Assignment 5+6: Ray Tracing a Scenegraph

Out: 16th November 2016
Assignment 5 due: 1st December 2016 (Thursday) at 11:59pm
Assignment 6 due: 7th December 2016 (Wednesday) at 11:59pm

1. **Read the assignment carefully! Pay attention to every instruction to save time.**
2. **Starting early and implementing it part-by-part as we discuss in class is the key to finishing this assignment on time and enjoying it!**

This write-up provides guidelines for Assignment 5 and 6 with clear boundaries for each. In assignment 5 you will implement a basic ray-tracing program. This program will be capable of rendering a 3D scene made of boxes and spheres with shading effects but no textures. In assignment 6 you will extend this program to include texturing, shadows, and reflective and transparent objects.

Step 1: Laying the foundation:

It would be useful to have the following small helper classes in your program to perform frequent operations:

1. 3D ray: contains a starting 3D point and a direction as a 3D vector.
2. HitRecord: Stores all the information that you will need to determine the closest object that was hit, and information about that object to compute color.

You may begin by creating the above classes before proceeding.

Step 2: Setting up the basic ray tracing:

Modify your Assignment 4 program so that it switches between OpenGL and raytracing modes. This will allow you to view a scene in OpenGL before ray tracing it. The output of the ray tracer should go to an image, not on the screen window.

Look at the overall design of your scene graph classes, and think about where a ray tracer would fit in. Note that a ray tracer provides an alternative to OpenGL to draw a scene graph.

Your program should have these functions somewhere:

2.1 raytrace(int w,int h,modelview)

Write a raytrace function that starts the ray tracing and exports the result accordingly. It should do the following:

For every pixel:

1. Compute the ray starting from the eye and passing through this pixel in view coordinates.
2. Pass the ray to a function called “raycast” that returns information about ray casting.
3. Write the color to the appropriate place in the array.

2.2 raycast(ray, modelview)

Write a function raycast() that takes a ray and the modelview stack. This function will determine if this ray hits anything in the scene graph computes the color accordingly.

It does the following:

1. Pass on the ray and the modelview stack to the root of the scenegraph.
2. If the ray did hit anything, call a function **shade** and pass to it all the relevant information from the hit record that the root returns, that will be useful in calculating the color of this pixel.

3. If the ray did not hit anything, return a background color.

Step 3: Computing the ray-object intersections:

The objects are in the leaves of your scene graph. You must write a function that descends the ray down the scene graph and determines the closest point of intersection with an object (if it exists). It must accordingly collect and return the relevant information from the hit that will help in computing the color.

Carefully think about how to implement this in the scene graph, which has several types of nodes. Hint: the `draw` function may inspire you.

How to test:

Start with a very simple scene (one solid, untransformed) and place the camera on the Z-axis looking directly at it. If the raycast returns that it was hit, write a white pixel. This will allow you to test whether your intersection code works by allowing you to step through and calculate expected answers manually. Now move the camera and test, and finally add several objects to test.

Step 4: Shade function

Write a shade function that takes all appropriate parameters necessary for it to do lighting. It must perform the same lighting as the shader you used in Assignment 4 (point/directional and spot light sources, ambient, diffuse and specular lighting), but without texturing. This involves essentially replicating your lighting shader in Java/C++.

The easiest way to test whether this is working is to switch between OpenGL and ray tracing modes and verify that the pictures are identical. You may find it better to test only ambient first, followed by ambient+diffuse and finally all three effects.

Extra credit (5 points)

To get 5 extra points (above what you will get for the same thing in Assignment 6), complete Step 5 of Assignment 6 in this Assignment. This is an all-or-none opportunity: you either get 5 points for working textures, or you get none.

Assignment 5 ends here.

What to submit

Program setup

Before you submit, please set up your program to read the provided input file "scene.xml" and the camera at (70, 100, -80) looking at (0,0,0). Doing this will expedite grading.

Submit your IntelliJ/Qt project with input model files and a high resolution ray-traced image (at least 800 by 800) of a scene that you have created. **You may submit several screen shots to show the features you have completed.** Set up your code so that the camera is set up for your input file.

Assignment 6

Step 5: Textures:

Texture coordinates can be part of what the leaf returns in the HitRecord if it was hit by a ray (you would need to determine them yourself). Use the TextureImage object to determine the pixel color from the texture.

Step 6: Shadows:

A point is in shadow if the light cannot see it. Conversely, if you place a camera at the point and look towards the light, you won't see the light.

In the shade function, before performing the shading calculations for each light, create a shadow ray from the point towards that light, and see if the point sees the light. If it does not, simply ignore that light in the shading calculations for that point. Don't forget to "fudge" the shadow ray a bit to avoid precision errors!

Step 7: Reflections:

The provided Material class has variables to store the absorption, reflection and transparency of a material. The XML parser already parses the relevant tags and adds this information to the Material object. The sum of absorption, reflection and transparency should be 1 for every object. **By default, a material is fully absorbent (i.e. reflection=transparency=0, absorption=1).** See scene.xml for an example.

In raycast(), after the shade function returns the color of the pixel, check if the object is reflective. If so, create a reflection ray and use raycast recursively to determine the color of the pixel. Finally, blend the color returned by shade and the color returned by this recursive call using the absorption and reflection coefficients of the material.

In order to prevent infinite recursion, add a parameter called 'bounce' to the raycast function. All recursive calls should be made by one more than this value. Introduce a condition at the beginning of raycast that simply returns the background if the bounce exceeds a maximum threshold (a value of '5' is more than sufficient).

Extra credit opportunities

Refraction (Extra credit 10 points)

Similar to spawning rays for reflection, spawn a ray for transmission if the object is not fully opaque (i.e. its transparency is greater than 0). Use Snell's law to compute the refraction ray as discussed in class. The final color of the pixel is a blend of its own color, the color determined by reflection and refraction. The blend is proportional to the object's absorption, reflection and transparency. Make sure that you specify these constants such that they add up to 1.

Remember that in order to calculate the correct refraction rays, you must remember the refractive index of the material that you are currently in (the refractive index of "air" is 1, so all rays start from the camera at this value). This would be another parameter to the "shade" function.

Creative modeling (Extra credit 5 points)

Use your creativity by creating an XML model showing your model and your ray tracer capabilities! Remember that you can make ellipsoids from spheres as well! Look at some outputs from previous years at http://www.ccs.neu.edu/home/ashesh/hall-of-fame.htm . Needless to say, a "close-enough" copy of any of these will earn no points.

THIS IS AN "ALL OR NONE" EXTRA CREDIT. That is, you either get 5 points or you get none. Your file must contain at least 10 objects, and must be meaningfully arranged. At least one sphere and box must be textured.

YOU MAY RECEIVE PARTIAL CREDIT FOR THE FOLLOWING:

Supporting cylinders (Extra credit 5 points)

Write the intersect function for the cylinder object so that it returns the correct hit record, along with texture coordinates.

Supporting cones (Extra credit 5 points)

Write the intersect function for the cone object so that it returns the correct hit record, along with texture coordinates.

In order to prove that you have done any of the extra credit, we expect you to include a ray traced image for each feature. We will not run your ray tracer to check each feature. We will rely on a screen shot you have submitted and a reading of your code to determine extra credit.

What to submit: Submit your IntelliJ/Qt project, texture files (images) and input model files that you have created. **You may submit several screen shots to show the features you have completed.**

Also submit interesting screen shots for the Hall of fame if you make it!