Assignment 7: Ray Tracing

In this assignment, we will explore the beauty of light by implementing our own ray tracing algorithm within a GLSL shader. This will be our first venture into achieving realistic shadow effects within a GLSL shader. You are tasked with implementing a comprehensive ray tracing algorithm encompassing functions such as ray-object intersection, shading, texturing, shadow, and reflection. Let us embark on this journey.

1 Reading

Before diving into the shader code, you may our course slides as well as the supplementary reading materials to get a comprehensive understanding of the ray tracing algorithm. Here is the reading list:

- Course Slides on Ray Tracing
- Tiger Book Chapter 13
- Ray Tracing in One Weekend

2 Starter Code

The starter code for this assignment, found under `assignments/a7/`, includes `main.cpp`, `common.vert`, `basic_frag.frag`, and `ray_tracing.frag`. Upon successful compilation and execution, you should see an image with black color, as depicted in Fig. 1. This shader implements a complete ray tracer within the GLSL pipeline. The high-level concept involves tracing a ray for each fragment using the fragment coordinate as the screen coordinate. Subsequently, the color of each ray is determined by calculating the ray-object intersection within the scene. This color is then returned as the fragment color to render the ray-traced scene on screen.

3 Requirements

For this assignment, you are expected to implement five features of a ray tracer: *ray-object intersection*, *shading*, *texturing*, *shadow*, and *reflection*. All these functions should be implemented within the same shader file `ray_tracing.frag`.

Step 1 and 2: Ray-Plane and Ray-Sphere Intersection As a warm-up practice, we show you the implementation of the logic for the intersection between a ray and a plane in the function `hitPlane()`. Your task is to uncomment this implementation, understand the underlying

Figure 1: The default display of a ray tracer without objects.

principles, and mimic this logic for the ray-sphere intersection you will implement next. The ray-plane intersection involves solving for the ray parameter t by substituting the ray's equations into the plane's equation. A positive t indicates that the ray intersects the plane. Subsequently, we calculate the intersection point with the t value. You will begin with a default implementation provided within the `hitPlane()` function, which is initially commented out. After uncommenting the implementation and running the program, you should see a red plane (its default color) on the background (Figure 2). *You are not required to write your own implementation for this step.* Instead, read the sample code and comprehend how a `hit` object is created and returned.

Next, you will implement the ray-sphere intersection function in `hitSphere()`. Before coding, we recommend you review the theoretical concepts behind ray-sphere intersections in our course slides and read through the provided `hitPlane()` function. For ray-sphere intersection, you will need to calculate the coefficients A, B, and C for the quadratic equation describing the intersection, assess the discriminant, and make sure its value is non-negative. If the discriminant is positive, you will calculate the two quadratic equation solutions, choose the smaller one, and check if it is greater than 0. If $t > 0$, calculate the intersection coordinates and normal, then create a `Hit` structure with these properties and return it; otherwise, return `noHit`.

If everything works correctly, you should see three blue spheres on the screen with a red background (the red plane), as shown in Figure 3.

Step 3: Phong Shading Your task is incorporating the Phong shading model into the ray tracing process. You may reuse and adapt your previous work to implement the function `shadingPhong()`.

Figure 2: A red plane rendered with ray tracing.

If everything works correctly, you should see ray-traced spheres and the ground with the Phong shading model, as shown in Figure 4.

Step 4: Texture Mapping In this task, you will implement texture mapping for the scene's ground in `sampleDiffuse()`. Compute UV coordinates using the intersection point, then retrieve the texture color from the `floor_color` sampler. This texture color will be multiplied by the material's base color to produce the diffuse term. The modified color should be returned for the ground (with `matId=0`), while other objects should maintain their original material colors. **After implementing `sampleDiffuse()`, call this function in `shadingPhong()` to replace its original diffuse component calculation.** If the texture is applied correctly, you should see a ground with floor texture as shown in Figure 5.

