Computer Graphics - Assignment 2

1 Overview

For this project, you will be implementing a simple raytracer that can handle shadows and reflections in a basic scene containing primitive shapes. All rendering can be done with software, only using OpenGL to display the final image to the screen (this is how the demo code is set up). Though a ray tracer can be written in a few hundred lines of code it is quite intensive and debugging can take a fairly long time so you are advised to start early. Do not wait until the last week to start.

2 Objective

This assignment is designed to give you an opportunity to implement techniques for rendering photorealistic images. Your task is to create a ray tracer capable of rendering a set of primitive shapes.

Input: We provide you with the basic setup code for the raytracer, outlines of `Object` and `Ray` classes that will be used for determining the final color for each pixel in the image, and OpenGL code for rendering the final image. Please see the sections below for more details on each item.
Output: You must write a ray tracer to render a scene composed of various primitive shapes including at least one sphere, one plane, and one triangle. This will require extending the `Object` and `Ray` classes to handle intersections between these primitives and the rays you cast in your scene. You will also want to define some material properties for these objects so that you can render them with the lighting equations you used in the last assignment. Your raytracer must include simple shadows and reflections up to a recursion depth of at least 3. You will also need to write a readme file explaining the techniques you have used in the assignment and provide screenshots of your work.
Requirements:

- Write intersection tests for spheres, planes, and triangles and include at least one of each these objects in the scene.
- Implement the basic ray tracing algorithm in the `CastRay` function by sending a ray from the eye through all objects in the scene, up to a recursion depth of at least 3.
- Add direct illumination and shadows by sending rays to point lights.
- Add specular reflections by sending reflected rays into the scene.
- Submit a few screen shots of your program's renderings.
- Use good code style and document well. We *will* read your code.
- Create a file named "readme.txt" containing the details of your implementation and instructions for compiling and running your code.
- The program must compile and run on DICE. If it does not, you run the risk of getting 0 marks.

Should you manage to achieve all of these requirements then you will receive a good mark. The following is a list of possible additional techniques that can be implemented for further marks:

- Add refractions using transmission rays
- Add intersection tests for other shape primitives e.g. Axis-Aligned Bounding Boxes, arbitrary convex volumes
- Implement acceleration structures e.g. grids, bounding volume hierarchies
- Soft shadows.
- Soft reflections.
- Depth of Field.
- Subsurface scattering

A number of these techniques are taught at the sites listed in the "Resources" section.

3 DICE Instructions for use

This demonstration program is written in C++. It needs to be compiled into executable code before running.

To compile the demo program, open up a terminal window and navigate to the directory containing the source code. Here you can run the command:

```
make run
```

This will compile the default source code and run the demo program with the teapot mesh as input. You can perform these actions separately by first compiling with:

```
g++ -o demo2 *.cpp -lglut -lGLU -lGL
```

This compiles the code and links libraries glut, GLU and openGL. You may then run the program with the following command:

```
./demo2
```

4 Resources

There are several resources online that teach basic raytracing concepts and provide example source code. These include:

- `http://www.scratchapixel.com/lessons/3d-basic-lessons/` - Covers a lot of the theory behind raytracing and provides examples of basic raytracers.
- `http://www.codermind.com/articles/Raytracer-in-C++-Introduction-What-is-ray-tracing.html` - Focuses more on the actual code used in a basic raytracer.
- `http://www.cs.utah.edu/~shirley/books/fcg2/rt.pdf` - Textbook chapter explaining the concepts of raytracing as well as the derivations of some intersection test equations.
- `http://www.youtube.com/watch?v=W2QrXv2yZhE` and `http://www.youtube.com/watch?v=s5m391a5HFg` - Video tutorials for ray-sphere and ray-plane intersection tests.

5 Notes on the source code

Below is a short description of the source code files provided to you for this assignment. The files themselves contain a number of comments on functionality of the program and should be largely self-explanatory. Should you have any issues then please contact the course teaching assistant (see contacts below).

- *demo2.h* - Used to include necessary utility code. Feel free to change this to your needs.
- *demo2.cpp* - Defines the entry point for the program. Sets up the OpenGL window and callback methods for displaying the rendered image and handling mouse input. Calculates the position of each pixel of the image in world space ready for ray casting. Defines the `CastRay` function where you will implement your recursive raytracing algorithm. Along with the "Ray.*" and "Object.*" files this will be where you implement the majority of your work.
- *Ray.h* - Defines the `Ray` class interface. The `Payload` class holds the information about the current state of the ray. Also provided is the `IntersectInfo` class that can be used to store details on the intersection between a ray and an object in the scene.
- *Object.h* - The `Object` class interface for objects in your scene. This class should be extended for primitive shapes and the `bool Intersect(const Ray &ray, IntersectInfo &info) const;` function overidden for each shape.
- *Object.cpp* Implementation of the `Object` and `Material` constructors

The source code includes headers from the OpenGL Mathematics library (`http://glm.g-truc.net/0.9.4/index.html`). These headers define basic vector and matrix classes that correspond to the classes used in OpenGL. Of particular interest will be the `glm::vec3`, `glm::vec4`, and `glm::mat4` classes.

6 Hints

- Read and understand demo2.h and demo2.cpp first. If you don't understand, contact the course's teaching assistant (see below for details) or ask a fellow student.

- Familiarise yourself with how raytracing works. Read through some of the basic tutorials at the websites listed in the "Resources" section. If you are having trouble then don't be afraid to ask!

- Start small. Begin with a single object in your scene and make sure your intersection tests are working. You don't have to render the shape with full lighting straight away. A simple change of pixel color based on the success of your intersection test should help you to check if you are doing it right.

- Use the `Payload` and `IntersectInfo` classes to help you store information on the current state of the ray and objects it may intersect with. This information can be useful for debugging the raytracer.

- Make sure your rays are performing correctly before trying to do any reflection.

- You can light the scene using the Phong shading equations from the previous assignment. The ray intersection test should give you the position and normal on the object. Note you should consider the ray direction to be equivalent to the viewing direction.

- Make sure your raytracing algorithm is in place before adding too many objects. A single sphere and plane should be enough to let you test this. If this is in place, you can create more complex scenes by simply extending the object class.

- Document your work well. Be sure to include details in your readme about the approach you took towards the coursework. A perfect solution with no documentation will not receive as high a mark. If you cannot get your raytracer to work try to explain why you think this might be happening.