# Project 5 Intersect

Introduction to Computer Graphics, Fall 2015

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 OpenGL. The support code is a wrapper around OpenGL libraries and hardware.

OpenGL uses vertices to form triangles to approximate shapes. Unfortunately, using this method, we would need an infinite number of triangles to render a truly curved surface. Because this method of rendering does not simulate how light travels through a scene, it is also difficult to achieve photorealistic results.

In this assignment and the next you will be implementing a basic raytracer, a renderer that breaks away from these limitations and more importantly, lets you make really cool looking images.

2 Requirements

2.1 The ray tracing 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; you are only required to support point lights

2.2 Implicit Equations

One of the 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 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.

The global lighting coefficients can be found in CS123SceneData.h. You should pay attention to this struct:

```c
struct CS123SceneGlobalData {
    float ka; // ambient
    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 $ks$ and $kt$ for now; these will be used 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 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. The main difference is in how the ambient term is calculated. For simplicity, we just use the object’s intrinsic ambient color and the global coefficient, instead of also counting the intensity of an ambient light.

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

Here,

- $I_{\lambda} =$ final intensity for wavelength $\lambda$; in our case the final R, G, or B value we want to color
- $k_a =$ global ambient coefficient; 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
\( \text{numLights} \) = 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 vector 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

Just to make everything absolutely clear, you do NOT need to handle attenuation, directional lights, spot lights, area lights, shadows, texture mapping, specular lighting, reflection, super sampling, or multi-threading in this assignment. You will want to use this equation to compute the R, G and B values independently for each 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 (smaller error correction term, i.e 1E-6) everywhere and that will make things even worse.

### 2.4 Design

A limited amount of support code is provided for ray tracing. Your final (rasterized) output image will reside on the 2D image canvas, the same one you used for Brush and Filter. Code carefully! You’ll (hopefully) be reusing all of the code you write in this assignment for Ray. Most of the design, this time, will be your own.

We provide you with a stencil class, RayScene, in RayScene.cpp/h. In this project you will be creating a raytracer rendering of whatever scene you currently have loaded. You should make use of your ScenceviewScene code. One option would be to let your ScenceviewScene load the scene first, and then create your RayScene based on the contents of your ScenceviewScene at the time the Render button on the Ray dock is clicked.

You will also need to figure out how to handle implicit shape intersections. You can create new classes for each implicit shape, or you can extend your existing Shapes classes to handle these intersections. Doing this may require modifications to the Shape.h interface. Consider the benefits and drawbacks of implementing your implicit shapes separately vs. alongside your shapes code.

### 2.5 Rendering the Scene

When the Render button on the Ray dock is clicked, the Canvas2D::renderImage method will be invoked. The current camera (whether the orbiting camera or your perspective camera) will be passed in as an argument to this function, which you should take into account in your render. You can get the current ray tracing settings from the global settings object, as usual. For now, you can ignore basically all of these settings. Note: You will need to modify MainWindow::renderImage() to load your RayScene into the canvas before you can render your scene.

You are not required to implement marquee selection for this assignment. When Render is clicked, you should render the entire scene, regardless of the marquee bounds.

### 3 FAQ

#### 3.1 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 if you have time, extending that octree to a kd-tree (much like the one you may have made in cs32). Look up the surface area heuristic online or see the lecture slides on aceleration data structures for more information. Many of the TAs have also implemented different optimizations, so you can ask them at TA hours for advice.

#### 3.2 Nothing shows up

Before you come to TA hours for help, please double check your math. Liberal use of assert statements will definitely help you find errors in your program. Please remember that the TAs are primarily here to help with conceptual problems, not to debug your code.
3.3 How can I design Intersect so that doing Ray is an upgrade instead of a rewrite?

When you 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.

4 Extra Credit and Half-Credit

4.1 [Optional] Extra Credit

Remember that half-credit requirements count as extra credit if you are not enrolled in the half-credit course.

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. On the other hand, Intersect is worth slightly more than Ray. Therefore, 1 point of extra credit you do now is worth more (in terms of your final grade) than 1 point of extra credit you do later. Additionally, 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 of both worlds!).

- 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. 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 multi-threaded 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.

- Anti-aliasing: 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.

4.1.1 Area Lights

You may notice that there is a material property called $cEmissive$. This property indicates the intensity of light being illuminated from the corresponding object. It allows you to design a scene that has area lights, which can result in some really cool effects (like soft shadows). If you want to implement this feature, you are going to have to trace multiple rays, one to each sample point on the area light source.

One way to do this is based on Radiosity (which you’ll learn about in lecture soon). Divide the surface of your area light (which will be a sphere or a cube, or a mesh triangle) into “patches.” Divide the $cEmissive$ evenly across the patches. This approximates the light passing through the corresponding solid angle. In addition to tracing a ray from the intersection point on an object to each light, also trace a ray from the intersection to each patch on each object with a $cEmissive$. You will also need to check for occlusion for each of these shadow rays, as usual.

4.2 Half-Credit Requirements

Remember that half-credit requirements count as extra credit if you are not enrolled in the half-credit course.

Important Note: Half-credit requirements for Intersect and Ray will be graded when you hand in Ray. However we strongly recommend that you begin work on these requirements with Intersect. Otherwise you will be very unhappy.

1. Make your implementation multithreaded! This will make your life a lot easier since scenes will load much faster. Check out the following link for documentation on how to do this: [http://qt-project.org/doc/qt-4.8/threads.html](http://qt-project.org/doc/qt-4.8/threads.html)

2. Implement an accelerated data structure of $O(n\log n)$ time.

3. Implement adaptive supersampling to antialias your scenes.
5 Handing In

To hand in your code, run /course/cs123/bin/cs123_handin intersect in the directory containing CS123.pro. Before you hand in, please create a README_Intersect.txt file which details any design decisions you made and identifies any bugs or known issues in your program. Make sure you thoroughly test and debug your program and ensure that it does not have any memory leaks.