Step 5: Shadow Your next task is to implement the logic for shadow computation by generating shadow rays from the intersection point to each light source in the function `isShadowed()`. Apply the `findHit()` function to ascertain whether any scene objects occlude the light, casting a shadow on the intersection point. You may also want to offset the origin of the shadow ray by a small measure (using the predefined `Epsilon`) to prevent self-shadowing. If everything works correctly, you should see shadows cast on the ground by the two light sources, as shown in Figure 6.

Step 6: Reflection and Recursive Ray Tracing Finally, we introduce reflections and recursive ray tracing to enhance the scene's visual depth and realism. This implementation consists of

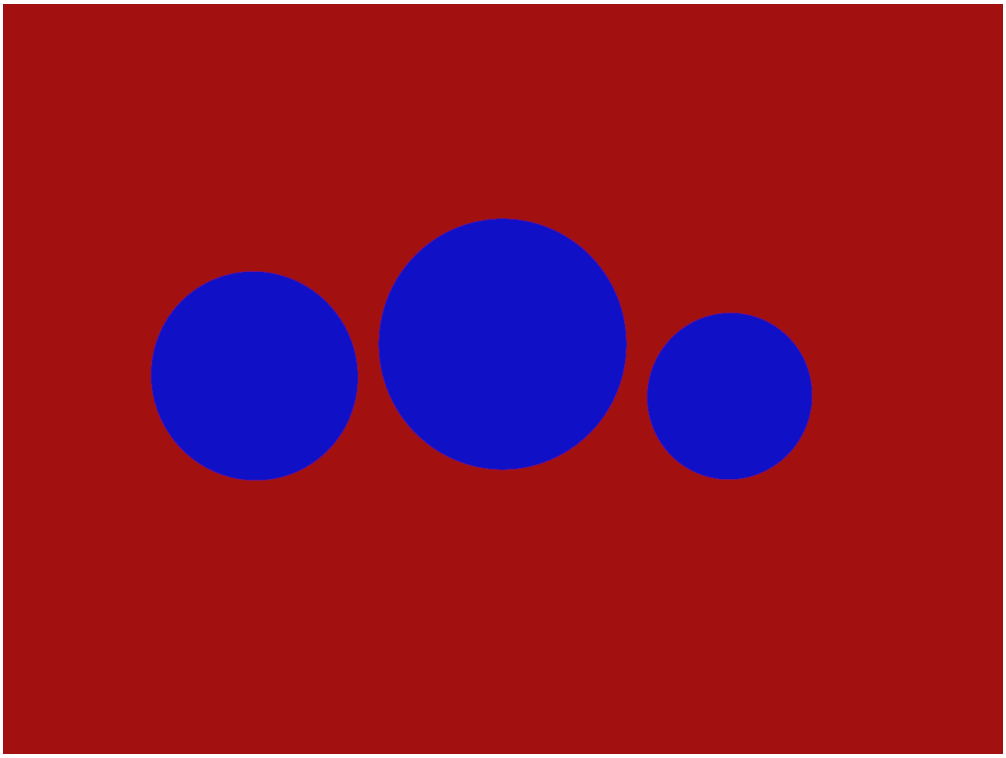

Figure 3: Three blue spheres on a red plane.

