

Assignment 7: Ray

Due: Friday, November 20th at 11:59PM
Help Session: Tuesday, November 12th at 7:00PM

1 Introduction

In Intersect, you saw a glimpse of what you could do with a rendering algorithm that stressed quality over speed. Curved surfaces are possible, and everything has a sort of “perfect” feel to it. As you may have noticed, there are a few things that our renderer does not yet handle. For example, what happens if you have a shiny surface? What about texture mapping surfaces? These issues and more are confronted in this assignment. In ray, you will be filling out the shell renderer you wrote for Intersect to support reflections, texture mapping, shadows, more advanced lighting, and perhaps even transparency, motion blur, spacial subdivision, or bump mapping.

2 Demo

The demo for this project can be found at `/course/cs123/demo/ray`. Scene files can again be found in `/course/cs123/data/scene`. To run the program, type `ray <filename> [<width> <height>] [<depth>]`, where *filename* is the one of the scene files, *width* is the desired width, *height* is the desired height, and *depth* specifies the maximum depth of recursion. These scenefiles should give you an idea of how reflection and texture mapping look.

3 Requirements

For this assignment you are required to write a program that, given a name of a scene file, will generate a high-quality image that correctly takes into account lighting and all surface characteristics, as well as shadows. This should be done within the structure of your Intersect assignment.

You must handle:

- Reflection
- Texture mapping for the following shapes:
 - Cube
 - Cylinder
 - Cone
 - Sphere
- Advanced Lighting
 - Specular highlights
 - Shadows

- Directional lighting
- Attenuation

To calculate the intensity of the reflection value, you need to determine the reflection vector based on an object's normal and the look vector. You then need to recursively calculate the intensity at the intersection point where the reflection vector hits. With each successive recursive iteration, the contribution of the reflection to the overall intensity drops off. For this reason, you need to set a limit for the amount of recursion with which you calculate your reflection. You must make it possible to change the recursion limit via the command line because the TA may want to change it during grading. To be consistent, this argument should be the 4th one. Also, you will want to terminate the recursion when the intensity of the contributed color drops below a reasonable threshold.

Just to review, the lighting model you will be implementing is:

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

I is the intensity of the light (or for our purposes, you can just think of it as the color) and the λ subscript is for each wavelength, that is red, green, and blue.

The subscripts a , d , s , and r stand for ambient, diffuse, specular, and reflected, respectively.

The k s are constant coefficients. For example, k_a is the ambient coeff.

O is the object being hit by the ray. For example, $O_{d\lambda}$ is the diffuse color at the point of ray intersection on the object.

m is the number of lights.

$f_{att\ i}$ is the attenuation for light i .

$I_{\lambda\ i}$ is the intensity of light i .

\hat{N} is the normalized normal to O at the point of intersection.

\hat{L}_i is the normalized vector from the intersection to light i

\hat{R}_i is the normalized, reflected light from light i

\hat{V} is the normalized line of sight

n is the specular exponent

$I_{r\lambda}$ is the intensity of the reflected light

Please note that this equation is slightly different that from the slides. You are expected to implement this equation.

4 Scenefiles

`/course/cs123/data/scene/ray` contains scenefiles that specifically test the new features in ray, but this doesn't mean you should ignore the other scenefiles. Make sure all of the other files (especially chess) work perfectly as well.

5 Texture Mapping SNAFUs

When texture mapping planes you need to be careful. If you're texture mapping the negative z face of the cube, you'll be mapping the intersection point's x position to the u in (u, v) space. The problem is when you go left-to-right on that face, your x values are actually going from positive to negative. This isn't the only cube face that something like this will happen on, so use `cube_test_master.xml` to check each face.

To texture map the cone, just do it the same way as a cylinder (except there's only one cap, of course).

6 Getting to Know Your Friend, the CS123Scene Library

Hopefully you're intimately acquainted with the scene library now that you've completed intersect. In ray, you'll have to access a few more features of it now that you are responsible for more things.

