Computer Graphic Rendering: Coursework 2 - Report

s2176963

November 25, 2024

1 Introduction

This report aims to explain the implementation and evaluation for implementation. The concerning application can be used by following command in terminal,

`./raytracer filePathToJson/test.json 64 outputFileName`,

where the first argument is the filepath to the json file to be loaded, the second number is the number of pixel used in pixel sampling, and the third argument is the output of file name in ppm.

Also, test suites shown in the figures included in this report can be found under `TestSuite/Report TestSuite`. While some test json file and their relative `.obj` and `.mtl` can be found under `Code/jsons` and `Code/models`.

2 Basic raytracer features

(a) Image writing

The image writing functionality is implemented through the `writePPM` function, which generates an image in the PPM (Portable Pixmap) format. The PPM format consists of a human-readable ASCII header followed by a sequence of RGB pixel values. The function writes this header, which includes the format identifier (P3), image dimensions (width and height), and the maximum color value (255). It then writes the pixel data as space-separated RGB values. In order to test the functionality and ensure it outputs a proper PPM image, we provided it with a test pattern of values that results in a gradient image that transitions smoothly from between colors. The test PPM image is converted by using *ImageMagick* to PNG image, demonstrated in figure 1.

Figure 1: Test Figure (PNG) for Image Writing function

The implementation is straightforward and leverages `std::ofstream` for efficient file handling. This approach ensures modularity by separating the pixel data computation and file writing into different functions.

However, the use of ASCII PPM (P3) results in larger file sizes and slower writing speeds, especially for high-resolution images. A binary PPM (P6) format could significantly reduce file size and improve performance. This can be adjusted when more functionalities are implemented.

(b) Virtual pin-hole camera

Before the implementation of actual camera functionality, some essential classes are implemented to assist later functionalities: `Vec3` and `Ray`.

The `Vec3` class represents a 3D vector or point in space and is a fundamental building block in a raytracer. It encapsulates the mathematical operations necessary for manipulating 3D coordinates and directions (Basic Arithmetic Operations, Dot Product, Length and Normalization, etc), which are essential for tasks such as defining rays, computing intersections, and handling shading and lighting.

The `Ray` class represents a ray in 3D space. A ray is defined by an origin point $\vec{O}$ and a direction vector $\vec{D}$ (based on `Vec3` objects).Rays are used to compute intersections with objects in the scene and to compute lighting effects by tracing paths from light sources.

The Camera class generates rays that simulate the projection of a 3D scene onto a 2D image plane. It defines the camera's position in the scene, its orientation, and its field of view. The implementation uses the `Look-At` method to establish a local coordinate system of camera itself, consisting of three orthogonal vectors:

- $\boldsymbol{w}$: Points from the camera position to the target, defining the viewing direction.
- $\boldsymbol{u}$: Perpendicular to both w and the up vector, representing the horizontal axis.
- $\boldsymbol{v}$: Perpendicular to both w and u, representing the vertical axis.

Using these vectors, the camera constructs the image plane, calculating its dimensions based on the field of view and aspect ratio. The lower-left corner of the image plane is computed as a reference point for generating rays . The `getRay(u, v)` method creates a ray that originates from the camera and passes through a pixel (u, v) on the image plane. This method transforms normalized device coordinates into world space coordinates, ensuring that rays accurately reflect the camera's perspective.

To test the camera's functionality and ensure correct ray generation, a light blue gradient background is generated. The color of each pixel is determined by the vertical component of the normalized ray direction, with upward rays producing a blue tint and downward rays leaning toward white. Any issues, such as inverted or skewed gradients, would immediately indicate errors in the implementation. The output image at this stage can be seen in Figure 2

Figure 2: Test Figure (PNG) for Virtual Pin-Hole Camera function

All three classes are modular and reusable, allowing them to be extended or reused in other parts of the program. However, it currently lacks advanced features such as depth of field, motion blur, or support for orthographic projection. Additionally, the focal length is implicitly defined, limiting fine-grained control over the camera's zoom capabilities. These could be improved and implemented in the next steps.

(c) Intersection tests

In this stage of the raytracer development, the focus was on implementing intersection tests for three types of geometric shapes: spheres, triangles, and cylinders. Additionally, the program was updated to parse a JSON scene file, construct the scene accordingly, and perform ray-geometry intersection tests to produce a binary image indicating whether a ray intersects any object in the scene.

Intersection test: Sphere

A `Sphere` class was created, inheriting from a base `Shape` class. The sphere is defined by its center $\vec{C}$ and radius r. The intersection method solves the quadratic equation derived from substituting the ray equation into the sphere's implicit equation.

An arbitrary point $\vec{P}$ on surface of sphere is satisifes the following equation:

$$\|\vec{P} - \vec{C}\|^2 = r^2$$

The ray is defined as

$$\vec{P}(t) = \vec{O} + t\vec{D}$$

By substituting ray equation into sphere equation, we can found a point, if any, that lies on the ray and lies on the surface of sphere, which is exactly the intersection point:

$$\|\vec{O} + t\vec{D} - \vec{C}\|^2 = r^2$$

$$at^2 + bt + c = 0$$

where:

$$a = \vec{D} \cdot \vec{D}, \quad b = 2(\vec{D} \cdot (\vec{O} - \vec{C})), \quad c = (\vec{O} - \vec{C}) \cdot (\vec{O} - \vec{C}) - r^2$$

Intersection Points:

$$t_0 = \frac{-b - \sqrt{\Delta}}{2a}, \quad t_1 = \frac{-b + \sqrt{\Delta}}{2a}$$

The discriminant of this equation $\Delta = b^2 - 4ac$ determines whether there exist a such point that satisifes the equation. If the discriminant is larger than 0, two intersection points are found (t_0 and t_1). The closest positive intersection is chosen. If the discriminant is equal to 0, only one intersection points is found, where the ray is tangent to the sphere. This implementation is efficient and handles all cases, including when the ray originates inside the sphere.

Intersection test: Triangle

Furthermore, a `Triangle` class, where a triangle is defined by its 3 vertices v_0, v_1 and v_2, each a `Vec3`. The intersection test uses the *Möller-Trumbore algorithm*[MT97], which computes the intersection point by solving for barycentric coordinates. We first compute Edge Vectors:

$$\vec{e}_1 = \vec{v}_1 - \vec{v}_0, \quad \vec{e}_2 = \vec{v}_2 - \vec{v}_0$$

According to *Möller-Trumbore algorithm*, we firstly check if the ray is parallel with triangle. If this is the case, the dot product between the ray's direction vector and the plane's normal vector will be zero. We compute the determinant a and check `if (std::abs(a) < EPSILON)`, where `EPSILON` is an infinitely small, positive constant used to handle numerical precision issues :

$$\vec{h} = \vec{D} \times \vec{e}_2, \quad a = \vec{e}_1 \cdot \vec{h}$$

If $a > 0$, then the ray intersects the plane on which the triangle lies, but not neccessary on the triangle. According to *Möller-Trumbore algorithm*, we should compute for v and u using *Cramer's rule* and check `if (u < 0.0 || u > 1.0)` and `if (v < 0.0 || u + v > 1.0)`. If one of the case is true, then the ray does not intersect with triangle.

$$\vec{O} + t\vec{D} = \vec{v}_1 + u(\vec{v}_2 - \vec{v}_1) + v(\vec{v}_3 - \vec{v}_1)$$
$$\vec{O} - \vec{v}_1 = -t\vec{D} + u(\vec{v}_2 - \vec{v}_1) + v(\vec{v}_3 - \vec{v}_1)$$

Barycentric Coordinates:

$$f = \frac{1}{a}, \quad \vec{s} = \vec{O} - \vec{v}_0$$
$$u = f(\vec{s} \cdot \vec{h}), \quad \vec{q} = \vec{s} \times \vec{e}_1, \quad v = f(\vec{D} \cdot \vec{q})$$

Intersection Point:

$$t = f(\vec{e}_2 \cdot \vec{q}), \quad \vec{P} = \vec{O} + t\vec{D}$$

The algorithm is widely used due to its efficiency and accuracy. It handles both front-facing and back-facing triangles, but it does not explicitly handle degenerate triangles (where vertices are collinear). Additionally, the reliance on an epsilon value to avoid division by zero may require tuning for different scenes.

