# Computer Graphics Coursework 2

Xi Zhang

April 2023

## 1 Parser

In order to finalise the parser, I have made modifications to the classes that extract data from the input JSON file. To verify the completion of the parser, I executed the command `./jsonexample ../examples/example.json` in the Terminal, and see the running result.

## 2 Pinhole Camera and Pointlight

Figure 1 illustrates the rendered image after integrating the pinhole camera and the point light into the scene. The image appears completely black as no shapes have been added to it yet. In the raytracing process, we calculate a ray from the camera for each pixel, cast this ray into the scene, and examine if it intersects with any objects. If a ray hits an object, we then determine if the point light source can also observe the object in order to generate shadows.

## 3 Ray-Sphere Intersection

Figure 2 has been derived from Figure 1 by incorporating the ray-sphere intersection. The algorithm employed for checking the intersection is based on the article "A Minimal Ray-Tracer: Rendering Simple Shapes (Sphere, Cube, Disk, Plane, etc.)" [2]. Additionally, the normal vector at the intersection point is computed by normalising the vector obtained from subtracting the hitting point from the sphere's center.

## 4 Ray-Plane Intersection

Figure 3 has been obtained by extending Figure 2 with the inclusion of the ray-plane intersection. The intersection is determined by comparing the hitting point with the edges of the plane and verifying if the intersection point is surrounded by all the edges. The normal vector at the hitting point is obtained from the normal vector of the plane.

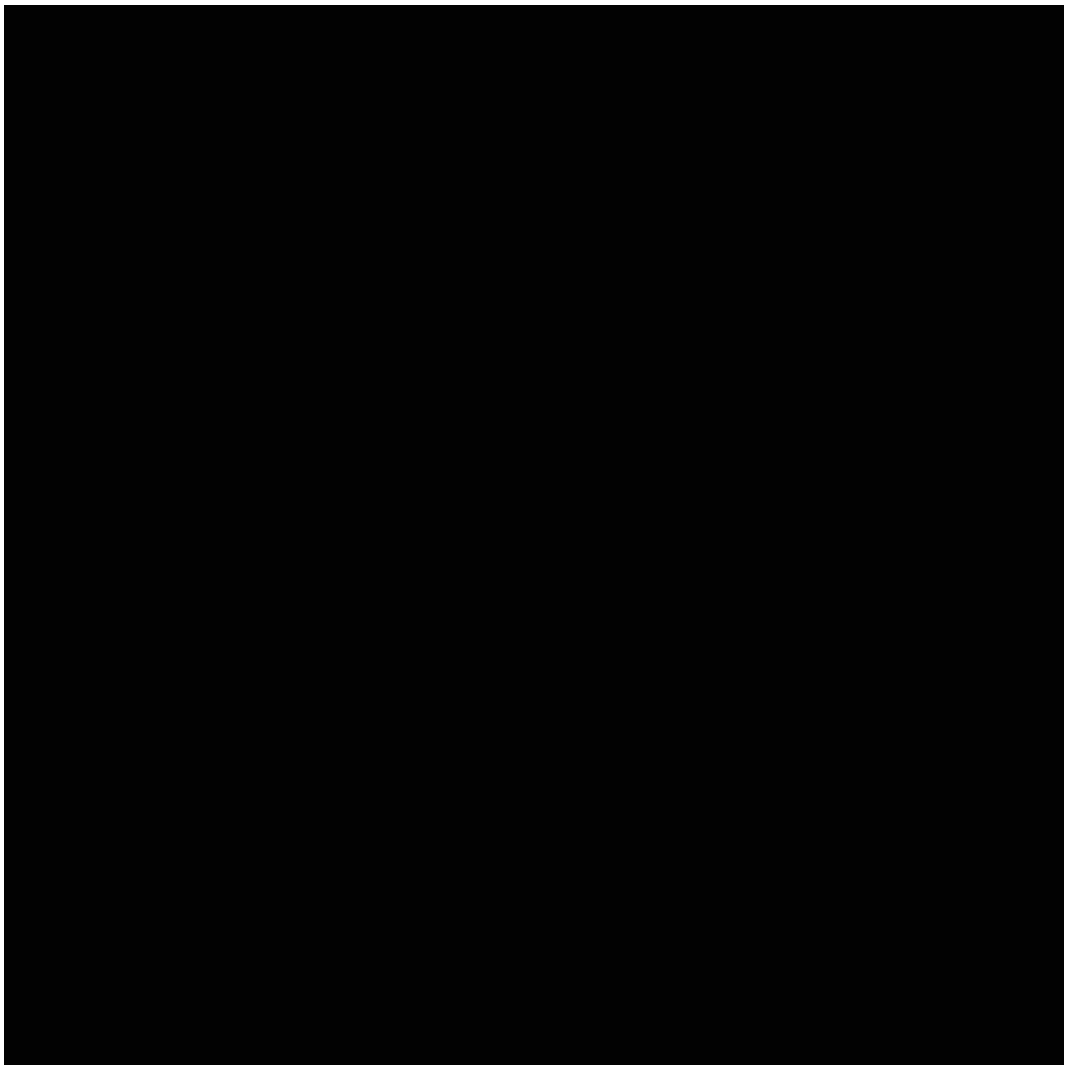

Figure 1: The output image with pinhole camera and point light implemented.

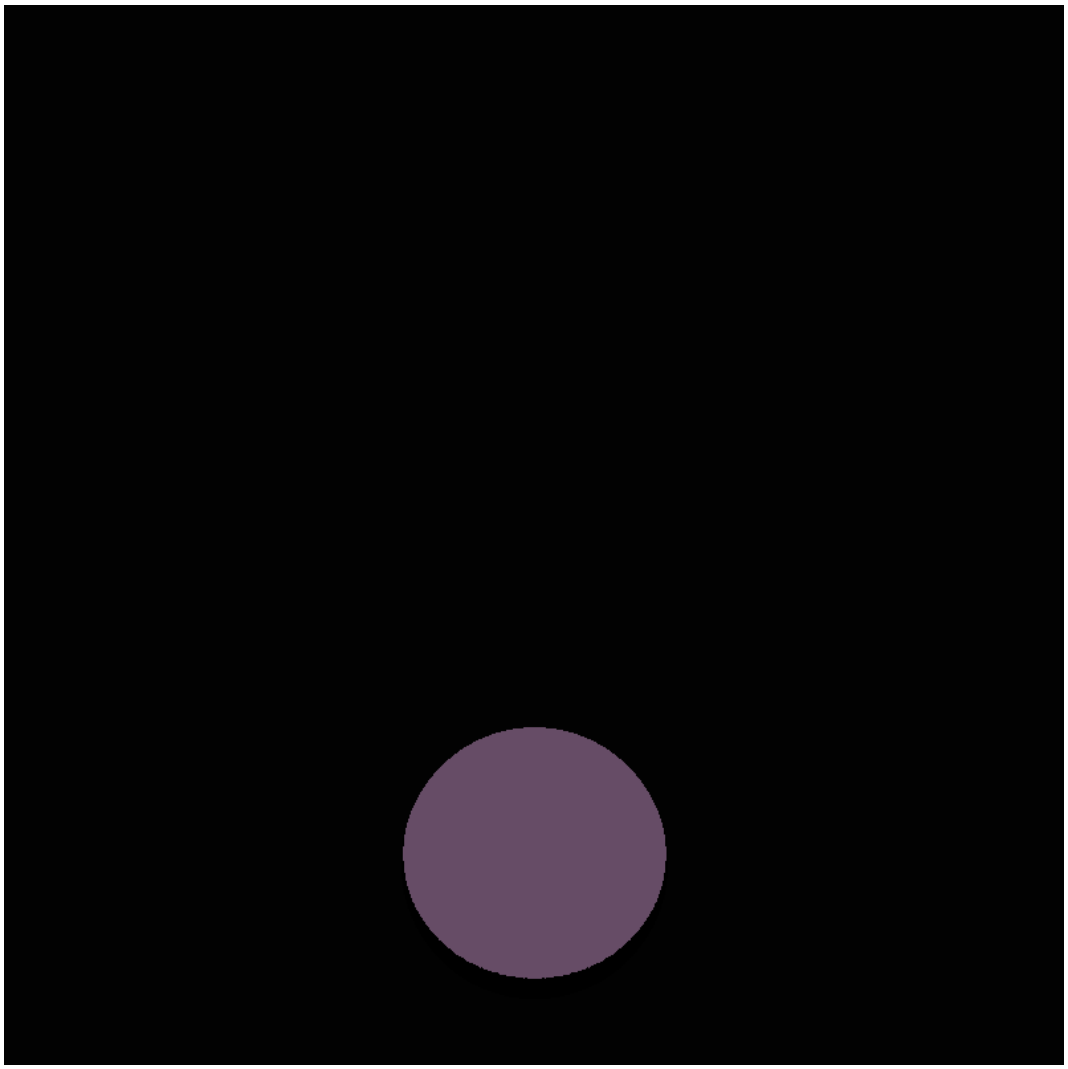

Figure 2: The output image with ray-sphere intersection added.

