Advanced Graphics 2022-2023 – Assignment 1

Tim Goedhuys, 6231926
Saiyang Zhang, 4664302

December 2022

1 Introduction

This is a report of the first assignment for the course Advanced Graphics 2022/2023. This assignment contains the following content:

- A generic and extendible architecture for a ray tracer.
- A 'free camera' with configurable position, orientation, FOV and aspect ratio.
- A basic user interface, or keys / mouse input handling to control the camera position and orientation at run-time.
- Support for (at least) planes, spheres and triangles.
- An extendible material class.
- A basic scene, consisting of a small set of these primitives. Use each primitive type at least once.
- A Whitted-style ray tracing renderer, supporting shadows, reflections and refraction with absorption (Beer's law).
- A Kajiya-style path tracing renderer, supporting diffuse interreflections and area lights.

The following section will describe where to find certain parts of the project to make it easier to review.

2 Implementation

2.1 Framework

We used the ray tracing framework from tmpl8rt_2022.

2.2 Camera/ User interface

To move the camera, we implemented user interface by detecting the keyboard interruption in the display window. The interruption will be detected when function **Renderer::Tick**(line 262) is called. We directly refer to the functions in **class Camera** when there is an interruption.

We use different transformation matrices for camera translation (**M, InvM**) and rotation (**R, InvR**). We only consider the translation of camera position and rotation of screen corners: we move the camera position after translation interruption; the **float3** vectors for screen corners represent their relative position to the camera, regardless of camera position, we only compute the relative position of screen corners after rotation interruption.

2.3 Shape primitive/ Material/ Scene

We used the shape primitives from the framework, and created a superclass **Shape** to make them easier to refer to, also added the **Triangle** subclass to **Shape** in **scene.h**. Currently the **Triangle** only supports ray intersection.

We use enumerators (**MatType**) to identify different materials. For each kind of material we define different rendering method in **Renderer::Trace** and **Renderer::PathTrace**. We implemented another material class in ADVGR-PathTracing, but the merge of repositories are not completed yet.

We are trying to use a vector of **Shape** pointer to refer to the shapes in the scene instead of refer to them one by one. This may also be necessary for further development.

2.4 Whitted ray tracing

We implemented Whitted-style ray tracing in **Renderer::Trace**. As mentioned before the render function will compute in different ways according to the material of the object.

We consider only point light here, so the main factor that affect the brightness will be the distance between object and the light source, so for objects with **Diffuse** material, they will be brighter if they are close to the light source and vice versa.

We cast shadow if there is an object between intersection point and the light source, but for **Mirror**, we will cast a reflecting ray and find the reflected color instead of casting shadow, so that there will be no shadow on the mirror as in figure 1 .

For **Glass**, we will mix the reflected color and refracted color. We cast rays by Snell's Law. And for rays enter the glass, we will use Beer's law to absorb certain brightness of the ray. As a result, the glass objects will render a shimmer of their original color as in figure 1.

With recursive tracing, the scene is realistic in reflection and refraction as in figure 2 and figure 3.

We applied anti-aliasing by casting multiple primary rays around the original primary ray as in line 196.