Intersection test: Cylinders

For a finite cylinder defined by its center $\vec{C}$, normalized axis vector $\vec{A}$, radius r and height h. Similarly, a ray is defined by its origin $\vec{O}$ and a direction vector $\vec{D}$. We assume the center of cylinder consist of geometric center of cylinder, not base center. The height therefore expand for both directions along direction axis. Similar to the computation of Spheres, The intersection method solves the quadratic equation derived from substituting the ray equation into the cylinder's implicit equation.
The ray is defined as:

$$\vec{P}(t) = \vec{O} + t\vec{D}$$

We first check the Intersection with lateral surface. We assume that the cylinder is infinitely long and an arbitrary point $\vec{P}$ projected to the surface perpendicular to $\vec{A}$ satisfies the following equation:

$$\|\vec{P} - \vec{C} - ((\vec{P} - \vec{C}) \cdot \vec{A})\vec{A}\|^2 = r^2$$

Substituting the ray equation:

$$\|\vec{O} + t\vec{D} - \vec{C} - ((\vec{O} + t\vec{D} - \vec{C}) \cdot \vec{A})\vec{A}\|^2 = r^2$$

Define:

$$\vec{oc} = \vec{O} - \vec{C}$$
$$\vec{D}_\perp = \vec{D} - (\vec{D} \cdot \vec{A})\vec{A}$$
$$\vec{oc}_\perp = \vec{oc} - (\vec{oc} \cdot \vec{A})\vec{A}$$

The quadratic equation becomes:

$$at^2 + bt + c = 0$$

Where:

$$a = \vec{D}_\perp \cdot \vec{D}_\perp, \quad b = 2(\vec{oc}_\perp \cdot \vec{D}_\perp), \quad c = \vec{oc}_\perp \cdot \vec{oc}_\perp - r^2$$

Solve using the discriminant:

$$\Delta = b^2 - 4ac$$

$$t_0 = \frac{-b - \sqrt{\Delta}}{2a}, \quad t_1 = \frac{-b + \sqrt{\Delta}}{2a}$$

Now we check if the point $\vec{P}$ is within the height of actual cylinder. For each t, compute:

$$y = (\vec{P}(t) - \vec{C}) \cdot \vec{A}$$

Check:

$$-h \leq y \leq h$$

Lastly, we check the intersection with bases. We first compute the intersection point with the plane where bases lie. For the top and bottom bases:

$$t_{\text{base}} = \frac{(\vec{P}_0 - \vec{O}) \cdot \vec{A}}{\vec{D} \cdot \vec{A}}$$

and we check if they are within the base:

$$\|\vec{P}_{\text{cap}} - \vec{P}_0\| \leq r$$

The implementation handles arbitrary orientations and accurately computes intersections, but it remains more complex than other shapes. Potential numerical instability due to floating-point operations could be improved with further refinement.

Managing the Scene

A `Scene` class was introduced to manage all shapes and facilitate intersection tests. It stores a collection of Shape objects and provides an intersect method to check for ray intersections with any object. The main rendering loop generates a ray for each pixel and checks for intersections using the `Scene` class. If an intersection is found, the pixel is colored white; otherwise, it is set to the background color.

The use of polymorphism via the `Shape` base class allows for easy addition of new geometric shapes, enhancing flexibility and scalability. However, the current implementation tests intersections sequentially, which may become inefficient for scenes with many objects. Introducing spatial acceleration structures, such as Bounding Volume Hierarchies (BVH), could significantly optimize performance in the next stages.

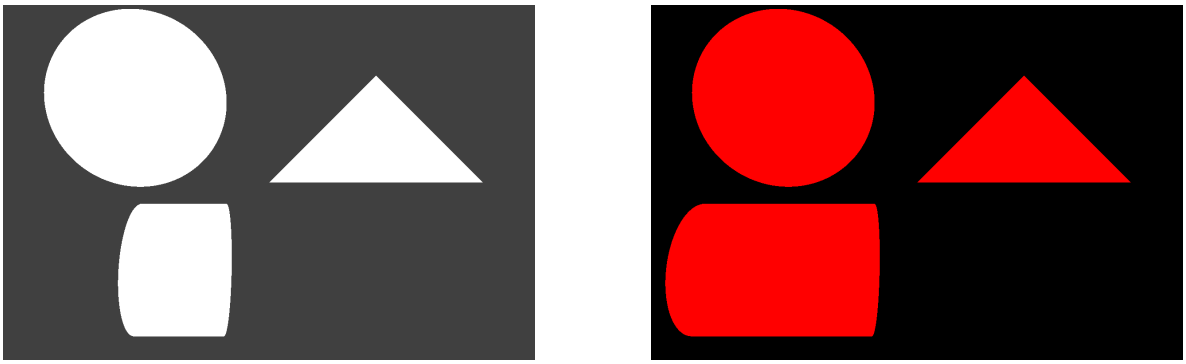
Json File reading

The program uses the `nlohmann/json` library to parse the scene configuration from a JSON file. The `loadScene` function reads camera parameters, background color, and shapes, dynamically constructing the scene based on the provided data. The `Camera` class was updated to include methods for setting image dimensions and recalculating its parameters.

The use of `nlohmann/json` library provides flexibility, enabling rapid testing with different scenes without recompiling the program. However, the implementation lacks robust error handling for malformed or missing JSON fields, which could lead to runtime errors. Adding validation and informative error messages would improve reliability.

Testing and Corrections

Several updates were made to address issues in the initial implementation, where the center of cylinder is interpreted as base center. This results in a incorrect image compared to the instruction, as seen in Figure 3a. The logic is then updated to interpret the center as the geometric center, correcting the cylinder's height and position. The final test result at this stage can be observed in Figure 5b



(a) Intersection test result before correction

(b) Intersection test result after correction

Figure 3: Test Figure (PNG) for Intersection test function

(d) Blinn-Phong shading

The Blinn-Phong shading model simulates how light interacts with surfaces, producing realistic lighting effects by combining three components: ambient, diffuse, and specular. These components are calculated using material and lighting parameters provided in the JSON scene file. The core logic for implementing this shading model is encapsulated in the `computeColor` function, which integrates these components to determine the final color of each pixel.

Diffuse Component: The diffuse term simulates light scattering evenly in all directions from the surface. The formula used is:

$$L_{\text{diffuse}} = k_d \cdot \max(\mathbf{N} \cdot \mathbf{L}, 0) \cdot \mathbf{C}_{\text{diffuse}} \cdot \mathbf{I}_{\text{light}} \tag{1}$$

where:

- k_d is the diffuse coefficient.
- $\mathbf{N}$ is the surface normal.
- $\mathbf{L}$ is the normalized light direction.
- $\mathbf{C}_{\text{diffuse}}$ is the material's diffuse color.
- $\mathbf{I}_{\text{light}}$ is the light intensity.

Specular Component: The specular term simulates the reflection of light in a specific direction, creating highlights. The formula used is:

$$L_{\text{specular}} = k_s \cdot \max(\mathbf{N} \cdot \mathbf{H}, 0)^{\text{specularExponent}} \cdot \mathbf{C}_{\text{specular}} \cdot \mathbf{I}_{\text{light}} \tag{2}$$

where:

- k_s is the specular coefficient.
- $\mathbf{H}$ is the halfway vector:

$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{V}}{\|\mathbf{L} + \mathbf{V}\|} \tag{3}$$

- specularExponent controls the sharpness of the highlight.
- $\mathbf{C}_{\text{specular}}$ is the material's specular color.

Final Color Combination: The final color at a surface point is computed as:

$$\text{Color} = L_{\text{diffuse}} + L_{\text{specular}} + L_{\text{ambient}} \tag{4}$$

By combining diffuse and specular terms, it provides a convincing approximation of how light interacts with surfaces, as we can see in Figure 4. Here we assume that the default ambient color is `(0, 0, 0)`, since no information regarding the ambient light is given by the json test file `simple_phong.json`.

