CS 465 Program 1: Ray I

out: 29 August 2005
due: 13 September 2005

1 Introduction

For this assignment, you will be writing a basic raytracer. We have provided a basic framework to handle I/O, provide an output viewer, and make the command-line interface uniform. The structural layout of the framework is also our suggestion of the typical layout of ray tracing code. However, if you find the framework cumbersome, feel free to change it to suit your needs. This assignment will contain material from Section 9.1-2 and 10.1-5 of Shirley.

2 Assignment Overview

Ray tracing is a simple and powerful alogrithm. Light travels in rays and interacts in geometrically predictable ways with surfaces. By following these rays and simulating these interactions, a ray tracer accurately simulates the physical process of photography. Within the accuracy of the scene and material models and with enough computing time, the images produced by a ray tracer are physically accurate and can appear indistinguishable from real images[1]. Your initial ray tracer will not be able to produce physically accurate images, but it should still produce high quality images with many interesting effects.

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

- Spheres
- Triangles
- Triangle meshes
- Lambertian materials
- Phong materials
- Point lights
- Simple cameras
- Shadows

[1] See http://www.graphics.cornell.edu/online/box/compare.html for a famous example

- Basic gamma correction

We have written a small amount of code to spare you from the details of I/0 and mathematics and to provide you with some convient data structures for representing common objects in rendering problems. However, the framework does not contain any code that does any actual ray tracing. You will develop that code yourself and implement the algorithms to create all the above effects.

Since you will writing most of the code yourself, now is a good time to mention design. We have left the framework simple enough to give you the freedom to determine the design you believe is best. So feel free to create classes as and when you see fit and lay out the parts of your ray tracer in a manner that you find convenient. Any solution that correctly meets the requirements below and is clearly written will recieve full credit.

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. General guidelines can be found in Shirley and in the lecture notes.
2. You need to fill in two methods in the camera class. One, `getRay()`, generates rays originating at the camera and another, `lookAt()`, defines the camera frame of reference.
3. You need to support spheres and triangles. Specifically, you need to be able to intersect a `Ray` with a `Sphere`, a `Triangle` and a `MeshTriangle` (see below for diffences).
4. You must support a Lambertian material. This is defined in Shirley 9.1 and in the lecture notes.
5. You must support a Phong material. This is defined in Shirley 9.2 and in the lecture notes.
6. The only lights you need to use are point lights. The intensity of these lights will fall off proportional to r^2.
7. You must gamma correct your images for a display device of $\gamma = 2.2$.
8. 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.
9. You **do** have to worry about hitting the back side of a surfaces. When shading a point, we would like you to detect this condition and color the back facing surface the color green $(0, 1, 0)$ (ignore all other shading effects). Since the back facing condition is determined from the surface normal, this error shading will allow you to determine if your normal calculations are correct. There are no back facing objects in any of the models provided.

4 Framework Information

The framework we have given you includes some of the classes you will need to finish this assignment. Here we give an explanation for each of the framework classes and some hints about how each can be used. We refer you to the code for more information about each class. The comments for methods and classes can give you a more detailed idea of their use or intent.

4.1 `Parser`

The `Parser` is the only class that we strongly recommend you do not alter. The parser class is based on a simple XML parser that is built into Java. The parser reads a XML document and instantiates objects in a scene based on the contents. The input format is described at the top of the parser class description, but the format is meant to be intuitive and it should be straightforward to create new input files from the provided ones.

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. It will treat all parameters as input files, parse them, render images and write these 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. The class also contains an method called `renderImage()` which decomposes the image into blocks and renders each block separately. This is done to support a viewer application that will show the progress of a rendered image as it completes. The private class `Spiral` is used to compute the spiral of image blocks computed during rendering. The entry point of your code is the method `renderBlock()` that must render a single block of the image.

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

We have created an abstract class called `Surface` to represent objects in the scene. It is generally a good idea to work at an abstract level when designing a ray tracing algorithm and we suggest that you work at the level of the `Surface` class when writing your ray tracing algorithms. Classes for specific types of surfaces can hide the differences in implementation. `Triangle` and `Sphere` are two such classes. In the framework they are mostly empty. They contain only the data needed to store the objects and an empty implementation of the `intersect()` method defined in `Surface`.

4.4 `Mesh` and `MeshTriangle`

`Mesh` and `MeshTriangle` also describe surfaces, specifically triangle meshes. To save memory, it is typical to store a mesh of triangles in a packed format rather than expanding the mesh into individual triangle objects. While none of the input files for this assignment will be large enough to stretch the limits of the computer's memory, we have included the files here so you will have some familiarity with the structure of such classes.

The two classes work in concert. The `Mesh` class stores arrays of packed mesh data and a `MeshTriangle` stores a reference to a `Mesh` object and the indicies of information specific to a single triangle. You should not need to edit the `Mesh` class, but you will be required to implement the intersection method in the `MeshTriangle` class. However, the implementation of this method should be identical to the implementation of the same method in `Triangle` except that the vertex data must be read out of the `Mesh` object in the former case. You are free to copy the code from one place to the other.