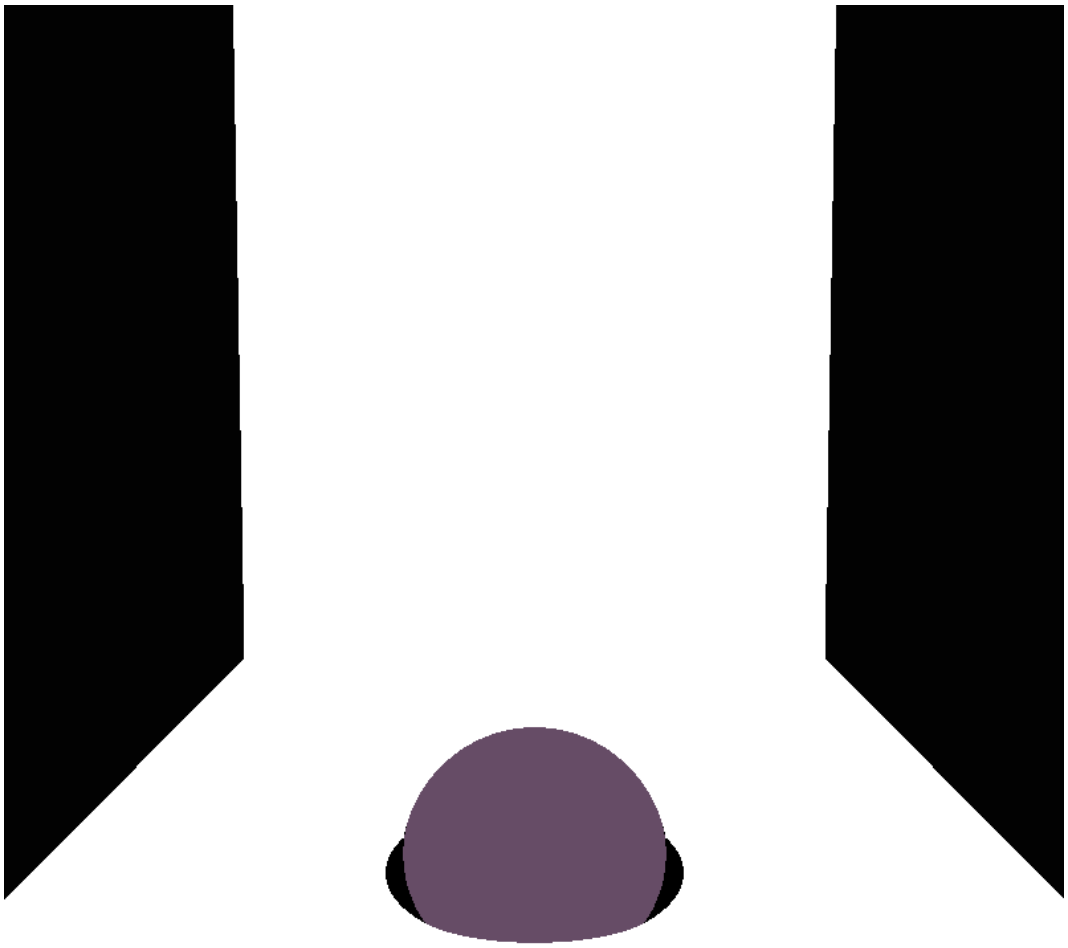

Figure 3: The output image with ray-plane intersection added.

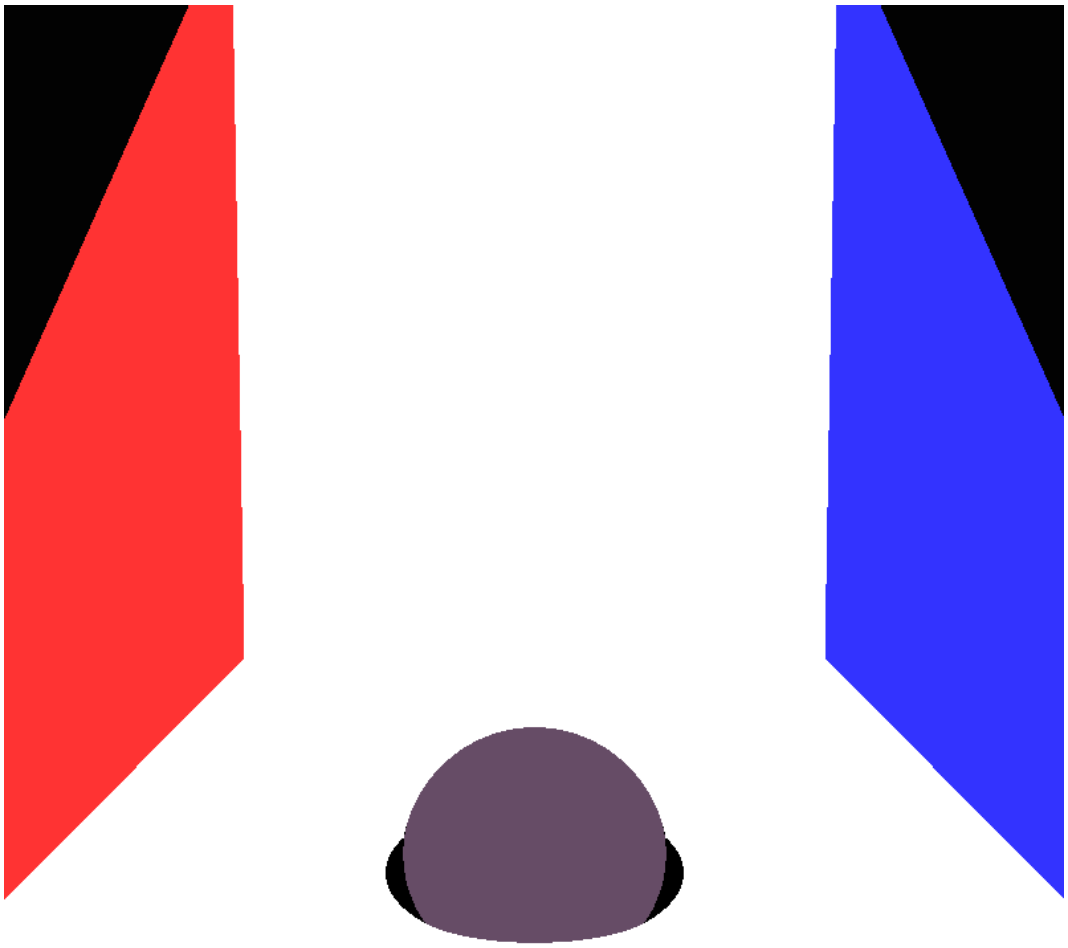

Figure 4: The output image with ray-triangle intersection added.