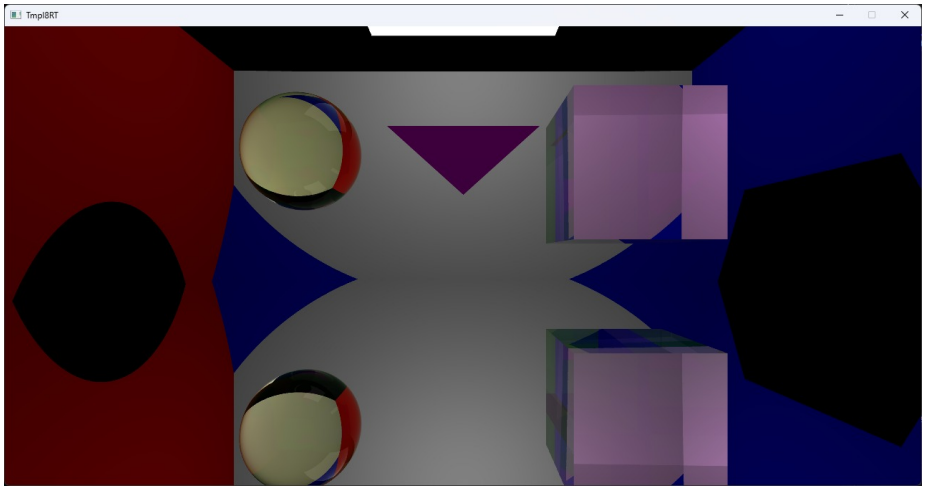

Figure 1: Rendering result of glass sphere and cube over mirror floor

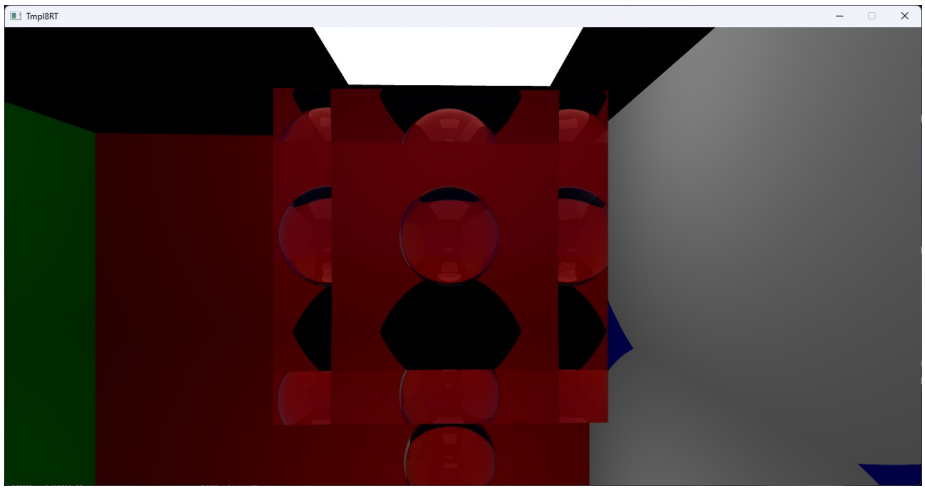

Figure 2: Rendering glass sphere through glass cube

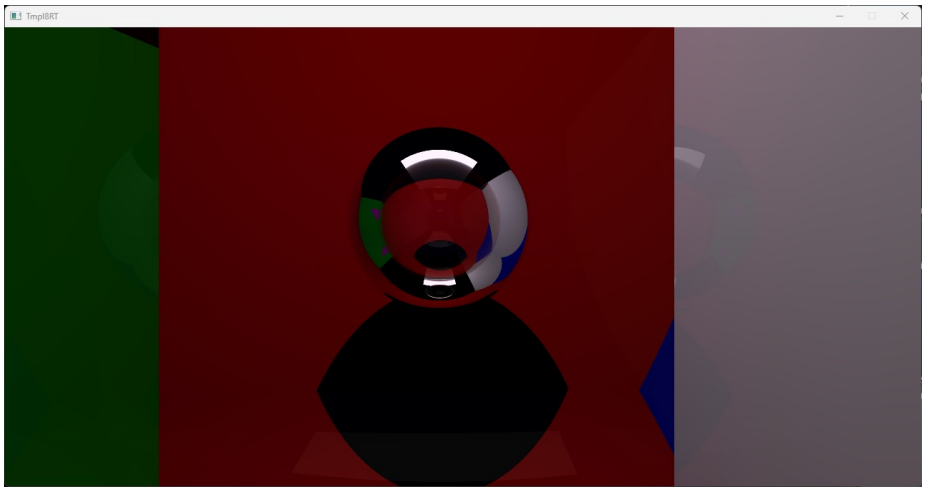

Figure 3: Rendering glass sphere within glass cube

2.5 Kajiya-style path tracing

We implemented Whitted-style ray tracing in **Renderer::PathTrace**. The color computation is not very different from that of Whitted-style ray tracing. However, we will use angle between object normal and intersection ray to compute brightness, instead of using distance. Also, the light source will be area light instead of point light.

We cast random rays when the primary ray hits the objects, and recursively call **Renderer::PathTrace** to take the diffuse rays into consideration.

To avoid too many recursions, we set a max depth for that. Also, in each recursion, there is certain probability that the computation is ended and the function returns no color. To keep the expectation of color correct, we will adjust the color according to the probability of ending recursion.

Rendering using path tracing will turn out to be more realistic in certain aspects, such as soft shadow and translucency as in figure 4 , the brightness will also change according to the surface area of the light source as in figure 5.

