

CPSC 314 Computer Graphics

Programming Assignment 4: Ray Tracing

Due 11:59pm, Dec 2nd, 2016

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 starting point of the program is defined in *main.cpp*; it uses the 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*) subsequently rendered by the ray tracing step.

The Raytracer (defined in *raytracer.hpp* and *raytracer.cpp*) is then used to render the scene. Rather than generating a screen rendering it renders the scene into an Image object and outputs the final image as a bmp file (saved by *image.hpp*).

The scene consists of one or more geometric objects including spheres, planes, triangle meshes, and conics (defined in *object.hpp* and *object.cpp*). Basic math constructs 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. The *referenceResults* directory provides example reference results.

Your code changes should be mostly limited to *object.cpp* and *raytracer.cpp*. You do not need to make any changes to the other source files (unless you wish to do so when implementing optional extra features). We also strongly recommend that you take a quick look at *basic.hpp*, *object.hpp*, *raytracer.hpp*, and *scene.hpp* to become familiar with the C++ classes used. This will help with your coding. The rest of the files: *image.hpp*, *parser.cpp*, and *parser.hpp* are less critical.

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 should render a scene with only a red cube and save the resulting image as *default_out.bmp* in the *scenes/* folder, where there will also be a depth image saved as *default_out_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_out.bmp* and *your_scene_file_path_out_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)

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. We strongly recommend that you follow the outlined sequence of the goals and implement them step by step. Make sure you understand what you are doing at each point - we will ask you about your code during marking. Please make sure you use the correct template as a starting point for your code. Failure to do so will result in a grade of zero.

Alternatively, you can instead implement the Path Tracing algorithm. In Path Tracing, implementing the core functionality is harder, but getting the 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. Set the depth to be the first intersection's depth. Once you implement one of the intersection test functions, you should be able to see some information on your depth output image.

2.2 Intersection Tests (15pts)

In this part you will first implement a number of intersection tests and use those in the previously written functions for casting primary rays. The result of these two steps is a depth image showing the depth information for a given scene. We will mark these two parts using the depth images 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 so new objects will appear).
3. Implement `Mesh::intersectTriangle`. This function calculates the point of intersection of a ray with a triangle. The difference in 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 does a ray intersection happen. 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 subsequent parts, then after you complete the whole system, 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. You can first assume that 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 colored resulting images 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. Shoot the shadow rays 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 list of intersection points, but do remember that the intersection points could include other points on the same object if the object is not convex (for example, the teapot). For points in the shadow, scale their original lighting induced color by the factor $(1 - \text{material.shadow})$.

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 was 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 adding newly determined light sources into the original pixel color computation.

2.6 Creative License (20pts)

Implement this part **only** after you are fully done with the previous tasks. Here we provide some suggestions on what you might explore (depending on complexity one or two of these may be enough for a full score):

1. **Conic Geometry:** Implement `Conic::localIntersect` to enable intersections between the rays and generalized conical surfaces ([http://en.wikipedia.org/wiki/Conical surface](http://en.wikipedia.org/wiki/Conical_surface)). Note that this task requires detecting the bounding circles of the conics and accurately handling those (to get finite cylinders/cones/ellipsoid parts).
2. **Refraction:** Implement 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** Implement texture support as part of your rendering scheme. 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** Speed up your method using any of the space-partitioning methods discussed in class. Compare your result to the original version of your raytracer. The dragon is a good test case here.
5. **Gloss** Use randomized direction estimation to account not only for specular but also for glossy surfaces.
6. **Other options:** Normal 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.

Hand-in Instructions: You do not have to hand in any printed code. Create a README.txt file that includes your name, student number, and login ID, and any information you would like to pass on to the marker. Create a folder called “a4” under your “cs314” directory. Put all the source files, your makefile, and your README.txt file for each part in the folder. Do not use further sub-directories. The assignment should be handed in with the exact command:

```
handin cs314 a4
```