Computer Graphics cw2 report

s1968779 Yilun Cheng

March 2023

Contents

1 Introduction

In coursework2, I implemented the raytracer as required, including all based functions as well as the extended parts including distributed ray tracing, AreaLight and Thinlens camera. I created a customized scene to show all these effects, for which the rendered image is shown in Section 3.

2 Workflow

2.1 Basic scene building

2.1.1 Basic shapes

I started the coursework by taking an overall view of the program. The first thing I did is to complete the implementation for all kinds of shapes.

The basic shapes, including Plane, Triangle and Sphere, were done quite easily. The only thing needed was to load the information from JSON files into the **createShape** function, and implement detailed classes inheriting Shape. As for Trimesh, I used a similar structure as the one used in **.ply** files. I store a vector of Vec3f data as the vertices, and a vector of Vec3i as the faces, each of them represents the index of the 3 vertices on the triangle-shaped face.

2.1.2 PLY file parsing and explicit definition for TriMesh

An important thing to mention is that I implemented two ways of constructing a TriMesh as required. While allowing to load the mesh information from the JSON input file directly as other shapes, I also implemented the functions for reading data from PLY files, as required by the documentation. The file **trimesh.json** shows the examples of the two different methods of loading data for a Trimesh.

2.1.3 Camera

For camera objects, most information are common between Pinhole cameras and Thinlens cameras, including the position, lookat and up vector. Information about the unique parameter for ThinLens camera is loacted in Section 2.7.2. Besides, in the camera specifications section, I have added choices for input files to specify the sample size and the sampler. Currently, only "jittered" and "random" sampler are supported.

2.1.4 Scene creation

After done the shape structure, I implemented the createScene() function, loading the background color, the LightSources, and the Shapes from the input file. Also, I added a detection for possible accelerator choice. If no accelerator is specified, the program uses BVH as default. The implementation detail of BVH is described in Section 2.6.Currently, as the coursework requirements specified, only BVH has been implemented as a possible choice as the accelerator, which inherits the class Shape. There could be a class named Accelerator implemented in the future if any other kinds of it are required as well.

In my scene structure, a BVH (or Accelerator, if needed in the future) is stored independently in the scene. When the scene is called to find a hit, it will first check if there exists an accelerator. If so, it will call the findHit() function inside the accelerator. Otherwise, it just performs the global hit detection on every Shape stored.

2.1.5 Material

Then I started working on the Material structure. As specified in the example.json file, my material constructor simply read all information inside the input file and create a BlinnPhong instance. Since the only difference between the BlinnPhong model and the regular Phong model is the way it calculated the specular reflection, there are no difference in the variables stored.

2.1.6 LightSource

The LightSource is the final class I built, where the constructor is simple as well. For point lights, only the information given in the example is loaded, including the position, the is and id value. The details about AreaLights are described in Section 2.7.5.

2.1.7 Ray and Hit

The ray struct is simple, only has three components: the raytype, the origin and the direction.

Along with the 3 given Raytypes, I added two extra Raytypes, REFRACT and BADRAY. REFRACT type is used for checking if the current ray is inside a transparent object or not. It affects the calculation of refraction angles. BADRAY type is simply used to indicate that this ray is not valid, and return a black color.

The Hit struct is more complex. The basic parameter is a boolean named isHit, which indicate if this Hit instance is empty or not. Instead of only having the hit position, my implementation of the Hit struct contains 3 parts: the position of hit point, the distance between the incoming ray origin and the hit point, and the normal of the hit surface. This information makes the later calculation of ray color more convenient.

2.1.8 Bounding Box

The BoundingBox struct is created for BVH construction. Technically, this is only a Box in 3D space, where each of its faces is parallel to one of the axis plane. The only information recorded is 6 float value, representing the maximum and minimum value in x, y, and z.

2.1.9 Basic class structure done

At this point, all variables in classes have mostly been done, but the functions are not implemented yet. I made the program to simply call a printf() function whenever an object has been build, to show that all object information in the input file are accessed and stored successfully.

2.2 Camera ray shooting implementation

To start build the basic render function, the first thing needed is the ray from the camera. Instead of transforming the scene data into camera space, I put the camera in the 3D scene space and shoot rays directly.

2.2.1 Pinhole camera

From the basic parameters stored in the pinhole camera, I first calculate the u, v direction of the screen using camera's 'lookat' and 'up' values, and then get the 3D space vector for every pixel on screen. Then for each pixel on the screen, I shoot a ray from the pixel point on screen to the camera "position", which is the position of the pinhole. This ray is then passed to the scene to perform the ray tracing functions.

2.2.2 Thinlens camera

Very like the structure of Pinhole cameras, Thinlens camera also holds all basic camera information like "position", "lookat" and "up". However, Thinlens camera has unique variables, which are "focal" for focal length, "aperture" for aperture size, and "d" for the distance between the Thinlens and the screen center. The main implementation of Thinlens camera is located in Section 2.7.2.

2.3 Ray-object intersection

To get the color of the ray, the next thing I did was to implement the intersection function between the ray and each shapes.

For plane and triangle, the algorithm is simple. I first calculate if the ray's direction is parallel to the infinite plane where the shape lies. If so, they have no intersection point. If there is a point of intersection between the ray and the infinite, I calculate the position of the point, and see if it is inside the shape. The algorithm for triangle is simple. But for Plane, considering the condition of some weird shape, I first calculate if the point is inside any of the triangles composed by any three of the vertices. If so, I check if the point is in the triangle composed by another combination of vertices. if so, the point should be guaranteed to be inside the plane.

The algorithm for Sphere is simple as well. I first found if the ray ever touches the surface of the plane by checking the distance from the Sphere center to the ray. If so, I found the nearest point of intersection.

For TriMesh, I simply get every Triangle face of it, each calculating the potential intersection point with the incoming ray, and find the Hit with shortest distance.

After finishing the ray-shape intersection, I first rendered an image with direct ray cast, where the output for each ray is simply the diffuse color of the hit shape, the result is shown in Figure 1.

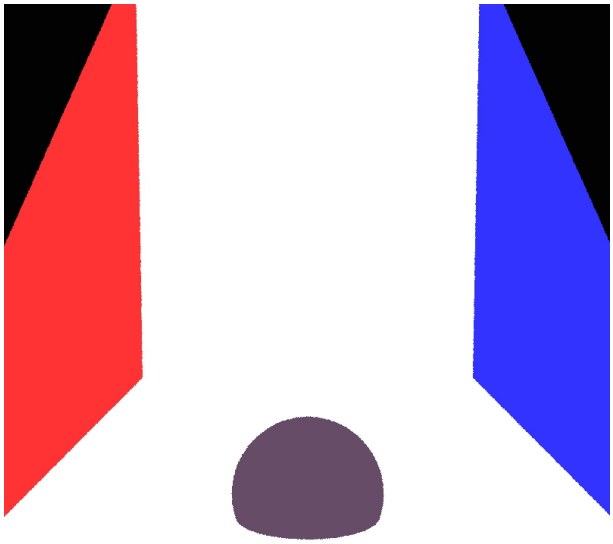


Figure 1: The ray cast image by simply returning the diffuse color of the first shape hit.

