CS 465 Program 1: Ray I

out: Wednesday 29 August 2007
due: Wednesday 12 September 2007

1 Overview

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 later in the semester you will extend it to produce nice looking images with many interesting effects. If you really catch the ray tracing bug, you can go on to take CS 665, where you will learn how to build a modern, physically accurate ray tracer.

In this assignment, your ray tracer will have support for:

- Spheres and axis-aligned boxes
- Lambertian and Phong shading
- Point lights with shadows
- Arbitrary perspective cameras

Some framework code is provided to save you the time to implement I/0 and vector operations. However, the framework does not contain any code that does any actual ray tracing—you will develop that code yourself.

When you write the ray tracing code, you have the freedom to determine the design you believe is best. You may want to add some classes to the program, and there are many choices about where to put the various parts of the computation. Any solution that correctly meets the requirements below and is clearly written will recieve full credit. The textbook, the lectures, and the course staff are all sources of information about good approaches to coding up a ray tracer.

This is not to say that you need to write a lot of new code. For reference, the framework contains 19 classes with about 2000 lines of code (1500 of which are in the parser and the vector math classes), and our solution contains three small additional classes and about 600 additional lines of code.

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

2 Requirements

1. Use a ray tracing algorithm.
2. Support arbitrary perspective projections as described in the file format section below.
3. Support spheres and axis-aligned boxes.
4. Support the Lambertian and Blinn-Phong shading models, as defined in Shirley 9.1–9.2 and in the lecture notes.
5. Support point lights that provide illumination that does not fall off with distance.

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.

3 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 contains 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 $[x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}] \times [z_{\text{min}}, z_{\text{max}}]$.
- `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, and negating its direction has no effect. By default it is equal to the 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 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.

4 Framework

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.

4.1 `Parser`

The `Parser` class contains a simple and, we like to think, elegant 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.

4.2 `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`, which you need to write, is called to do the actual rendering.

4.3 `Image`

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

4.4 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.

4.5 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` and `Box`; 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.

5 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.