CS 4620 Program 4: Ray II

out: Wednesday 4, November 2009
due: Wednesday 2, December 2009

1 Introduction

In the first ray tracing assignment you built a simple ray tracer that handled just the basics. In this assignment you will build a more capable ray tracer that can handle more substantial models and can produce much more interesting renderings.

The framework for this assignment is the solution to the first ray tracing assignment. If you are happy with your Ray I solution you are encouraged to use it; otherwise you can start from our solution. You are free to design and implement the extensions in any way you like.

2 Requirements

Your ray tracer will read files in a standard file format and output PNG images (like the first ray tracer). It should support the basic features given below. The starred problem is to implement two extensions (you may suggest your own as well, but do so by November 16th).

2.1 Basic features

Your ray tracer should implement the following features beyond what the first ray tracer did:

1. *An acceleration structure*. Your program should be capable of rendering large models (up to several hundred thousand triangles) with basic settings in a few minutes. Achieving this requires a spatial data structure that makes the time to trace a ray sublinear in the number of objects. We recommend implementing an axis-aligned bounding box hierarchy (AABB), which is a simple and effective way of speeding up ray traversal.

2. *Recursive ray tracing*. You should implement a "Glazed" shader that acts like a thin layer of dielectric over another material. It should compute a reflected ray, trace it recursively through the scene, and use the Fresnel reflection factor R to weight its contribution. It should also call another shader which represents the substrate below the glaze to get that contribution (weighted by the Fresnel transmission factor, $1 - R$).

   In the input file the glazed shader is specified by an index of refraction, through a parameter named `refractiveIndex`, and another shader for its substrate:

```
<shader type="Glazed">
  <refractiveIndex>1.5</refractiveIndex>
  <substrate type="Lambertian">
    <diffuseColor>0.4 0.5 0.8</diffuseColor >
  </substrate>
</shader >
```

Having a shader that uses recursively computed rays means that your renderer will generate a chain of rays (or a tree of rays if you choose to do glass), which is potentially slow and infinite in depth. In addition to a maximum-depth cutoff, you should also implement a maximum-attenuation cutoff by keeping track of how much a given ray will contribute to the image (i.e. what is the factor it is being multiplied by before it is added to the image). When that factor drops below a user-determined threshold, you should terminate recursion.

3. *Transformations*. Your ray tracer should support transformations using an approach similar to the modeler. You should introduce a new type of surface, "Group", that contains a transformation and a list of surfaces. The transformation is specified as a sequence of rotations, scales, and translations, which are combined in the order given to define the transformation that is applied to all members of the group. The transformation that appears first in the file is on the outside. All transformations apply to all objects, even if the transformations and objects are intermixed in the file (it makes most sense to put the transformations first, then the objects). Transformations are described in exactly the same way as in the modeler: translations and scales have components for x, y, and z; a rotation is actually a sequence of three rotations about the three coordinate axes, with the x rotation on the inside and the z rotation on the outside.

   The file format can be defined by example. For instance, if the a transformation in the modeling assignment was given as "T: 1 2 3; R: 40 50 60; S: 0.7 0.8 0.9," the same effect can be specified in the ray tracer as follows:

```
<surface type="Group">
  <translate>1.0 2.0 3.0</translate>
  <rotate>40 50 60</rotate>
  <scale>0.7 0.8 0.9</rotate>
  <surface type="Sphere">
    <!-- ... -->
  </surface>
  <!-- more surfaces... -->
</surface>
```

4. *Triangle meshes*. In order to allow for more interesting geometry than spheres and boxes, you should support triangle meshes. Since meshes can be quite large, it is not practical to process them using the parser, so they are stored in a simple text format in separate files. These files contain standard indexed triangle meshes, with optional texture coordinates and surface normals at the vertices. If the mesh contains vertex normals, you should shade it with interpolated normals; otherwise you should shade it with the triangles' geometric normals.

   The input format for a mesh is just a filename reference:

```
<surface type="Mesh">
```

```
  <shader><!-- ... --></shader>
  <data>filename.msh</data>
</surface>
```

The mesh file contains text as follows, one word or number per line:

- The number of vertices in the mesh.
- The number of triangles in the mesh.
- The keyword `vertices`
- The 3D coordinates of the vertices, ordered by vertex number: $x_0, y_0, z_0, x_1, y_1, z_1, \ldots$.
- The keyword `triangles`
- Three integer vertex indices per triangle.
- (Optional) The keyword `texcoords`, followed by u and v coordinates for each vertex.
- (Optional) The keyword `normals`, followed by x, y, and z components of a normal vector for each vertex.

You can find code that reads meshes in this format in the `Mesh.readMesh` method of the Model assignment framework.