2.4 Shade on Material: BlinnPhong

2.4.1 Diffuse

The first and easiest shade color to be implemented is the diffuse color. When an incoming ray hits and object, I gett the normal of the hit point, and for each light source (at this point only point lights are considered) I shoot a shadow ray to the lightsource to check if the blocked by other shapes. If not, the raytracer calculates the dot product between the shadow ray direction and the surface normal, and get the diffuse factor by multiplies this value with the diffuse color of the shape's material. Finally the diffuse color is calculated by combining the diffuse factor, the light intensity for diffuse, and the distance between the light and the hit point.

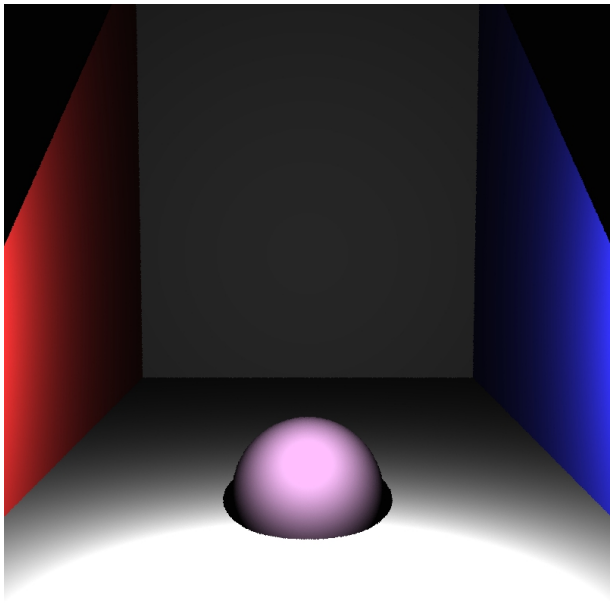


Figure 2: The diffuse reflection based on BlinnPhong model.

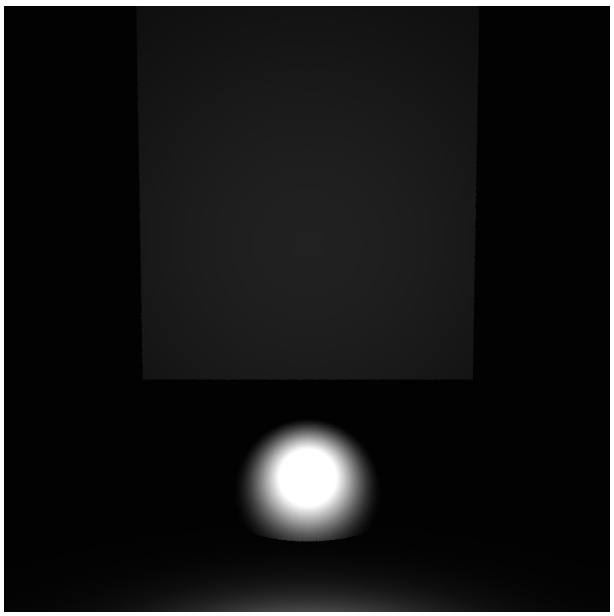
However, if the shape has a texture specified, the renderer will get the pixel value of the texture according to the position of the hitpoint on the shape, then use this value to replace the diffuse color of the shape's material.

2.4.2 Specular

As the BlinnPhong model introduces, the specular reflection of a light ray is calculated with the dot product between the normal and the average of the viewing angle and the light source direction. This can be done easily witn my program because my implementation of Hit Struct contains the information about the normal of the hited point and the direction of the incoming ray. So the only thing needed for the shape to get the specular reflection color is to pass the light source information, whihc are the position and the 'id' variable. The specular component in the given example input file is shown in Figure 3, while Figure 4 shows the combined result of diffuse reflections and specular reflections.

2.4.3 Reflect

After the calculation of direct reflections from lightsources, I implemented the reflection to scenes to achieve mirror-like effect. The raytracer simply get the direction of the reflected ray, and shoot this ray starting from the last hit position. The color got is then multiplied by the kr property of the current hited shape.



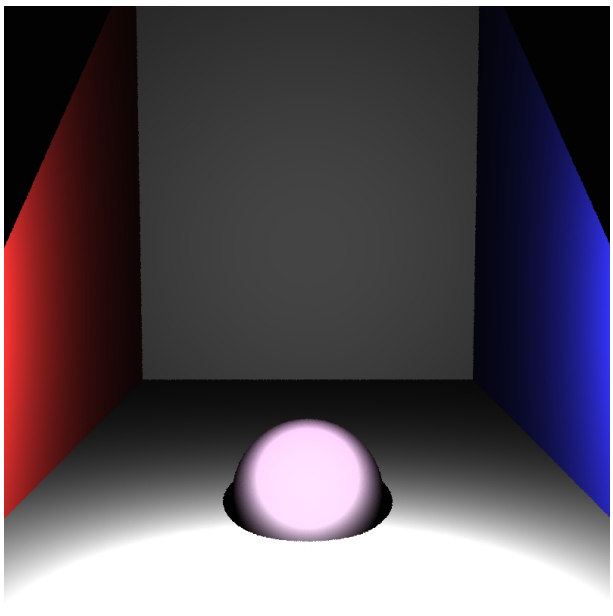


Figure 3: Left: The specular reflection based on BlinnPhong model

Figure 4: Right: The completly shaded image based on BlinnPhong model.

2.4.4 Refract

Many likes to the reflect rays, the raytracer simply calculate the refract ray with the necessary information, and adds up the refracted color with the kt value of the material. Note that refract rays are only traced when teh material is Refractive, which means the kt is larger than 0. In this case the material wll be assigned an extra variable named 'eta', to calculate the ratio of angles between incoming ray and refracted ray.

2.4.5 A Problem Solved

An interesting thing to mention here is that during the implementation of ray-shape intersection, an error occurs very often, where that are black lines showing up on different shapes, just like Figure 6 shows. I tried many different ways , including re-writing the reflection algorithm, changing the diffuse color or changing the position of the shape, but this error still shows up.

In the end it turns out to be the precision error for floating points. During my implementation I used float and Vec3f for most cases. When shooting shadow rays to the light sources, I used the hit point as the origin of the new rays. However, the precision limit of floating points makes t possible for the calculated position to be slightly different from the actual value, so there is a chance where the origin of a ray is "inside" the last hit object, therefore can not reach any light source, resulting in black colors.

My solution to this is to add a very slight amount of surface normal to the hit point, to ensure that the point will never fall into the other side of the surface of the shape.

2.5 Texture mapping

When considring texture mapping, the first hing o do is to decided the u-v position of the pixel from the hited position on the shape.

For plane and triangle, I decided to use the vertex v0 as the left-top corner of the texture, and use the (v1-v0) direction as the u-axis. All other calculation of the texture position are based on this strategy.