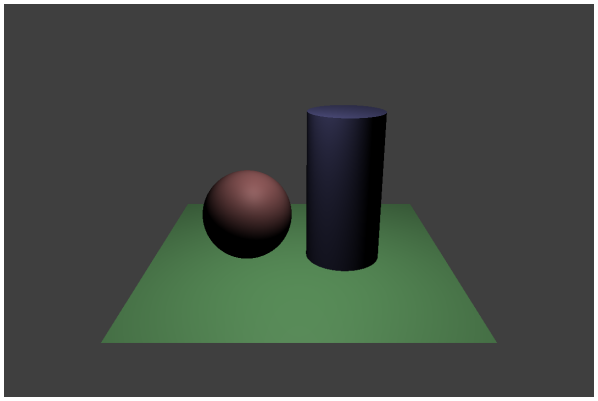


Figure 4: Test Figure (PNG) for Blinn-Phong Modelling

The model is also computationally efficient, making it well-suited for applications requiring real-time rendering. Despite its strengths, the current simple Blinn-Phong model has inherent limitations. It simplifies the lighting process by not accounting for global illumination, which includes ambient lighting and light bouncing between surfaces. As a result, it struggles to accurately represent certain materials, such as metals and rough surfaces. To overcome these limitations, several enhancements could be made. Introducing ambient lighting or precomputed global illumination could simulate indirect light more realistically. Additionally, shadows will be added at next steps.

(e) Shadows

Shadows simulate areas where light is blocked by objects, adding depth and realism to the scene. For each light source, a *shadow ray* is cast from the intersection point to the light source. The shadow ray is defined as:

$$\text{ShadowRay} = \text{Ray}(\mathbf{P} + \epsilon \cdot \mathbf{N}, \mathbf{L}) \tag{5}$$

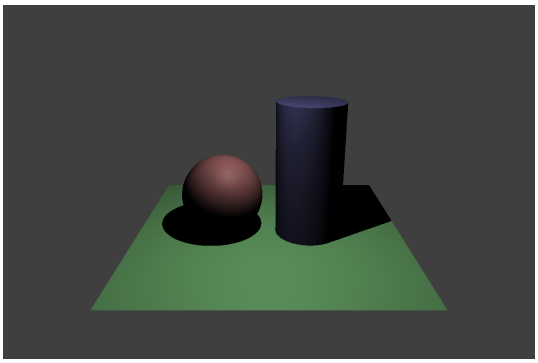
where:

- $\mathbf{P}$ is the intersection point.
- ϵ is a small bias to prevent self-intersection.
- $\mathbf{N}$ is the surface normal.
- $\mathbf{L}$ is the light direction.

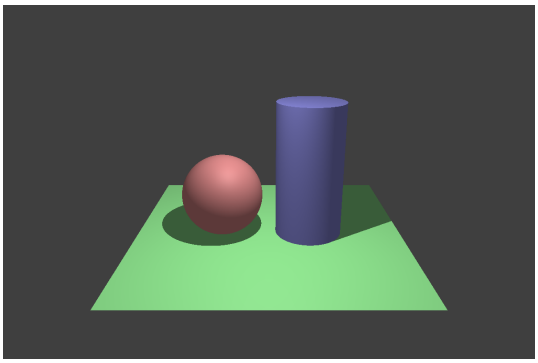
Intersection Test: If the shadow ray intersects any object before reaching the light source, the point is considered to be in shadow. The shading is then modified accordingly:

$$\text{Color} = \begin{cases} L_{\text{diffuse}} + L_{\text{specular}}, & \text{if not in shadow} \\ \mathbf{C}_{\text{ambient}}, & \text{if in shadow} \end{cases} \tag{6}$$

We first test the `simple_phong.json` with default ambient light of `(0, 0, 0)`, the result is shown in Figure 5a. To better demonstrate the simulation of shadow, an ambient light of `(0.5, 0.5, 0.5)` is given, as demonstrated in Figure 5



(a) Shadow test before applying visible ambient light



(b) Shadow test after applying visible ambient light

Figure 5: Test Figure (PNG) for Shadow casting function

One of the key strengths of this approach is its dynamic nature adapting the possible movements of light source and objects. Moreover, the integration of shadow rays fits naturally into the ray tracing framework. However, the current implementation has several limitations. The shadows produced are hard-edged because lights are modeled as point sources. This simplification excludes soft shadows or penumbra effects. Additionally, the computational cost of shadow rays is significant, as each light source requires an additional ray cast for every intersection point, effectively doubling the workload. To address these limitations, several improvements could be implemented. Soft shadows can be achieved by sampling multiple points on an area light, creating a gradient between light and shadow regions. This would be implemented in the advanced steps.

(f) Tone Mapping

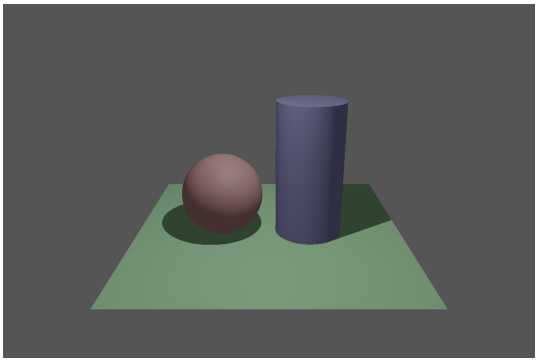
Tone mapping is essential in rendering high dynamic range (HDR) scenes for display on low dynamic range (LDR) devices. As shown in Figure 6c, the color are brighter and highly contrasted. To compare, we employed different tone mapping strategies: Reinhard Tone Mapping [RSSF23] and Exponential Tone Mapping. The setting for those scene are kept equally, with a exposure of 2, and an ambient light

of `(0.25, 0.25, 0.25)`. Figure 6a demonstartes the effect after applying the reinhard tone mapping technique, which compresses luminance values using the formula:

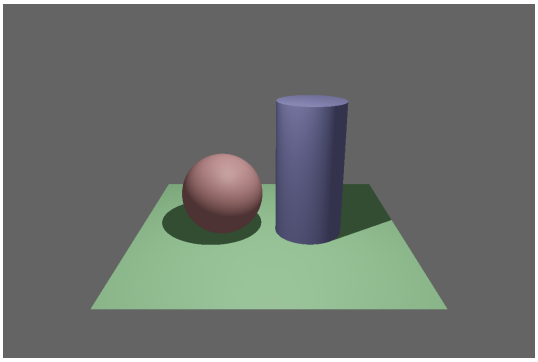
$$L_{\text{mapped}} = \frac{L_{\text{original}}}{1 + L_{\text{original}}}$$

Meanwhile, exponential tone mapping calculates mapped color using:

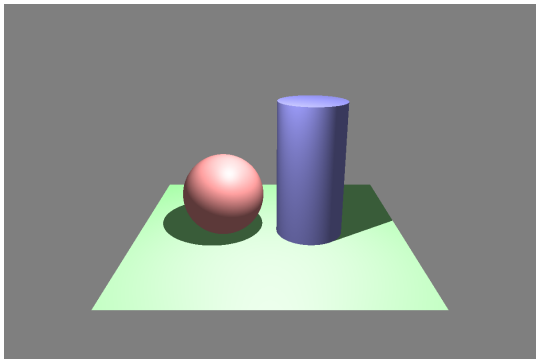
$$L_{\text{mapped}} = 1 - e^{-L_{\text{original}} \times \text{exposure}}$$



(a) After applying Reinhard Tone Mapping



(b) After applying Exponential Tone Mapping



(c) Without the application of Tone Mapping

Figure 6: Test Figures (PNG) for Tone Mapping function

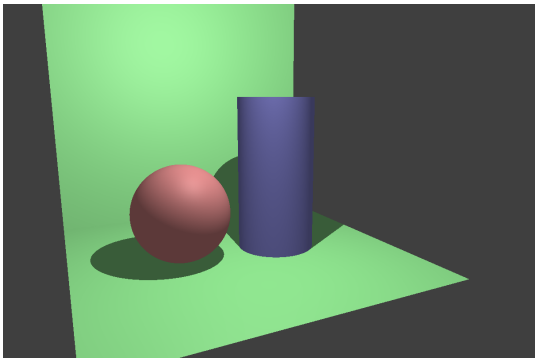
We could observe that although reinhard tone mapping applies uniformly across all luminance values. This aggressive compression may mute bright highlights, losing details in the color and lights. Meanwhile. exponential tone mapping operator compress less low luminance values, which preserving shadow details. Also, high luminance values are smoothly compressed, retaining natural brightness. Therefore, at this stage, Exponential Tone Mapping is adopted.

