CS 465 Program 5: Ray II

(version 3, revised November 23, 2004)

out: 19 November 2004
due: 3 December 2004

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 fancier ray tracer that can handle more substantial models and can produce much more interesting renderings.

This assignment is rather open-ended relative to the earlier ones. You have free rein to design your program in any way you like, but you can also use whatever pieces of earlier assignments are convenient. Obviously the Ray I assignment contains lots of code you can reuse; the other important pieces are in the Model assignment (surface of revolution, file format) and the Pipeline assignment (triangle meshes, texture mapping).

2 Requirements

Your ray tracer will read files in a standard file format and output PNG images (like the first ray tracer). It has to support the basic features given below, plus several extensions that you choose from a list.

2.1 File format

Your ray tracer must support a file format that extends the format used by the Model assignment. This means that if you build a scene with the Model solution and save it, you should be able to read it in and render it with your ray tracer (except that surfaces of revolution will only show up if you chose to implement them).

All extensions to the file format to support ray tracing features should be done using keyword parameters. After the basic parameter list for a particular object, a sequence of parameters can appear, each consisting of a string followed by a value consisting of a sequence of tokens enclosed in square brackets. The value can't contain any brackets itself, so it's easy for a program that doesn't recognize a particular parameter to skip it (by reading the string, deciding it's unsupported, then ignoring the tokens inside the square brackets). For instance, a light source in the old format looks like:

```
Light [(3, 4, 5) (1, 1, 1) (0, 0, 0)]
```

and it could be extended with a radius and a spotlight direction as follows:

```
Light [(3, 4, 5) (1, 1, 1) (0, 0, 0)
       radius [1.2] direction [(1, 0, 0)]]
```

Note that the newline, like other white space, is not significant in this file format.

In order to set the resolution of the image, the camera has an optional parameter `image` with two integer values that are the width and height, respectively, of the image in pixels. For instance, `image [300 200]` sets the image size to 300 by 200 pixels. The aspect ratio is determined by the image size, and the aspect ratio that is written by the modeler should be ignored, as should the near plane and the far plane

On the assignment page you can find a modified version of the parser from the previous assignment, which should be useful as a starting point for your ray tracer's parser.

2.2 Basic features

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

1. Support for cylinders and cubes identical to the ones used in the Model assignment.
2. Groups and hierarchical geometric transformations like those in the Model assignment.
3. Antialiasing by regular supersampling with box filtering. Add an integer parameter `samples` to the camera that controls the number of samples. For example, if the parameter is `samples [2]` then you should use a 2 by 2 pattern, with four samples.

The Lambertian and Phong materials used by the original ray tracer should still be supported, but the modeler only writes out a single color for every object. If no other information is available, your ray tracer should assign a Lambertian material using the given color as the diffuse reflectance. The material can be explicitly set by replacing the single color with a material specification consisting of a string followed by parameters in square brackets. "Lambertian" introduces a Lambertian material, which has just a diffuse color, and "Phong" introduces a Phong material, which has diffuse and specular colors plus an exponent. For instance:

```
Cube [Phong [(0.2, 0.2, 0.7) (0.8, 0.8, 0.8) 40.0]]
```

specifies a cube with a Phong material that is red with a white highlight and an exponent of 40.

2.3 Extension features

Your ray tracer must also implement enough extensions from the following list to add up to 5 units. Features that are likely to take more time are assigned 2 units, and ones that require less effort are assigned 1 unit.

1. *Ray intersection acceleration* (2 units). You should do this using an axis-aligned bounding box hierarchy covering the whole model. If you are also implementing triangle meshes, either they will need to maintain their own hierarchies or your acceleration hierarchy will have to treat the triangles within a mesh as individual surfaces (not as one big surface), otherwise it won't accelerate large meshes.

2. *Triangle meshes and spline revolutions* (2 units). Triangle meshes are specified using a reference to a separate data file, which contains the mesh data in the same simple format that was used by the "mesh" scene in the Pipeline assignment. They can contain vertex normals or not; if they do, then interpolated normals should be used for shading. They can also contain texture coordinates, which should be used for texture mapping if you are also implementing that.

   Triangle meshes are introduced in the input file by the string "TriangleMesh" followed by the filename of the mesh, in brackets. For example:

   ```
   TriangleMesh [ship.msh Lambertian [(0.5, 0.5, 0.5)]]
   ```

   creates a mesh with a gray material.

   The whole point of using triangle meshes, rather than just a scene containing a lot of triangles, is efficiency—so we'll expect reasonable attention to speed and memory usage in your implementation. When you store your triangle meshes in memory, you should not take up significantly more memory than what's required to store the data. This means you can't afford to store arrays of Java objects; you need to store the data as arrays of primitive types. For instance, making an array of Vector3f objects to hold all the vertex positions will use up at least twice as much memory as necessary, because of the pointers and object overhead for all the dynamically allocated objects. You also can't do something like expanding the mesh into separate triangles.

   If you choose triangle meshes, you also need to choose ray intersection acceleration, since nontrivial meshes will be intolerably slow otherwise. The acceleration structure will need to be able to see the individual triangles. One way to do this without undue wastage of memory is to create a "mesh triangle" class that acts as a surface (can intersect rays and compute its bounding box) but actually just consists of a reference to a mesh and a triangle ID. Another (possibly more efficient) way is to have the triangle mesh object present itself to the main acceleration structure as a single surface, and then maintain its own bounding box hierarchy internally to accelerate ray–mesh intersections.