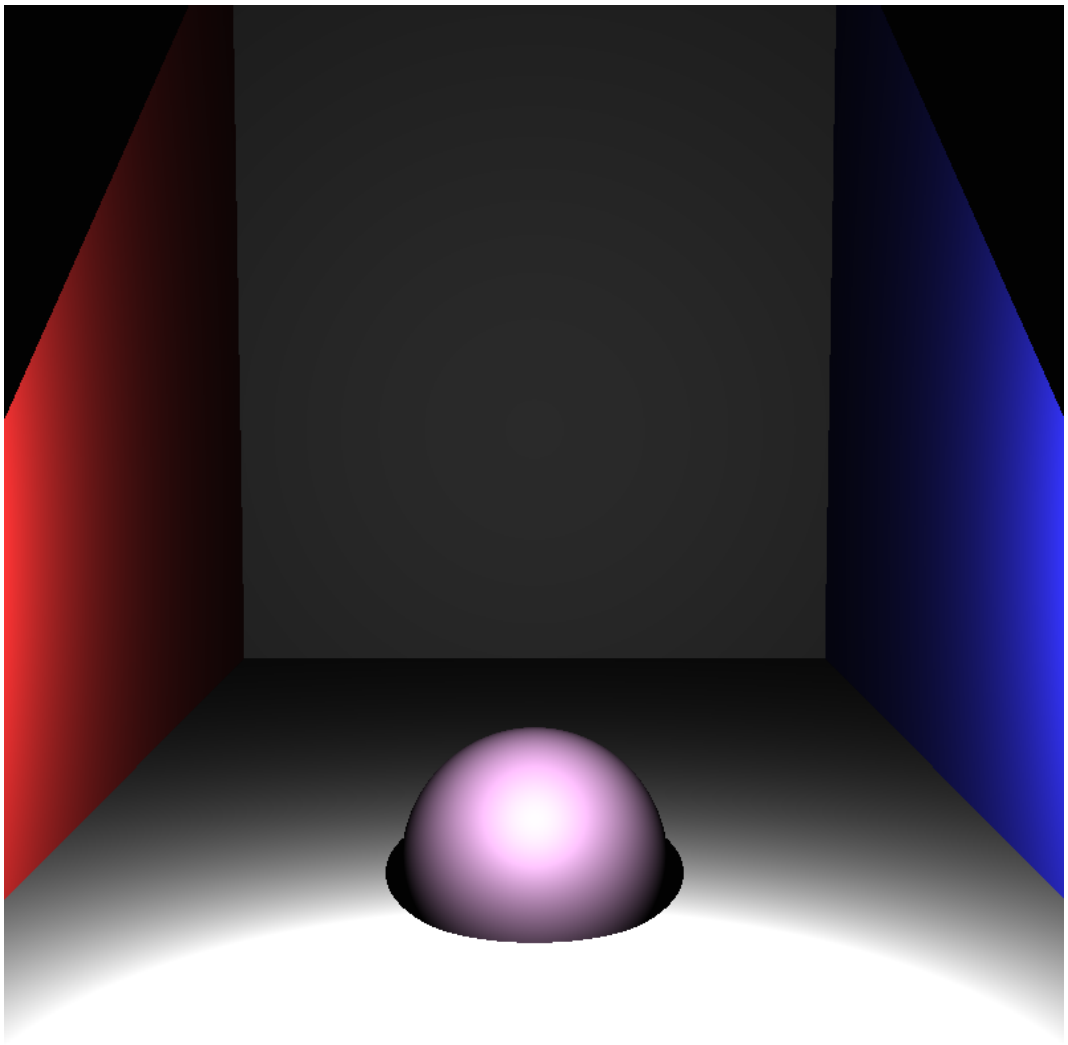

Figure 5: The output image with blinn-phong shading added.

## 5 Ray-Triangle Intersection

Figure 4 is derived from Figure 3 by incorporating the ray-triangle intersection. We calculate the u and v values for the intersection point, as well as a t value, and use these values to determine if the ray intersects the triangle.

## 6 Blinn-Phong Shading

Figure 5 is obtained by extending Figure 4 with the inclusion of Blinn-Phong shading. We compute the Ambient (Ia), Diffuse (Id), and Specular (Is) components of the hitting point if the ray intersects a shape, and add them to the pixel's colour in ray-casting.

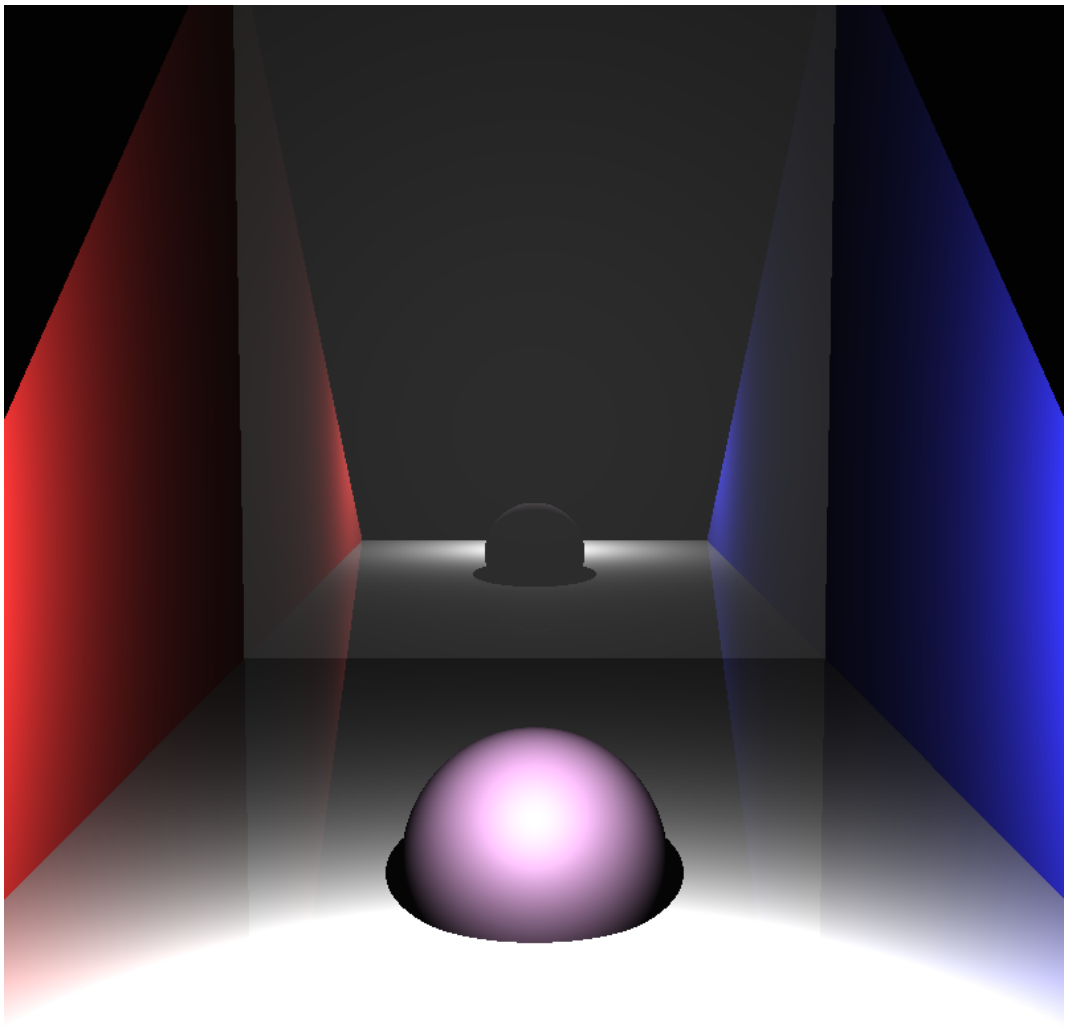

Figure 6: The output image with reflection added.

# 7 Reflection and Refraction

## 7.1 Reflection

To calculate the reflection of a ray, the direction of the ray and the normal of the hit point are used. The reflection direction is computed based on the reflection formula, with the hit point serving as the origin of the reflection ray. This reflection ray is then cast using the `castRay` function with depth+1 to obtain the reflection colour. Figure 6 has been modified from Figure 5 by incorporating the reflection.

## 7.2 Refraction

To calculate refraction, the `isInside` property is added to the ray, indicating whether the ray is inside a shape. This property is used solely for computing the relative refractive index