(g) Reflection

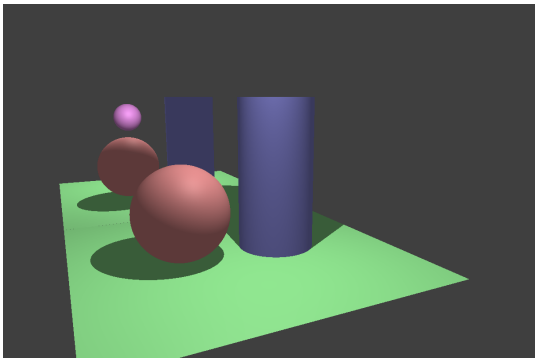
Reflection is implemented by calculating a new reflection ray at the point of intersection. The direction of the reflected ray is computed using the formula:

$$\text{reflectionDir} = \text{I} - 2(\text{I} \cdot \text{N})\text{N}$$

where I is the incident ray direction and N is the surface normal. After computing this direction, a new ray is cast recursively to determine the reflected color. This color is blended with the local color based on the material's reflectivity property..In the test suite `mirror_image.json`, the triangles simulating the mirror have a reflectivity of 1.0, representing a perfect mirror surface, shown in Figure 7b.



(a) Before Applying Reflection



(b) After Applying Reflection

Figure 7: Test Figure (PNG) for reflection function

In the shading function, a recursion depth check ensures the algorithm does not exceed a maximum recursion limit, preventing infinite recursion. The reflection ray is slightly offset from the surface to avoid self-intersection, ensuring accurate results.

The reflection implementation effectively enhances the realism of rendered scenes by simulating mirror-like surfaces. The use of recursive ray tracing allows for accurate computation of multiple levels of reflection, capturing complex visual phenomena such as reflections within reflections. However, the current implementation does not account for the Fresnel effect, which would make reflections more physically accurate by varying the reflectivity based on the angle of incidence. Additionally, reflection increases the computational load due to recursive ray tracing, particularly in scenes with multiple reflective surfaces. This can lead to longer rendering times, especially with high recursion depths. In next steps, Fresnel equations could be incorporated, dynamically adjusting reflectivity based on viewing angles. Performance optimizations, such as implementing bounding volume hierarchies (BVH) could reduce computational costs without compromising visual quality.

(h) Refraction

In this implementation, we extended the raytracer to handle refraction by calculating the direction of refracted rays, managing total internal reflection, and combining reflection and refraction contributions using Fresnel equations.

To correctly calculate refraction, it is essential to determine the correct orientation of the surface normal. For rays entering a material, the normal is used as is. However, if the ray is inside the material, the normal must be inverted, and the refractive indices must be swapped. This is achieved by computing the dot product between the ray direction $\mathbf{D}$ and the normal $\mathbf{N}$:

$$\cos\theta_i = \mathbf{D} \cdot \mathbf{N}.$$

If $\cos\theta_i > 0$, the normal is inverted, and the indices of refraction are swapped:

$$\mathbf{N} \leftarrow -\mathbf{N}, \quad \eta_i \leftrightarrow \eta_t.$$

Using Snell's Law, the refraction direction $\mathbf{T}$ is calculated:

$$\mathbf{T} = \eta\mathbf{D} + (\eta\cos\theta_i - \cos\theta_t)\,\mathbf{N},$$

where $\cos\theta_t$ is computed as:

$$\cos\theta_t = \sqrt{1 - \eta^2(1 - \cos^2\theta_i)}.$$

The parameter k is defined as:

$$k = 1 - \eta^2(1 - \cos^2\theta_i).$$

If $k \geq 0$, refraction occurs, and the refraction direction is normalized.

If $k < 0$, total internal reflection occurs. The reflection direction $\mathbf{R}$ is computed as:

$$\mathbf{R} = \mathbf{D} - 2(\mathbf{D} \cdot \mathbf{N})\mathbf{N}.$$

A reflection ray is traced recursively to calculate its contribution to the final color.

Fresnel Equations: To accurately combine reflection and refraction, the Fresnel equations are used. The reflection coefficient F_r is computed as:

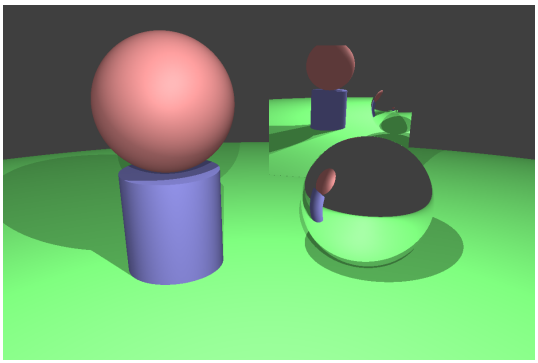
$$F_r = \frac{1}{2}\left(\left(\frac{\eta_t \cos\theta_i - \eta_i \cos\theta_t}{\eta_t \cos\theta_i + \eta_i \cos\theta_t}\right)^2 + \left(\frac{\eta_i \cos\theta_i - \eta_t \cos\theta_t}{\eta_i \cos\theta_i + \eta_t \cos\theta_t}\right)^2\right).$$

The refraction coefficient is $F_t = 1 - F_r$. These coefficients balance the contributions of reflection and refraction.

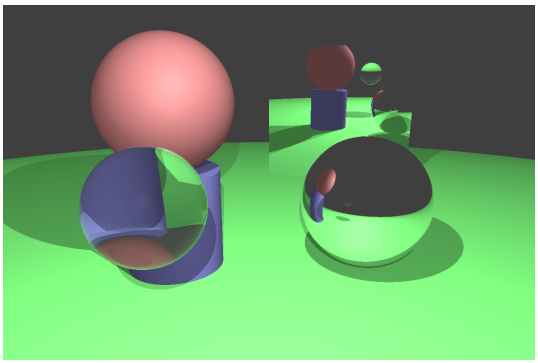
Materials that lack refractive components are handled differently, bypassing the Fresnel equation calculations. Instead, their reflectivity is directly controlled by the material's reflectivity parameter, as described before. Figure 10, demonstrates a combination of reflective and refractive materials within a scene. A perfect mirror maintains its distinct reflective properties, while a simulated glass sphere is introduced with the following property given in Table 1.

ks	0.05
kd	0.05
specularexponent	128
diffusecolor	[0.05, 0.05, 0.105]
specularcolor	[1.0, 1.0, 1.0]
isreflective	true
reflectivity	0.8
isrefractive	true
refractiveindex	1.5

Table 1: Parameters for Glass Sphere



(a) Before Applying Refraction



(b) Before Applying Refraction

Figure 8: Test Figure (PNG) for Refraction function

The implementation accurately simulates refraction and total internal reflection based on physical principles. Transparent materials like glass are convincingly rendered, with visible distortions of background objects. The inverted image produced by the glass sphere effectively demonstrates these principles in action. The use of Fresnel equations adds realism by dynamically balancing reflection and refraction.

While the implementation is computationally intensive, recursion depth is limited to prevent infinite loops. Further optimization, such as using bounding volume hierarchies (BVH) or parallel processing, could improve performance. Currently, the implementation assumes non-absorptive materials, neglecting light attenuation within refractive objects. Introducing Beer-Lambert's law could enhance realism. Additionally, incorporating wavelength-dependent refractive indices would allow for dispersion effects, simulating phenomena like chromatic aberration.

3 Intermediate raytracer features

(a) Textures

The texture feature was implemented by introducing a new `Texture` which encapsulates the logic for loading texture images and sampling colors based on UV coordinates. It supports PPM format textures. The `getColor` method maps normalized UV coordinates to pixel values, accounting for texture wrapping by allowing UV values to repeat beyond the $[0, 1]$ range.

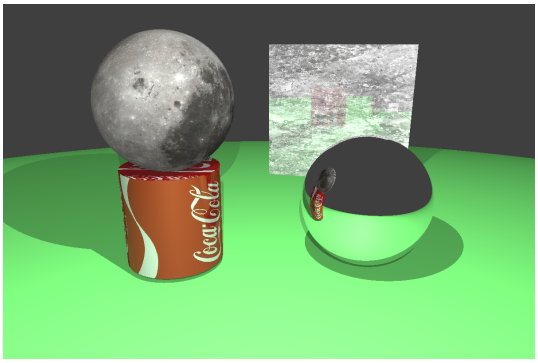
The `Material` class was changes to incorporate a `std::shared_ptr<Texture>` for texture storage and scaling parameters `scaleU` and `scaleV` to control the size and repetition of the texture on the surface. When a texture is present, the diffuse color for shading is sampled from the texture using the computed UV coordinates. Otherwise, a default diffuse color is used.

