Assignment 6: Ray-Triangle Intersection
CS180 CS 280 Fall 2022

Professor: Lingqi Yan

University of California, Santa Barbara

Assigned on Nov 15, 2022 (Tuesday)

Due at 11:59 PM on Nov 21, 2022 (Monday)

Notes:

- Be sure to read the Programming and Collaboration Policy on course website.
- Any updates or correction will be posted on EdStem, so check there occasionally.
- You must do your own work independently.
- Read through this document carefully before posting questions on EdStemn.
- Submit your assignment on GauchoSpace before the due date.

---

1 Overview

In the second half of the class, we will focus on rendering image by ray tracing. One of the most important operations in ray tracing program is to find the ray-object intersection. Once we find an intersection of one ray, we can perform the shading and return the pixel color. In this assignment, we need to implement two parts: ray generation and ray-triangle intersection. The general work flow of the current code base is:

1. Start from `main` function. We define the parameters of our scene. Add objects (spheres or triangles) to our scene, and set their material. Add the light sources to the scene.
2. Call `Render(scene)` function. Inside a loop which iterates through all the pixels, generate rays, and save the returned color inside a framebuffer. Finally the framebuffer is saved inside an image.
3. Later, we generate a ray through the pixel and call the `CastRay` function, which calls `trace` to find the intersection of ray and the closest object in the scene.
4. Then we perform shading at this shading point. We have three different shading cases, and already provide the code for you.

The functions you should modify are:

- `Render()` **in the Renderer.cpp**: You need to generate one ray for each pixel here, and call the function `castRay()`, which will return the color, and store the color in the corresponding pixel of the framebuffer.
- `rayTriangleIntersect()` **in the Triangle.hpp**: v0, v1, v2 are the three vertices of the triangle, orig is the origin of the ray, dir is the direction(normalized) of the ray. tnear, u, v are the parameters (t, b1 and b2, respectively) that you need to update as the output of the Moller-Trumbore algorithm we derived in the class.

2 Getting started

In this assignment, you will have a new code base, the only dependency of the start code is CMake. We believe all of you are able to work on your own machine now.

Please download the project's skeleton code, and build the project by using the following commands:

```
$ mkdir build
$ cd ./build
$ cmake ..
$ make -j
```

After this, you should be able to run the given code by using **./Raytracing**. There a few classes in our code base now. Some general introductions are:

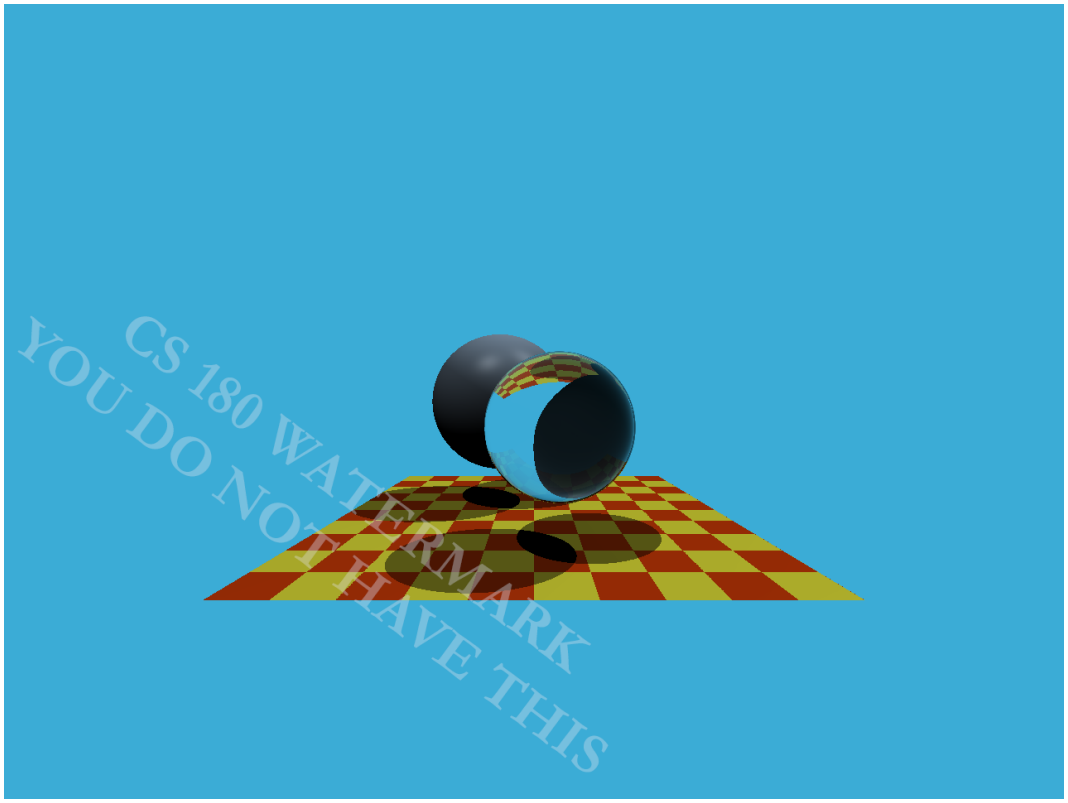
- global.hpp: Some functions or variables we use across the classes.
- Vector.hpp: Since we no longer use the Eigen library, we provide the common vector operation here, such as: dotProduct, crossProduct, normalize.
- Object.hpp: The parent class of a primitive we can render. Triangle and Sphere classes are inherit from this base class.
- Scene.hpp: Define the scene to be rendered. Including setting options and object list and lights list.
- Renderer.hpp: The main renderer class, which implements all ray tracing operations.