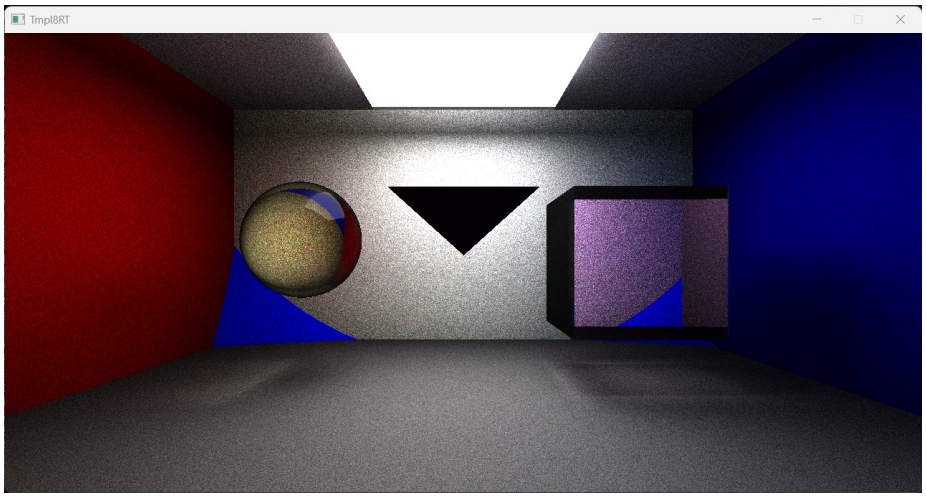

Figure 4: Rendering using path tracing

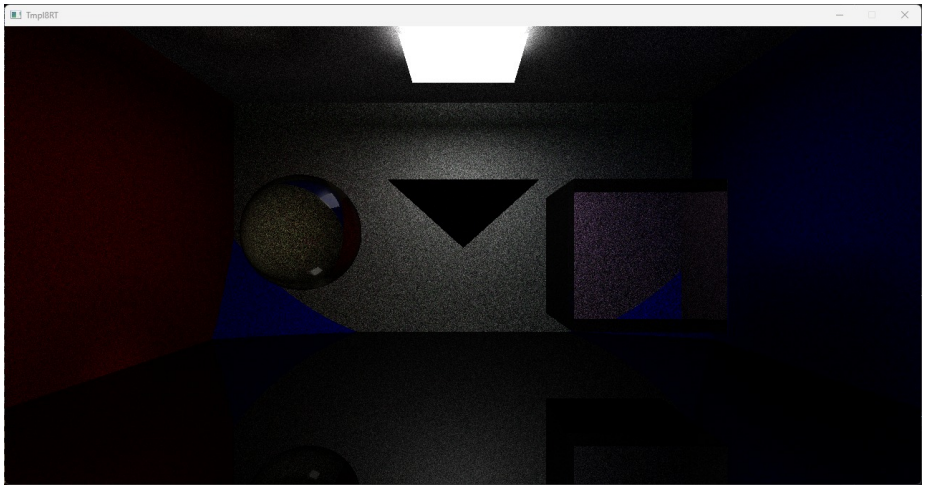

Figure 5: Rendering with smaller light source

In ADVGR-PathTracing, we implemented another workflow for path tracing, but the result is not ideal as in figure 6.

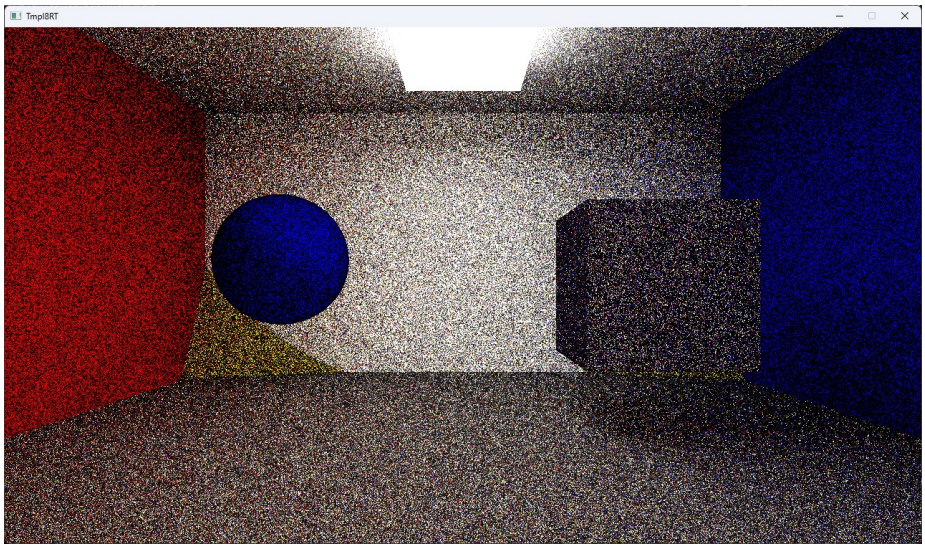

Figure 6: Rendering with another workflow

3 division of work

- Whitted style raytracer: Tim-50%, Saiyang-50%.
- Kajiya style ratracer: Saiyang 100%.
- Report: Tim-50%, Saiyang-50%.