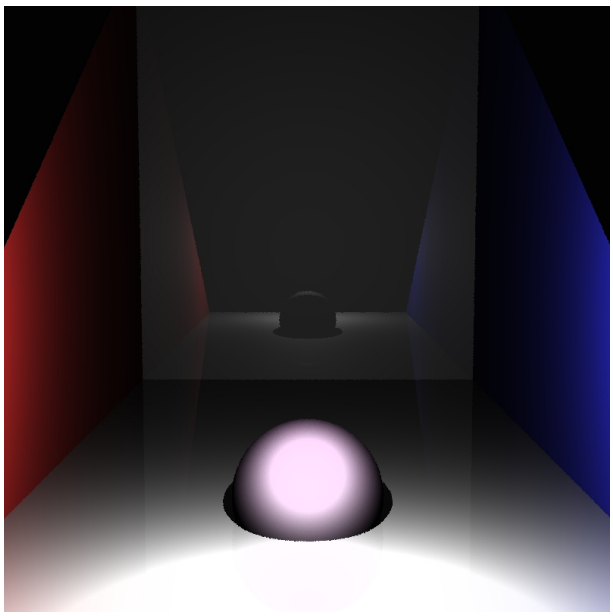


Figure 5: Light intensity adjusted by distance based on Figure 4, while adding reflection of other shapes.

For sphere, I use the highest point, which is the center position plus the radius times vector(0,0,1), as the top-bound of the texture, and the scene space x-axis as the left bound. The v value of the texture is decided by the angle between the hited point to the normalized z-axis direction, and the v-value is deicded by the angle between the hited point and the normalized x-axis ad y-axis direction.

For Trimesh, I found it hard to really implement a global texture mapping algorithm for all faces. The final solution is to do the texture mapping for every triangle faces, following the algorithm described above.

Here in Figure 7 is a sample of the three main texture mapping examples. It shows the texture mapping methods for Plane, sphere and TriMesh. The mapping algorithm for Triangles are just a simplified version of the one for TriMeshes, so I haven't made a separated Triangle shape for it.

2.6 BVH implementation

My implementation of BVH class is similar to a tree structure, where every node contains 4 parts: a Shape pointer, a BoundingBox, and two child nodes where each is a pointer to another BVH.

When createBVH() is called by the scene constructor, a vector of shape pointers will be passed to the BVH constructor, which first creates a huge bounding box that covers all shapes. Then for the longest axis in x,y and z, a new BoundingBox is generated with half of the length in this axis. It checks for shapes inside the new box. Shapes that could not be covered will be used to construct another sub-BVH. By doing so, the shapes are divided into two groups. Then the current constructor calls a new BVH constructor for each of the group, and stores the newly generated BVHs as its two child nodes the shape pointer in current node will be set as null pointer.

Note that there is a special case where one of the vectors of shape for child BVH contains no shapes, while the other one contains them all. In this case, before calling the new BVH constructors, the current constructor will push the first shape in the full-vector into the empty one, to ensure both BVHs in child nodes have at least one shapes.

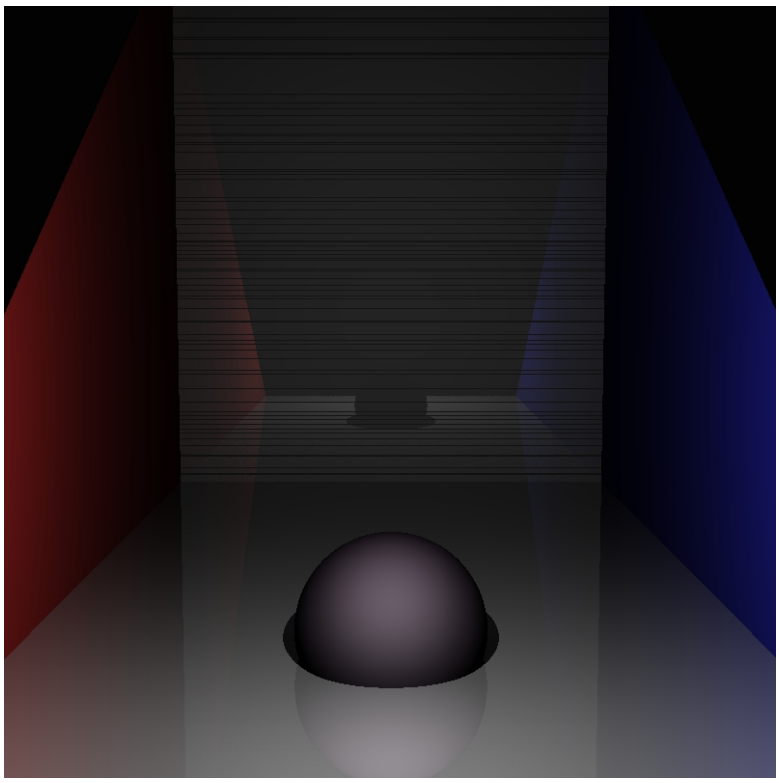


Figure 6: An error caused by the insufficient precision of floating point number. It can be seen that there are horizontal black lines showing on the back wall.

When there is only one shape left passed to the constructor, the constructor uses its bounding box as the bounding box in the current node, store the shape's pointer, and set its two child nodes to null pointers.

After BVH has been implemented, I ran several tests to see if this accelerator does work. I ran a simple scene with only a point light and several shapes and a complex scene with both AreaLights and Thinlens camera. The scene were rendered with different sample size in 100,200,300, and the time taken with BVH on and off were recorded. The result is shown in the table

Scene	Simple			Complex		
Sample Size	100	200	300	100	200	300
BVH On	23	54	79	38	87	123
BVH Off	30	70	105	52	114	167
Ratio(%)	76.7	77.1	75.2	73.1	76.3	73.7

Table 1: Comparison between rendering times with and without BVH

It can be seen that the BVH saves roughly 25% of the rendering time. This value is not as high as expected, because that the current scene is not widely extended in space, and many objects are partly overlapped in axes, making the BVH harder to construct efficient bounding boxes. Also, a large amount of time could be used on other functions like generating random sample for Arealight, and detecting ray-shape Hit for TriMeshes.

Though not in a large extent, the BVH structure indeed accelerates the ray-tracing task, while only store very few extra information other than several pointers and float numbers, taking hardly no memory space that can barely seen in the memory usage. In a word, BVH structure is both efficient and cheap.

