Assignment 3: Intersect

Comp175: Introduction to Computer Graphics – Fall 2022

Algorithm Due: **Friday March 28th** at noon (12:00pm)
Project due: **Monday March 21th** at midnight (11:59pm)

1 Introduction

Throughout this semester, you have written code that manipulated shapes and cameras to prepare a scene for rendering. These preprocessing steps made your scene ready to be rendered by one of the most popular APIs out there today: OpenGL. The support code is a wrapper around OpenGL libraries and hardware.

OpenGL uses a Phong illumination model[1], a simple lighting model explained in the lecture slides. In addition, OpenGL uses a traditional rendering pipeline. It requires that you break up your scene into polygons. Then it takes each polygon, determines which pixels that polygon affects, and paints those pixels according to its illumination model. These operations can be done quickly in hardware. Unfortunately, because the renderer is in hardware, it is difficult to extend[2]. Furthermore, the renderer requires that you create a discrete approximation to your scene, by approximating perfect curves with flat triangles.

In this assignment, you will have the opportunity to break away from these limitations by writing your own renderer implementing a basic ray tracer. The result will be a rendering technique that looks better and is more extensible than OpenGL.

From now on, we're leaving OpenGL behind[3]. When you are done with Assignment 4, you will have realized that you have mastered 3D rendering because you have implemented every step of the process (geometry, camera, rendering) yourself!

2 Requirements

2.1 The Raytracing Pipeline

In lecture you learned the basic steps of the ray tracing pipeline's inner loop:

- **Generating rays** (simplest case: for `Intersect`, just shoot a ray through the center of each pixel)
- **Finding** the closest object along each ray
- **Illuminating** samples; for `Intersect` you are only required to support basic lighting

2.2 Implicit Equations

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. You need to be able to render the following objects using their implicit equations: `Cube`, `Cylinder`, `Cone`, and `Sphere`. Basically this means that you will use math to solve exactly where a ray intersects a cone instead of approximating the cone with a lot of triangles and sending it to OpenGL.

2.3 Simple Illumination Model

Like OpenGL, you will be using a limited illumination model. We only expect you to handle the ambient, diffuse, and specular lighting terms of the simple illumination model (no attenuation or shadows yet). In this sense, your rendering will look a lot like the output from `Sceneview`, but likely with better shading. You should however leave room in your design for a more complex model of illumination. The next assignment, `Ray`, will extend what you do in this assignment to handle a recursive illumination model.

The global lighting coefficients can be found in `SceneData.h`. You should pay attention to this struct:

```
// Scene global color coefficients
struct SceneGlobalData {
     float ka; // ambient
```

[1] OpenGL uses a Phong lighting model to color vertices, and a Gouraud shading model to fill in all of the interior pixels.

[2] Although this has changed thanks to the advances in programmable hardware. We will revisit this in future labs.

[3] The only OpenGL call that remains is to put an array of pixels onto the screen. Technically we don't need OpenGL for thta, but OpenGL still does that faster than most.

```
    float kd; // diffuse
    float ks; // specular
    float kt; // transparent
};
```

In the lighting equation, `ka` is k_a and `kd` is k_d. You do not have to worry about and `kt`; this will be used as extra credit in `Ray`, the next assignment.

We give out a lot of equations for various kinds of illumination, but here is the one we expect you to use for `Intersect`. Notice that there is recursive term 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

$$I_\lambda = k_a O_{a\lambda} + \sum_{i=1}^{m} \left[l_{i\lambda} (k_d O_{d\lambda} (\hat{N} \cdot \hat{L}_i) + k_s O_{s\lambda} (\hat{R}_i \cdot \hat{V})^f) \right]$$

Here,

- 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 = the global intensity of ambient light; `SceneGlobalData::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, `SceneMaterial::cAmbient` in the support code
- m = the number of lights in the scene; `SceneParser::getNumLights()` in the support code
- $l_{i\lambda}$ = intensity of light i for wavelength λ; in our case the R, G, or B value of the light color for light m, `SceneLightData::color` in the support code
- k_d = the global diffuse coefficient, `SceneGlobalData::kd` in the support code
- $O_{d\lambda}$ = object's diffuse color for wave length λ; in our case the object's R, G, or B value for diffuse color, `SceneMaterial::cDiffuse` in the support code
- $\hat{N}$ = the unit length surface normal at the point of intersection; this is something you need to compute
- $\hat{L}_i$ = the unit length incoming light vector from light m; think how this might change depending on whether the light is a point or directional light, also make sure this vector is oriented in the correct direction
- k_s = the global specular coefficient, `SceneGlobalData::ks` in the support code
- $O_{s\lambda}$ = object's specular color for wave length λ; in our case the object's R, G, or B value for specular color, `SceneMaterial::cSpecular` in the support code
- $\hat{R}_i$ = the unit length reflected light from light i
- $\hat{V}$ = the normalized line of sight
- f = the specular component (often referred to as "shininess")

You will want to use this equation to compute the R, G and B values independently for the current image pixel. Note that you need to figure out which object the current ray intersects with before you can use this illumination equation. Keep in mind that this assignment is math heavy and it's easy to get confused. Please make sure you understand the concepts in lecture before attempting to code the assignment.