5. *Antialiasing*. You should support antialiasing by regular supersampling. The number of samples is specified by the `samples` property of the `Scene` class. For example, the following input specifies a 640 by 480 pixel image rendered with a 3x3 grid of subpixel samples for each pixel.

```
<scene>
  <camera>
    <!-- ... -->
  </camera>
  <image>640 480</image>
  <samples>9</samples>
</scene>
```

You are free to round the number of samples to a convenient number (for example, to the nearest perfect square). So if the user inputs 10, you should assume 9 samples - not 100.

2.2 Starred Problem Extensions

The starred problem is to implement two of the following extensions.

1. *"Glass" shader*. A glass shader simulates an interface between air and a dielectric material. It should compute the directions of the reflected and refracted rays using Snell's law, compute the reflection factor using Fresnel's formulas, then trace reflected and refracted rays recursively and combine the results using the reflection factor. It needs to work for rays coming from both sides of the surface; you can always tell which side is air because the air is on the outside, the side toward which the normal points.

Transparent objects should not cast black shadows; they should attenuate the illumination but not block it entirely. You should work out a way of computing the attenuation that produces shadows with an appearance you like.

In the input file the glass shader is specified just by its index of refraction, through a parameter named `refractiveIndex`:

```
<shader type="Glass">
  <refractiveIndex>1.5</refractiveIndex>
</shader>
```

Alternately, you could implement a "Colored Glass" shader by also including color attenuation using Beer's Law.

2. *Cube-mapped backgrounds*. A ray tracer need not return black when rays do not hit any objects. Commonly, background images are supplied that cover a large cube surrounding the scene. The directon of rays that do not intersect objects are used to as indices into these images and the color of the image in the rays direction is returned rather than black. The techinique is commonly called cube-mapping. To implement cube-mapping in you ray tracer you will need to extend the `Scene` class to contain an image used as the cube map background. You will also need to write code that maps ray directions into cube-map pixels. A short introduction to cube-maps can be found at `http://panda3d.org/wiki/index.php/Cube_Maps` and many actual maps can be found here `http://www.debevec.org/Probes/`.

3. *Output HDR images to OpenEXR format*. OpenEXR is an open standard file format for high-dynamic range images. Once your image is in the format, there are many tools for producing interesting images from the EXR file. There is existing code for outputting to it, but it is written in C++. So most of the work for this will be in making your Java code interact with the C++ library (using JNI). More information and a link to ILM's free code and tools can be found at `http://en.wikipedia.org/wiki/OpenEXR`.

4. *Bilinearly filtered texture mapping*. Implement bilinearly filtered texture mapping for *triangle meshes*. You can use the ship models from the Pipeline project since they have texture coordinates. You will need to interpolate these when you shade a ray that hits a mesh triangle and sample the texture. You may re-use your texture class from the Pipeline project.

5. *Camera depth of field*. A real camera exhibits depth of field effects, such that objects far away from the focal distance are blurry. This can be simulated in a ray tracer using distributed rays. Refer to section 10.11.13 in the book (Shirley et al., Second Edition) for more details.

6. *Spotlights*. Extend your point light source to be a circular spotlight. A spotlight has a direction, a beam angle θ_b, and a falloff angle θ_f, in addition to the usual position and intensity. For directions that make an angle less than θ_b with the spotlight's direction, it produces the same intensity as a regular point light. For directions that are more than an angle of $\theta_b + \theta_f$ from the spot direction, it produces no illumination. In the falloff zone it drops off smoothly according to a C^1 function of angle.

   In the input file an overhead spotlight with a full beam width of 30 degrees might look like this:

```
<light type="Spot">
```

```
    <position>0 10 0</position>
    <intensity>1 1 1</intensity>
    <direction>0 -1 0</direction>
    <beam>15</beam>
    <falloff>5</falloff>
  </light>
```

7. *Propose your own.* You can propose your own extension based on something you heard in lecture, read in the book, or learned about somewhere else. Doing this requires a little extra work to document the extension and come up with a good test case. If you want to do your own extension, email your proposal to the course staff list before Sunday, November 16th.

3 Implementation hints

3.1 Axis-aligned bounding boxes

To implement an AABB you could take the following approach:

1. Create a class that defines an axis aligned bounding box. Design the class as you feel appropriate, but you will likely, at minimum, need methods to: determine if the box intersects a ray, determine if the box intersects another box, grow the box to include a point, and grow the box to include another box. You will also need to extend the `Surface` class to include a method that can grow a bounding box to include each surface type.

2. Design a class to represent a node in the heirarchy. Each node should have a pointer to a bounding box object, pointers to two children and a list of pointers to objects contained in the box.

3. Implement a method of building the heirarchy. The most straightforward approach is to create an AABB that encloses all the objects in the scene and then create a bounding volume node for this box and that includes all objects. Then recursively split this box and its children until the number of objects in each node is less than a constant (usually around 10). To split a node, choose an axis to split along, sort all the objects in the box along the axis and put each half in each of the children. The sort will require that you implement a method of sorting `Surfaces`. A good method is to sort by an approximation of their center: the center for Spheres and the average of the vertices for triangles.

