CPSC 314 Computer Graphics
Programming Assignment 4: Ray Tracing

Due 11:59pm, Nov 30th, 2015

1 Introduction

In this assignment, you will implement a simple ray tracer (or a path tracer if you're feeling brave) that renders local illumination, shadowing, and reflection. The supporting geometric objects will be spheres, planes, triangle meshes, and optionally other types of surfaces.

1.1 Template Code Description

The entry of the program is defined in *main.cpp*, it uses Parser (defined in *parser.hpp* and *parser.cpp*) to parse the scene files (in scenes/ folder, the description of the format could be found in scenes/basic.ray) and create a Scene object (defined in *scene.hpp*) accordingly for the ray tracing step.
Then Raytracer (defined in *raytracer.hpp* and *raytracer.cpp*) is used to render the scene. It renders the scene to an Image object and outputs the final image as a bmp file (saved by *image.hpp*).
The scene object may contain geometric objects like spheres, planes, triangle meshes, and also conics (defined in *object.hpp* and *object.cpp*). The basic math toolkit such as Vector, Matrix, Ray, and Intersection are defined in *basic.hpp*.
There are three subdirectories: *scenes/*, *meshes/*, and *referenceResults/*. The scenes directory contains scene descriptions in the .ray format, describing the following scene parameters: Dimensions, Perspective, LookAt, Material, PushMatrix, PopMatrix, Translate, Rotate, Scale, Sphere, Plane, Mesh, and PointLight. The comments in those files describe the format. A few triangle meshes in OBJ format are provided in the meshes directory, and most of the scene files depend on one or more of these. In the referenceResults directory, there are reference results provided for you.
Your job will be mostly limited to *object.cpp* and *raytracer.cpp*. You do not need to make any changes to the other source files (sure you can if you wish when implementing optional extra features). Besides, you are strongly recommended to have a quick look at *basic.hpp*, *object.hpp*, *raytracer.hpp*, and *scene.hpp* to get a good knowledge of the C++ classes, which will help with your coding. The rest of the files: *image.hpp*, *parser.cpp*, and *parser.hpp* are less important.

1.2 Instructions for Compilation and Execution

If you have a favorite IDE, create an empty project and add all the header and source files in the main directory to it and run. The project doesn't need any 3rd party libraries. For Windows users we also provide a *sln* file for Visual Studio 2013. For Linux (e.g. all the department undergrad machines), we provide a Makefile to compile the project. To compile, simply type **make** command in the main directory. *Attn Mac users: Due to a bug in standard library in the latest MacOS, scene files may not be loaded properly. If that's the case, you can still debug it on your Mac, but not using scene files. You'll have to perform actual testing on some Linux or Windows machine.*
To run the executable:

```
./a4.exe
or ./raytracer
```

By default, this will render a scene with only a red cube and save the resulting image as *default_output.bmp* in the *scenes/* folder, where there will also be a depth image saved as *default_output_depth.bmp*, in which white means far, dark means near.
To run the executable and render a different scene, pass it as the first argument:

```
./a4.exe your_scene_file_path
```

The resulting images will also be saved in *scenes/* folder named *your_scene_file_path_output.bmp* and *your_scene_file_path_output_depth.bmp*. For simple scenes, it needs several seconds to render a scene. However, once you have meshes, you will quickly realize why we are using C++ for this assignment.

2 Assignment (Total: 100pts)

Here we strongly recommend you to follow the sequence of the goals and implement step by step. The raytracer should cast primary rays into the scene, which spawn shadow rays and secondary reflection/refraction rays. Note that rendering very complicated scenes with many primitives (eg: the provided teapot mesh has thousands of triangles) can take a long time! Do not change any of the scene files, we will compare your results with our reference images to do the marking.
Alternatively, *you can instead implement Path Tracing algorithm. In Path Tracing, implementing the core functionality is harder, but getting creative part done is much easier.*

2.1 Shooting Rays (20pts)

1. Implement the missing parts of Raytracer::render for basic ray casting for all pixels in the image, using the camera location and the coordinates of each pixel. You can test this code by re-computing the pixel as the intersection of the ray and the view plane and testing that you obtain the same coordinates back.