Note that for this assignment, you only need to worry about "spot lights." In other words, you don't have to implement directional lighting. We will leave that requirement for your next assignment.

2.4 Design

As usual, we have provided for you some support code that is built upon the previous assignments. However, it is relevant to note that this new approach of rendering requires little to no OpenGL! All the math and the matrix manipulation are done in software (in the code that you have written or will be writing). Our only use for OpenGL/FLTK is to get a 2D canvas and to have some GUI control.

When you compile the support code, you will see that there are now two buttons. The first is to "Load" the xml file, and the second, "Render!" is to start the ray tracing. Note that the use of the "isectOnly" checkbox is a convenient debugging mechanism. I suggest that you start with getting intersection to work right by turning on isectOnly (a good example input file for testingis "data/general/isect.xml"). Once intersection works, you move on to implementing lighting.

As with all of your assignments, you should be reusing code from previous assignments (and labs). This includes, but is not limited to, your camera, shapes, intersection code for spheres from the lab, etc.

3 New Scene Files

You will notice that in addition to the xml scene files that you used in Assignment 3, we have provided a new directory of scene files. These scene files are specifically designed for testing recursive ray tracing. You will be using them again in the next assignment.

4 FAQ

4.1 Is my specular highlighting working?

One way to test it is to run the test file "data/recursiveRayDemo/shadow_test1.xml". If your specular highlighting is not working, you will see a pitch-black sphere. If it's working, you will see shading at the bottom right of the sphere (as shown in the demo).

4.2 I am getting some crazy rainbow colors, what's going on?

If you set a pixel's (r, g, b) value to be above 255 in any of the channels, you can get into negative numbers, which is interpreted in unpredictable ways. So don't forget to cap your colors!

4.3 There are random (black) dots, why?

Chances are you need to look into changing your EPSILON value. EPSILON is used in a variety of ways because we cannot check for equality between two floating points. So changing EPSILON effectively means changing your evaluation of whether two floating point values are the same.

4.4 My objects seem to be more translated than the demo?

This is likely happening because when you turn vector of vec3 into a vector of vec4, you added a 1.0f as the fourth value (when it should have been a 0.0f). Recall that if the 4th value is a 1.0f, then the translation component of a matrix would have an effect. In other words, if you use 1.0f as the 4th value of your vector, you are translating the vector (which obviously should not be allowed).

4.5 My raytracer seems to be running slowly

Speeding up ray tracing 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. For each ray you shoot through the film plane (i.e. for each pixel in your output image) you must test for intersections for each object in your scene. Thus, as the number of objects in the scene increases, the time it takes to render the scene increases exponentially. For extra credit, you can address this problem by implementing a spatial (bounding volume) acceleration data structure. We recommend starting with a basic `octree` and then extending that `octree` to a `kd-tree`, if you have time. Look up the surface area heuristic online or see the lecture slides on spatial acceleration for more information.

4.6 Nothing shows up

Much like the other assignments, "Nothing Shows Up" will be a very common problem when you first start the assignment. I would recommend: (a) start with a simple scene graph with only one object located somewhere down -z; (b) don't worry about colors yet, start with casting rays and intersecting them with objects (by turning on the isectOnly checkbox); (c) focus on one geometry at a time. Get one shape to work before moving on to another.

5 Extra Credit

Here are some examples of extensions you can do for extra credit. Note that we won't give out double extra credit between `Intersect` and `Ray`. If you do multithreading now, for example, you'll get extra credit for it on `Intersect`, but not on `Ray`. Other extra credit options will be offered on `Ray` (in addition to these), so you can do extra credit both times (which is clearly the best option!).

- **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 precompution 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 (like a chess piece) 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 in `Intersect` (where rays don't bounce around), but not as much in `Ray`. The second method will help with both `Intersect` and `Ray`, 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`. Better yet, you can extend your `octree` to divide at an optimal split position (vs. at the midpoint each split); an `octree` implemented in this way is known as a `kd-tree`. Having an `octree` or `kd-tree` will really pay off in `Ray`.

- **Multithreading**: if you make your code multithreaded and then find a multiprocessor machine you can have multiple threads ray tracing different parts of the image concurrently.
- **Other implicitly-defined shapes**: the torus, for example, is described by a quartic on x, y, and z.
- **Antialiasing**: you might try rendering, edge detecting, and then blurring the edges that were detected. You can also shoot out multiple rays per pixel to get less aliasing (supersampling). Supersampling will produce better results than blurring edges (you should know why by now), but it's much slower. Instead of brute force supersampling, try adaptive supersampling for a speed boost.
- **Intersect polygonal meshes**: Wouldn't it be nice to see the cow ray traced? The cow is composed of a lot of triangles, so you may want to think about some of the optimizations you can do. There are a couple of famous techniques you can implement.

6 How to Submit

Complete the algorithm portion of this assignment with your teammate. You may use a calculator or computer algebra system. All your answers should be given in simplest form. When a numerical answer is required, provide a reduced fraction (i.e. 1/3) or at least three decimal places (i.e. 0.333). Show all work.

For the project portion of this assignment, you are encouraged to discuss your strategies with your classmates. However, all team must turn in ORIGINAL WORK, and any collaboration or references outside of your team must be cited appropriately in the header of your submission.

When finished, hand in the worksheet and assignment on Canvas.