

CS 354, Fall 2012
Project 6: Ray tracer
Due: Dec 7, 11:59pm

Introduction

In this assignment, you will flesh out a basic raytracer to make it capable of rendering images with lighting, reflection, refraction, and shadows.

1 Running the code

Build using the command `make -f Makefile.campus`.

Run using the command `./ray`

2 Requirements

A ray tracer is a complicated piece of software, so we've provided starter code that takes care of a significant amount of the work for you. When you build and run the project, you'll get a full GUI-based system that allows you to load and render scene files. Sample scene files are downloadable from the class webpage, where there is also a link to a description for the file format. Data structures for all the scene data are already in there, and are filled out by the scene loader for you. Camera ray generation is already set up, as is intersection with boxes, cones, cylinders, and spheres. The lighting model is stubbed out to just return a color. This brings down your list of requirements to the following:

- Implement the Blinn-Phong lighting model, including ambient, diffuse, and specular terms. All the inputs needed for the model have been provided, you just need to compute the result correctly. Your modifications should go in `scene/material.cpp`.
- Implement point light distance attenuation, using the reciprocal constant / linear / quadratic formulation. This goes in `scene/light.cpp`.
- Implement reflection and refraction. This includes generating the new rays, casting them into the scene, and summing in their results. This mostly goes in `RayTracer.cpp`.

- Implement shadows, which has many of the same requirements as above.
- Implement intersection for triangle meshes, including Phong interpolation of normals. This mostly goes in `SceneObjects/trimesh.cpp`.
Caveat 1: Trimesh materials be specified in the triangle, the mesh, or the scene. Be aware of this - if you do not pay attention to it your code can end up with a segmentation fault.
Caveat 2: Normals may be specified in the mesh or they might not. Make sure to check for this and use the right normal in each case.

3 Bells & whistles

If you made it through all that, you've finished the project. No additional features are required, however, there are many ways to make it more awesome if you're so inclined. If you impress the grader, it's possible to earn a good amount of extra credit! Here are some suggestions:

- **Spatial data structure:** As you'll see, a basic raytracer is unbearably slow. The way to make it faster is to implement a spatial data structure, which can quickly cull objects a given ray is not likely to hit instead of linearly searching through them all. Examples include spatial grids, BSP trees, and kd-trees.
- **Texture mapping:** Implement texture mapping. Most of the code for this (including the texture loader) is already included in your project, but you will need to implement your own bilinear interpolation.
- **Anti-aliasing:** Try firing several rays per pixel and averaging the results to implement anti-aliasing. Just four rays makes a huge difference in image quality, but even that takes four times as long to render.
- **Lighting effects:** Add normal mapping, bump mapping, Cook-Torrance lighting, or any number of other effects to your raytracer. There's a virtually unlimited number of ways to change your rendering.

4 Screenshot

10% of your grade in this project is a screenshot that you submit along with your code. Generate the best image you can (a custom scene file is recommended). Call this screenshot **original.png**.

Skeleton code

Starter code has been provided on the class website. It's quite extensive; it sets up a GUI using the FLTK library and provides you with the bulk of the framework necessary to set up your raytracer. It is not based on the usual GLUT / GLM setup that you're used to, so be prepared to spend a

little bit of time figuring out its matrix library and so forth. The best feature is the embedded debugger, which will render the scene from any arbitrary view using OpenGL. If you click on the raytraced image, the ray fired from that pixel will be shown in the debugging view, allowing you to much more easily debug your ray casting. There's also a directory of scene files available for you to test your raytracer with.

We only can really guarantee that this code works on the department Linux machines, although we have taken steps to ensure that it builds on Windows and Mac systems as well, as long as you have the FLTK libraries available. Caveat emptor.

Logistics

You may work in a group of up to 2 people for this project. You may work alone if you prefer.

In case you needed a reminder: **START EARLY**. If you manage your time badly, you will have a serious problem.

Turnin and grading

You can develop your application on whatever operating system you like, but before you submit it, you **must** make sure that it builds and runs properly on the department Linux machines! All grading will take place on these machines, so if your code doesn't work on them, you're in trouble.

Grading breakdown:

- 15% Phong lighting model
- 10% Point light distance attenuation
- 15% Reflection
- 15% Refraction
- 10% Shadows
- 15% Triangle meshes
- 10% Screenshot
- 10% Code quality

Your code will NOT be checked with `check-code` (`cpp lint`). But it will be visually inspected using the same criteria as always.

To submit your code, use the department's turnin script, like so:

```
turnin --submit edwardsj project6 raytracer/
```

Replace `raytracer/` with whatever your code directory is named. Make sure that *all* the necessary code is submitted. On the other hand, make sure that you're including just the files for this project, and not several hundred megs of extra junk.

Make sure you have included your (and your partner's) name and UTCS ID in `README.txt`, which should also explain extra features of your program.