4.5 `Material`, `Lambertian`, and `Phong`

As `Surface` is to surfaces, `Material` is to materials. The implementation classes `Lambertian` and `Ward` contain only data declarations and empty implementations of the `shade()` method that must be supported by all materials.

4.6 `Camera`

The `Camera` class represents the vantage point and orientation for viewing a `Scene`. The data and its accessors have been provided in the `Camera` class. The class models the camera's orientation with three orthonormal vectors, traditionally called forward (out from the image), right (to the right in the image plane) and up (to the top of the image plane). You must implement the method `lookAt(Point3 inEye, Point3 inTarget, Vector3 inUp)` which takes a new eye point of the camera (it's location), a new target point (the point in the center of the image) and a new up vector (a vector close to the up direction of the image) and constructs orthonormal values for forward, right and up.

The size of the image is defined by the fields of view in the X and Y image directions. The field of view is a measure of the angular extent that can be seen in each direction. As implemented, the class also continuously updates cached values of the tangent of half the field of view in each direction. These values will be useful when you implement `getRay(Ray outRay, double inU, double inV)`. This method takes a point in the image `(inU, inV)` and sets `outRay` to a ray starting at the camera location and pointing in the direction of what is seen at that image point. The image plane coordinates are defined abstractly for all image shapes and sizes and lie in in the unit square $[0, 1] \times [0, 1]$. The lower left hand corner of the image is $(0, 0)$. You must convert pixel coordinates into this abstract coordinates before calling `getRay()`.

4.7 `Light`

The `Light` just a holder for a location and power.

4.8 `Scene`

This class is only a container for the surfaces, materials, lights, camera and image defined by an input file.

4.9 `Point3`, `Vector3`, and `Tuple3`

Basic mathematics classes for 3D points and vectors. Both Point3 and Vector3 derive from Tuple3, mainly, for convenience in parsing. See the class descriptions for more information.

4.10 `Ray`, `Color` and `Image`

Basic utility classes for rays, colors and images.

4.11 `IntersectionRecord`

A class to hold the information computed during a Ray/Surface intersection. It is initially empty—you must determine the necessary information to return.

4.12 `PanelDisplay` and `QuickViewer`

These classes support the rendering progress viewing window. There should be no reason to edit these files. You can change the size of the render blocks and turn the viewer on and off using constants at the top of the `RayTracer` class.

4.13 Hints

- Start by getting a working camera. Try to implement the needed camera methods and test to make sure that your camera can generate correct eye rays.
- Start simply—you can get your first picture with just a working `RayTracer`, `Sphere` and `Camera` class.
- There is a saying: "Make it work then make it fast". While we won't say that efficiency is not an important part of a ray tracer, for this assignment, the scenes will be small enough to render fast regardless of how you implement things. Write a working version of the ray tracer and then think about how you can make it work faster afterwards. You will not be graded on how fast your ray tracer is. Our test implementation can render all of the test inputs in a minute or less (For the bunny scene, Test4. The other scenes render in less than 10 sec.), but its pretty streamlined. Do not be surprised if your code takes several times longer, however if your code takes longer than 10 minutes (a minute for Tests 1-3) to render, something is probably wrong.
- Try to design and write your code cleanly. It will help you understand what you have written and how it works.
- Get a good IDE and use it. We recommend Eclipse (its free, already installed in the lab, and fully featured), but any environment should do.
- Encode debugging information in images. Think of an image as the the print statement of graphics. When you are having trouble rendering, make images where the colors stand for various program parameters then look at the images and see if they make sense. (But remember that you might be gamma correcting the images and that colors are clamped from 0 to 1 by Java when saved.)
- 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 First steps

The easiest and most frequently rendered image in computer graphics is a black screen. Try to render something informative as soon as you can. If you have the black screen problem, try to make the image colors depend on as little code as possible. Once you have *something* in the output, you can add elements one at a time looking for problems. Your first tasks should be to render images that test the camera `getRay()` method and your first intersection methods. A good basic test loops over all of the pixels and tests for the presence of an object in each. You can do this without even implementing a material.

Of course, you will have to write an intersection method. We suggest just using spheres initially. The intersection routine for spheres tends to be simpler. After this works, move onto triangles.

Finally, you should assure yourself that the ray tracer is correctly computing the intersection information. Determine what information you need to have about an intersection point to perform correct shading, place fields so you can return the information in the intersection record, and add code to compute it. Draw images to check that the data you compute is correct. If you can assure yourself that the intersections are correct, you can isolate future problems to other areas of your code.

5.2 Materials and Shadows

Your first step should be to recognize front vs. back facing objects. After you can assure yourself that you can correctly recognize front and back facing, start implementing a material shader.

The Lambertain material is the simplest and a good place to start working on shading. The first test scene contains only spheres and Lambertain materials. Get this image working correctly and move on and implement Phong. Worry about shadows only after all this is working correctly.

If you have all everything else working, shadows should be almost trivial to add to your ray tracer. It shouldn't involve much more than another ray cast and an if statement.

5.3 Finish Up

If you've gotten this far, you've almost done. Make sure you implement gamma correction and do plenty of testing. The 4 scenes we have given you are *not* a complete test set. There are subtle problems that can arise if you are not careful. Think about what your ray tracer is doing and try to test each part as robustly as possible.

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.