15-462 Project 4: Path Tracing

Release Date: Thursday, March 27, 2014
Due Date: Tuesday, April 15, 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 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. 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-s14-users/andrewid/p4/.
   /afs/cs.cmu.edu/academic/class/15662-s14-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 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 slightly different from the reference images. We require that you simulate global illumination using Bidirectional Path Tracing, but you are free to explore other options such as photon mapping, or Metropolis Light Transport if you wish. To improve the efficiency of intersection tests, the second part of the assignment requires you to implement a spacial data structure: either BVH, or k-d trees. If you already implemented BVH in p3, you would need to implement k-d trees to receive credit for this assignment (and vice versa).

The two parts of the assignment are quite independent from each other, though you would benefit from the gain in speed if you choose to implement the spatial data structures first. The first part of the assignment can be achieved in three steps.

- Path tracing in one direction (starting from the eye)
- Direct sampling
- Bidirectional path tracing

3 Path Tracing and Direct Sampling

Path tracing is a solution to approximating global illumination. Similar to ray tracing, it also involves tracing rays from the camera into the scene.

The difference lies in the computation of the ambient term. Instead of assuming a constant ambient term in the scene, we would like to compute it by bouncing the ray between diffuse surfaces. For the purpose of this project, you can assume that this is not transmissive.

At the point of intersection between a ray and an opaque geometry, in addition to computing the direct illumination component (i.e. computing the shadow rays by sampling light sources) and the specular component, you would also need to calculate the light contributed by the environment. This is done by casting another ray in a cosine weighted direction on the normal hemisphere, which will be discussed in the ensuing section.

4 Cosine weighted sampling

Define the normal hemisphere as one centered at the intersection point, and extruding in the direction of the surface normal.
In order to sample the direction of the recursive ray, we compute a point on the normal hemisphere that is more likely to be close to the normal than far from it.

Intuitively, this is motivated by the heuristic that a light ray is more likely to be reflected in a visible direction. If we define all points on a unit disc on the surface of the geometry as equally visible, when we project these visible points onto the normal hemisphere, there would be more of these points closer to the normal (Feel free to research more on the monte carlo sampling methods).

We provided the code for generating a random direction towards a random point on the cosine weighted hemisphere (given the normal) in `cos_weighted_hemi(normal)`.

5 Bidirectional Path Tracing

As your intuition might warn you, it can be unlikely for a ray casted from the eye to eventually reach a light source. As a result, the initially rendered scene could contain a lot of noise (dark pixels).

If you are patient, the scene eventually becomes less noisy as the color of each pixel averages out.

As a computer scientist who prefers more efficient algorithms, in this section, you are going to optimize the situation.

To reduce the noise, we cast a sub-ray from both the light source (the shooting ray), and the eye (the gathering ray). After a certain number of bounces on both ends, we join the two intersection points to complete the path.

The subpath lengths (i.e. number of bounces) for the shooting ray and the gathering ray should be determined by a form of russian roulette. To bound the total path length, you can set this to be two random numbers between 0 and some maximum subpath length.

Note that the method for the computation of color is unchanged, we only determine the direction of the path in advance.
6 Spatial Data Structures

Please implement one of the data structures that you did not implement for p3.

In this section, we are essentially “wrapping things that are hard to check for intersection in things that are easy to check.” (Lecture 12)

We can preprocess the geometries in the scene by wrapping each geometry in a bounding box, or a bounding sphere. Since the bounding volume encapsulates the geometry, if a ray does not intersect with it, the ray would not intersect the geometries in it. In this way, some geometries can be ignored from intersection tests.

6.1 \( k\)-d Tree

A \( k\)-d tree is much better than either a uniform grid or an oc-tree because in general our distribution of geometries in our scene is not uniform. 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.

At each node, all geometries in the left child are to one side of the splitting plane, and all geometries in the right child on the other. Since we split along the major axes, it means that, for example, all geometries 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.

For simplicity’s sake, if a geometry is split by the splitting plane, we can duplicate it in both child nodes.

6.2 Bounding Volume Hierarchies

Each node in the bvh tree denotes a bounding box for the geometries contained within. The root node is thus the box for all the geometries in the scene, and the leaf node the box for an individual geometry. We apply the Surface Area Heuristic (SAH, discussed later) to split the parent tree node into child nodes. The splitting terminates when only one geometry is contained in a bounding box. At a parent
node, the number of geometries in the left child is the same as that in the right.

When doing intersection tests, we intersect the ray with the bvh of the whole scene and traverse the tree to determine intersections with geometries.

### 6.3 Construction With Surface Area Heuristic

In both k-d tree and bvh, we need to split the bounding volumes by one of the x,y or z planes. We chose the axis that yields the smallest total surface area of the resulting bounding boxes.

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. Alternatively, you can also use `std::partition`.

### 7 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** Explore photon mapping and progressive photon mapping as an alternative solution to global illumination.

- **Metropolis Light Transport** We can 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.

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

### 8 Words of Advice

#### 8.1 Programming Hints

Since path tracing may take 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:

- **Pay attention to code reuse.** Not only can spatial data structures be constructed for the whole scene, they can also be used on complex geometries (i.e. meshes). Can you reuse some of the code to construct a spatial data structure for each individual mesh?

- **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.
9 Appendix

Reference images, with path tracing.

Figure 3: Cornell Box scene

Figure 4: Ring scene

Figure 5: Dragon scene

Figure 6: Glass dragon