Hw6: (Baby) RayTracer

Out: Thu Nov 8
In: Thu Nov 20, 11:59pm

1. Introduction

So far in this course we have written code that manipulates 3D shapes and cameras to prepare a scene for rendering. These preprocessing steps make your scene ready to be rendered by one of the most popular APIs out there today: OpenGL. The support code is, in fact, a wrapper around OpenGL libraries and hardware.

OpenGL uses a traditional rendering pipeline, which is why it requires that you break up your scene, including curved surfaces, into polygons (ie, approximate perfect curves with flat triangles). Flat polygonal approximations, however, don't look so nice at the end of the day.

OpenGL also uses a Phong illumination model, the simple lighting model we discussed in class (remember that OpenGL uses a Phong lighting model to color vertices, and a Gouraud shading model to fill in all of the interior pixels). We are really only using the diffuse and ambient terms of that simple illumination model. Unfortunately, because the OpenGL renderer is tightly linked with the hardware, it is pretty difficult to extend to more sophisticated models of illumination.

These are serious limitations.

In this assignment, we are giving you the opportunity to circumvent these limitations by writing your own renderer: a baby ray-tracer. By doing so you will see a glimpse of what you could do with a rendering algorithm that stresses quality over speed. Curved surfaces are possible, and everything has a sort of "perfect" feel to it.

Note: There are a few things that our renderer will not handle (but could, with a bit more work), such as specular reflections.

2. Approach

One of the real advantages of ray-tracing is that you don't have to work with approximations to the objects in your scenes. When your objects are defined by an implicit equation, you can render that object directly with a resolution as high as your image allows. For this assignment, you only need to be able to render spheres. Basically this means that you will use math to solve exactly where a ray intersects a sphere instead of approximating the sphere with a lot of triangles and sending it to GL.

You will be using a limited, non-recursive illumination model. We only expect you to handle the ambient and diffuse lighting terms of the simple illumination model (no attenuation or shadows). Just to make everything absolutely clear, you do NOT need to handle attenuation, directional lights, shadows, texture mapping, specular lighting, or reflection in this assignment.

Foley, van Dam et al give out a lot of equations for various kinds of illumination, but here is the one we expect you to use for this assignment (there are no recursive or specular terms and you are not expected to take light attenuation or shadows into account). This is slightly different from the equation in class so please use this one:

CS1566 Homework 6 – RayTracer

$$I_\lambda = k_a O_{a\lambda} + \sum_{m=0}^{numLights-1} \left(I_{m\lambda} [k_d O_{d\lambda} \vec{N} \bullet \vec{L}] \right)$$

I_λ = final intensity for wavelength λ; in our case the final R, G, or B value of the pixel we want to color

k_a = global intensity of ambient light; ka in the support code

$O_{a\lambda}$ = object's ambient color for wavelength λ; in our case the object's R, G, or B value for ambient color

m = the current light; to compute the final intensity we must add up contributions from all lights in the scene

numLights = the number of lights in the scene;

$I_{m\lambda}$ = intensity of light m for wavelength λ; in our case the R, G, or B value of the light color for light m

k_d = global diffuse coefficient

$O_{d\lambda}$ = object's diffuse color for wave length λ; in our case the object's R, G, or B value for diffuse color

N = the unit length surface normal at the point of intersection; this is something you need to compute

L = the unit length incoming light vector; make sure this vector is oriented in the correct direction

You will want to use this equation to compute the R, G and B values independently for the current image pixel. So that is, you need to compute one I_{red}, one I_{green}, and one I_{blue}. Note that you need to figure out which object the current ray intersects with before you can use this illumination equation. Please make sure you understand the concepts in lecture and hw5 before attempting to code the assignment.

You will also need to compute N and L at the intersection point. Hints: Since we only ask you to work with spheres, your N could be the intersection point (in world coordinates) minus the current (world) center of the sphere. Remember to normalize! Similarly, L would be the light position (in world coordinates) minus the intersection point (in world coordinates). Easy peasy.

If you'd like to raytrace cylinders, cones, etc. for EC, use a similar approach to figure out the normal at the intersection point.

Finally, here is the high-level pseudo-code for a non-recursive ray-tracer. You obviously need to implement this code, where Scene stand for the scene objects (spheres, light sources, camera) contained in the spec file, and Canvas is an RGB pixel array of let's say size 256 by 256 (you may go up to 512 by 512 if you wish, but please keep it a power of 2 and reasonably small).

```
Ray-Trace(Scene,Canvas)
1 for each point in Canvas
2     do cast a ray to find the nearest object
3     if ray intersects an object
4           then for each light that illuminates the point
5                 do Canvas[pt] ← color with only diffuse/ambient components
6     else Canvas[pt] ← background color
```

3. Algo and Support Code

As you probably have noticed already, there is no Algo part for this assignment, and the Support Code is truly just an example of parsing a spec file and of creating in your program and transferring to the frame buffer an image array. So long, OpenGL renderer, vertices and polygons.

