CS 465, Fall 2003, HW 3b

Andy Scukanec

October 14, 2003

1 Introduction

For this assignment, you will be writing a raytracer that will work with basic shapes, lights, and materials. As with the other assignments, 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. If you find the framework cumbersome, feel free to change it. But be warned that we will be most adept at helping you if you stick within the framework, and if you find yourself creating completely new classes and methods because the framework doesn't make sense, you might want to step back and ask yourself if you are missing something.

2 Requirements

Your ray tracer will read a scene description from an input file containing a camera, objects, lights, and some other parameters, and it will render an image that can be displayed and/or saved in PNG format. (The framework handles the I/O; you just have to implement the algorithm.)

You're required to support the following types of objects in the scene:

- Surfaces, which can be spheres or triangles, and which have an associated material.
- Three materials:
  - A diffuse material, which has an ambient color k_a and a diffuse color k_d and is rendered with Lambertian shading:

    $$L = k_a I_a + k_d (I/r^2) max(0, \mathbf{n} \cdot \mathbf{v}_L)$$

    where L is the light reflected from the surface, I_a is the scene's ambient intensity, I is the intensity of the light source, r is the distance to the source, $\mathbf{n}$ is the surface normal, and $\mathbf{v}_L$ is the direction toward the light source.

  - A phong material, which has an ambient color k_a, a diffuse color k_d, a specular color k_s, and a specular exponent, and is rendered using the Blinn-Phong model:

    $$L = k_a I_a + k_d(I/r^2)max(0, \mathbf{n} \cdot \mathbf{v}_L) + k_s(I/r^2)max(0, \mathbf{n} \cdot \mathbf{v}_H))^p$$

    In addition to the above symbols, $\mathbf{v}_H$ is the half vector.
  - A glazed material, which adds a mirror reflection to another material, known as the "base" material, of the other models:

    $$L = L_{\text{base}} + L_m$$

    where L_m is the light seen along the mirror reflection ray and L_{base} is the light reflected by the base material.

- Point light sources. These sources have just a position and an RGB intensity, and you must support any number of lights.

- A perspective camera. Its pose is specified in the file using the eye/target/up vector approach we discussed in class, and its field of view is specified by giving the vertical field of view and the aspect ratio (the ratio of the width to the height of the image).

Given a scene populated with these types of objects, your program must be able to render an image with the following features:

- Shadows cast by the point light sources.

- Antialiased output computed using a n_s by n_s grid of samples within each pixel (where n_s is a parameter).

3 Framework

The framework consists of a set of classes in the default package. They make use of a number of Java 1.4 features, so make sure you are using JDK 1.4 or later. This homework does *not* use the GL4Java library (although having it around won't hurt anything). It does, however, use the `vecmath.jar` from the previous homework. There are some things to be said about the classes we are giving you in order to make sure you understand how to use the framework well.

3.1 RayTracer

This class is really the starting point for the entire program. It doesn't do a lot, but it does tie everything together. You can pass any number of parameters to the RayTracer class as filenames of scene descriptions to render. The resulting images will have the same name as the scene file, but will be in .PNG format. This image format should be easy to open on any modern Windows/*nix OS. If you don't give the main method any parameters, an interactive GUI window will open. From here, you will have the option to open a scene file and watch it render.

3.2 Light, Surface, and Material

These three classes are the abstract superclasses that you will have to subclass at least once. They represent just what you would expect them to in a ray tracer and have abstract methods for doing basic operations. Each subclass should have some additional fields, and perhaps some additional methods. For instance, if you were creating a spot light, you would create a Light subclass with an intensity drop-off factor, a radius, and a direction. Keep in mind that how you declare variables in these subclasses is *very* tightly tied to how the Parser deals with these objects in a scene file.

3.3 ViewingFrame

This is the main GUI class that you see if you start the program without any parameters. No reason to worry about this class unless you're just interested.

3.4 IntersectionRecord

This class contains all the necessary information about a particular ray-surface intersection. It keeps track of the location of the interesction, the primitve that was intersected, the normal at the intersection, and the distance t along the ray where the intersection occurred. This record is to be passed back and forth from one object to another during program execution.

3.5 Scene, Camera

The scene object is simply a collection of attributes for a particular scene. This includes a single camera, a background color, the ambient light, and the lists of materials, lights, and surfaces that compose the scene. The main interest that the scene object should hold for you is in 3 important methods:

```
public Vector3d[][] renderAll();
public Vector3d renderPixel(int x, int y);
public Vector3d trace(Ray r);
```

Right now, these methods just return null ... obviously unacceptable. More on this in the actual assignment description.

The Camera object contains information vital to actual image production. It holds the information such as the location of the camera, the direction in which it is looking, etc.

3.6 Ray

The Ray object is a pair of Vector3d objects. One represents the origin of the ray, and the other represents its direction. This object comes with a particularly useful method called `evaluate(double t)`. This evaluates the ray at time `t` and returns the location at that time. This class should be used whenever you are casting rays for reflections, intersections, etc. The other fields inside a Ray object are two double values, `tMin` and `tMax`. These tell you the distance over which the ray is valid. The default values for tMin and tMax should be `Ray.EPSILON` and `Double.POSITIVE_INFINITY` respectively.