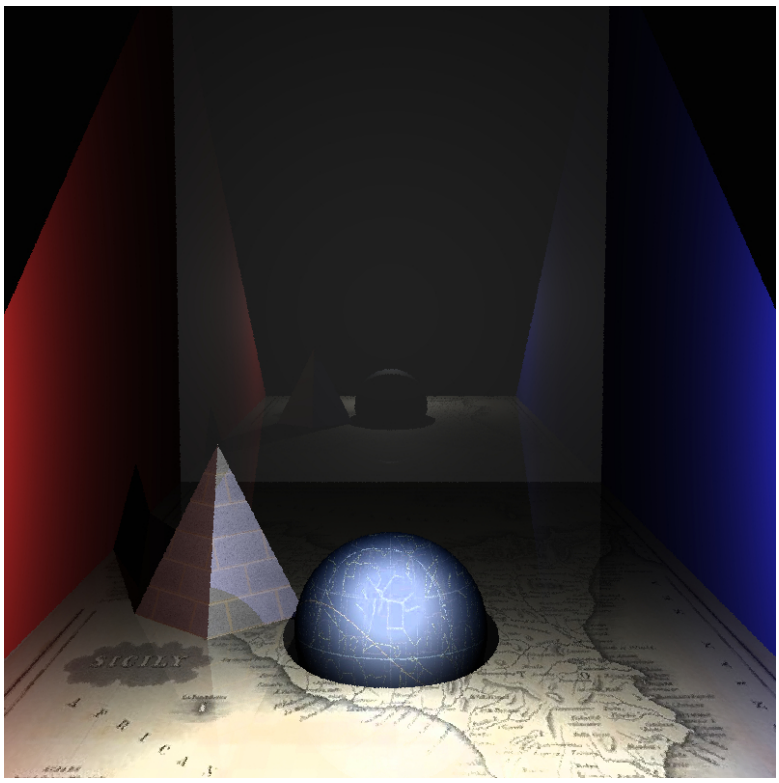


Figure 7: Based on Figure 5, while adding texture mapping to Sphere and Plane and TriMesh.

2.7 Distributed Ray Tracing

When implementing distributed raytracing, effects light depth-of-field and Thinlens camera could be achieved. Also, the lighting appears to be more realistic.

2.7.1 Overall effect view and Advanced Lighting

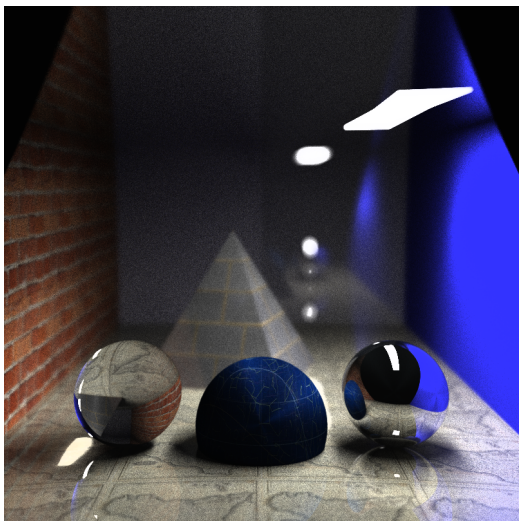
As the distributed ray indicates, instead of shooting rays directly from the light sources to the point, the ray tracer shoots random rays from the hit point and see if it ever reaches the shape of a light source. This allows more advanced lighting effects, like lighting through transparent objects.

Figure 8a shows that the distributed ray tracing implementation was a success. All effects are seen clearly, including Thinlens camera and depth-of-field effect. But here, the one to be focused is the advanced light effect with transparency. As shown in Figure8b, there is a clear light spot showing on the floor, which is generated by the area light being refracted by the translucent sphere. This is an advanced lighting effect that can only achieved by distributed ray tracing.

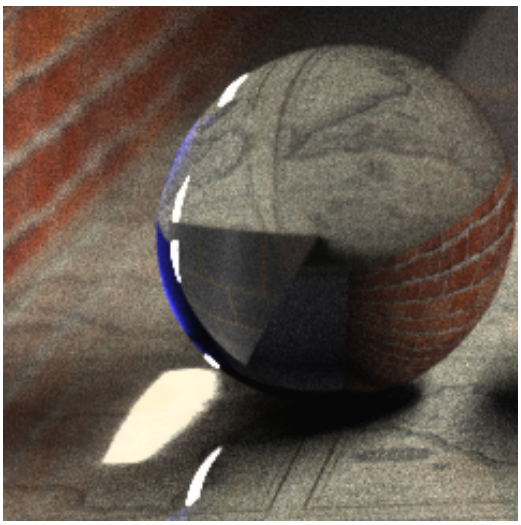
However, this lighting effect requires a higher bounce limit (higher than 3) than the one required by the coursework specification, where only a maximum of 1 bounce is allowed for distributed ray tracing. Therefore, the following examples and the final output presentation will be shown as combinations of Whitted ray tracing and distributed ray tracing. The distributed ray bounce will be limited to 1, while the original Whitted ray bounce will be set to about 3 to make the scenes more clear and realistic, as well as to lower the render time.

2.7.2 Thinlens camera

The main difference between a Thinlens camera and a Pinhole camera is that the ray shot from the screen will pass through a thinlens bounded by an aperture, therefore creating the effect of



(a) An image based purely on distributed ray tracing with n-bounce = 4



(b) Advanced lighting effect with transparency

Figure 8: Detailed effects achieved with the distributed ray tracing.

depth-of-field. In my implementation, like the one I did for pinhole camera, I first calculate the screen position and the thinlens center. Then for each pixel on screen, the camera shoot a ray to the thinlens, and calculate the refracted ray based on the focal length of the thinlens and the relative position between the screen pixel and the position where the ray hits the thinlens. The refracted ray is then passed to the scene to perform render function.

To achieve the blurring effect in depth-of-field, my Thinlens camera shoots a fixed number of random rays for the current screen pixel to different locations on the Thinlens, and each of them gets refracted into different directions. By doing this, the program can simulate the real light-ray behaviour, resulting a depth-of-field effect.

A straightforward comparison between the Pinhole camera and the Thinlens camera is shown in Figure 9. In Figure 9a, every shape on every pixel is clearly visible. In Figure 9b, however, only the shapes at a certain distance from the camera, including the spheres and part of the floor, are visible. All other parts of the image are blurred.

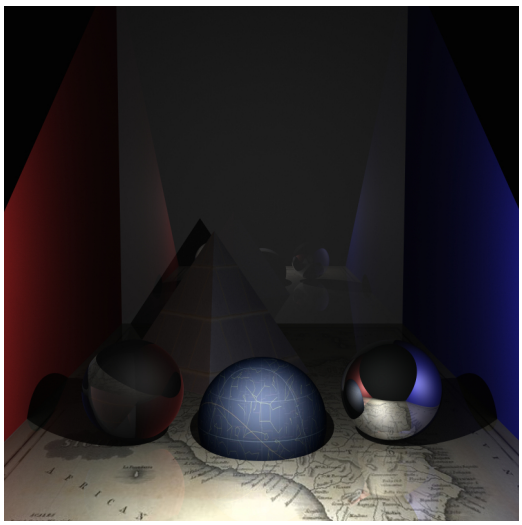
2.7.3 Thinlens vs Pinhole

From the two images shown in Figure 9, we can tell that both camera models have their advantages. For more descriptive scenes where people want every detail to be clear and straightforward, a Pinhole camera would do the job with high speed and clearer details. However, if the desired output is a more realistic iamge, more like a human-eye's view, then Thinlens camera will be the better choice.