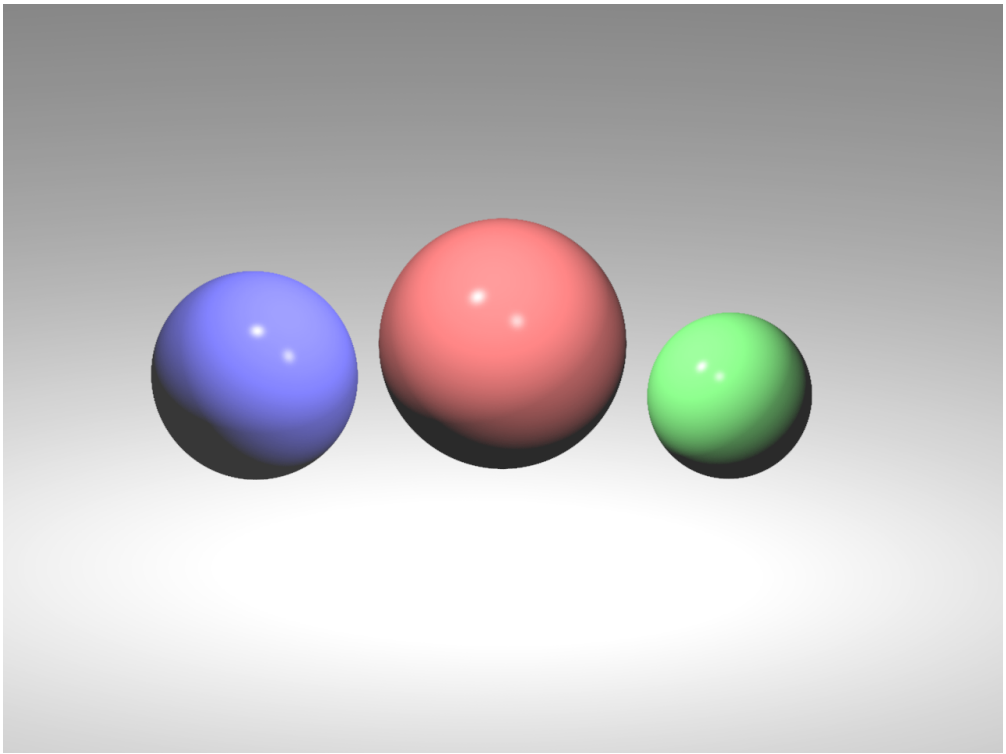

Figure 4: Three spheres on a plane with Phong shading.