Each geometric shape—sphere, cylinder, and triangle—was updated to compute UV coordinates specific to its geometry. For spheres, UV mapping uses spherical coordinates derived from the hit point's relative position, while cylinders calculate u based on the angular position around the axis and v along the height. Triangles use barycentric coordinates to interpolate UV values across their surface, enabling smooth texture transitions over complex surfaces, such as meshes.

The rendering pipeline in `computeColor` was modified to utilize texture sampling during shading. When a ray intersects a shape, the computed UV coordinates from the hit record are used to sample the texture if available.

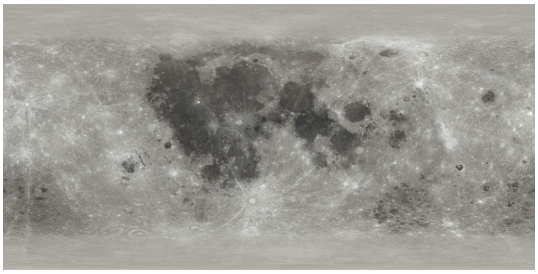
To support textures in scene configuration, the `loadScene` function was updated to read texture paths and scaling parameters from the JSON scene file. If a texture is specified for a material, it is loaded and linked to the corresponding shape.

We observe that these features are primarily tested on basic geometric shapes, as shown in Figure 9. The texture integration works with both reflection and refraction. For instance, when reflectivity is applied to a mirror surface with a stone texture 9d, the surface still reflects other shapes in the scene. Additionally, the textures appear continuous across the surface, unaffected by the fact that the surface comprises two distinct triangles.

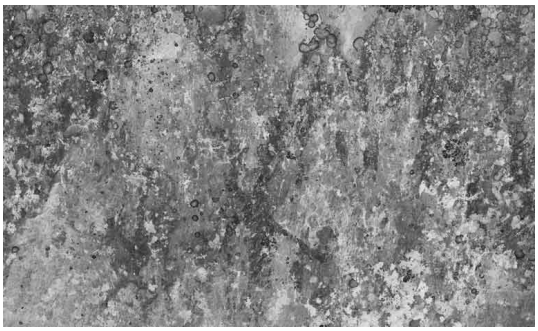


(a) After applying textures

(b) Coca Cola texture applied to cylinder



(c) Moon Texture applied to sphere

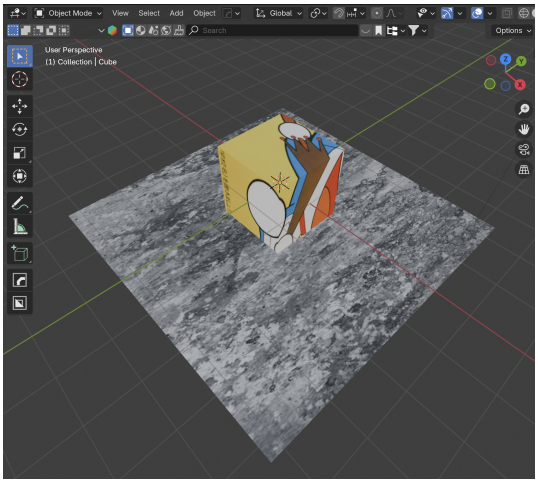


(d) Stone texture applied to Mirror Surface

Figure 9: Test Figures for Texture implementation on basic geometric shapes

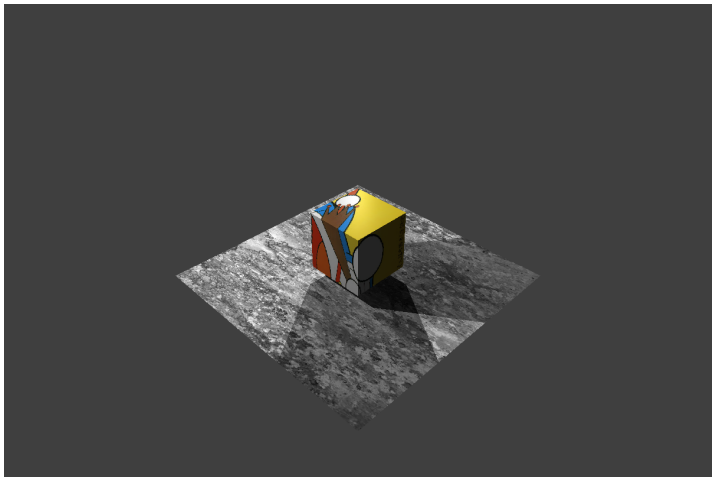
Based on the previous implementations, we took the idea that several triangles can make more complex objects, extends the possibility of objects. we treat triangle as unit for mesh, and added class `Mesh` to convert `.obj` meshes and their relative `.mtl` material files to triangles. During mesh construction, texture coordinates are associated with each vertex and interpolated using barycentric coordinates during ray-triangle intersections. This ensures smooth and continuous texture mapping across the model's surface. The shading function then uses these interpolated texture coordinates to sample the texture image, applying the texture color to the surface based on lighting conditions.

A more complex test case is made with Blender, as demonstrated in Figure 10c by constructing a basic cube and simple plane with different image texture (10b and 9d). We can observe that at this stage, textures are applied correctly to the meshes, similar to one demonstrated by Blender. However, our rendered images exhibit a flipped orientation compared to Blender's default viewport, as shown in Figure 10a. This discrepancy arises because our raytracer's coordinate system has the Y-axis positive out of the screen, whereas Blender's Y-axis points backward into the screen. This difference means that when rendering from the same camera angles, the sides of objects appear swapped. To maintain consistency with previous test cases, we retained this axis orientation in our raytracer. While this may lead to perceived flipping when comparing directly with Blender, it preserves compatibility with test cases and expected results in our implementation.



(a) Sample Objects in Blender

(b) Texture used for cubes



(c) Rendered .obj object

Figure 10: Test Figures (PNG) for Texture implementation on .obj objects

Additionally, there are areas for potential improvement. The current nearest-neighbor interpolation can result in pixelated textures when viewed closely. Implementing bilinear filtering would smooth out

these textures, enhancing their appearance.

Future improvements could focus on optimizing performance using acceleration structure, enhancing texture sampling quality, and expanding material capabilities to include more advanced features. Additionally, refining code organization and adding comprehensive error handling would strengthen the implementation.

(b) Acceleration hierarchy

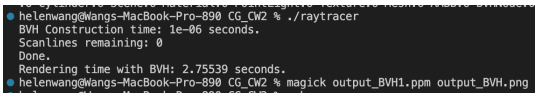
Once we start to insert complex objects with multiple faces in the scene, the efficiency starts to lower down. To efficiently handle complex, **Bounding Volume Hierarchy (BVH)** is implemented. BVH organizes the scene into a tree-like structure, where each node represents a bounding volume that encompasses a subset of the scene's objects. By leveraging the hierarchical structure, the number of intersection tests is reduced from a linear complexity (testing all objects) to logarithmic complexity in well-balanced trees. This enables the raytracer to handle high-poly meshes and dense scenes efficiently.

Axis-Aligned Bounding Box (AABB): The `AABB` class is used to represent the bounding volumes in the BVH. It defines an axis-aligned rectangular box that tightly fits an object or a group of objects.

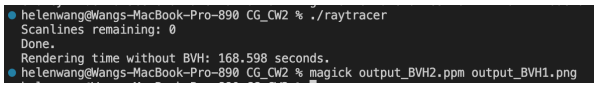
BVHNode: The `BVHNode` class represents the nodes in the BVH tree. Each node can either be a **leaf node**, containing a single object, or an **internal node**, which stores two child nodes. Each node also has an associated bounding box (`AABB`) that encompasses all the objects within its subtree.

BVH Usage: The scene is augmented to include the BVH as its acceleration structure. After loading the scene's objects, the BVH is built by passing the list of shapes to the `BVHNode` constructor. Meshes constructs their internal MeshBVH structure where individual triangles are leafs of the structure. During rendering, rather than directly testing a ray against every object in the scene, which is computationally expensive, the ray first checks for intersections with the `AABBs` associated with the BVH nodes. Starting from the root, the ray is tested against the bounding box of the node. If no intersection is detected, the ray can safely skip all objects within that node's subtree. Conversely, if an intersection occurs, the ray is recursively tested against the child nodes. This hierarchical approach reduces the number of intersection tests performed, quickly narrowing down the potential objects that the ray might intersect, thereby improving overall rendering efficiency.

