Assignment 7+8: Ray Tracing a Scene graph

Assignment 7 due: 9th April 2019 at 11:59pm
Assignment 8 due: 18th April 2019 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 7 and 8 with clear boundaries for each. In Assignment 7 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 and textures. In Assignment 8 you will extend this program to include 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 calculate shading. More specifically, it will contain (a) the time 't' on the ray where it intersects an object (b) the 3D point of intersection in view coordinates (c) the 3D normal of the object at that point in view coordinates (d) The material properties. (e) Texture coordinates and a Texture object, if applicable.
3. Create small scenes with single or a few objects for testing purposes. Do not try to test directly on a big scene!

You should begin by creating the above classes before proceeding.

Step 2: Setting up the basic ray tracing:

Modify your Assignment 6 program so that it switches between OpenGL and raytracing modes, possibly with a keypress. 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.

2.1 raytrace(int w,int h,modelview stack)

Write a raytrace function that creates and exports an image file, as an output to ray tracing (look at the design of your program to see where your raytracer, and this function would fit well). The modelview stack currently contains the camera transform that is set in the View::draw method (as with the OpenGL rendering). 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()

Create the function raycast(…) that takes the following parameters: the ray to be cast in the view coordinate system and the modelview stack. This function will compute the resulting color. It returns true/false depending on whether the ray hit some object in the scene or not.

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 the background color.

Step 3: Computing the ray-object intersections:

1. Add a method to your node classes that will determine and compute the intersection of a ray with various nodes. It should also populate the HitRecord with the **nearest** point of intersection, the normal at that point **(in the view coordinate system)**, material and texture properties.

Define this function:

a. For the leaf node, identify the object associated with the leaf (Box or Sphere) and process the ray accordingly.
b. For the Group and Transform nodes, it merely passes everything to each of its children after altering the modelview suitably.

When this function returns from the root, it should have information about the closest intersection, and other relevant information.

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 true, 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 6 (point/directional and spot light sources, ambient, diffuse and specular lighting), <u>but without texturing</u>. This involves essentially replicating your lighting shader.

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. <u>You may find it better to test only ambient first, followed by ambient+diffuse and finally all three effects.</u>

Step 5: Textures:

The intersect functions of the respective objects should also return the appropriate texture coordinates for the point of intersection in the HitRecord. Use the TextureImage object to determine the pixel color from the texture.

Assignment 7 ends here.

What to submit: Submit your IntelliJ/Qt project with input model files. Also submit:

1. An image showing the final rendering (the best output) of your ray tracer (at least 800x800).
2. An image showing the OpenGL rendering of the same scene and the same camera position as your final raytraced image above. This will allow us to quickly verify that everything works correctly.
3. An image showing that your ray tracing works for spheres (a black and white image, or the final rendering)
4. An image showing that your ray tracing works for boxes (a black and white image, or the final rendering)
5. An image showing that lighting works (a suitable image, or the final rendering)
6. An image showing that your texture mapping works (a suitable image, or the final rendering)

Assignment 8

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 input.txt 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 project, texture files (images) and input model files that you have created. **Also submit interesting screen shots for the Hall of fame if you make it!**