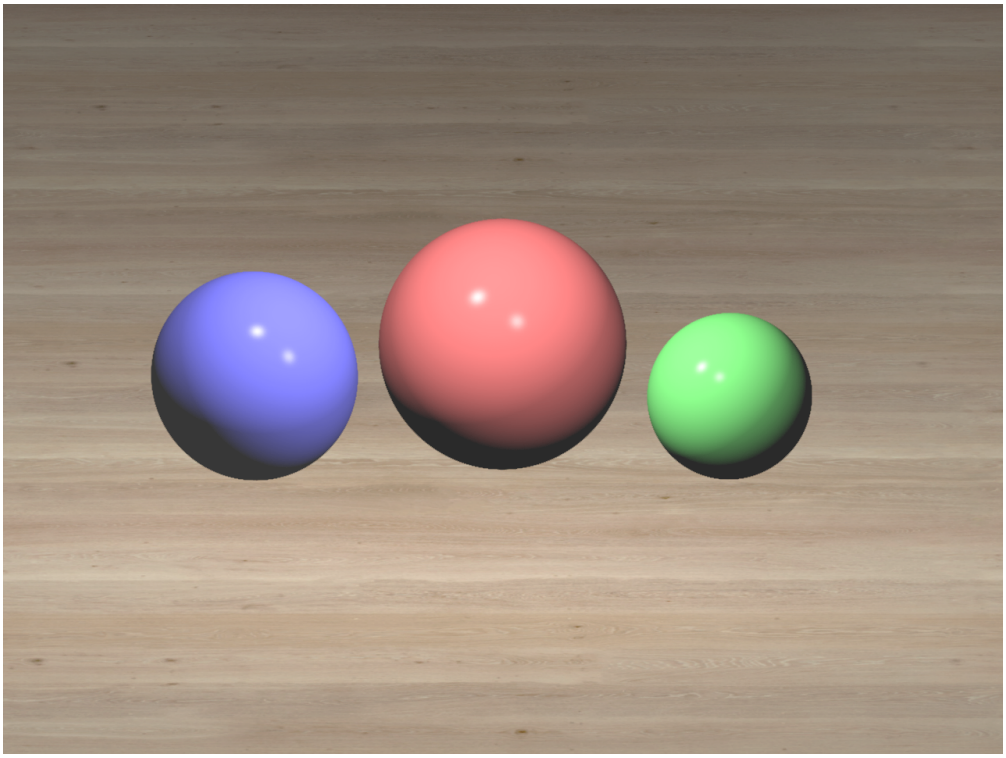

Figure 5: Texture the ground with a floor texture.

two parts. First, you are asked to increase the `recursiveDepth` from 1 to 50 to allow for extensive reflections. Second, you will calculate the direction of reflected rays based on the surface normal and incident ray direction at each point of intersection and then update the `recursiveRay` accordingly for subsequent iterations. This implementation happens within the `main()` function. After implementing this final piece of code, you can see spheres and the ground with reflective surfaces, which significantly enhance the realism of the rendering, as shown in Figure 8.

4 Creative Expression

In the Creative Expression section of this assignment, you are invited to customize the *camera, materials, textures, lights, and objects* in the default ray-tracing scene to create a unique ray-tracing image. While it is not required, you are encouraged to implement additional ray-object intersection algorithms to introduce new shapes into the ray tracer. Your rendering should exhibit complex lighting, shading, and shadow effects that distinctly differentiate between a ray-traced image and one rendered via rasterization. We provide a sample image as shown in Figure 8. This assignment's creative expression theme is the **Magic of Light**.

5 Submission

Submit the following components for evaluation:

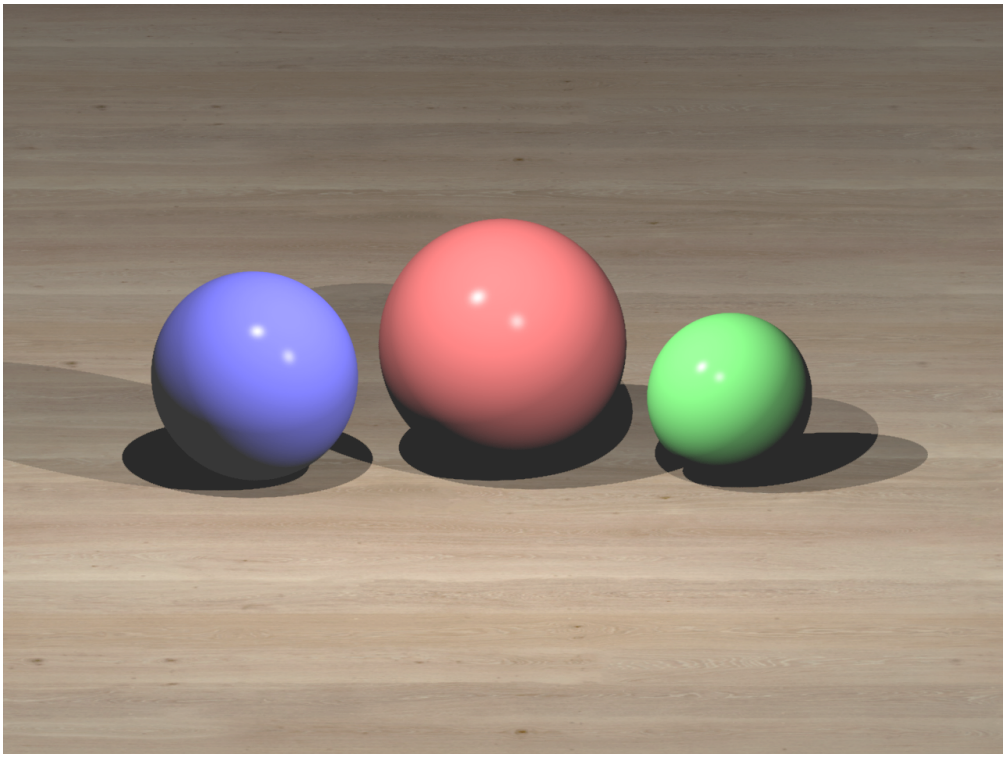

Figure 6: Adding shadow effects to the ray-traced iage.