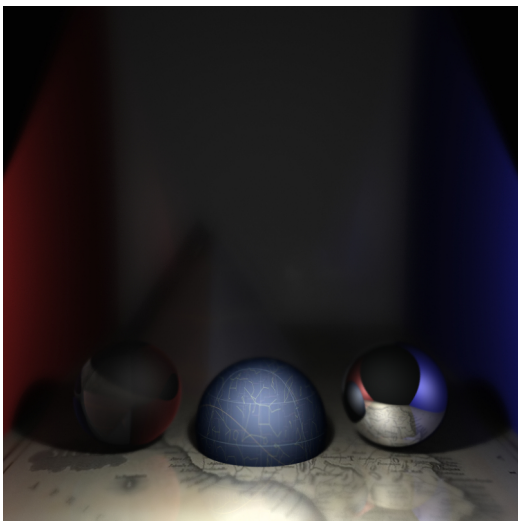
2.7.4 Impact of the parameters of a Thinlens camera

Apart from the three basic parameters held by all cameras, Thinlens camera has three unique parameters, which are: f for focal length, d for the distance from the screen to the lens, and a for the aperture size. Here I have listed a series of rendered images to show the impact of these parameters.

In rendering, f and d decide the distance from the camera at which the objects will be clearly visible on the screen. Figure 10 shows the images of an identical scene under 3 Thinlens cameras with different f and d values. In Figure 10a, only the closest sphere on the left side



(a) A completed scene under Pinhole camera



(b) The same scene, but with a Thinlens camera

Figure 9: An identical scene with different camera used.

can be seen clearly. In Figure 10b, the one in the middle becomes the clearest. And in Figure 10c, we can onnly see the furthest sphere on the right side clearly.

The a value decides the range of the clear zone, or in another work, the range of distance at which the object is not too blurred to be seen. Figure 11 shows the images of an identical scene under 3 Thinlens cameras with different a values.

In Figure 11a, the a value is set to $f/4$, and the clear zone is only a small range of distance near the distance of the middle sphere.

In Figure 11b, we can see things around the middle sphere clearer, but the further objects, like the reflected image on the back wall, are still blurred.

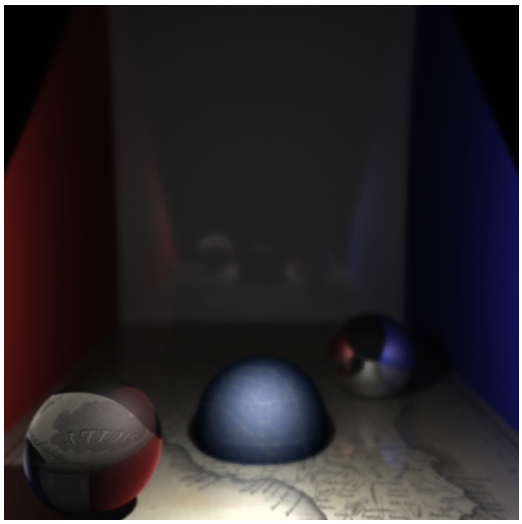
In Figure 11c where the aperture size is $f/64$, almost all parts of the image are clearly visible. In this case, the Thinlens camera behaves just like a Pinhole camera at a close distance.

2.7.5 Area light source

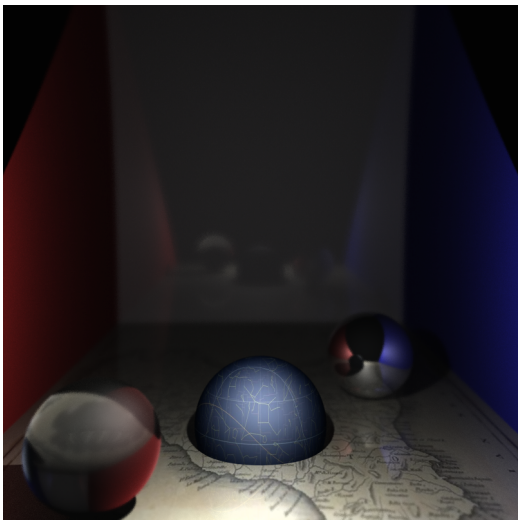
The biggest difference between an Area light source and a point light is that the AreaLight has a physical shape that could intersect with other rays. In the implementation, I separated the AreaLight info and its shape. The key to combine these two object is the shapeId. The AreaLight will need to specify the ID of its shape, while the shape must have the same ID as specified. When the ray-tracer tries to get the reflections on a hit point from an area light source, instead of shooting one single ray, the AreaLight chooses a fixed number of random position on its shape, and the hit point will shoot rays towards all this positions, to see how many of them actually reaches the area light shape. The ratio of shadow rays hit the area shape will be the light strength of the point from this light source. With these methods, the program is able to achieve soft shadows. In my implementation of the raytracer, with random sampling functioning in primary rays, the AreaLight simply choose one random position on its shape and shoot a ray to the destination point. The integrated value of all PRIMARY rays seeing at this point will be the final value of light intensity here, just like shown in Figure 12

2.7.6 AreaLight vs PointLight

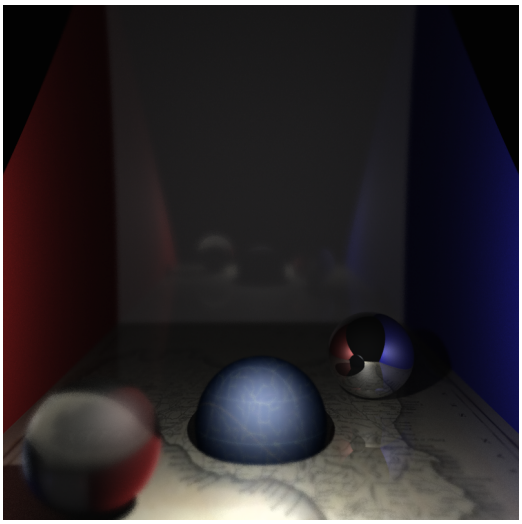
After several experiments with the two LightSources, the difference between Pointlight and AreaLight is more like a trade-off. PointLights are more effective, requires no sample size



