CS 4620 PA 1: Basic Ray Tracer

out: Sep 9, 2011
due: Sep 23, 2011

1 Introduction

Ray tracing is a simple and powerful algorithm for rendering images. Within the accuracy of the scene and shading models and with enough computing time, the images produced by a ray tracer can be physically accurate and can appear indistinguishable from real images[1]. Your ray tracer will not be able to produce physically accurate images, but it will be capable of rendering basic geometry as well as more complex mesh models consisting of smaller triangle meshes. Your ray tracer will also support recursive ray tracing to produce more realistic results when rendering transparent or reflective materials.

Due to this semester's tight schedule, a more populated framework code is provided to save you from spending time on class hierarchy design, and rather to have you focus on implementing the core ray tracing components. However, you also have the freedom to redesign the system as long as your ray tracer meets our requirements and produce the same images for given scenes.

2 Requirement Overview

Your ray tracer will read files in an XML file format and output PNG images. We have split the ray tracing assignment into two parts. For now you will implement "basic ray tracing". Later in the semester (in PA 3) you will implement advanced features in ray tracing. We will release a single code base to you that you will build on for both parts. In the code base the code for Part A and Part B are clearly marked. For now ignore everything for Part B. In this document we will focus on what you need to implement for Part A.

It should support the features given below.

1. *Basic geometry*. Spheres, cylinders, cones, and axis-aligned boxes, which can be transformed under geometric transformations.(Shirley 4.4,13.2 and Homework 1)
2. *Basic shader*. Implement Lambertian and Blinn-Phong shaders (Shirley 4.5, 10.1-10.2)
3. *Basic shadows*. Add "direct illumination" with shadowing by implementing point lights with shadows.

[1] Cornell pioneered research in accurate rendering. See http://www.graphics.cornell.edu/online/box/compare.html for the famous Cornell box, which exists in Rhodes Hall.

4. *Basic viewing*. Arbitrary perspective projections as described in the file format below.

3 Implementation

We have divided the assignment into two parts A and B. Once you have finished part A, your ray tracer should be able to render a scene with basic geometry and basic shaders.

3.1 Part A

We have marked all the functions or parts of the functions you need to complete in part A with `TODO(A)` in the source code. To see all these TODO's in Eclipse, select *Search* menu, then *File Search* and type "`TODO(A)`." The following explanations are to give you a rough overview of the tasks. Most instructions, however, are provided in the comments in the source code.

1. *Camera.java*. This class represents the camera in the scene. You need to complete two functions which will initialize the camera's basis and return a ray corresponding to a given pixel position in the rendered image. Note that projNormal and viewDir are anti-parallel to each other in our examples (and hence redundant), and further, projNormal is perpendicular to the image plane. This is all we expect in the examples we will use to test your implementation.

   However, you are welcome to both implement and try out settings where this is not the case to support oblique camera viewing.

2. *RayTracer.java*. Once you have completed *Camera.java*, you need to find out a way to call your camera class to shoot the correct ray to the scene.

3. *Box.java*. This class is a subclass of Surface. It contains `minPt` and `maxPt`, which are 3D points. If the two points are $(x_{\text{min}}, y_{\text{min}}, z_{\text{min}})$ and $(x_{\text{max}}, y_{\text{max}}, z_{\text{max}})$ then the box is $[x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}] \times [z_{\text{min}}, z_{\text{max}}]$. The only function you need to complete for part A is the `intersect()` method which will take the shooting ray as input and return true if the ray hits the box. Moreover, this function will fill an `IntersectionRecord` whose information will be used later on.

4. *Sphere.java*. This class is a subclass of Surface. It contains `center`, a 3D point, and `radius`, a real number. The only function you need to complete for part A is the `intersect` method which performs a similar task to the `intersect()` in Box.java.

5. *Cylinder.java*. This class is a subclass of Surface. It contains `center`, a 3D point, and `radius`, `height`, both real numbers. The only function you need to complete for part A is the `intersect` method which performs a similar task to the `intersect()` in Box.java. Assume that the cylinder is capped, i.e., the top and bottom have caps (it is not a hollow tube). Also, assume it is aligned with the z-axis, and is centered around `center`; i.e., if center is $(center_x, center_y, center_z)$, the cylinder's z-extent is from $center_z - \frac{height}{2}$ to $center_z + \frac{height}{2}$.