Directional lighting, attenuation, reflection, specular highlights, and texture mapping were things you previously ignored but now must handle. Here is where the parser support code stores the information:

```
struct CS123SceneLightData
{
    LightType type;           // This will specify the type of light
    Vector4 function;        // The attenuation function coefficients in 1+x+x^2 order
    Vector4 dir;             // If the light is directional, use this instead of position
    ...
};

struct CS123SceneMaterial
{
    CS123SceneColor cSpecular; // The specular color
    float shininess;           // The specular exponent
    CS123SceneFileMap* textureMap; // The full path to texture file and uv repeats
    float blend;              // Blend between texture color and diffuse
                                //color (0 is 0% texture, 100% diffuse)
    float ior;                //index of refraction
    ...
};

struct CS123SceneGlobalData
{
    float ka; // The constant ambient coefficient for the scene
    float kd; // The constant diffuse coefficient for the scene
    float ks; // The constant specular coefficient for the scene
    float kt; // The constant transparent coefficient for the scene
};
```

This is just a quick overview of the new items. For more information, check out the library documentation on the website.

7 Support Code

For this project, you will receive significantly less support code than in previous projects. For the most part, the “support code” is your intersect, since this assignment builds off of it. It behooves you to build off your intersect code because a complete ray can be quickly built from a well-designed intersect.

The only support code you will receive is a spatial acceleration data structure. The point of this is to allow you to implement complex rendering techniques, such as supersampling, if you did not write any acceleration structures of your own. Without an acceleration structure, it would take an inordinate amount of time to render most scenes.

The acceleration structure can be accessed and created using the following classes:

```
class CS123Intersectable {
    virtual bool getIntersection(const double *dir,
                                const double *start,
```

```

        double &intersectT);
virtual void getBounds(double &minX, double &maxX,
                    double &minY, double &maxY,
                    double &minZ, double &maxZ);
};

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

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

```

8 The Book

The book has a lot of great stuff on raytracing and reading up a bit will make the assignment easier. You should read through most of chapter 16 and pay particular attention to pages 780 and 781 if you're having trouble designing ray.

9 Extra Credit

Ray is one of the coolest projects you'll ever write at Brown. You, yes, you, can make it even cooler by doing some sweet extra credit. Here are some ideas (book sections are included if there's significant discussion of the topic):

- Antialiasing
Brute force supersampling isn't hard to do and antialiased images look really sexy. If you're feeling brave, try your hand at adaptive supersampling (15.10.4) or stochastic supersampling (16.12.4)
- Transparency (16.5.2)
- Motion blur
- Depth-of-field
- Fewer intersection tests (15.10.2)
Bounding volumes, hierarchical bounding volumes, octrees, or kd-trees are all things to try that will get big speedups on complex scenes since most of the clock cycles go to intersection tests. Mucho bonus points if you do one of these. Really fast intersection tests might get you a few points too, but only if they're really good. It is highly *encouraged* that you do some sort of spacial subdivision, but certainly not required at all.
- Constructive solid geometry (a.k.a. CSG, 12.7)
- Bump mapping
Like texture mapping, except each texel contains information about the normals instead of color values. It's a great way to "add" geometry to an object without having to actually render the geometry.

- Texture mapping and/or intersecting other shapes, like the torus
- Spotlights
Spotlights have position, direction, and an aperture in degrees. If a light is a spotlight, `SCLightData`'s `m_type` will be equal to `LIGHT_SPOT` and `m_aperture` will contain the aperture size.
- Any spiffy optimizations
Be careful here. “Premature optimization is the root of all evil” – Knuth. You’ll learn that he’s right at some point in your career, but let’s not learn that lesson on ray. Get the basic functionality done then go for the gusto (and don’t forget to profile with `gprof`!). This isn’t as important as it once was now that we have blazing fast machines, admittedly, but it’s still awful fun.
- Multithreading
Raytracing is “embarrassingly parallel” because a ray does not depend on the outcome of any of the other rays. Each ray cast per scanline can be made into its own thread. If a TA knows `pthread`s and is feeling particularly nice, he might help you with it.
- Texture filtering (bilinear, trilinear, what have you)
- Whatever else you can think of!

Remember to make a few scenefiles to show off the extra credit you did. One last tip: write your ray assignment with modeler in mind (think of ray not as a final program but a component of a future program). Integrating ray with modeler makes modeler a lot more useful. If you plan ahead, it’s trivial to combine the two.

10 Handing In

To hand in your assignment, type `make handin` in your ray source directory.

Remember to document anything cool or unusual in your `README` file.

| | | |
|------------------------|------------|---------|
| Algo Handin: | 11/12/2009 | 5:00PM |
| Program Handin: | 11/20/2009 | 11:59PM |