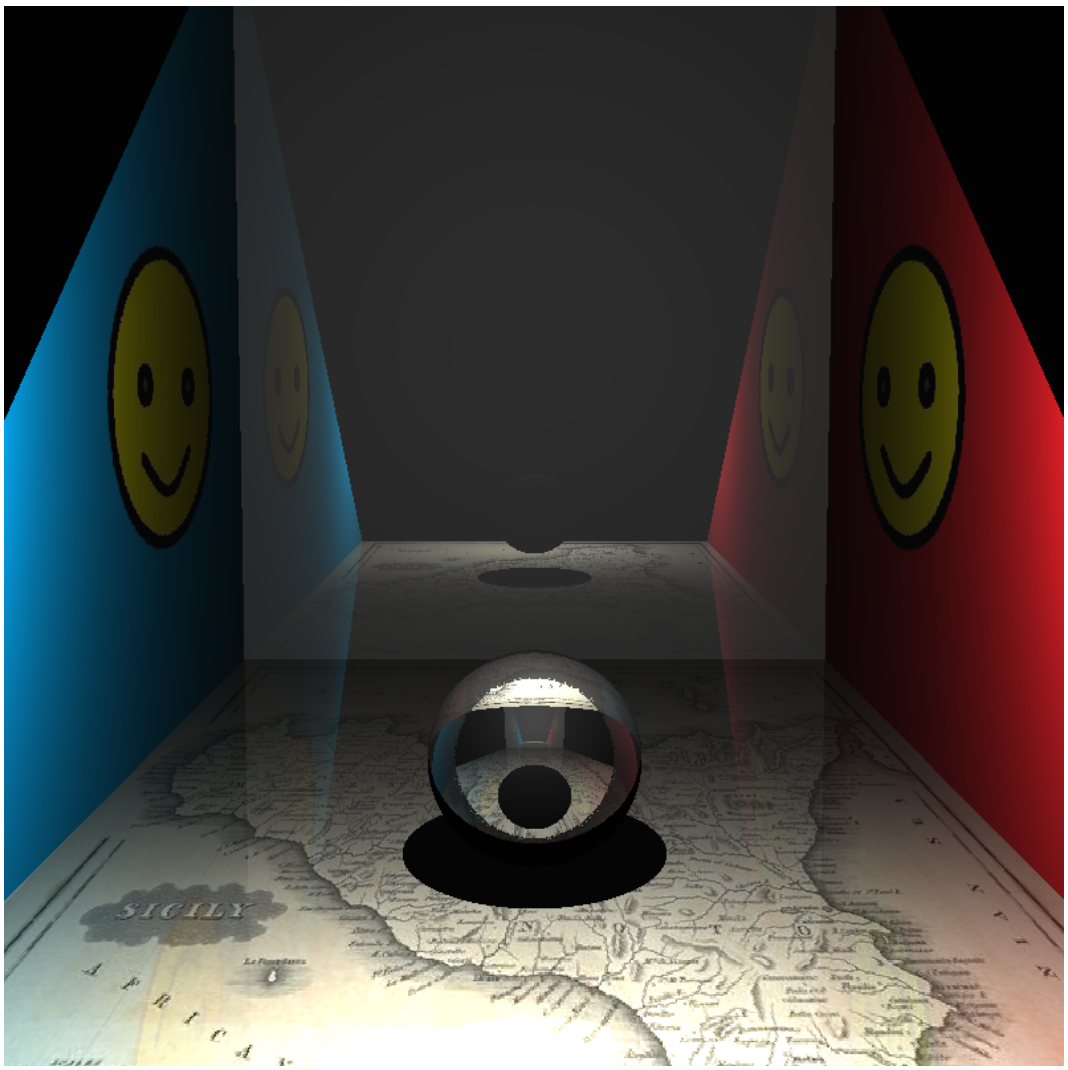


Figure 7: An output image with refraction, this image was generated after ray intersections and texture mappings are finished.

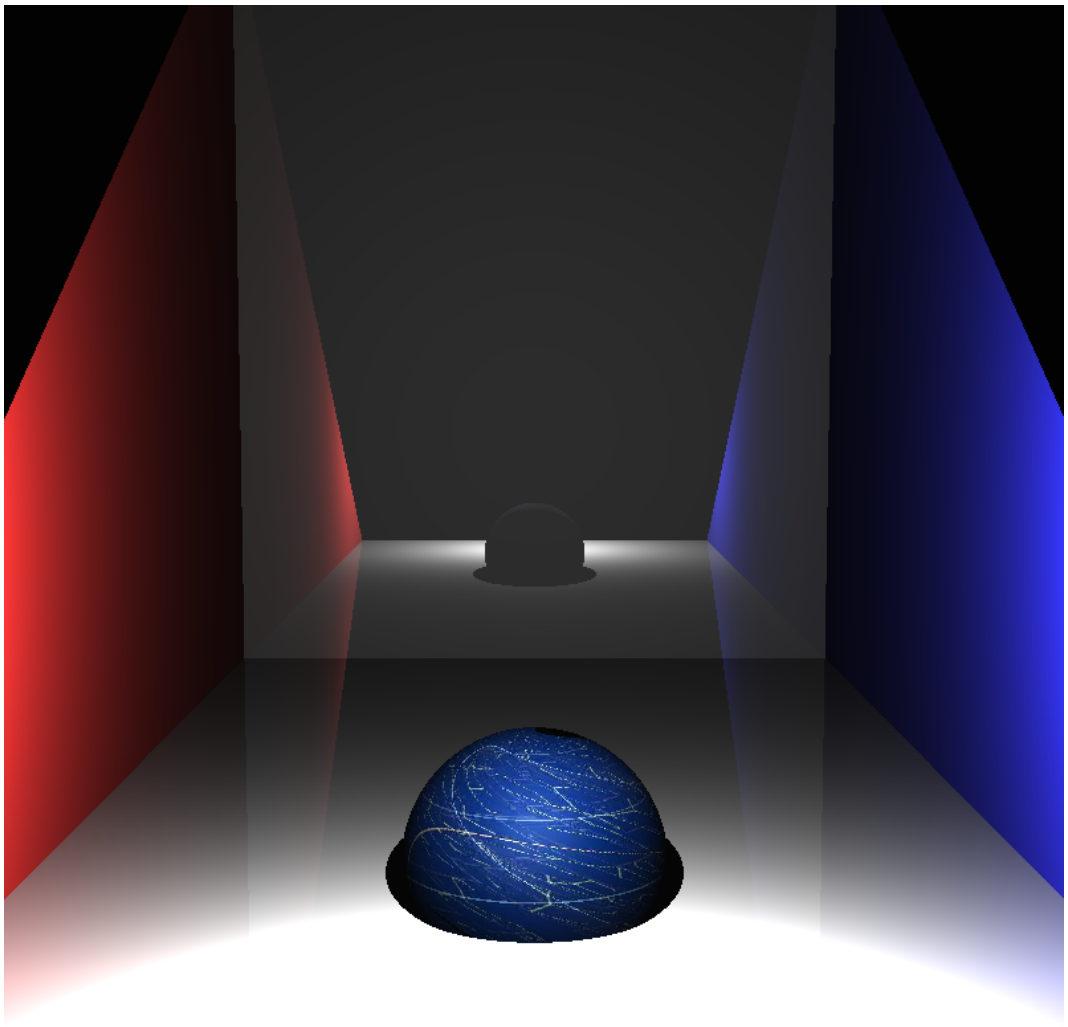

Figure 8: The output image with texture mapping in sphere added.

for refraction. Figure 7 has been modified from Figure 10 by substituting the textured sphere with a smaller glass sphere that has a displaced position. The rendered image obtained from this modification will demonstrate the effect of refraction. It is important to note that internal reflection should also be considered, as there are cases where the ray undergoes internal reflection inside a shape instead of refraction, if the conditions for internal reflection are met.

## 8 Texture Mapping in Sphere

Figure 8 has been modified from Figure 6 by incorporating the texture mapping of the sphere. First, the texture information is read from the JSON file if a shape has it. The stb_image.h file from the stb C/C++ library [3] is used for loading the texture. The u and v values for the hit point are calculated and used for the sphere's texture mapping. Furthermore, the `castRay` function has been adjusted so that if the material has a texture, the u and v values

