

# Assignment 6: Intersect

Due Friday, November 13, 2009 at 11:59pm  
Help Session on Thursday, November 5, 2009 at 7:00pm

## 1 Introduction

### 1.1 The Old Way: Rendering with OpenGL

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<sup>1</sup>, the simple lighting model in the lecture slides. The support code is set up to use diffuse surfaces, so we are really only using the diffuse and ambient terms of that simple illumination model.

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<sup>2</sup>. Furthermore, the renderer requires that you create a discrete approximation to your scene (ie, approximate perfect curves with flat triangles).

### 1.2 The New Way: Rendering with Raytracing

Finally, we are giving you the opportunity to break away from these limitations, to free yourself from the bondage of someone else's swilly code. You are going to write your own renderer. You want a rendering technique that looks better and is more extensible than OpenGL. For these reasons you will implement a basic raytracer for this assignment.

## 2 Requirements

### 2.1 The Raytracing Pipeline

In lecture you learned the basic steps of the raytracing 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**

---

<sup>1</sup>OpenGL uses a Phong lighting model to color vertices, and a Goraud shading model to fill in all of the interior pixels

<sup>2</sup>Though this is changing thanks to the rapid advance of programmable hardware

## 2.2 Implicit Equations

One of the real advantages of raytracing 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 GL.

## 2.3 A Simple Illumination Model

Like OpenGL, you will be using a limited illumination model. We only expect you to handle the ambient and diffuse 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. 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.

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.

# 3 Administrative Support

## 3.1 Demo

The demo for this project is `/course/cs123/demos/ray`. You should run this program as well as your own program on as many scene files as you can. Scene files that we provide are in `/course/cs123/demos/data/scene`. You should test all of the files in the “general” and “intersect” directories. Writing your own scene files can help as well because the provided scene files may not test all possible cases.

## 3.2 Support Code

Your Intersect program must be able to take the name of a valid scenefile on the command line. It must also start properly if no scenefile is specified.

When the program begins, you will want to bring up a subclass of RayCanvas, into which you will be rendering. The interface for RayCanvas contains the following methods:

```
RayCanvas(int width, int height);  
virtual void render() = 0;  
virtual void render(const CS123Rectangle&) = 0;  
virtual void loadScene(const char *sceneFile) = 0;
```

When the user clicks on the “Render” option in the “Ray Options” menu, the `render()` method of RayCanvas will be called. Override this method in your subclass to begin rendering.

You may also render a rectangular region of the screen (similar to filter) by dragging a marquee selection rectangle over the area of interest. `loadFile` should replace the current scenefile in memory with a new one, given a filename.

We also provide you with an implementation of a spatial acceleration data structure. The idea is, you can use this spatial acceleration data structure to help you render scenes faster. This will allow you to do extra credit, such as supersampling, and not do extra credit to accelerate rendering, such as implementing a kd-tree, and have your scenes render in a reasonable amount of time. Of course, you are all strongly encouraged to implement your own spatial acceleration data structure. You may also use the interfaces we provide you to aid in the design of your own data structure. We provide you with the following interfaces:

```

class CS123Intersectable {
    bool getIntersection(const double *dir,
                        const double *start,
                        double &intersectT);
    bool hasIntersection(const double *dir,
                        const double *start);
    void getBounds(double &minX, double &maxX,
                  double &minY, double &maxY,
                  double &minZ, double &maxZ);
};

class CS123SpatialAccel {
    void build(const CS123IntersectableList &intersectableList);
    bool findFirstIntersection(const double *dir,
                              const double *start,
                              double &firstT,
                              CS123Intersectable *&firstObject);
};

class CS123SpatialAccelFactory {
    static CS123SpatialAccel *createOctTree(unsigned maxRecursionDepth,
                                             unsigned minObjectsPerNode);
};

```

The corresponding header files can be found in `/course/cs123/include/accel`. The comments in the header files explain how these interfaces should be used.

There's something else you should pay attention to in the `CS123SceneData.h`:

```

// Scene global color coefficients
struct CS123SceneGlobalData
{
    float ka; // ambient
    float kd; // diffuse
    float ks; // specular
    float kt; // transparent
};

```

In the lighting equation,  $k_a$  is  $k_a$  and  $k_d$  is  $k_d$ . You do not have to worry about  $k_s$  and  $k_t$  for now. You can populate the global scene data for a given scene by calling the scene parser's `getGlobalData` method defined in `CS123XmlSceneParser.h`.

We give out a lot of equations for various kinds of illumination, but here is the one we expect you to use for intersect (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:**

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

$I_\lambda$	=	final intensity for wavelength $\lambda$ ; in our case the final R, G, or B value of the pixel we want to color
$k_a$	=	global intensity of ambient light; CS123SceneGlobalData::ka in the support code
$O_{a\lambda}$	=	object's ambient color for wavelength $\lambda$ ; in our case the object's R, G, or B value for ambient color, CS123SceneMaterial::cAmbient in the support code
$m$	=	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; CS123XmlSceneParser::getNumLights() in the support code
$I_{m\lambda}$	=	intensity of light $m$ for wavelength $\lambda$ ; in our case the R, G, or B value of the light color for light $m$ , CS123SceneLightData::color in the support code
$k_d$	=	global diffuse coefficient, CS123SceneGlobalData::kd in the support code
$O_{d\lambda}$	=	object's diffuse color for wave length $\lambda$ ; in our case the object's R, G, or B value for diffuse color, CS123SceneMaterial::cDiffuse in the support code
$\hat{N}$	=	the unit length surface normal at the point of intersection; this is something you need to compute
$\hat{L}_m$	=	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

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.

Please make sure you understand the concepts in lecture before attempting to code the assignment. This assignment is math heavy and it's easy to get confused. If you get confused you'll just end up throwing EPSILONs everywhere and that will make things even worse.

### 3.3 Design

To help with your design, we are providing some of it! There is an abstract class defined in `CS123SpatialAccel.h` for a spatial acceleration data structure (as described in the raytracing lecture). We suggest you subclass and implement this interface for intersect, as it will make it much easier for you to swap in an optimized, super-fast datastructure later on. Plus, it can make testing and debugging easier. Note that you are not required to implement anything fancier than a simple linear search, though it makes fantastic extra credit.

### 3.4 Other Notes

As with sceneview, you should *not* copy your entire linear algebra package over into your intersect project. If you make changes to your linear algebra package specifically for intersect, please hand it in again and make a note in your README file that you have done so.

### 3.5 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. You can also try implementing some of the tricks mentioned in *Introduction to Computer Graphics* section 16.2, the visible surface lecture, or ask a TA for optimization tips.

**How can I design Intersect so that doing Ray is an upgrade instead of a rewrite?** When you start to do reflections in Ray you will not only cast rays out from your eye point but also from reflected surfaces. Try to break up your program into different functions to make this transition easier. *Introduction to Computer Graphics* has an example structure in section 16.12.

### 3.6 Handing In

To handin your assignment, type “make handin” at a shell prompt.

### 3.7 Extra Credit

If you are interested in these topics you should talk to a TA for more details.

- **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 (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. An octree will really pay off in ray.
- **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. If you’ve taken CS167, multi-threading this should be a snap.
- **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.
- **Do ray now:** If you get this assignment done early and want to get ray or a good part of ray out of the way, go for it and you’ll get extra credit. The ray assignment will be going out while intersect is still out.