new tree with the list of all photons L and the current axis as x:

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 Store the median element in 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; see the online STL documentation. A good data structures programmer would note that because the tree is balanced, it can be implemented entirely inside an array of photons, with no pointers or auxiliary data, using binary heap layout. While not required, you may want to try a similar layout to avoid allocating individual tree nodes with child pointers.

5.3 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 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 than 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 as if you were inserting p. That is, go left if p is less than the split value (for the current dimension) and right otherwise.

3 Once you reach a leaf node, store that photon 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 L is full *and* the distance from p to the nearest point on the splitting plane is *greater* than the farthest photon in L. In this case, all photons on the other side of the splitting plane are farther than the n photons already collected, so we can simply return. Otherwise we must check the other branch, using the same recursive algorithm with our current L.

5 Once we return from the unwinding of the root node, we return L.

6 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, and is usually performed 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. Since this information is independent of camera position, you can compute the photon map once, and re-use it for any viewpoint (as long as the lights and objects do not move).

6.1 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 in we pick a random point on the surface of the sphere and set that as the start position and the direction. Recall that to pick p, a random point on the surface of a unit sphere we let p be defined as:

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

In order to generate an initial ray for each photon let the ray be `Ray(p + light_position, p)`.

6.2 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 vast majority of your trace function without much modification. In fact you may wish to pass in a flag to say whether or not you are tracing a photon or a ray; however the code for intersecting photons with objects will be slightly different.

For dielectrics, you should use Russian Roulette sampling to decide in which direction to continue tracing the photon. Transparent objects in our scenes have a reflective and refractive component; the contribution of each is determined by the Fresnel coefficient. For photons, rather than tracing both, you should choose only one (this is more efficient and maintains the distribution of light paths) with probability determined by the Fresnel coefficient. A simple way to do this is to generate a random number $x \in [0, 1)$; if $x < R$ where R is the Fresnel coefficient, you should reflect the photon, otherwise use refraction. This ensures a probability R or reflection and $1 - R$ of refraction, as desired. For perfectly transparent objects, you do not need to modify the color of the photon; simply continue tracing in the chosen direction.

For diffuse objects, Russian Roulette sampling is slightly more complicated. Photons should either be absorbed (stop tracing), perform a diffuse bounce, or perform a specular bounce, in accordance with the material properties. A standard technique is to compute a probability for each case from the diffuse and specular albedos; see the lecture slides and accompanying SIGGRAPH course notes for details. For a specular bounce, the new direction is the reflected direction; for a diffuse bounce, you should randomly select a direction from the hemisphere around the surface normal. This supports the model of materials as perfectly Lambertian - the incoming light is scattered equally in all directions. One simple way to choose a uniform random direction is as follows:

```
Vector3 uniformSampleHemiSphere(const Vector3& normal)
{
  Vector3 newDir = uniformPointOnSphere();
  if (dot(newDir, normal) < 0.0) newDir = -newDir;
  return newDir;
}
```

In both cases, you should multiply by the texture color before continuing. Note that you do not need to multiply by the specular color for reflective bounces (the distribution of light energy will be correct due to Russian Roulette), or by the cosine of the incident angle for diffuse bounces.

At each diffuse intersection point, you should also store the photon in your photon map. The choice of which map to insert into is determined by the previous bounce. If the previous intersection was

a specular bounce (reflection or refraction, including reflection off of an opaque object) you should add the photon to the caustic map. Otherwise (if it was a diffuse bounce) you should add the photon to the global illumination map. This separation is motivated by the fact that reflections/refractions tend to concentrate photons from many incoming directions onto a small point (caustics) while diffuse inter-reflection naturally scatters photons across the scene. Photons should not be stored on the first intersection after leaving the light source; these paths will be handled by direct lighting in the raytracing part.

Lastly, you must ensure that tracing photons eventually terminates; this may be a simple maximum-depth in your recursion.