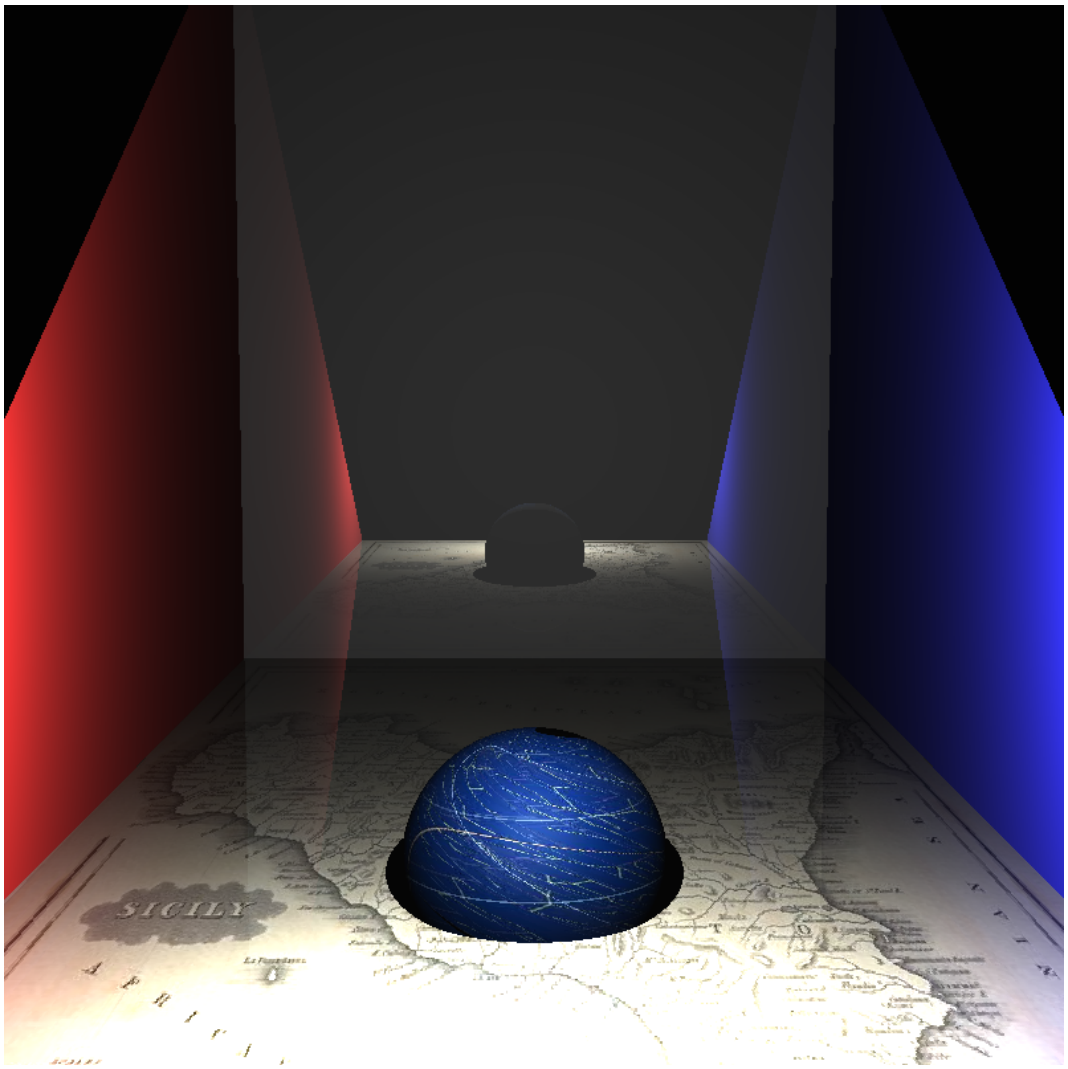


Figure 9: The output image with texture mapping in planar quad added.

are employed with the material's texture to determine the colour of the hit point, instead of utilising the diffuse colour when computing the Id value.

## 9 Texture Mapping in Plane

Figure 9 has been derived from Figure 8 by incorporating the texture mapping of planes. The u and v values for the hit point are calculated and used for the plane's texture mapping.

## 10 Texture Mapping in Triangle

Figure 10 has been obtained by extending Figure 9 with the inclusion of texture mapping for triangles. The u and v values that were previously computed during the ray-triangle intersection are now utilised for the texture mapping of triangles.

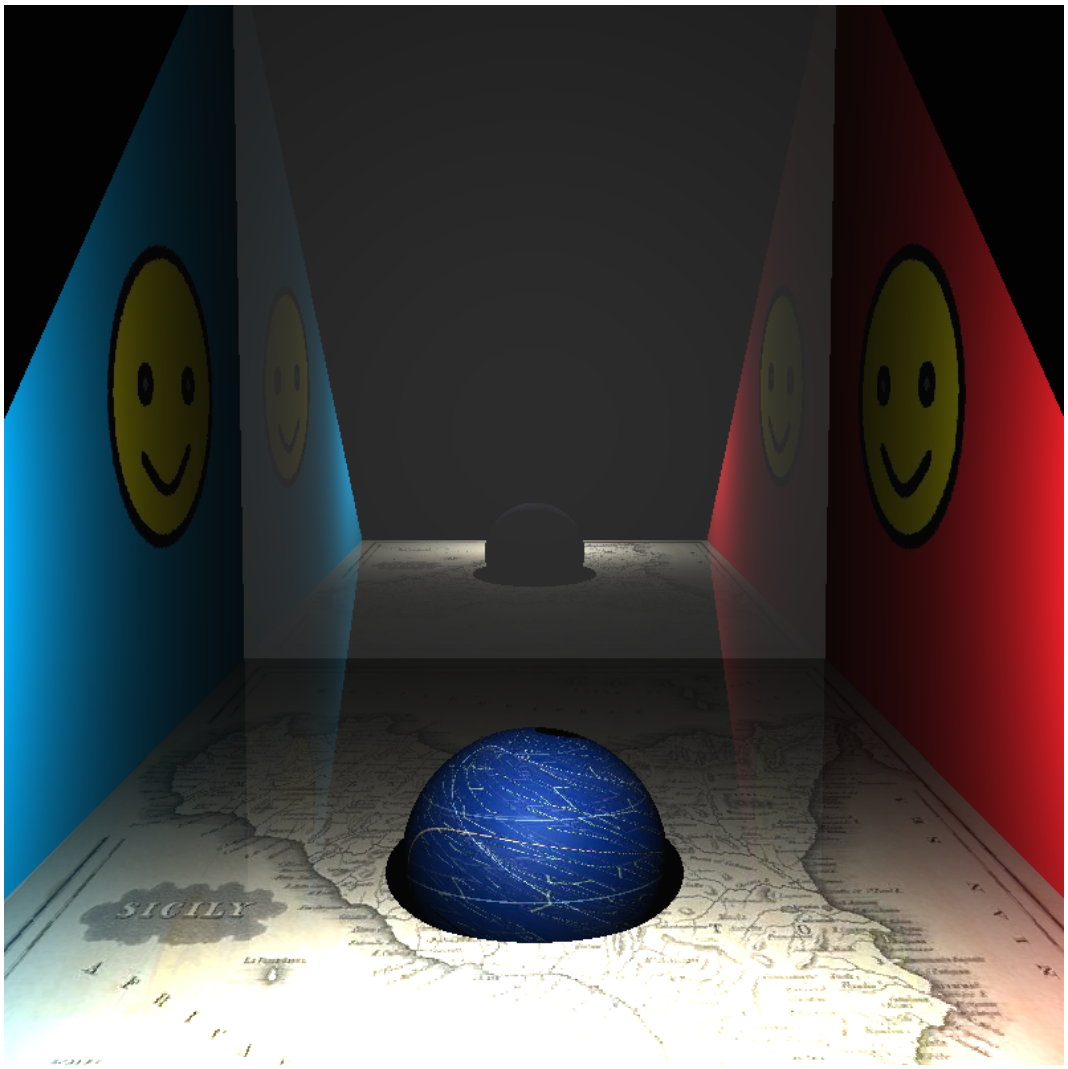


Figure 10: The output image with texture mapping in triangle added.