(a) $f = 2, a = 4, d = f/16$

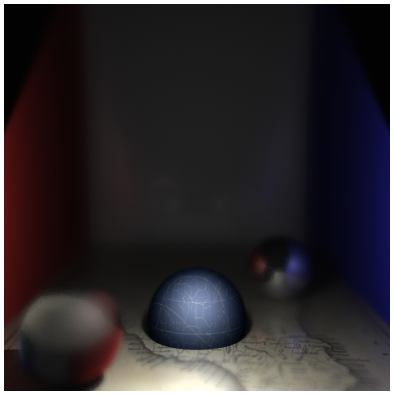


(b) $f = 3, a = 6, d = f/16$

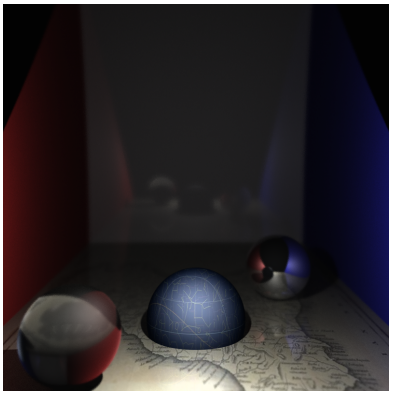


(c) $f = 4, a = 9, d = f/16$

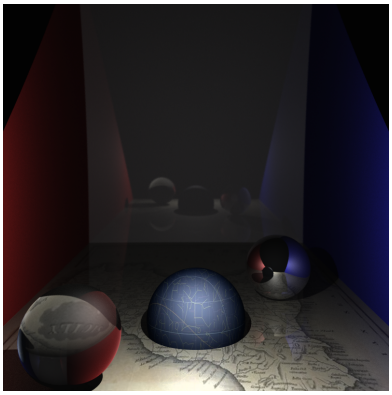
Figure 10: Identical scene with 3 different set of f and d values.



(a) $f = 3, a = 6, d = f/4$



(b) $f = 3, a = 6, d = f/16$



(c) $f = 3, a = 6, d = f/64$

Figure 11: Identical scene with 3 different aperture settings.

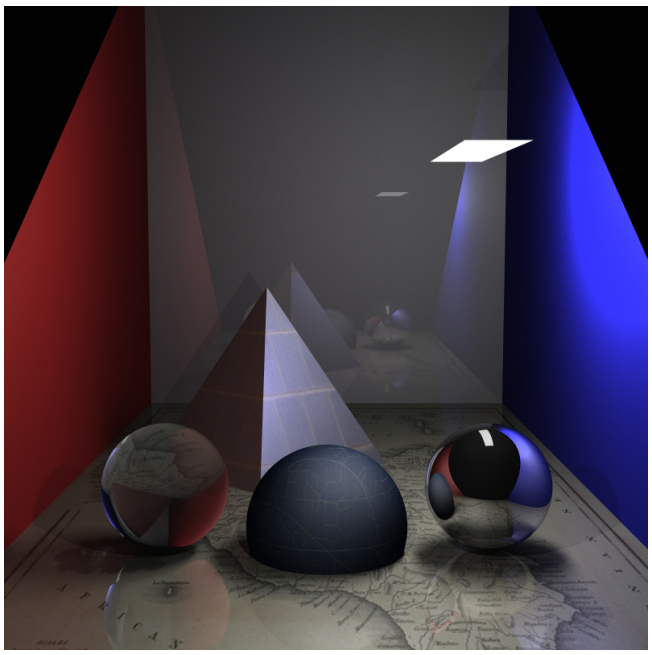


Figure 12: Arealight implemented and shown in the scene, casting soft shadows on the objects.

to produce a clear result, being much more efficient than AreaLight. However, in real-world senarios there are no real 'PoingLight', all lights have a shape, and thus produces soft shadows, not to mention the diffraction of lights. AreaLights, in this aspect, ca better simulate the real-world light sources, but requires much higher sample size, as well as an ideal sampling strategy, to produce a nice-looking result.

2.7.7 Sampler comparison

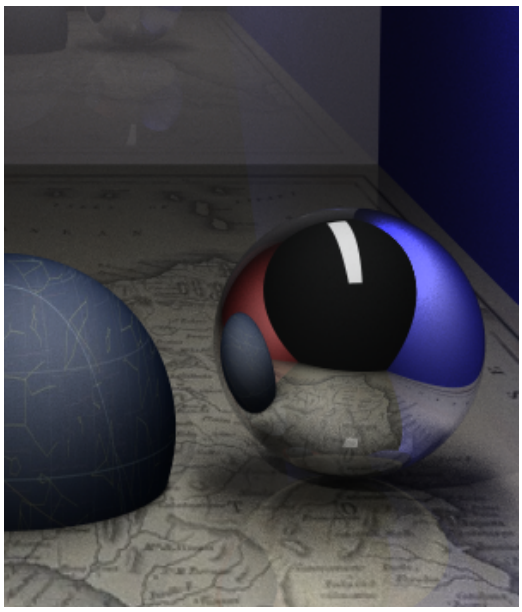
After the implementation of Thinlens camera and Arealight, I performed the comparison between uniformly random sampling and jittered sampling as required.

The coursework specification requires that we perform the tests starting with a sample size of 10, and increase by a step size of 50 until it reaches 2000. However, since jittered sampling requires that each pixel is divided in a $n*n$ grid, the requested non-square sampling sizes (such as 10,60,110,etc.) will result in partially sampled pixels or sampling overflows. Therefore, I decided to use sample sizes from $3^2, 4^2, 5^2...44^2$ (i.e. $9, 16, 25...1936$)to perform the tests. Also, the reference image is rendered based on a sample size of 4,900 instead of 5,000 for the same reason.

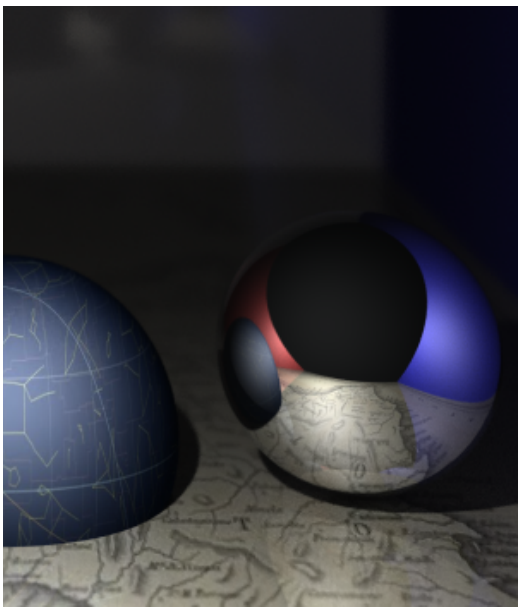
The tests are based on a customized scene with one point light. For tests about Thinlens camera, I changed the camera type to Thinlens camera and set proper parameters for it. And for the tests about AreaLight, I simply added one more area light source to the scene. To speed up the tests, I only rendered a small part of the image as the specification required. Figure 13 shows two examples of the rendered image regions for both tests.

In the case of Thinlens camera, the test result is shown in Figure 14. It can be seen that At low sample sizes, jittered sampling holds a certain advantage over random sampling. But as the sample size gets larger, the MSE of the two sampling methods get closer. This is because that at low sample sizes, jittered sampling can achieve a more uniformed sampling positions, while random sampling might result in certain regions left blank, while some other regions full of sampling points. This could result in a higher MSE for random sampling. But as the sample size gets higher, the chance of such biased sampling gets lower, so the MSE of random sampling gets closer to the one for jittered sampling.

