Project 6 Ray
Introduction to Computer Graphics, Fall 2015

1 Introduction

With Intersect, you saw a glimpse of what you could do with a rendering algorithm that stressed quality over speed. Truly 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? These issues and more are addressed in this assignment. In Ray, you will be filling out the 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.

The demo for this project is the same as Intersect. Run cs123-demo at a shell prompt, and load a scene file using the menu. This time, try some of the files in the “ray” scene folder, and be sure to enable the more advanced options in the ray tracing toolbox.

There is no additional support code for Ray; in this assignment, you are building entirely on your Intersect code.

2 Requirements

For this project you are required to extend your Intersect code to implement additional features in your ray tracer. Your ray tracer must be able to handle:

- Reflection
- Texture mapping for, at least, the cube, cylinder, cone, and sphere
- Specular highlights
- Shadows
- Directional lighting
- Light fall-off (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 easily because the TA may want to change it during grading. 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:

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

Here, the subscripts \( a, d, s, \) and \( r \) stand for ambient, diffuse, specular, and reflected, respectively.

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

\( k \) is a constant coefficient. 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. The main difference is in how the ambient color is calculated. We just use the color of the object instead of also counting the intensity of an ambient light.

\( \footnote{By “easily,” we mean that we should only have to change it in one place. A \texttt{const int} in a common header file would be an excellent example of a good location to store such a parameter. You should document the location of this parameter in your README_Ray.txt file.} \)
2.1 Texture Mapping

A “blend” value may be supplied as a material property for objects that are texture mapped. If the blend value is 1, then the texture entirely replaces the diffuse color of the object. If the blend value is 0, then the texture will be invisible. If the blend value is between 0 and 1, then you should perform linear interpolation in the same fashion as the flow rate in Brush.

3 Testing

Your ray tracer’s output should look exactly like the demo (for a given scene file and render settings). In addition, the directory /course/cs123/data/scene/ray contains scenefiles that specifically test the new features in Ray. That said, you should not ignore the other scenefiles. Make sure all of the other files (especially chess) work perfectly as well. Also, pay particular attention to the orientation of your textures.

If you create your own additional scene files, please put them in /course/cs123/data/scene/shared so that other students may benefit from them. We’ll also be awarding extra credit for particularly interesting scenes.

4 FAQ

4.1 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.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).

4.2 QImage and QString classes

We strongly recommend you look at the documentation for the QImage and QString classes. The QImage class provides an image representation that allows direct access to pixel data (hint, hint) and this class can require a QString to be passed into it. You can also search the support code for uses of QImage for reference.

5 Extra Credit and Half-Credit

5.1 [Optional] Extra Credit

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

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, and 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 spatial 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, CS123SceneLightData’s m_type will be equal to LIGHT_SPOT and m_aperture will contain the aperture size.

**Optimizations** Be careful here. “Premature optimization is the root of all evil.” 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. You may find profiling tools like the Valgrind Function Profiler under QtCreator’s Analyze tab useful. This isn’t as important as it once was now that we have

---

2 Knuth
blazing fast machines, admittedly, but it’s still awful fun. CS123 used to render ray-traced images on Apollo workstations in the Sun lab; in some cases, it would take an entire night to generate a 16x16 pixel image!

**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. Feel free to use QThreads to help.

**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. If you create your own additional scene files, please put them in `/course/cs123/data/scene/shared` so that other students may benefit from them. We’ll also be awarding extra credit for particularly interesting scenes.

### 5.2 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://doc.qt.io/qt-4.8/threads-starting.html](http://doc.qt.io/qt-4.8/threads-starting.html)

2. Implement an accelerated data structure of O(nlogn) time.

3. Implement adaptive supersampling to antialias your scenes.

### 6 Handing In

To hand in your code, run `/course/cs123/bin/cs123_handin ray` in the directory containing CS123.pro. Before you hand in, please create a README_Ray.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.