3.7 Parser

The parser class is responsible for reading in the scene description, and creating a Scene object. This is the most complicated class you will be using that you did not develop. The parser will be very picky about the parameters of objects in the scene, because of how it actually creates and alters the objects. For each object, it reads the parameter information (name, type, and value), and writes that information directly into the object. For this reason, you need to be very precise when you create a scene file.

Let's go over the format of a scene file. A scene consists of a listing of object descriptions. The template for an object description is:

```
<ObjectType> <Classname> [<name-of-object>]
<param-type-1> <param-name-1> <param-value-1>
...
<param-type-n> <param-name-n> <param-value-n>
```

`<ObjectType>` can be one of "Light", "Surface", "Material", or "Camera". Depending on what `<ObjectType>` is, the Parser will put the resultant object into the a appropriate list in the Scene. `<Classname>` is the actual name of the class that will be created, and that class must be a subclass of Light if it is a Light, etc. For instance, if you are placing a sphere in your scene, the first line of your sphere object would look like `Surface Sphere`. The `<Classname>` value must be typed exactly and is case sensitive. The other restriction is that the Sphere class must be a subclass of the Surface class. The optional `<name-of-object>` information is used if you want to reference this object later. This will be particularly useful for associating a material to a surface.

All of the parameters that go with a class declaration must be on the following lines, with no blank lines in between! If the parse encounters a blank line, it will assume that a new object has been started. For each of the parameters of the class, we have three parts: the type, the name, and the value. The type must be either “int”, “double”, “String”, “ref”, or “Vector3d”. The parameter name must be typed *exactly* as the parameter is declared in the class. So if we had a material with a variable called “diffuseColor” in its definition, then we set the diffuse color in the scene file by adding the line `Vector3d diffuseColor [1.0 0.0 0.0]`. Note that `Vector3d diffusecolor [1.0 0.0 0.0]` would have failed because the names are case sensitive.

For the parser to work, any variable that will be set by the parser must be declared as either protected, public, or package visible. Private variables will cause exceptions to be thrown if the Parser tries to modify them. Vector3d values need to be formatted like `[<x-value> <y-value> <z-value>]`.

There are some settings that don’t belong to any object. The background color, ambient light, the image height and width, and the number of samples per pixel (only used with anti-aliasing). To set these values, simply add the following lines and substitute in appropriate values:

```
BGColor <bg-color-vec>
AmbientLight <ambient-light-vec>
Height <num-pixels>
Width <num-pixels>
AASamples <samples-per-pixel>
```

4 Assignment

So . . . now that we’ve told you what’s been done for you, its time for us to tell you what you need to do. There are a number of things to do in order for you to produce a full fledged working raytracer, and we have come up with a sequence of these things that we hope will help you out. Generally, it is difficult to write a raytracer because you don’t start seeing any output for a very long time. By the time you do get to see output, many things could have gone wrong. We have tried to mitigate this problem by giving you steps that will give you output pretty early on.

4.1 Intersections and Cameras

First off, lets get your intersections working with some basic shapes. Fill in the sphere and triangle intersection methods. Don’t forget to alter the intersection record! Please note that triangle intersections should only be valid on the *positive* side of the triangle. Which side is the positive side depends on the order in which the vertices were specified. The normal for a triangle should be $(\vec{v_1} - \vec{v_0}) \times (\vec{v_2} - \vec{v_0})$.

Now, you need to fill in two methods in the Camera object. First, write the necessary code for `recalculate()`. This method should recalculate the $\vec{u}$, $\vec{v}$, and $\vec{w}$ vectors for the scene. Next, you need to fill in the `getRayThrough(double x, double y, double width, double height)`, which as you might have guessed, returns a Ray whose origin is at the location of the camera, and whose direction is such that it will pass through the imaginary point (x, y) on the pixel plane. Notice that this method takes in doubles, meaning that it can return rays that go through fractional pixels. This is useful for anti-aliasing and for making sure that you cast the ray through the center of the pixel.

Last, go to the scene object's `renderAll()`, `renderPixel(int x, int y)`, and `trace(Ray r)` methods. The first method should just repeatedly call the second method inside of a double loop. The `renderPixel` should call `trace` after obtaining a ray from the Camera object. The `trace` method should be a bit more complicated. It needs to find the closest object in the scene, and query that object to find out what the color is at that point. For now though, let's just test for intersection and turn the pixel white if the ray intersects any object at that point, and black otherwise.

If you are having a hard time getting your camera to compute the $\vec{u}$, $\vec{v}$, and $\vec{w}$ vectors correctly, simplify your problem and make your camera work so that it is centered at the origin and facing in the -z direction.