As for distributed ray tracing for AreaLights, both uniformly random sampling and jittered



(a) Image region for sampling test with AreaLight



(b) Image region for sampling test with Thinlens camera

Figure 13: Examples of test image regions. Both are with sample size of 128.

sampling hold similar MSEs across all sample sizes, as shown in Figure 15. This is probably because that when using distributed sampling to simulate the effect of AreaLight, the factor that mostly affects the error of the result is the chance of the randomly distributed ray from the hit point to reach the area light directly. In this case, the sampling method can hardly affect the result of each sample ray. Therefore, the MSEs from random sampling and jittered sampling are close to each other. Even in this case, jittered sampling still have slightly smaller MSEs than the ones of the random sampling.

By comparing the results in Figure 14 and 15, we can see that the MSEs for AreaLight case begins much higher, but end up lower than the ones for the Thinlens case. This is probably because that the error caused by AreaLight shadows are kind of a 1/0 error, where the random position of the AreaLight can either light up the certain position or not. This could result in huge errors when the sample size is low. However, as the sample size increases, the portion of rays where the selected random point on the AreaLight lights up the given position gets closer to the actual value, which is the portion of the area of the AreaLight that can be seen from the position. Therefore, the MSE could reach relatively low value.

By contrast, the error from Thinlens camera comes mainly from the mechanics of the Thinlens. Apart from the changes in sampling positions, the random selection of ray direction toward the lens could also result in huge differences in the out coming ray. This makes it harder for rendering with Thinlens camera to reduce the MSE. Therefore, the MSEs for Thinlens camera case ends up higher than the ones for AreaLight case.

2.8 JPGLoader

To get the RGB values in texture images, I used **stb_image** library, and implemented a **JPGLoader** class based on it. The loader will, at the beginning of the rendering phase, go through all shapes'materials to check for necessary texture file paths. It then reads the texture images, and store the RGB values in local vectors. By doing so, whenever the renderer needs a pixel value of a texture, it can simply pass the texture name as well as the u, v coordinate to the JPGLoader, and the loader will return the RGB value it has stored.

3 Final Output Presentation

Figure 16 is the final output image of my customized scene, which shows all the implemented functions in my ray tracer. The image is re-producable with **final_output.json**. The bullet points are shown below:

3.1 Basic Shapes and BVH

All types of shapes are shown in the image, including Triangle (left and right wall), Plane (floor), Sphere and TriMesh (The pyramid-shaped grey brick). Though not directly shown in the image, the scene uses BVH to accelerate the ray tracing.

Obvously, all shapes can correctly interset with the rays, which indicates the correct implementation of **Ray** and **Hit**, as well as the **intersect** function for all shapes.

3.2 Diffuse and Specular reflection

As the images shows, the diffuse reflection and specular reflection from light sources are properly implemented.

3.3 Refraction

The sphere on the left shows the transparency effect, with the implementation of refraction. We can see an up-side-down refraction image within the sphere, where the red brick wall appears on the right and the grey TriMesh on the left and the textured floor appears at the top. This sphere is created with $eta = 1.5$.

3.4 Reflection of surrounding objects

The sphere on the right shows a total reflection, where we can see all the surrounding shapes, as well as the AreaLight hanging over it.

3.5 Texture Mapping

It is shown in the image that objects from all types of Shape are able to be properly textured, including: Triangle (left wall), Sphere (middle sky-map), Plane (floor) and TriMesh (The pyramid-shaped grey brick).

3.6 AreaLight

The AreaLight lights up the surrounding shapes, also casting soft shadows on the floor. There are soft shadows for the spheres showing on the floor that can be easily seen. Also, we can see the shape of the light itself from the camera, as well as its reflections on other shapes.

3.7 Thinlens camera and Depth-of-Field

Although the three spheres and the closer part of the walls and floor are clear at the front of the scene, the shapes further away are all blurred. The pyramid-shaped grey TriMesh behind the spheres has a brick texture applied, but it cannot be seen clearly. All these blurring effect is because the camera used here is a Thinlens camera with $f = 3$, $d = 6$ and an aperture of $f/16.0$. (Explanation of these parameters are introduced in Section 2.7.2)

The default sphere in the middle is textured with a sky-map and can be clearly seen, while the shapes far away, like the TriMesh and the back wall, are all blurred. Also, we can see from the left wall that the bricks closer to the camera are clearer than the ones further away. These are the 'depth-of-field' effects achieved with my implementation of Thinlens camera.

3.8 Ray Tracing Combination

As discussed before in Section 2.7.1, to lower the rendering time as well as minimizing the noise, the final output comes from a mixed algorithm of distributed ray tracing and the classical Whitted ray tracing. This allows the scene to accept both AreaLights and PointLights at the same time. While only one AreaLight is seen, there is also a point light in front of the camera, which is at the same position as the one in the given example.json to light up the front side of the objects. However, to show the soft shadow more clearly, the light intensity of this point light is reduced by 50%.

Tips

I'm using **Visual Studio 2019 Release amd64_X86** compiler to build the ray tracer program through **Visual Studio Code** on Windows, therefore the default executable program is generated inside **.\build\Release**. Once generated, move the **RayTracer.exe** file into the **.\example** folder, open the Windows terminal here and run command:

.\raytracer.exe 'inputfilename' 'outputfilename'

Note that the **JSON** files are also included in the **example** folder. Please do not modify the structure of the "source code" folder.

To speed up the rendering process, my implementation of the RayTracer includes <**execution**> to perform paralleled rendering. If you have to remove the **build** folder for re-build, please use **MSVC** compiler and move the new **RayTracer.exe** file into the **example** folder so that it can be executed as expected. Thank you!

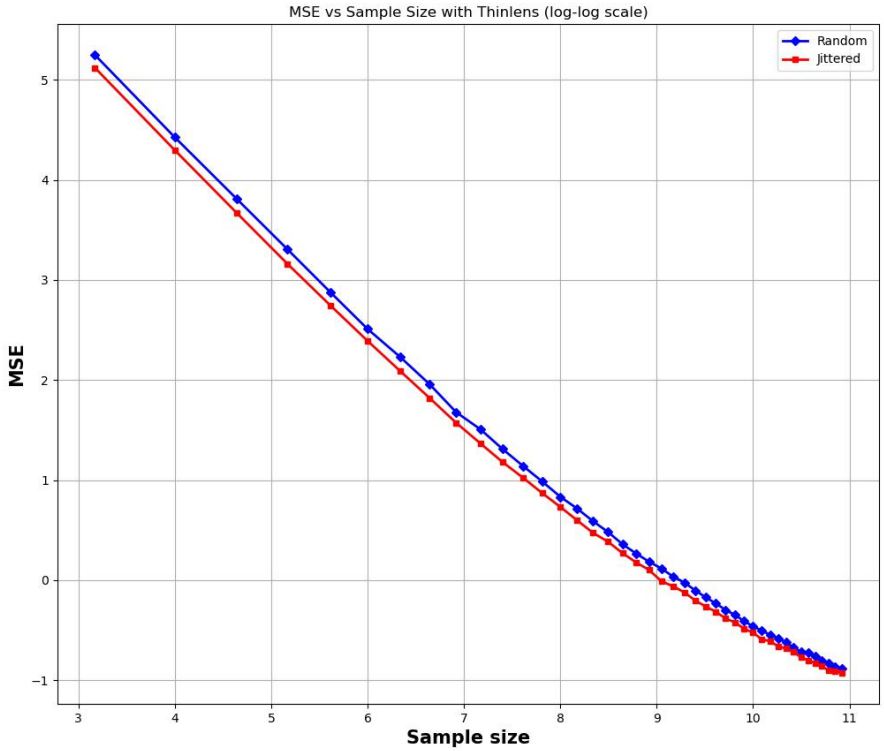


Figure 14: For a scene rendered with Thinlens camera, the MSEs against the sample size. Both X and Y axis are in log scale.