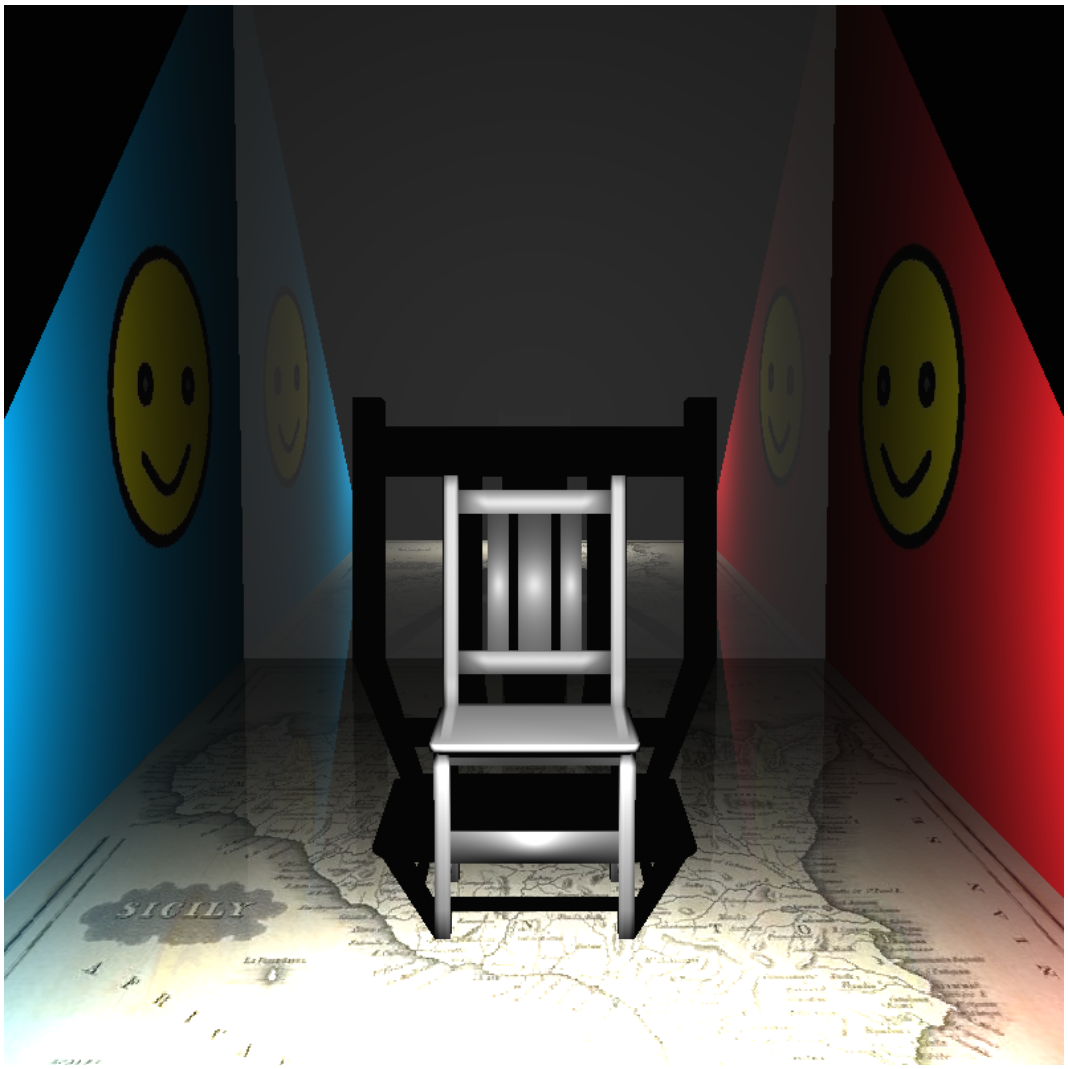


Figure 11: The output image with ray-triMesh intersection added.

## 11 Triangle Meshes

Figure 11 illustrates a chair without texture, represented as a TriangleMesh. To read the .ply files and create triangles for the faces of the triangle mesh, we utilise the happly C++ library [1]. Subsequently, a Bounding Volume Hierarchy (BVH) is generated for these triangles to optimise the ray-tracing process. As the triangles in the .ply files already have their normal values, we compute the barycentric coordinates for the hit points and combine them with the triangle's normal value to obtain the normal value of the hit point.

## 12 Texture Mapping in TriangleMesh

Figure 12 is adapted from Figure 11, where texture mapping has been added to the chair mesh. Barycentric coordinates are computed for the hit points of each triangle within the

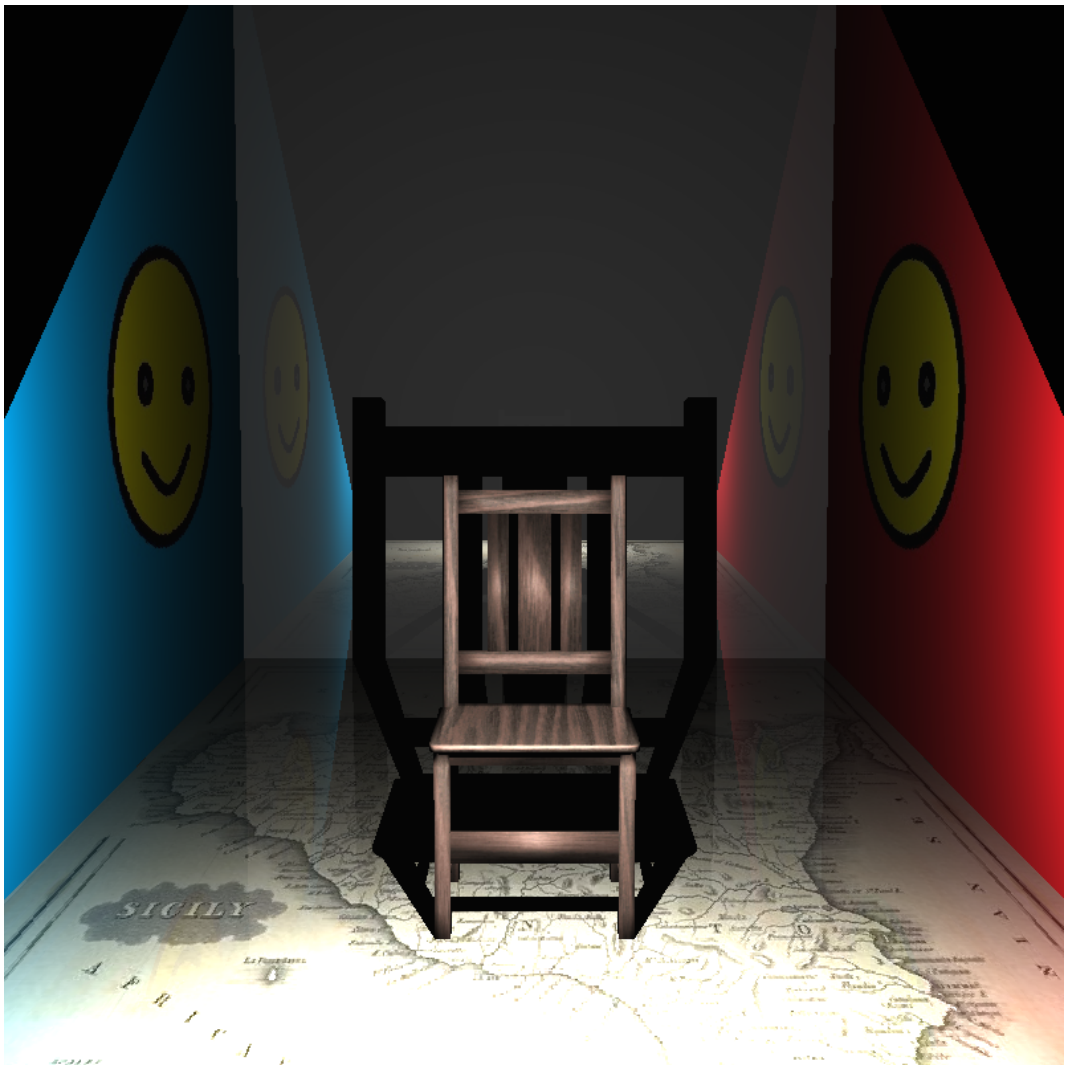


Figure 12: The output image with texture mapping in triMesh added.

triangle mesh, and they are combined with the model's UV mapping to compute the u and v values for displaying the texture on the chair mesh.

# 13 Bounding Volume Hierarchy (BVH)

The bounding volume hierarchy (BVH) is a binary tree data structure used for speeding up ray tracing. In my implementation, each tree node stores its bounding box and child nodes, and only the bottom-most leaf nodes store shapes. The tree is built by recursively splitting shapes based on their centers into left and right parts, and then calling the `BuildBVHTree` function separately with the left and right parts as input parameters. The bounding box for nodes that contain their own shape is computed from their shapes, while the bounding box for other nodes is computed from the bounding boxes of their child nodes.

When tracing a ray using the BVH, I first check if the bounding box of the current node intersects with the ray, and then recursively check the right and left child nodes separately. I also compare the distance from the hitting points to the ray's origin to get the closest hit. Comparing the rendering times with and without BVH on the scene from Figure 10, I observed no significant difference in rendering time. However, when checking ray intersections for triangle meshes, the rendering time is significantly reduced when using BVH for the triangles, as expected.

# 14 Distributed Raytracing

For distributed ray tracing, I have implemented a thin lens camera model along with camera jittering using random sampling.

## 14.1 Thin Lens Camera

Figure 13 displays the scene from Figure 10, but with the implementation of a Thin Lens camera model and 5x5 sampling per pixel. The Thin Lens camera model is achieved by generating multiple camera rays for each pixel, and the color of each pixel is computed as the mean of the ray-tracing results from all sample rays within that pixel.

To implement the Thin Lens camera, we compute the lens point and the focus point for each camera ray based on the camera's aperture size and focus distance. The lens point is used as the origin of the ray, and the direction of the ray is obtained by normalizing the vector from the lens point to the focus point. This allows for simulating depth of field effects, where objects at different distances from the camera may appear blurred or out of focus, similar to the behavior of a physical camera lens.