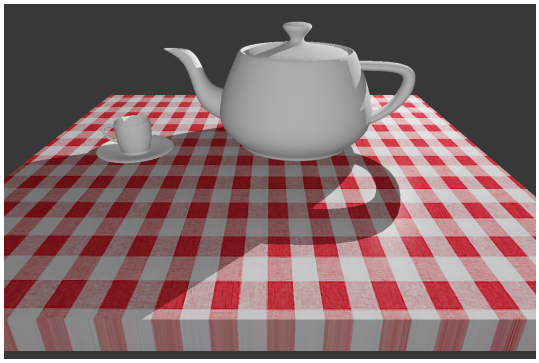
If the BVH is not built (e.g., for testing purposes), the raytracer defaults to the previous brute-force approach, iterating over all objects in the scene. This ensures that the BVH integration is modular and non-disruptive to existing functionality.



(a) Rendering time with BVH enabled



(b) Rendering time with BVH disbaled



(c) Output of both trials 11a and 11b

Figure 11: Test Figures (PNG) for BVH implementation on .obj objects

The BVH implementation significantly improves the performance of the raytracer, especially for complex scenes. This can be shown in Figure 11, by comparing 11a with 11b. We could observe that with BVH, the speed of rendering increases from 168.59 seconds to 2.75 seconds both with `-3o` tag in the `Makefile`.

Although the BVH implementation is effective, there are areas for refinement. The current construction strategy partitions objects by sorting them along a randomly chosen axis. While simple, this may result in unbalanced trees for certain scenes. Implementing more advanced heuristics, such as the Surface Area Heuristic (SAH), could improve tree balance and traversal efficiency.

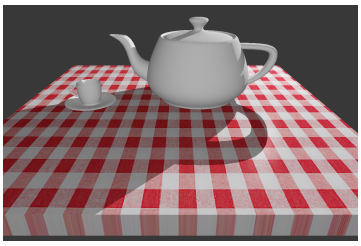
4 Advanced raytracer features

(a) Pixel sampling

We use stratified sampling to enhance pixel sampling accuracy. It divides each pixel into a grid of subpixels (in our test case, a 4x4 grid, resulting in 16 subpixels per pixel) and casts multiple rays through these subpixels.

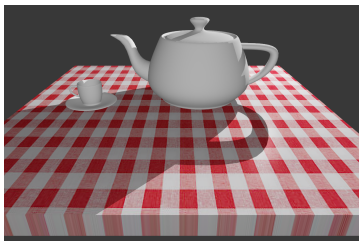
For each pixel, the render loop iterates through a grid of 16 subpixels. Within each subpixel, a random offset is generated using a uniform distribution. This ensures that the ray originates from a unique point within the subpixel's boundaries, allowing for a more accurate and evenly distributed sampling.

After generating and tracing all rays for a pixel, their color contributions are accumulated and then averaged to determine the final color of that pixel. Additionally, exponential tone mapping are applied to the averaged color, as demonstrated in figure 12c, to recreate the accurate brightness and contrast as the original figure 12a.

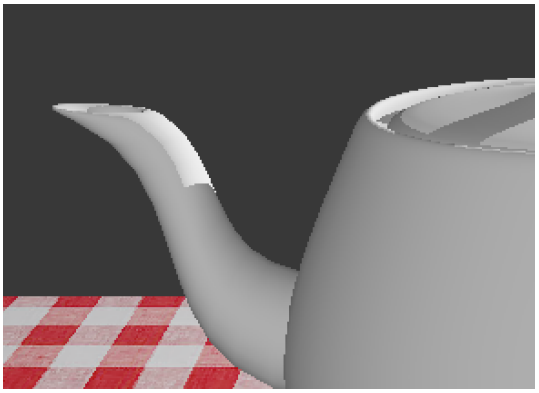


(a) Rendered Scene without Pixel Sampling

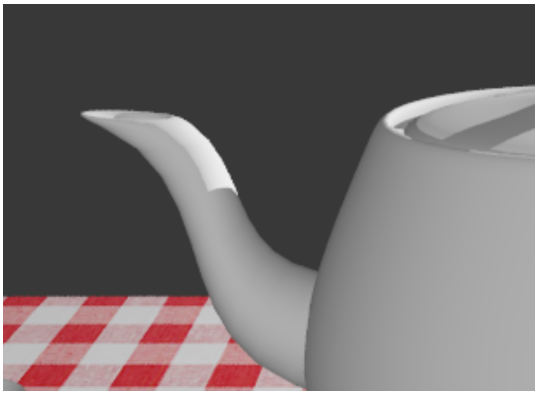
(b) Rendered Scene with Pixel Sampling



(c) Figure 12b with Exponential Tone Mapping



(d) Close Look of 12a



(e) Close Look of 12c

Figure 12: Test Figures (PNG) for Pixel Sampling implementation on 11c

The stratified sampling reduces aliasing artifacts like jagged edges, as demonstrated by the comparison of Figure 12d and 12e. However, setting the number of samples too high, such as 100 samples per pixel, can lead to excessive averaging, which may result in a blurred appearance as finer details become overly smoothed out. To address this, we start with a moderate number of samples per pixel (e.g., 16), which brings a satisfactory rendering quality. Furthermore, implementing stratified sampling

ensures a more consistent and reliable distribution of samples compared to purely random sampling, which can inadvertently cause uneven coverage and introduce noise.

(b) Lens sampling

The effect of depth of field is achieved through lens sampling, where rays are generated from a finite-sized aperture instead of a single point, converging at a defined focal plane.

To support lens-based depth of field, aperture and focus distance are introduced to `Camera` class. Also, we can now set `aperture` parameter and `focusdistance` parameter in json files to set the camera. In the default mode without giving a specific aperture and focus distance in `json` files, aperture is 0, indicating a pin hole camera, and focus distance is given by the distance between camera and camera's focus point `(cameraPosition - lookAt).length()`. Aperture controls the amount of blur for objects outside the focus plane. While, focus distance specifies the distance from the camera where objects appear in focus. In the default mode without giving a specific focus distance in `json`, focus distance is given by the distance between camera and camera's focus point `(cameraPosition - lookAt).length()`.

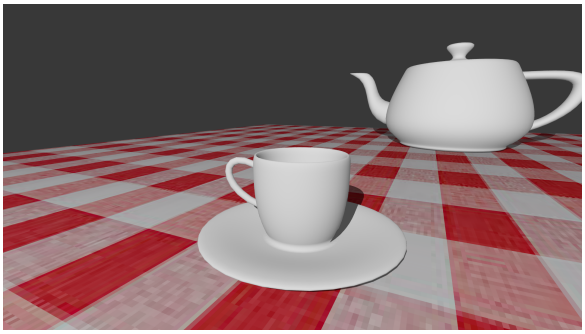
The image plane's lower-left corner, horizontal, and vertical extents are adjusted to calculate based on these parameters, ensuring correct ray direction computation.

To simulate lens sampling, a random point is sampled within a unit disk defined by aperture:

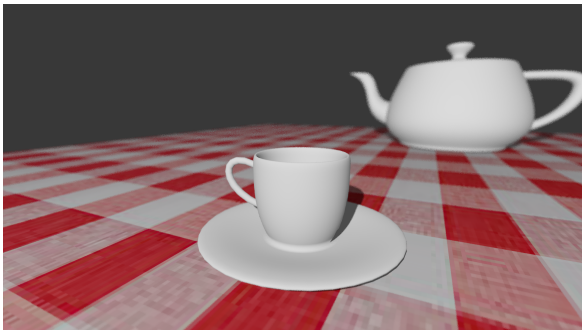
$$p = \text{Vec3}(x, y, 0), \quad x, y \sim U(-1, 1)$$

A rejection sampling technique ensures that points lie within the unit disk by rejecting samples where $x^2 + y^2 \geq 1$. The sample is scaled by the lens radius to calculate the lens offset. The ray origin is offset by the lens sample, while the ray direction is adjusted to pass through the corresponding pixel on the image plane. Finally, the rays are normalized to ensure consistent intersection behavior.

