CS 465 Program 1: Ray I

out: 1 September 2004
due: 14 September 2004

1 Introduction

For this assignment, you will be writing a raytracer that will work with basic shapes, lights, and materials. We have developed a framework for you to use, to save you from having to muck around with details that have little to do with graphics, like I/O and file parsing. If you find the framework cumbersome, feel free to change it. In fact, we have tried to give you a relatively minimal framework. Chapter 9 of the Shirley book, and in particular section 9.4, should be a great reference for doing this assignment.

2 Assignment Overview

Ray tracing is, in some sense, the most natural way to generate images. By generating and casting rays, the ray tracer is able determine what color we would see if we looked in some direction from some point of view. The ray tracer you'll write for this assignment is very simple, containing just the key elements regarded to be a ray tracer.

In the end, your ray tracer will have support for:

- Spheres
- Triangles
- Lambertian materials
- Phong materials
- Point lights
- A simple camera model
- Basic gamma correction

We have written a small amount of code to spare you from the mundane details of I/O, vector arithmetic, etc. Also, since we'll examine cameras in detail later, we're providing a simple camera. You will need to code all of the methods of the ray tracer yourself. You should feel free to create whatever new classes you think are necessary, change parameter lists, etc. In the end, if your program works, and is understandable, you will receive full credit. Be aware, though, that while there is no one "right" way to do this assignment, there are many incorrect ways to do it.

3 Requirements

1. You need to use a ray tracing algorithm. This isn't going to be defined explicitly, but if you are unsure about your algorithm ask one of the TA's. A general guideline can be found in Shirley and in the lecture notes.
2. You need to support spheres and triangles. Specifically, you need to be able to intersect a `Ray` with a `Sphere` or `Triangle`.
3. You must support a Lambertian material. This is defined in Shirley and in the lecture notes.
4. You must support a Phong material. This is also defined in Shirley and in the lecture notes.
5. The only lights you need to use are point lights. There is no r^2 falloff.
6. You must gamma correct your images for a display device of $\gamma = 2.2$.
7. You do not need to worry about malformed input, either syntactically or semantically. For instance, you will not be given a sphere with a negative radius or a scene without a camera.
8. You do not have to worry about the camera being inside a sphere.
9. You **do** have to worry about hitting the back side of a triangle. The back of a triangle should be shaded the same as the front side.

4 Framework Information

The framework we have given you includes some of the classes you will need to finish this assignment. Here, we given an explanation for each of the classes as well as some hints about how each is to be used.

4.1 `Parser`

The `Parser` is the only class that we recommend you do not alter. If you add new shapes or materials, then clearly it will be necessary to change the parser, but this assignment does not require that. The parser is a simple top down parser, and the only methods that you should use are `parseFile(String)`, `parseFile(File)`, and `parseStream(InputStream)`. In fact, the `RayTracer` class already calls one of these methods.

4.2 `RayTracer`

The `RayTracer` is the entry point for the entire program and ties the system together. The `main` method has been written for you completely. It will treat all parameters as input files and attempt to parse them, render them and write the images to .png output files. PNG files should be relatively cross platform — they are natively supported in Windows XP as well as Mac OS X and *nix. This class will benefit the most from the pseudo-code in Shirley's book (hint hint).

4.3 `Surface`, `Sphere`, and `Triangle`

Using the power of polymorphism, we have created an abstract class called `Surface` to represent objects in the scene. The only concrete feature of a `Surface` is that it must have a reference to a `Material`. `Sphere`s and `Triangle`s are natural surfaces, and are some of the easiest to intersect with. These are all that you will be required to implement for this assignment. Information on how to determine an intersection with either object can again be found in Shirley.

4.4 `Material`, `Lambertian`, and `Phong`

We have created another type hierarchy in order to express different types of materials. `Lambertian` and `Phong` materials are the two simplest materials to render. Both materials should be implemented exactly as they are described in the Shirley book.

4.5 `Camera`

The `Camera` class represents the vantage point and orientation for viewing a `Scene`. The `getRay(int, int)` method is the key method for this class. It returns a `Ray` object starting at the camera's location and going through the indicated pixel. Pixels are ordered increasing from left to right and top to bottom (using the image convention, not the math convention). To keep things simple, the "up" vector is fixed at (0, 1, 0). This means the camera is unable to point directly down or up.

4.6 `Light`

The `Light` just has a location and power, and nothing else. Although physically implausible, this is the simplest type of light to use in a ray tracer.

4.7 `Scene`

This class, as it stands right now, serves mostly as a convenient return mechanism from the `Parser` class. It is simply a grouping of all the objects necessary to describe a particular input file.