You are finally free from the tyranny of earlier assignments (except the know-how of homework 5). All you need for your baby ray tracer is to know how to set up a camera, build rays, how to intersect spheres with rays, and how to compute the normal at the intersection point – and you don't need tessellated spheres for that. Go out and have fun. Write your own code, from scratch if you wish.

We do provide a few spec files. Their structure is very similar to the structure of your hw4 and hw5 spec files: Here is an example specification file (Some rather long lines have been split into 2 lines so that it would fit on one page. Your program does not necessarily have to handle object definitions that span more than one line. It sets up one sphere and two lights. It also sets up the camera. You will need to let the user decide which specification file to parse and display. You will also need to parse the file they specify.

```
# object definition:

# The next 2 lines should really fit on 1 line; rotation given in degrees

# object_type (tx ty tz 1) (sx sy sz 1) (rx ry rz 1)

#    (mat_shiny) (mat_emission) (mat_amb) (mat_diffuse) (mat_specular)

# where object_type is always = 3 (a sphere)

# The next 2 lines should really fit on 1 line.

3 (2 0 10 1) (1 10 1 1) (0 0 0 1)

        (5) (.2 .2 .2 1) (0 0 0.2 1) (0.5 0.5 0.5 1) (0.8 0.8 0.8 1)

# light definitions;

# l (amb) (diffuse) (specular) (pos) [optional (dir) (angle)]

l (1 1 1) (0.4 0.4 0.4) (0.5 0.5 0.5) (0 0 0 1) (0 0 .1 0) (25)

# l (amb) (diffuse) (specular) (pos) [optional (dir) (angle)]

l (1 1 1) (0.4 0.4 0.4) (0.5 0.5 0.5) (-10 0 10 1) (0 0 .1 0) (25)

#camera definitions:

#c (posx posy posz 1) (lookAtx lookAty lookAtz 1) (upx upy upz 1)

c (0 0 -20 1) (0 0 0 1) ( 0 1 0 1)
```

FAQ: My raytracer seems to be running slowly, what should I do? Speeding up raytracing is difficult. Usually, it is not enough to optimize inner loops, as there is just too much work that needs to be done for ray creation, intersection testing and illumination. You need to cut away large amounts of this work. Check out some of the optimization suggestions in the extra credit section.

Please use whatever tricks you need to debug your program without waiting for it to fully render the entire scene in perfectly computed color for each debugging iteration; this should save you some time. You may want to start by just writing a white (as opposed to colored pixel) for each intersection you get. You might want to ray-trace just the small central part of the scene in the beginning, just to make sure you get something on the screen at the correct location. **Remember that a program that produces no image will get you no points; you want to build your program incrementally, and be able to document its output.**

4. Expected Behavior and Grading

Your program should take as argument the name of a spec file (the grader needs to be able to test your code on a variety of spec files.)

Depending on your coding abilities, your code may run remarkably slow. For this reason, we ask you to grab and submit a snapshot of your ray-traced outputs for each spec file, and submit those image files (.jpg format) along with your code. The staff will still test and verify your code against a spec file of their choice.

Ray-Traces correctly unit center sphere spec file (shape)	20 pts
+ Illumination values are computed correctly (i.e., colors!)	15 pts
Ray-Traces correctly world sphere spec file	10 pts
+ Illumination values are computed correctly (i.e., colors!)	10 pts
Ray-Traces correctly multiple object spec file	10 pts
Ray-Traces correctly multiple lights spec file	10 pts
+ Illumination values are computed correctly (i.e., colors!)	10 pts
Program completes in reasonable time	10 pts
README completed	5 pts
Total	**100**

5. Extra Credit

- Effective optimizations: Think about how to reduce the overall number of intersection tests required for a scene. The biggest speed gain can be found by making a "bucket" for each pixel (or small group of pixels) that stores what objects could possibly lie "underneath". This involves a precomputation step where object bounding boxes are projected into screen coordinates (think backwards mapping!). You can also put 3D bounding cubes or spheres around master objects so that you don't have to check every sub-object if the ray is nowhere close to the master object. The first method will help a lot, since rays don't bounce around in this assignment. The second method will help even with bouncing rays, but actually makes things a bit slower for small scenes. A slightly more complicated, but better than bounding spheres/cubes, solution is to partition your scene with an octree.
- Multithreading: if you make your code multi-threaded and then find a multi-processor machine you can have multiple threads raytracing different parts of the image concurrently.
- Other implicitly-defined shapes: cones, cylinders etc.
- Antialiasing: You can shoot out multiple rays per pixel to get less aliasing (supersampling). Supersampling will produce better results, but it's much slower. Instead of brute force supersampling, try adaptive supersampling for a speed boost.
- Do a full-fledged ray-tracer now.