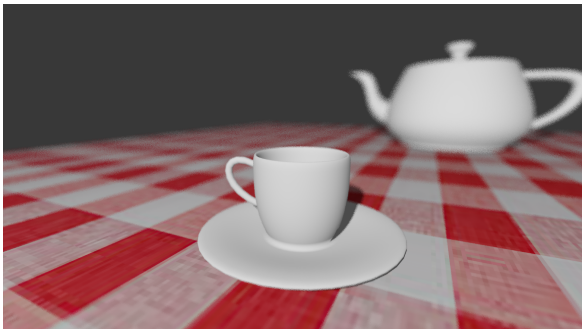
For each pixel in the image, multiple rays are generated using the lens sampling technique. Each ray's color is computed by tracing it through the scene and averaging the results to produce anti-aliasing effects and realistic depth of field.



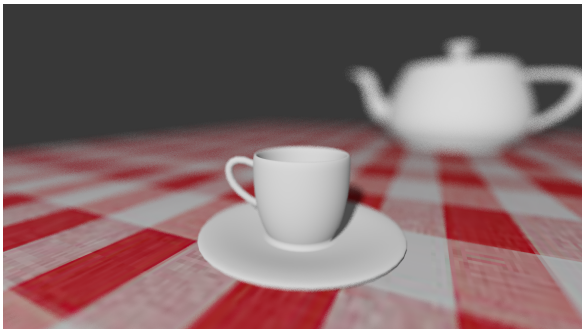
(a) Rendered Image with Aperture = 0 (pin-hole)



(b) Rendered Image with Aperture = 0.05



(c) Rendered Image with Aperture = 0.1



(d) Rendered Image with Aperture = 0.2

Figure 13: Test Figures (PNG) for Lens Sampling implementation

The results of lens sampling are illustrated in Figure 13. The rendered image demonstrated realistic depth of field effects, confirming the implementation's correctness. By comparing Figure 13a

with Figure 13b, we observe that the second image shows noticeable blurring on the tablecloth and teapot, which are outside the focus plane. Furthermore, Figures 13c and 13d demonstrate progressively stronger blurring effects as the aperture size increases.

Currently implementation allow allow for flexible control over the depth of field effect through lens radius and focus distance parameters in json files. However, if the focus distance is not set accurately to match the intended focus object's distance, both near and far objects may appear blurred.

(c) BRDF sampling

We now introduces the `pathtracing` render mode. This BRDF sampling approach is designed to accurately model the interaction of light with different material surfaces, accommodating both diffuse and specular components. The implementation leverages a combination of cosine-weighted hemisphere sampling for diffuse reflections and the GGX microfacet model for specular reflections, enabling the simulation of realistic glossy effects observed in materials like porcelain.

Diffuse Reflection Sampling Diffuse reflection is modeled using cosine-weighted hemisphere sampling, which aligns with Lambertian reflectance principles. This method ensures that rays are more likely to be sampled in directions where the surface normal aligns with the incoming light, adhering to the Lambertian assumption of uniform scattering.

The function `sampleCosineHemisphere(const Vec3& normal)` generates random directions over a hemisphere oriented around the surface normal, with a probability density function (PDF) proportional to the cosine of the angle between the sampled direction and the normal. Uniform random samples are transformed into spherical coordinates, followed by an orthonormal basis transformation to align with the surface normal. According to this , the diffuse BRDF is computed as:

$$\text{BRDF}_{\text{diffuse}} = \frac{k_d \times \text{diffuseColor}}{\pi}$$

where k_d is the diffuse reflection coefficient and `diffuseColor` represents the material's base color.

Specular Reflection Sampling with GGX Microfacet Model Specular reflections, responsible for glossy highlights and mirror-like surfaces, are modeled using the GGX microfacet distribution. The GGX model provides a more physically accurate representation of surface roughness and the distribution of microfacets compared to traditional models like Phong or Blinn-Phong. Thia can be achieved thanks to the `pr` and `pm` parameter given by `.mtl` file when exporting from Blender with PBR extension. Due to complex nature of realization, implementation are divided into different functions:

- **Sampling Function:** The function `sampleGGX(const Vec3& normal, const Vec3& viewDir, double roughness)` samples the half-vector h based on the GGX distribution, considering the surface roughness parameter. The sampled half-vector is then used to compute the reflected direction `outDir` via the reflection equation:

$$\text{outDir} = 2 \times (\text{viewDir} \cdot h) \times h - \text{viewDir}$$

- **Normal Distribution Function (NDF):**
  `ggxNormalDistribution(const Vec3& h, const Vec3& n, double roughness)`
  The GGX NDF quantifies the distribution of microfacet orientations, influencing the sharpness and spread of specular highlights:

$$D(h) = \frac{\alpha^2}{\pi \left[(\vec{n} \cdot h)^2(\alpha^2 - 1) + 1\right]^2}$$

  where $\alpha = \text{roughness}^2$ and $\vec{n}$ is the surface normal.

- **Geometry Function:**
  `ggxGeometryFunction(const Vec3& v, const Vec3& l, const Vec3& n, double roughness)`
  This function accounts for shadowing and masking effects:

$$G(v, l) = G_1(v) \cdot G_1(l), \quad G_1(x) = \frac{\vec{n} \cdot x}{(\vec{n} \cdot x)(1 - k) + k}$$

with $k = \frac{(\text{roughness}+1)^2}{8}$.

- **Fresnel Term:**
  `fresnelSchlick(const Vec3& F0, double cosTheta)`
  The Fresnel-Schlick approximation models the angle-dependent reflectance:

$$F = F_0 + (1 - F_0) \cdot (1 - \cos\theta)^5$$

  where F_0 represents the base reflectivity at normal incidence.

- **BRDF Calculation:**
  The specular BRDF is computed as:

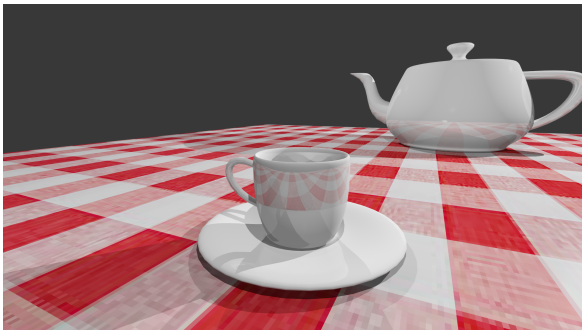
$$\text{BRDF}_{\text{specular}} = \frac{D(h) \cdot G(v,l) \cdot F}{4 \cdot (\vec{n} \cdot v) \cdot (\vec{n} \cdot l)}$$

The final BRDF combines diffuse and specular components:

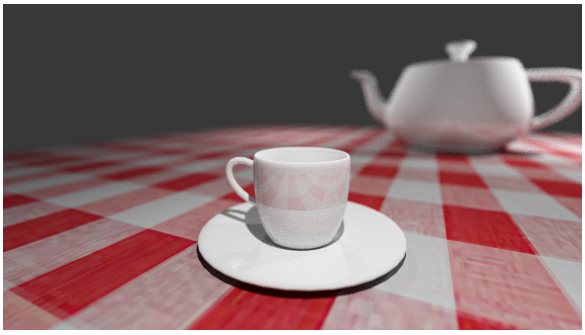
$$\text{BRDF} = \text{BRDF}_{\text{diffuse}} + \text{BRDF}_{\text{specular}}$$

For metallic materials, the diffuse component is attenuated based on the metallic factor, and the specular color is derived from the diffuse color.

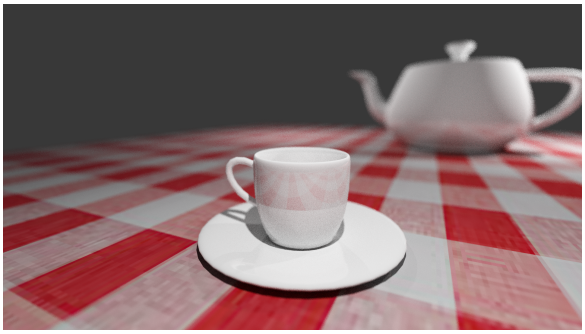
The test set in Figure 14 illustrates the impact of the BRDF implementation. The tests were conducted under identical scene conditions, with minor adjustments to light intensity. Compared to the rendered image under Phong shading in Figure 14a, the shadow in Figure 14b appears more natural and realistic. While in the 14a, a shadow can be barely seen from point light in sides. The proximity of the light source leads stronger and more defined shadows, showcasing the improved realism achieved with BRDF-based sampling. In terms of reflective surface, 14b handles the reflection better with the integration of diffuse color, which look more realistic. Also, we can see from the comparison of Figure 14d and 14d, that as sample increases, the noise in the Image decreases.