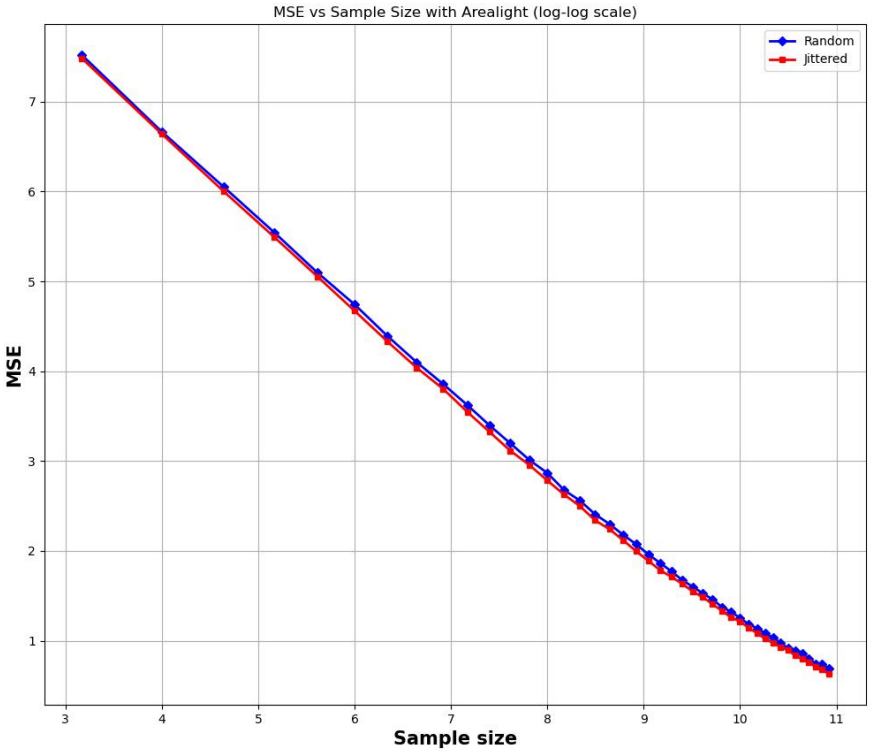


Figure 15: For a scene containing an AreaLight, the MSEs against the sample size. Both X and Y axis are in log scale.

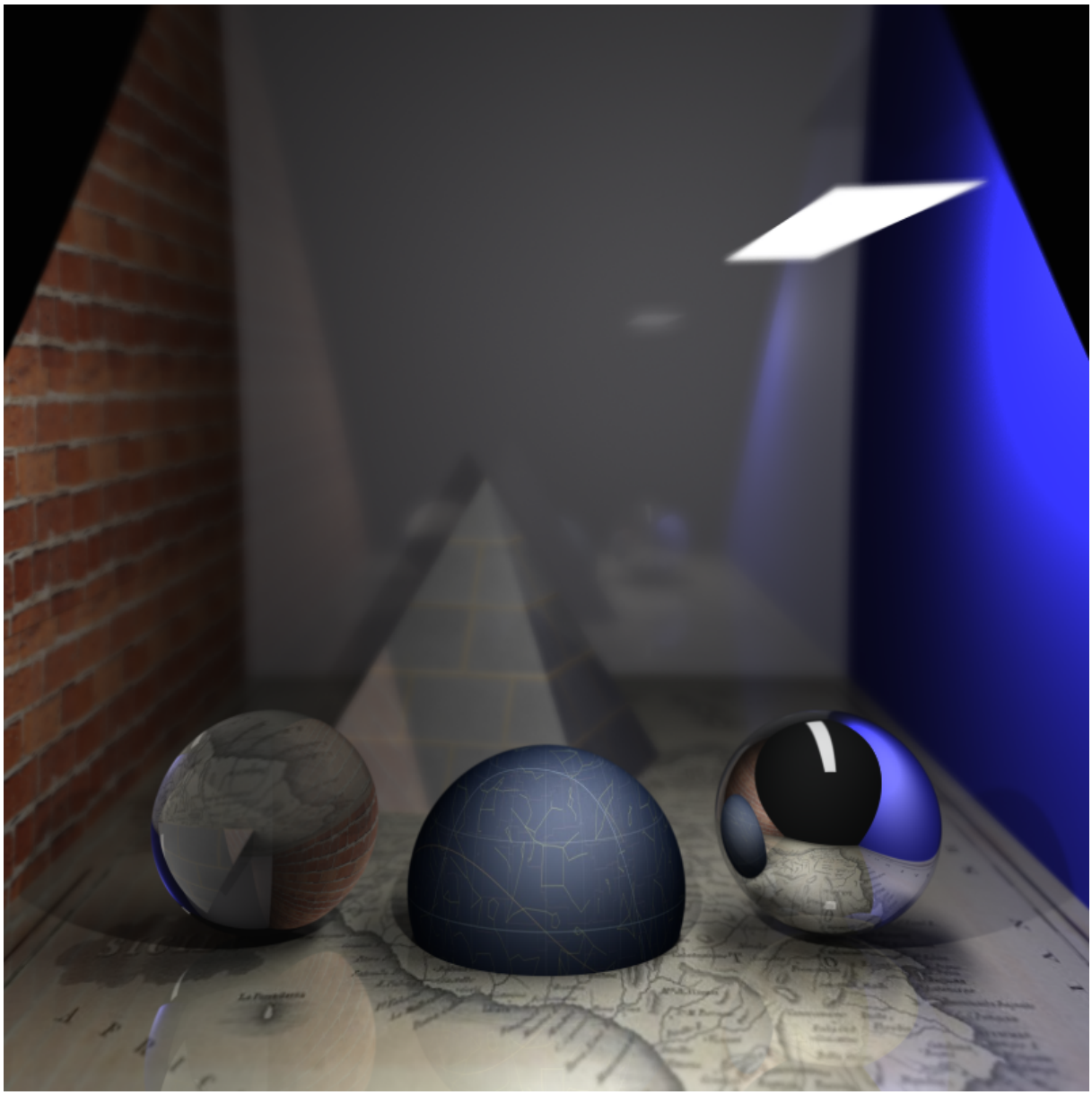


Figure 16: This is the final output.