3 Grading and Submission

Grading:

1. (5 points) Submission is in the correct format, and includes all necessary files. Code can compile and run.
2. (10 points) Ray Generation:
   You successfully implement the ray generation part and are able to see the two spheres in the image.
3. (15 points) Ray-Triangle intersection:
   You implement the Moller-Trumbore Algorithm correctly and are able to see the floor in your image.
4. (Bonus 5 points) Sample area light to get soft shadow. Please refer to Section.4 for more details.

After you implement the basic parts, you will see the image like this:



4 Bonus part

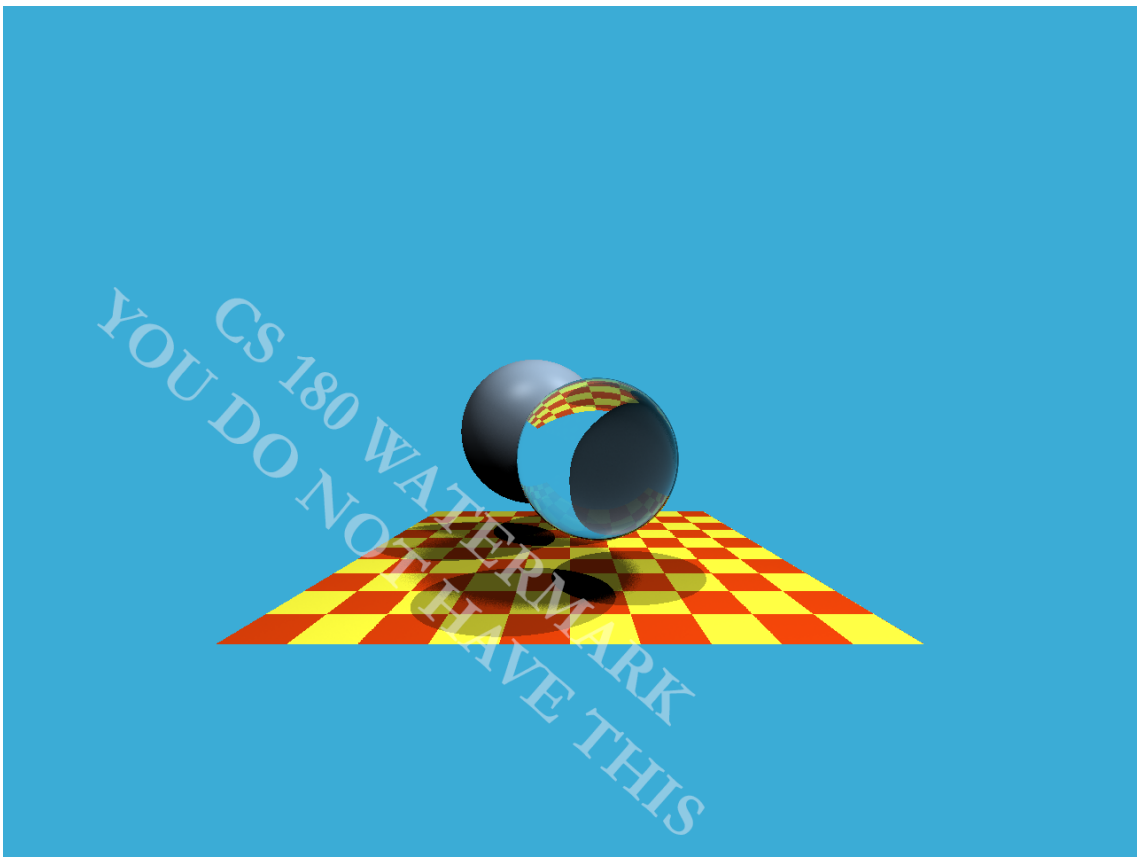
For the bonus part, you should implement area light for the shadows. Currently, the code contains support only for point light sources, where checking if a point is in shadow is straightforward. At the point of intersection, we create a new ray through the exact position of the light source, and check to see if there is an object that's occluding the light source.

With area lights, you know the position of a corner of the light source, two vectors that define its two dimensions, and the length in both of these dimensions (defining a parallelogram). For this part, for a single shading point, you should calculate **N** random points inside the area light source, and shoot rays through those points, and count how many of them are in shadow (then divide it to the number of samples to normalize between 0-1).

There are two parts you should update to make this work. First, you should implement point sampling the area light source, and then the actual shadow checking part.

- `SamplePoint()` **in the AreaLight.hpp**: You need to calculate two random floating point numbers to make this work. There's already a function ready to be used called **get_random_float** that returns a random value between 0-1. You should use those values to find the point inside the area light.
- `castRay()` **in the Renderer.cpp**: In this function where the shading is done by iterating through all the light sources, you will see an if statement that checks whether the current light source is an area light source. Inside this function, you can write a function that calculates the color of the current hit position. For this portion, you can check how a single ray is sent through a point light to check if it's in shadow or not. For bonus, you should extend the idea of single shadow ray to shoot multiple random rays into the area light and count how many of them are in shadow. This way, you should get softer shadows. (This will make your code work **A LOT slower**, so start with less number of samples, like 10, and after you get it done, increase the number of samples to get smoother shadows).

You will be able to see results like this:



Submission:
After you finish the assignment, clean your project, remember to include the CMakeLists.txt in your folder, whether you modified it or not. Add a short report file in the directory, write down your full name and perm number inside it. Tell us whether you did the bonus part or not, and **briefly describe what you have implemented in each function.** Then compress the entire folder and rename it with your name, like "Goksu.zip". Submit only **ONE** .zip file on GauchoSpace.