- Your source code `ray_tracing.glsl`;
- Your default ray-traced image after finishing Step 6;
- Your customized ray-traced image for creative expression;
- A concise paragraph that provides a technical explanation of your implementation for the customized scene.

6 Grading

This assignment is worth a total of 8 points, with the grading criteria outlined as follows:

1. Technical contribution (6 points): The core of the grading is based on the correct implementation of ray-tracing functions. The distribution of points is as follows:
   - Step 1-2: 3 points
   - Step 3: 1 point
   - Step 4: 1 point
   - Step 5: 1 point
   - Step 6: 1 point
2. Creative expression (1 point): This aspect focuses on your ability to create new lighting and shadow effects with ray tracing.

Figure 7: Implementing reflection with recursive ray tracing.

7 Sharing your Work

You are encouraged to share your graphical work with the class. If you want to do so, please upload your image to the Ed Discussion post **A7 Gallery: Magic of Light**. This is an excellent opportunity to engage with your peers and gain recognition for your work. Share with us the magic of light you create!

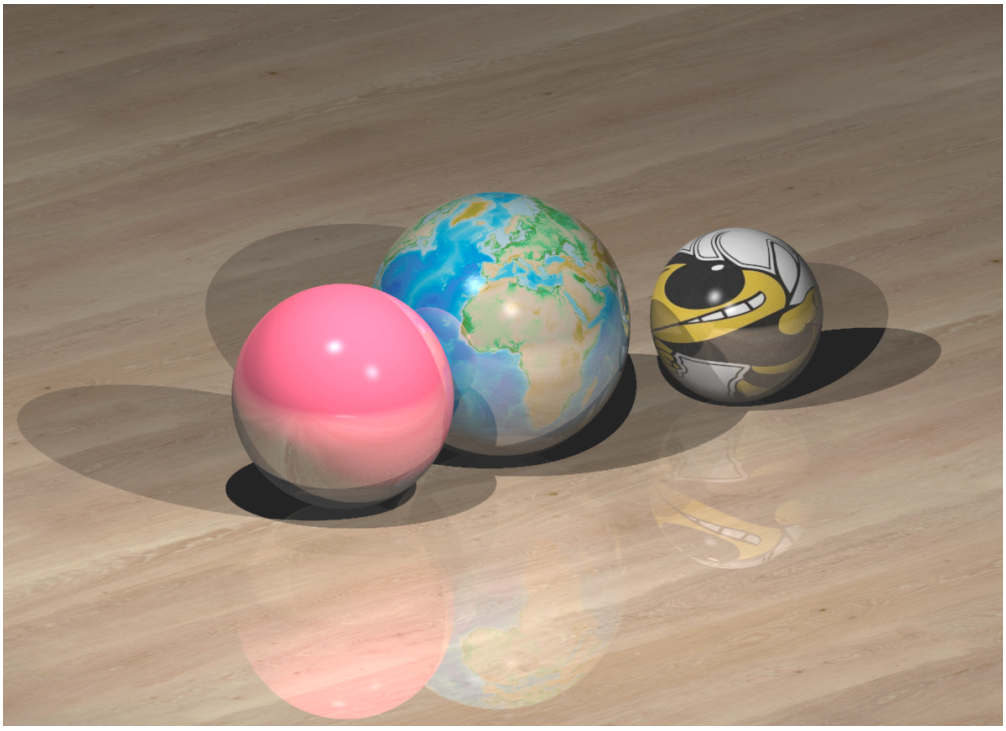

Figure 8: Customized scene with different camera angles and textures.