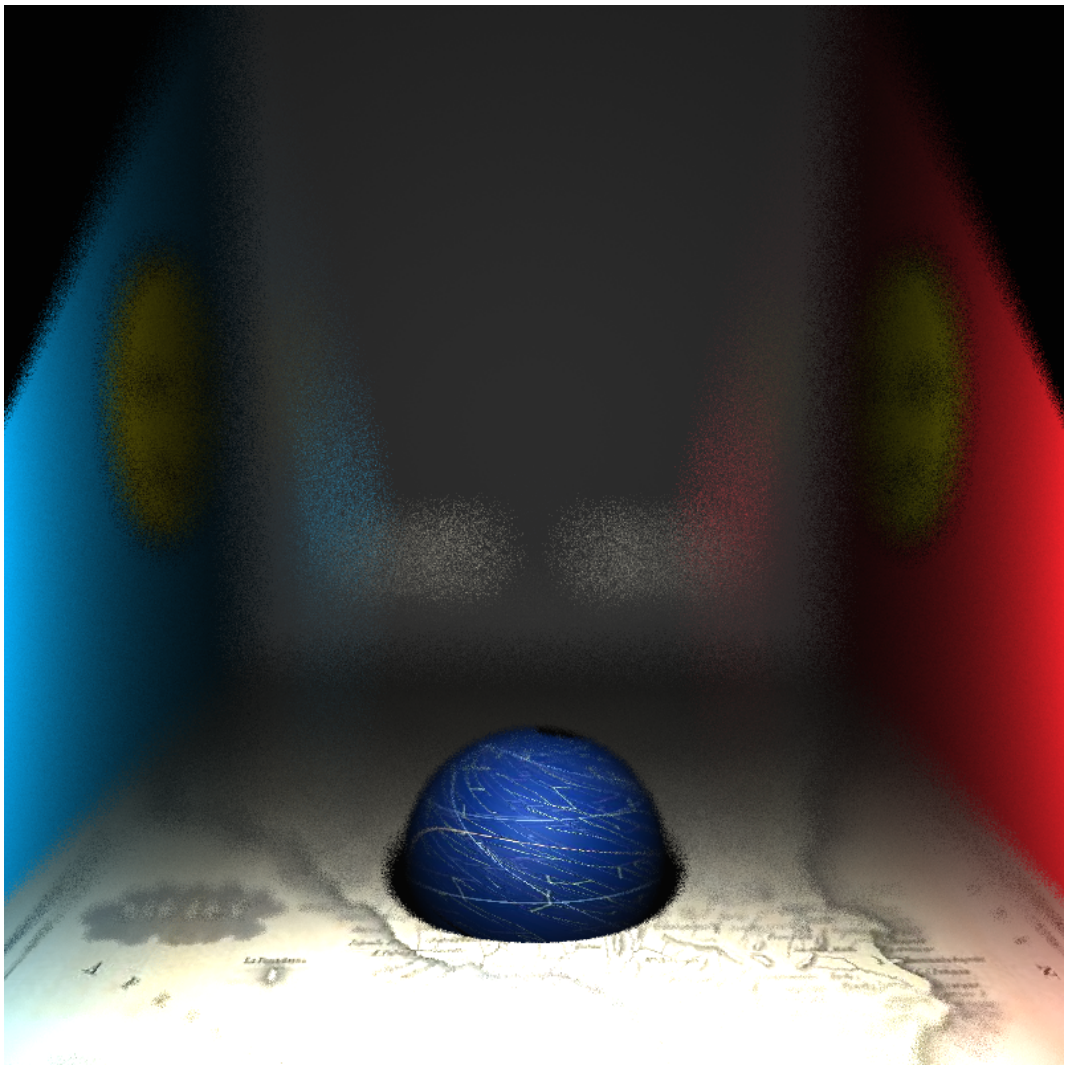

Figure 13: The output image using thin lens camera with 5 * 5 samples.

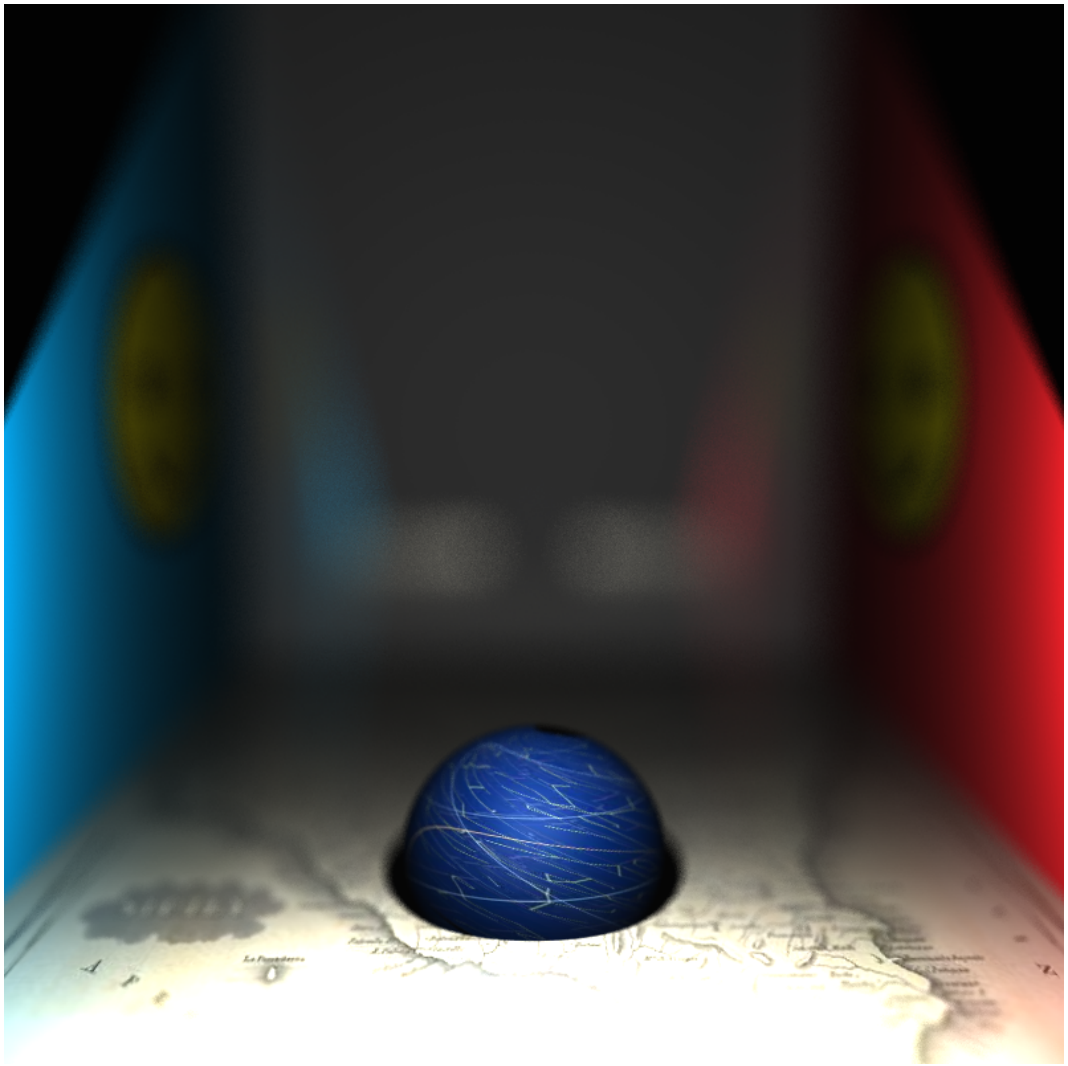

Figure 14: The output image using thin lens camera with 5 * 5 samples, and 25 camera jittering samples.