4. Implement a method of traversing the heirarchy and finding the first object intersecting a ray. The method is described both in the lecture notes and in Shirley Chapter 10.

3.2 File format

This assignment will require you to make several extentions to the existing code. Of course, you will need to be able to make test cases that can exercise the new features you will be adding. The framework's `Parser` class is designed to support this type of extension without change, but it requires that you implement the new features in a certain way. The requirements are:

1. Any class that will be instantiated by the Parser must implement a public constructor that takes no arguments.
2. Any class described by a block of xml that includes sub-tags must have public methods called either `setXXX()` or `addXXX` where XXX is the exact name of the sub-tag used in the description. These methods must take exactly one argument. The data will be parsed as if it is the same type as the argument. The `Parser` can correctly parse all primitive types, `String`s, `Color`s and sub-classes of `Tuple3`.

For example, if you wanted the following input to parse correctly:

```
<foo>
    <bar>
        <cat>Lucky</cat>
    </bar>
</foo>
```

You would need to have classes with the minimum definitions:

```
class Foo {
    public Foo() {}
    public setBar(Bar inBar) {
        //Do something with inBar
    }
}

class Bar {
    public Bar() {}
    public setCat(String inName) {
        //Do something with inName
    }
}
```

Finally, the most common case is that you will be adding new shaders, surfaces, or lights. In this case, in addition to the requirements above, the new classes should extend `Shader`, `Surface`, or `Light`. You can then instantiate the correct class using the `type` argument. For example:

```
<shader type="MyShader">
</shader>
```

is the correct way to specify an instance of a new shader class named `MyShader`.

You can find comments with a more detailed description of the Parser in `Parser.java`.

4 Handing in

When you hand in your ray tracer, in addition to the code you need to hand in input files that demonstrate its abilities. A fraction of the grade for this assignment will be set aside for the quality

of your test cases: do they test your features well, so that we can tell for sure that they work, and do the images just look nice. None of the test images should take more than about 10 minutes to compute on a recent PC (such as the ones in the lab). You are required to submit at least 2 test input files for each of the starred extensions you implement above (except we've provided some simple glass scenes).

Also hand in a text file (a page or so) with simple user documentation that explains how to use your program. For example, we need to know how to set any options or parameters that are not set through the input file, and we need to know about any extra extensions you made to the file format.

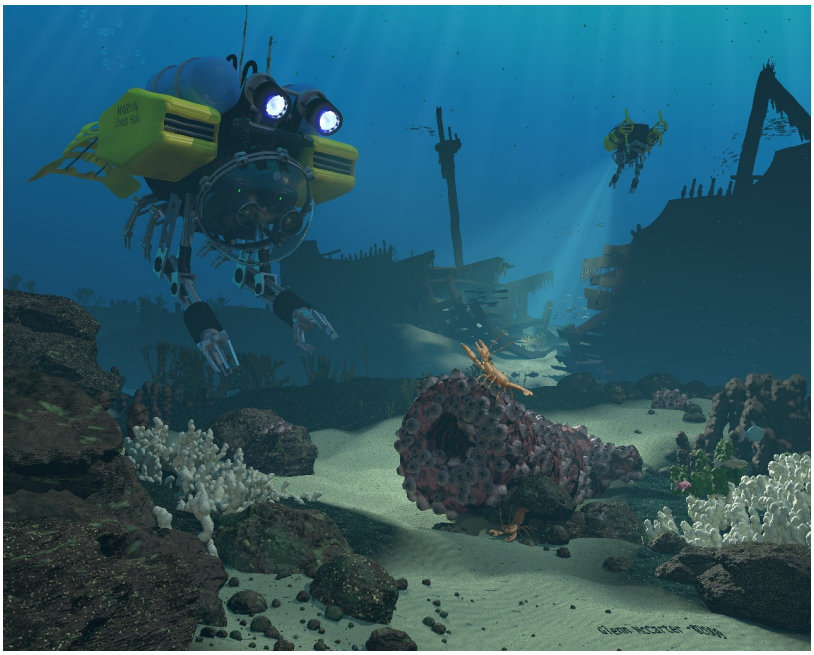

Figure 1: **"Galleon's Reef" by Glenn McCarter** from the Internet Ray Tracing Competition.

Rendering Contest! An exciting part of computer graphics is rendering things never seen of imagined before, and ray tracing contests are a great excuse to do so. To show off the best your program can do, please also submit:

1. one image rendered at high quality and at high resolution (1280 pixels across), and
2. the corresponding XML file used to generate the image.

Make the model interesting, and make the image aesthetically pleasing. As discussed in class, there will be a significant two-homework point credit awarded to the "best image" (on combined technical and aesthetic grounds) we receive, and a one-homework credit to each of two runners-up. There is no limit on CPU time to create your image, so have fun!