6. *Cone.java*. This class is a subclass of Surface. It contains `center`, a 3D point, and `height` and θ, which are both real numbers. The only function you need to complete for part A is the `intersect` method which performs a similar task to the `intersect()` in Cylinder.java.

This is a truncated cone; again assume it is capped (both at the top and bottom). The cone is centered at `center`, its axis is along the z-axis, the cylinder's z-extent is from $center_z - \frac{height}{2}$ to $center_z + \frac{height}{2}$. Further, if the cone were to go to a point, the angle it would make is θ. See the figure. Note that you will have to work out the correct θ and `height` to make the cone converge to a tip.

7. *Lambertian.java* and *Phong.java*. In these two classes, you need to complete `shade()` method which will shade the material using the information from `IntersectionRecord`, light sources, and eye's position.

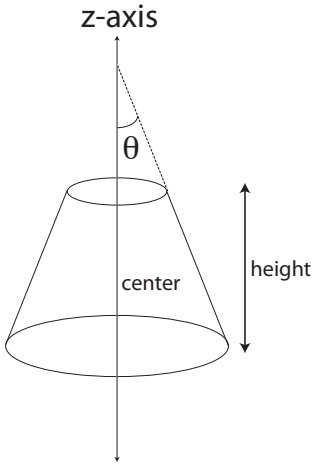

Figure 1: Truncated Cone

.

4 Framework

The rest of the framework is described at the end of this document in Appendix A. You do not need to spend time trying to understand it and it is not essential to read this to get started on the assignment. Instead, you can reference it as needed.

5 Handing in

All submissions will be through CMS. We will also use CMS to schedule demo slots to grade the ray tracer as we get closer to the due date. We will post clarifications on Piazza if needed.

6 Appendix A

The framework for this assignment includes a simple main program, some utility classes for vector math, a parser for the input file format, and stubs for the classes that are required by the parser.

6.1 `Parser`

The `Parser` class contains a simple parser based on Java's built-in XML parsing. The parser simply reads a XML document and instantiates an object for each XML entity, adding it to its containing element by calling `set...` or `add...` methods on the containing object.

For instance, the input

```
<scene>
    <surface type="Sphere">
        <shader type="Lambertian">
            <diffuseColor>0 0 1</diffuseColor>
        </shader>
        <center>1 2 3</center>
        <radius>4</radius>
    </surface>
</scene>
```

results in the following construction sequence:

1. Create the scene.
2. Create an object of class `Sphere` and add it to the scene by calling `Scene.addSurface`. This is OK because `Sphere` extends the `Surface` class.
3. Create an object of class `Lambertian` and add it to the sphere using `Sphere.setShader`. This is OK because `Lambertian` implements the `Shader` interface.
4. Call `setDiffuseColor(new Color(0, 0, 1))` on the shader.
5. Call `setCenter(new Point3D(1, 2, 3))` on the sphere.
6. Call `setRadius(4)` on the sphere.

Which elements are allowed where in the file is determined by which classes contain appropriate methods, and the types of those methods' parameters determine how the tag's contents are parsed (as a number, a vector, etc.). There is more detail for the curious in the header comment of the `Parser` class.

The practical result of all this is that your ray tracer is handed a scene that contains objects that are in one-to-one correspondence with the elements in the input file. You shouldn't need to change the parser in any way.

6.2 `RayTracer`

This class holds the entry point for the program. The `main` method is provided, so that your program will have an command-line interface compatible with ours. It treats each command line argument as the name of an input file, which it parses, renders an image, and writes the image to a PNG file. The method `RayTracer.renderImage` is called to do the actual rendering.

6.3 `Image`

This class contains an array of `floats` and the requisite code to get and set pixels and to output the image to a PNG file.

6.4 The `ray.math` package

This package contains classes to represent 2D and 3D points and vectors, as well as RGB colors. They support all the standard vector arithmetic operations you're likely to need, including dot and cross products for vectors and gamma correction for colors.