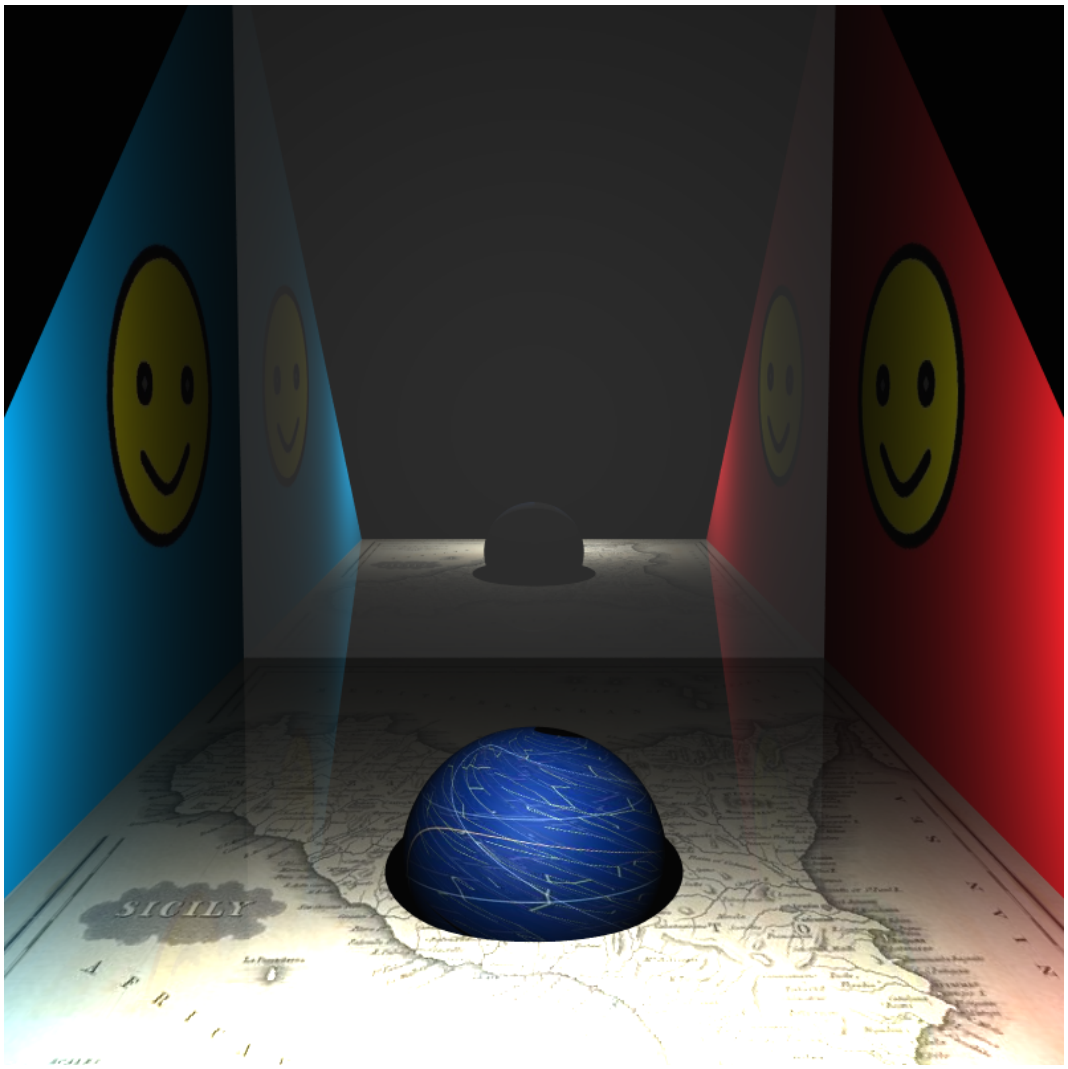


Figure 15: The output image using pinhole camera, and 25 camera jittering samples.

### 14.2 Camera Jittering

The camera jittering technique is achieved by dividing each pixel into sub-pixels and computing the mean of the sub-pixels for each pixel. Random sampling is also applied by changing the ray's direction with a random minor offset, which helps to reduce aliasing and produce smoother images.

Figure 14 is adapted from Figure 13, where camera jittering with 25 jittering samples per pixel has been added. Similarly, Figure 15 is adapted from Figure 10, where camera jittering with 25 jittering samples per pixel has also been applied. By comparing Figure 14 with Figure 13, and Figure 15 with Figure 10, it can be observed that camera jittering helps to produce smoother output images that looks more realistic.

## 15 My Own Scene

Figure 16 is the final output scene that showcases all the implemented features, except for the thin-lens camera due to limitations of having only one camera at a time. This scene is an upgrade from the image shown in Figure 10. The changes made to the scene include moving the sphere with texture to the left side, adding a clock triangle mesh with texture to the left of the sphere, adding a glass ball sphere for displaying refraction, and introducing the chair model that was previously shown in Figure 12. The clock and chair models are free resources obtained from the CGTrader website [4], and they both have UV-mapping with their Ply models.

## References


[1] happly Development Team. Happly - a header-only C++ reader/writer for the PLY file format. https://github.com/nmwsharp/happly, Accessed: April 21, 2023.

[2] Scratchapixel. A Minimal Ray-Tracer: Rendering Simple Shapes (Sphere, Cube, Disk, Plane, etc.). https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection.html, Accessed: April 21, 2023.

[3] stb Development Team. stb - single-file public domain (or MIT licensed) libraries for C/C++. https://github.com/nothings/stb, Accessed: April 21, 2023.

[4] The CGTrader Development Team. CGTrader Website - Free 3D Models, 2023. https://www.cgtrader.com/free-3d-models, Accessed: April 21, 2023.

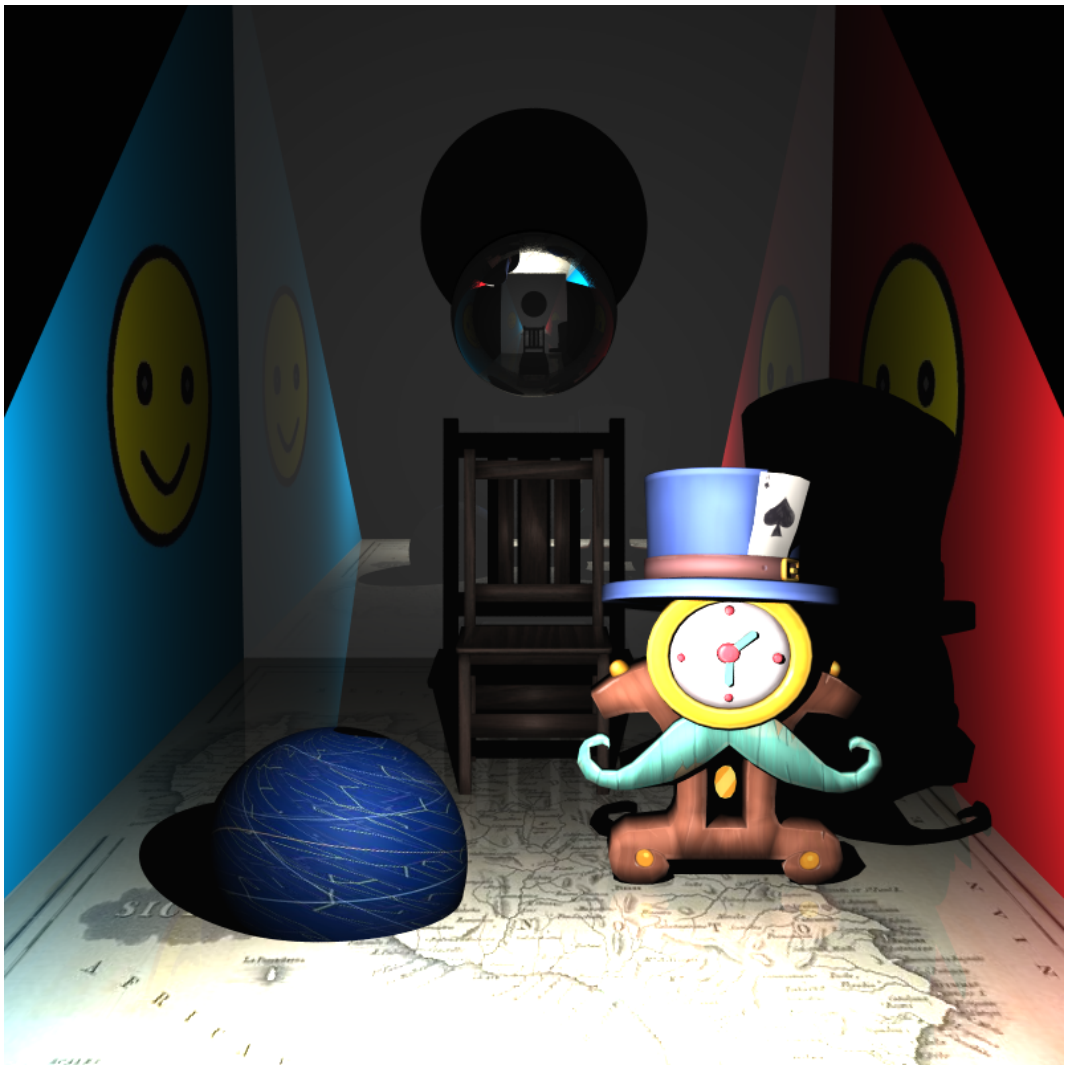

Figure 16: The final output image using pinhole camera with 25 camera jittering samples.