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 p in our scene (which we will get from the second pass where we ray trace and intersect an opaque object at a point), the radiance estimate is the estimate of how much light reaches p 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. To perfectly simulate the incoming light, we would like to have all the incoming rays which intersect the surface exactly at p. Instead, we will approximate this by collecting the n nearest photons to p; we account for the error introduced from the photons not intersecting exactly at p by dividing the incoming intensity over the area of a circle centered at p which encompasses all the photons. Given n photons, let r be the distance to the farthest photon from p. The contribution from a single photon with incoming color c_i and negated incoming direction ω_i is $c_i \max(N \cdot \omega_i, 0)$ where N is the normal to the surface at p, accounting for the incoming angle in a standard Lambertian model. Then the final radiance estimate is

$$\frac{1}{\pi r^2} \sum_{i \in [0,n)} c_i \max(N \cdot \omega_i, 0)$$

One caveat to the above is that we want n photons which lie in a *disc* around p; this approximates the light which hits the surface close to p, which is why we divide by the area of the surface hit. If you simply implement nearest-neighbor search as described previously, your search will return the n nearest photons in a *sphere* around p, which can yield artifacts along borders between surfaces at angles to each other. You will need to augment your search to exclude photons which do not lie on-or-near the plane defined by p and the normal of the surface at p.

Another question is what value of n to use in the 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

While raytracing, intersections with transparent and reflective objects should be computed as in p3.

For opaque 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 and the global component. The contribution of each is just the radiance estimate from the corresponding photon map. You simply need to multiply each of these values by the texture color and add it to the final result.

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 100000 photons in both maps with 500 in the radiance estimate.

8 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.

9 Words of Advice

9.1 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.

10 Appendix

Reference images, with global illumination.

Notice the caustics in Figure 1 and Figure 2. This is an effect Photon Mapping is particularly good at capturing.

Also notice the diffuse inter-reflection between the colored and white walls of the cornell box, and the visible color in shadowed areas.

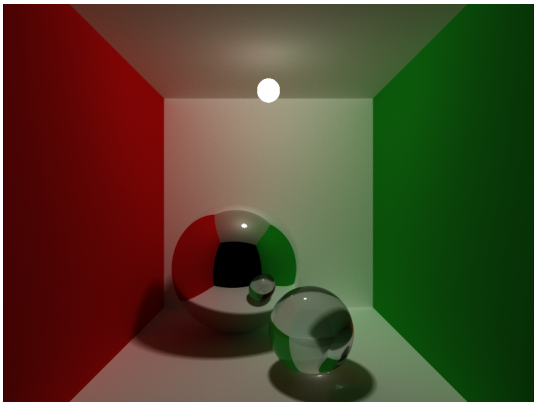

Figure 1: Cornell Box scene

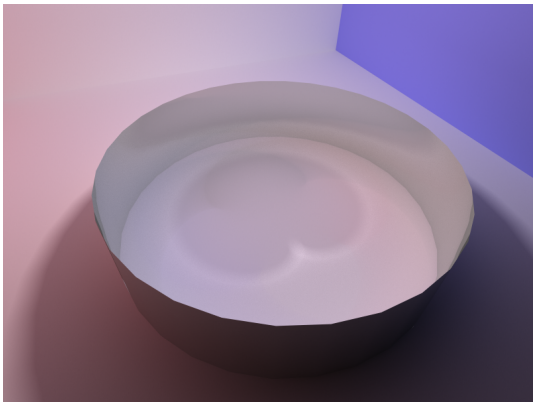

Figure 2: Ring scene

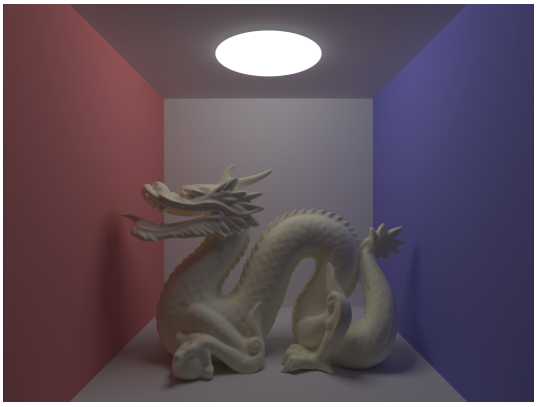

Figure 3: Dragon scene

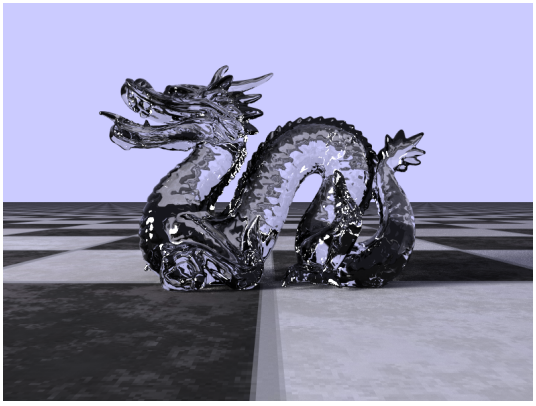

Figure 4: Glass dragon