6.5 Other classes

The other classes in the framework all exist because they are required in order for the parser to decode files in the input format described above. Since the XML entities in the file correspond directly to Java objects constructed by the parser, there is a class for every type of XML tag that can appear in the file, including `Scene`, `Image`, `Camera`, `Light`; `Surface` and its subtypes `Sphere`, `Box`, `Cylinder`, and `Cone`; and `Shader` and its subtypes `Lambertian` and `Phong`. These classes generally contain only the fields and set/add methods required to implement the file format.

6.6 File format

The input file for your ray tracer is in XML. An XML file contains sequences of nested *elements* that are delimited by HTML-like angle-bracket tags. For instance, the XML code:

```
<scene>
    <camera> </camera>
    <surface type=Sphere>
        <center>1.0 2.0 3.0</center>
    </surface>
</scene>
```

contains four elements. One is a `scene` element that contains two others, called `camera` and `surface`. The `surface` element has an *attribute* named `type` that has the value `Sphere`. It also contains a `center` element that contains the text "1.0 2.0 3.0", which in this context would be interpreted as the 3D point $(1, 2, 3)$.

An input file for the ray tracer always contains one `scene` element, which is allowed to contain tags of the following types:

- `surface`: This element describes a geometric object. It must have an attribute `type` with value `Sphere` or `Box`. It can contain a `shader` element to set the shader, and also geometric parameters depending on its type:
  - for sphere: `center`, containing a 3D point, and `radius`, containing a real number.
  - for box: `minPt` and `maxPt`, each containing a 3D point. If the two points are $(x_{\text{min}}, y_{\text{min}}, z_{\text{min}})$ and $(x_{\text{max}}, y_{\text{max}}, z_{\text{max}})$ then the box is $[x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}] \times [z_{\text{min}}, z_{\text{max}}]$.
  - for cylinder: `center`, containing a 3D point, `radius` and `height`, each containing a real number.
  - for cone: `center`, containing a 3D point, `height`, and θ, each containing a real number.
- `camera`: This element describes the camera. It is described by the following elements:
  - `viewPoint`, a 3D point that specifies the center of projection.
  - `viewDir`, a 3D vector that specifies the direction toward which the camera is looking. Its magnitude is not used.
  - `viewUp`, a 3D vector that is used to determine the orientation of the image.
  - `projNormal`, a 3D vector that specifies the normal to the projection plane. Its magnitude is not used. By default it is anti-parallel to view direction.
  - `projDistance`, a real number d giving the distance from the center of the image rectangle to the center of projection.
  - `viewWidth` and `viewHeight`, two real numbers that give the dimensions of the viewing window on the image plane.
  - `image`: This element is just a pair of integers that specify the size of the image in pixels.

  The camera's view is determined by the center of projection (the viewpoint) and a view window of size `viewWidth` by `viewHeight`. The window's center is positioned along the view direction at a distance d from the viewpoint. It is oriented in space so that it is perpendicular to the image plane normal and its top and bottom edges are perpendicular to the up vector.
- `light`: This element describes a light. It contains the 3D point `position` and the RGB color `color`.
- `shader`: This element describes how a surface should be shaded. It must have an attribute `type` with value `Lambertian` or `Phong`. The Lambertian shader uses the RGB color `diffuseColor`, and the Phong shader additionally uses the RGB color `specularColor` and the real number `exponent`. A shader can appear inside a surface element, in which case it applies to that surface. It can also appear directly in the scene, which is useful if you want to give it a name and refer to it later from inside a surface (see below).

If the same object needs to be referenced in several places, for instance when you want to use one shader for many surfaces, you can use the attribute `name` to give it a name, then later include a reference to it by using the attribute `ref`. For instance:

```
<shader type="Lambertian" name="gray">
    <diffuseColor>0.5 0.5 0.5</diffuseColor>
</shader>
<surface type="Sphere">
    <center>0 0 0</center>
    <shader ref="gray"/>
</surface>
<surface type="Sphere">
    <center>5 0 0</center>
    <shader ref="gray"/>
</surface>
```

applies the same shader to two spheres.

Really, the file format is very simple and from the examples we provide you should have no trouble constructing any scene you want.