4.8 `Ray`, `Vector3`, and `Color`

These utility classes are provided to you pretty much fully developed. The methods should be easy to understand from the comments. While none of the classes are necessarily full featured (or efficient) versions of what they could be, all have enough code in them to get by. Once again though, feel free to add/modify/remove as you see fit. For a simple ray tracer like this one, you can simplify things by keeping the direction vector of a `Ray` normalized.

4.9 Hints

- Start simply—you can get your first picture with just a working `RayTracer`, `Sphere`, and `Camera` class, as well as any necessary support classes that you have written.
- Do not worry too much about efficiency (just take a look at the `Vector3` class we supplied you with . . .), but a scene of half a dozen objects should take no more than a few seconds to a minute to render.
- Do not worry too much about correct object oriented design either. We will not be grading you on whether or not you have accessors/modifiers in place, but rather on whether or not your code works and your general algorithm is well thought out.
- Get a good IDE and step through code that is broken. We recommend Eclipse (its free, already installed in the lab, and fully featured), but any environment should do.
- Use very basic test scenes initially. We are supplying some simple test scenes, but coming up with your own is useful when hunting a specific bug.

5 Suggested Path

There are many many ways to write a ray tracer, and an open-ended task like this can seem daunting at first. To help you get started, we have a suggested path that you can take in order to complete the assignment. Please understand that this is **not** a requirement, just a suggestion.

5.1 Avoiding the black screen

The easiest and most frequently rendered image in computer graphics is the dreaded black screen. It is to graphics people what the blue screen was (is?) to Windows users. Traditionally, a lot of work was required in order to get off the ground with a ray tracer and actually start producing images. To avoid this unfortunate side effect, your first step should be to write a basic version of the `RayTracer#renderImage(Scene)` method. It should simply loop over all of the pixels, and test for the presence of an object through each pixel. Don't worry about shading or using materials yet; do something much simpler. If an object is hit, return white; otherwise, return black.

For this to work, you will need to code an intersection method too. We suggest just using spheres initially. The intersection routine for spheres tends to be simpler. You might find it convenient, or even necessary, to create some sort of simple object for returning values from an intersection routine. It might not be necessary now, but eventually you'll want to know where in space the interesection happened, the normal at that location, etc.

When you've gotten the spheres detection to work ok, then try and get triangles to work as well. Make sure that you can't see surfaces behind the camera, and that they look as they should as you move the camera around. A lot of subtle bugs in graphics are hard to find when using nice numbers like (0, 0, 0) for the origin, or (0, 0, -1) for the viewing direction.

5.2 Avoid shallow thoughts

At this point, your ray tracer should be giving you images that tell you whether or not you hit an object. Now its time to add depth information. Rather than just returning a color for object hit/object not hit, find some way of associating a color with an object (ie - like hard coding triangles as green and spheres as red) and returning the correct color for the **closest** object that was found in a particular pixel.

Now it is necessary to know how far along the generated eye ray the intersection occurred. If the direction vector of your `Ray` objects are normalized, then your chore becomes much easier, as the methods in the Shirley book solve for t along a ray, which is exactly the distance between the origin of the ray and the intersection location. Of course, regardless of the scale factor on your direction vector, a smaller t still means an object is closer, but t loses physical meaning.

5.3 Let's add color

We should be in good shape to colorize things now. The `Lambertian` material will be the easiest to complete. Lambertian shading requires that you know the normal and color at the point of interest. The normal can most easily be determined in the intersection routine, so you'll have to find some way to get that information to the shading code.

Some common pitfalls for shading routines are not to correctly handle dot products less than 0 (i. e. illumination from behind), using non-normalized vectors for dot products, and using the wrong vectors.

One of the more difficult tasks will be to get shadows integrated into the ray tracer. If one sphere is occluding another from a light source, then part of the occluded sphere shouldn't be shaded. If you follow Shirley and pass minimum and maximum t values to your intersection code, then you can re-use the code you're already using for viewing rays. Look out for another classic bug here: a surface that sometimes shadows itself because roundoff errors make it seem that something is blocking the shadow ray a tiny, tiny distance from the surface. If you get mysterious dark blotches all over your surfaces as soon as you turn on shadows, this is the likely culprit. To avoid this you will want to start your shadow rays a small distance (say `ray.epsilon`) from the shading point.

5.4 Almost done

If you've gotten this far, you've done well. After getting the Lambertian material done (with shadows), getting Phong shading done shouldn't be too difficult. Finally, make sure to gamma correct the final image. This can be done very easily by modifying the `Color` class a little bit.

6 Submission and FAQ

Submission will be through CMS. A FAQ page will be kept on the course website detailing any new questions and their answers brought to the attention of the course staff.