2. Implement the missing parts of Raytracer::trace to iteratively test all the object's intersection with the given ray. Update the depth as the first intersection's depth, then after implementing one of the intersection test functions, you will be able to see some information on your depth output image.

2.2 Intersection Tests (15pts)

In this part you will first implement the functions of intersection tests and the functions of casting primary rays. The result is the depth image showing the depth information of a given scene. We will mark this part according to the depth image generated by your program.

Steps:

1. Implement Sphere::localIntersect. For this part you are required to calculate if a ray has intersected your sphere. Be sure to cover all the possible intersection scenarios(zero, one, and two points of intersection). Test your result by comparing the output depth image of your algorithm with the provided example solution's results.

2. Implement Plane::localIntersect. The implementation of this part is similar to the previous part in that you are calculating if a line has intersected your plane. Test this function in a similar way to the previous part (note that as you do new objects will appear).

3. Implement Mesh::intersectTriangle. This function calculates the point of intersection of a ray with a triangle. The difference of this part when compared to the plane intersection is in the bounds you must check. Think back through the course and try to decide what equations might help you decide on which side of the bounding lines of the triangle the ray intersects. Test this part just like the above two parts; when triangle intersection is working properly, you should be able to see full meshes appear in your scenes. *Think how would you compute a normal at the intersection point.*

You can first implement the intersection test for some of the geometries and focus on the following parts, then after you complete the whole idea, you can go back to implement the tests for other geometries.

2.3 Local Illumination (10pts)

Implement the missing part of Raytracer::shade that does a lighting calculation to find the color at a point. First you can assume all the light sources are directly visible. You should calculate the ambient, diffuse, and specular terms. You should think of this part in terms of determining the color at the point where the ray intersects the scene. After you finished, you will be able to get the colored resulting image with local illumination, just like in programming assignment 3. Test your results by comparing to the ground truth ones.

2.4 Shadowing (10pts)

Implement the shadow ray calculation in Raytracer::shade and update the lighting computation accordingly. Emit the shadow ray from a point you're computing direct illumination for to determine which lights are contributing to the lighting at that point. Be careful to exclude the origin of the ray from the intersection points, but do remember that the intersection points could be another points on the same object if the object is not convex (for example, the teapot). For points in the shadow, scale their original lighting color by the factor $(1 - material.shadow)$.

2.5 Reflection (25pts)

Implement the secondary ray recursion for reflection in Raytracer::shade. Use the rayDepth recursion depth variable to stop the recursion process. (The default used in the solution is 10.) Update the lighting computation at each step to account for the secondary component. You can think of this part as an extended shadow ray calculation, recursively iterating to determine contributing light (and weighting newly determined light sources into the original pixel).

2.6 Creative License (20pts)

Implement this only once you're done with the previous assignments. Now it's time to play more and get the full score! Here we provide some suggestions on what you might explore:

1. **Conic Geometry:** Implementing Conic::localIntersect to enable intersections between the rays and generalized conical surfaces (http://en.wikipedia.org/wiki/Conical surface). Note that this requires detecting the bounding circles of the conics and accurately handling those (to get finite cylinders/cones/ellipsoid parts).

2. **Refraction:** Implementing a secondary ray recursion for refraction rays. Use the same recursion depth variable rayDepth as for reflection to stop the recursion process. Update the lighting computation at each step to account for the secondary component.

3. **Texturing** You may need to use a 3rd party library, like OpenCV or libpng, to import textures and access the texture during ray tracing to get a local diffuse color.

4. **Speed** Consider speeding up your method using any of the space-partitioning methods discussed in class. Compare your result to the original version of your raytracer.

5. **Gloss** Use randomized direction estimation to account not only for specular but also glossy surfaces.

6. **Others:** Nomal Mapping, GPU Raytracing, Ambient Occlusion, or Soft Shadows.

Note that creating new scenes by writing your own scene files does not count as a creative license, but is certainly fun to do.