   Spline revolutions (as saved out by the Model assignment) should be converted to triangles (with vertex normals, and with texture coordinates if you support texture maps) in exactly the same way as was done in that assignment, using the tessellation threshold saved in the file. After that, they should be treated the same as any other triangle mesh for rendering.

3. *Loop subdivision surfaces* (2 units). A subdivision surface is specified by a triangle mesh and a subdivision level k. Your program should subdivide the mesh by applying Loop's rules k times, then treat the resulting mesh as a general triangle mesh. In the input file, a subdivision surface is just a triangle mesh with an extra integer parameter "subdivide" that gives the number of subdivision steps:

   ```
   TriangleMesh [octahedron.msh (0.2, 0.5, 0.8) subdivide [3]]
   ```

makes a blobby, generally spherical blue surface.

4. *Dielectrics and glazed materials* (2 units). Implement a dielectric interface material that represents an interface between air (on the outside, the side toward which the normal points) and a denser dielectric. You should also implement a "glazed" material that acts like a thin layer of dielectric over another material—that is, it behaves like a dielectric, but rather than computing and tracing a refracted ray it just calls another material and scales its contribution by the Fresnel transmittance.

   A shader that uses recursively computed rays means that your renderer will generate a tree of rays, which needs to be pruned to keep the program from becoming too slow. In addition to the 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.

   In the input file a dielectric material is specified just like the other materials; its single parameter is the index of refraction. For instance, `dielectric [1.5]` for glass. The glazed material is the same but also expects to see another material for its substrate, as its second parameter:

   ```
   Glazed [1.5 Lambertian [(.4, .5, .8)]]
   ```

5. *Texture mapping* (1 unit). Implement 2D texture mapping based on images. This involves defining texture coordinates for all the geometry types and reading in texture maps that then modulate the diffuse component of whatever shading model you are using. All of the geometry types other than triangle meshes can have their own native (u, v) coordinates; for triangle meshes you should interpolate the texture coordinates from the vertices (if they are not given, the results of texture mapping the mesh are undefined).

   The texture coordinates should be as follows:

   - Cylinder: u goes from 0 to 1 as you go counterclockwise around the surface (viewed from the direction of the +y axis), and $v = (y + 1)/2$. For the end caps the texture coordinates should follow the same rules—v is constant for each cap and u is constant along the radii of the cap.
   - Sphere: u is longitude, going from 0 to 1 as you go counterclockwise around the equator, and v is latitude, going from 0 at the $y = -1$ pole to 1 at the $y = +1$ pole. (Note that v should *not* be $(y + 1)/2$).
   - Triangle mesh: u and v are interpolated from the (u, v) coordinates that are given for the vertices.

   Texture maps should replace the diffuse component of the Lambertian and Phong materials. This means that the original diffuse color is ignored, and the color looked up from the texture is used instead. Textures don't affect dielectrics, which don't have a diffuse component. The glazed material would be textured by texturing its base material, so a texture on the glazed material itself also has no effect.

   In the input file textures are specified by naming an image file. You should be able to read at least PNGs (go ahead and crib from the Pipeline texture mapping code). With texture mapping implemented, all materials have an optional parameter called "texture" that takes a single string as its argument. For example:

```
Lambertian [(.4, .5, .6) texture [image.png]]
```

You can find lots of nice textures on this web site:

http://astronomy.swin.edu.au/∼pbourke/texture/

6. *Fancy point lights* (1 unit). Extend your point light source so that it can be switched between it's old behavior and $1/r^2$ falloff of brightness and can also be used as a spotlight. For the spotlight, the input file specifies a direction and a Phong-like exponent; the intensity of the light is then scaled by $\cos^n \theta$ where θ is the angle between the spotlight direction and the shadow ray.

   In the file format, add optional parameters "falloff" (a boolean), "direction" (a vector), and "exponent" (a float).

7. *Distribution ray tracing* (2 units). Using distribution ray tracing with random sampling, add soft shadows, glossy reflection, and depth of field to your ray tracer. Also alter your antialiasing code to use jittered samples. Details:

   - Area light sources: generalize your point light source by adding a radius, and treat the light source as a disc that faces the shading point. This approximates a spherical source but is much simpler.
   - Glossy reflection: generalize your glazed material by randomly perturbing the surface normal by a vector in the plane perpendicular to the normal. Choose the perturbation uniformly from a circle, or do something smoother (see Shirley's later chapters).
   - Depth of field: generalize your camera by making it generate rays with randomly chosen origins, but always pointing toward the appropriate point on the camera's focus plane.

   These features require a few optional parameters in various places: lights should have a "radius" parameter; glazed surfaces need to have a normal perturbation angle "angle"; cameras need to have an aperture diameter "aperture" and a focus distance "focus". Counter to the tradition in photography, measure the focus distance from the viewpoint rather than from the film plane, because it's simpler that way.

   It's fine for you to use independent random samples to generate your points and directions. But you will consume a lot less CPU time if you use jittered samples instead—but watch out to make sure they are not correlated with the image plane samples. See Shirley, ch. 20.

3 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).

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.

Finally, hand in one image, rendered at high quality and at high resolution (1280 pixels across) that shows off the best your program can do. Make the model interesting, and make the image aesthetically pleasing. We will award 10 extra credit points to the best image (on combined technical and aesthetic grounds) we receive, and 5 points to each of two runners-up.