4.2 Shades of a Raytracer

Now, we need to see if we can't get somewhere and actually create a scene with textures. In the diffuse material class, and implement the `shade` method. This material should have only diffuse shading available to it (note that this includes the ambient light!), and should be fairly easy to implement. The biggest problem here will be making sure your vectors are in the correct direction.

You will also need to implement the basic point light. Again, fill in the empty method (`getIntensity` this time).

Before you start seeing any colors, you will need to modify your `renderPixel(int x, int y)` code to get the material from a surface and ask it for the color. If you get the light and material code right, you should be able to create nice diffuse spheres and triangles.

At this point, more errors from the previous section could easily show themselves. Things like incorrect normals etc. only become apparent now that you have code that uses this information. Expect to spend a fair amount of time debugging this.

Note about Colors - we are using `Vector3d` objects to represent color, since the `java.awt.Color` class isn't as easily modifiable. Be careful of the specifics of the `Vector3d` methods though - read the documentation carefully. In particular, be careful of `scaleAdd`. Don't forget to look in the

`Tuple3d` class for most of the methods either! The components line up when translating from a Vector3d to color - x = r, y = g, z = b. The range for each of the colors should be $[0.0, 1.0]$.

4.3 The Next Steps

Now we get to the last few steps in writing your raytracer. The hardest part has ironically been done – debugging the vector math. Now, we want you to implement a BlinnPhong material as from the lecture slides. Just use the formula from the Shirley book, and fill in the shade method.

At this point, it is prudent to test to see if a light is occluded or not. Use the incoming ray information in a light's `getIntensity()` method to see if going back along the ray hits some object in the scene before it hits the surface in the IntersectionRecord. If it does, then the current light is occluded and should return an intensity of $\vec{0}$. There are actually a number of places where you could handle light occlusions - anything reasonable is acceptable though.

Finally, implement your `GlazedMaterial` class's shade method. The color returned by this class should be $c = c_l + c_r k_m$, where c_l is the color returned by the base material, c_r is the reflected color, and k_m is the mirror coefficient. Make sure your code handles the case when there is no base material!

4.3.1 Anti Aliasing

Rather than just shooting a ray through the center of each pixel, make your render pixel method shoot multiple rays through each pixel and average them together. You should shoot `scene.samplesPerPixel` in each direction. So if `samplesPerPixel` was 3, then you would really shoot 9 rays. Please be aware that this can *really* slow down the rendering process.

4.4 Testing

Don't forget to test with as many cases as you can! Put an object behind the camera, put lights behind objects, objects behind lights, make objects intersect each other, etc. We will be using a large number of test cases so make sure you test thoroughly! We will be giving you some sample input scenes and their output, but passing the test cases we give you won't guarantee your code is correct!

4.5 Extra Credit

If you are brave enough, you can try handling refraction, area lights, spotlights, and other interesting surfaces too. If you do any of this, please submit a scene file for each of the extra credit ideas you attempt.

4.5.1 Dielectric Materials

Make a `DielectricMaterial` class in a reasonable way that implements the model of dialectric materials presented in the Shirley book.

4.5.2 New Primitives

New primitives are also worth extra credit. Some suggestions are discs, tori, CSG object, triangle meshes, cones, cylinders, polygons, etc. Again, please be sure to submit example test cases with your code if you attempt any of this.

4.5.3 Area Lights/Spotlights

There are numerous resources available on how to implement area lights and spotlights on the web. Implement (at most) one of each and include these with your submission. Be sure to make a readme.txt somewhere giving a link to the model that you used (book name and author, URL, etc.).

4.6 Recap

- Methods to implement:

```
Sphere.intersects(Ray, IntersectionRecord, Scene);
Triangle.intersects(Ray, IntersectionRecord, Scene);

Camera.recalculate();
Camera.getRayThrough(double, double, double, double);
Scene.renderAll();
Scene.renderPixel(int, int);
Scene.trace(Ray);

PointLight.getIntensity(IntersectionRecord, Scene);
Diffuse.shade(Ray, IntersectionRecord, Scene);
BlinnPhong.shade(Ray, IntersectionRecord, Scene);
GlazedMaterial.shade(Ray, IntersectionRecord, Scene);
```

- Make sure you are using `Vector3d` functions correctly
- Use System.out.println() to debug - we have implemented .toString() in almost every class.
- Make sure you differentiate between the front and back of triangles
- Check the FAQ page (see below)

5 Submission

You should submit through CMS as with the other coding assignments thus far. Zip up all of your .java files *and* your scene files as well. Submit them through CMS. Do ***NOT*** submit .class files. JAR files are fine too if you know how to create them.

6 FAQ Page

We will be maintaining a FAQ page linked from the assignment page. If you have any questions about how to use the framework code, what we intended something to mean, etc., please check the FAQ page and newsgroups first. If your question is still not answered, then email the course staff at cs465-staff@cs.cornell.edu. We will *not* be posting to notify you that we have updated the FAQ page because we anticipate that this will happen frequently.

If you have any questions regarding the use of lab equipment, a supported IDE, or anything ancillary that does not directly involve this assignment, then please talk to either Andrew Butts or Pet Chean.