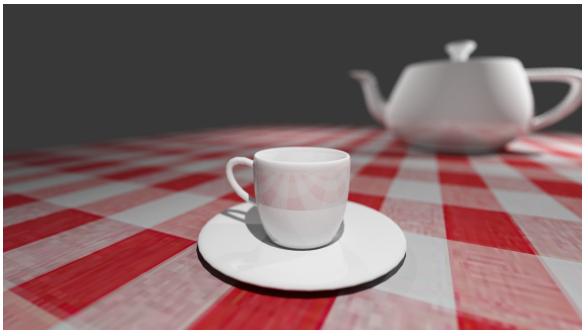
(a) Rendered Image under phong mode with pixel sample = 16



(b) Rendered Image under path tracing mode with pixel sample = 16



(c) Rendered Image under path tracing mode with pixel sample = 32



(d) Rendered Image under path tracing mode with pixel sample = 128

Figure 14: Test Figures (PNG) for BRDF Sampling implementation

However, as shown in Figure 15, increasing the number of samples significantly increases rendering time. Additionally, by comparing Figure 15a with Figure 15b, it is evident that the path tracing rendering mode generally requires more time to compute than the Phong shading mode.

```
helenwang@Wangs-MacBook-Pro-890 CG_CW2 % ./raytracer
focalLength 2.37908 seconds.
BVH Construction time: 0 seconds.
Scanlines remaining: 0
Done.
Rendering time with BVH: 155.431 seconds.
```

(a) Rendering Time for 14a

```
helenwang@Wangs-MacBook-Pro-890 CG_CW2 % ./raytracer
focalLength 2.37908 seconds.
BVH Construction time: 1e-06 seconds.
Scanlines remaining: 0
Done.
Rendering time with BVH: 378.721 seconds.
```

(b) Rendering Time for 14b

```
helenwang@Wangs-MacBook-Pro-890 CG_CW2 % ./raytracer
focalLength 2.37908 seconds.
BVH Construction time: 0 seconds.
Scanlines remaining: 0
Done.
Rendering time with BVH: 774.354 seconds.
```

(c) Rendering Time for 14c

```
helenwang@Wangs-MacBook-Pro-890 CG_CW2 % ./raytracer
focalLength 2.37908 seconds.
BVH Construction time: 0 seconds.
Scanlines remaining: 0
Done.
Rendering time with BVH: 3100.3 seconds.
```

(d) Rendering Time for 14d

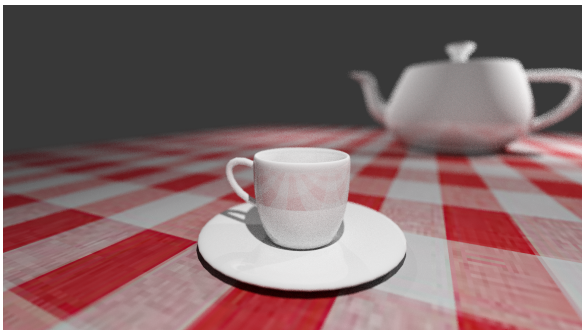
Figure 15: Rendering time for BRDF Sampling implementation

The implemented BRDF sampling strategy offers high realism by accurately modeling both diffuse and specular reflections, leveraging the GGX microfacet model to handle surface roughness and metallic properties effectively. It ensures energy conservation, supports diverse material types, and uses importance sampling to reduce noise and accelerate convergence, making it versatile and physically plausible. However, the implementation has limitations, including increased computational overhead due to the complexity of the GGX model, a simplified Fresnel-Schlick approximation that may not capture all light-material interactions, and the absence of advanced scattering mechanisms like subsurface scattering, which are crucial for certain materials such as skin or marble.

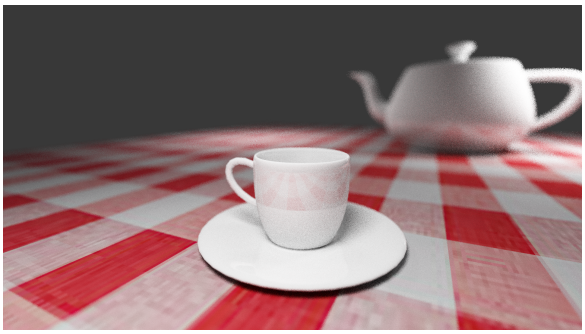
(d) Light sampling

Soft shadows are achieved by sampling across the area of the light source instead of treating the light as a single point. We first randomly sample multiple points within the area light's geometry (e.g., a rectangle or mesh triangle). Then we cast a shadow ray toward each sampled point and compute its contribution. Finally we aggregate the contributions from all sampled points and average them to compute the soft shadow.

This approach produces soft shadow edges by distributing light over an area rather than concentrating it at a single point. This effect is demonstrated by replacing point lights with area lights, which cast softer shadows, as shown in 16. In particular, the shadows cast by the teacup appear noticeably smoother in 16b compared to 16a. Additionally, the transition between the shadowed and illuminated regions of the teapot is more natural and gradual in 16b, enhancing the overall realism.



(a) Rendered Image with point light resources



(b) Rendered Image with area light resources

Figure 16: Test Figures (PNG) for Light Sampling implementation on .obj objects

The system's Modularity is an advantage, allowing for easy extensions to support additional light types such as area lights and spotlights. This modular design facilitates future enhancements and adaptability to more complex lighting scenarios.

Nevertheless, as more light samples are needed, the efficiency of the renderer is also a concern, especially in scenes with numerous light sources, since casting shadow rays for each light significantly increases computational overhead.

5 More Advanced raytracer features

(a) Transmission Filter

The Transmission Filter modulates the light passing through transparent or translucent materials, allowing for realistic rendering of objects such as colored glass or tinted plastics. It adjusts the color and intensity of transmitted light based on the material's properties, contributing to more visually accurate simulations of light interaction.

The transmission filter is applied conditionally, only influencing the light transmission for materials with non-zero transparency. When applied, the transmission filter interacts with the transmitted light by:

- **Sampling a transmission direction:** A direction for light transmission is calculated based on the material's properties.
- **Scaling transmitted radiance:** The transmitted light is modulated by the material's transmission filter, ensuring that light passing through is tinted or altered according to the material.
- **Contributing to final color:** The filtered transmitted radiance is added to the material's final appearance, appropriately scaled by the transparency factor and geometric considerations.

The transmission filter ensures energy conservation by applying modulation only where physically appropriate. This allows materials to realistically interact with light, producing effects such as color tinting for transparent surfaces. A demonstration of the effect is illustrated in Figure 17 and 18, where the teacup is modeled as a black-tinted glass cup.

Figure 17: Rendering Result of Transmission Filter and BSSRDF

(b) BSSRDF (Surface Scattering)

BSSRDF extends the traditional BRDF by accounting for the subsurface scattering of light. While the BRDF models the reflection of light directly at the point of incidence, the BSSRDF captures how light penetrates the surface, scatters beneath it, and exits at a different location. This is essential for rendering translucent materials such as skin, marble, or wax, where light interaction involves multiple scattering events within the material.

The implementation of BSSRDF relies on several key material properties:

- **Absorption Coefficient (σ_a):** Represents the rate at which light is absorbed per unit distance within the material.
- **Scattering Coefficient (σ_s):** Denotes the rate at which light is scattered per unit distance.
- **Anisotropy Factor (g):** Determines the preferential direction of scattering, influencing whether scattering is more forward or backward.

The total extinction coefficient (σ_t) is the sum of absorption and scattering coefficients:

$$\sigma_t = \sigma_a + \sigma_s$$

To simulate subsurface scattering, the renderer first determines how far light travels within the material before scattering. This is achieved by sampling the scattering distance (d) from an exponential distribution, adhering to the Beer-Lambert law, which governs the attenuation of light as it propagates through a medium:

$$d = -\frac{\ln(1-u)}{\sigma_t}$$

where u is a uniformly distributed random variable in the interval $(0, 1)$, ensuring numerical stability by preventing the logarithm from evaluating to zero.

Once the scattering distance is established, the new origin point for the scattered ray is computed by moving the original hit point along the direction of the incoming ray by the sampled distance:

$$\texttt{scatteredOrigin} = \texttt{hitRecord.point} + \texttt{ray.direction.normalized()} \times d$$

This offset ensures that the scattered ray originates from a point within the material, accurately representing the subsurface interaction.

The direction of the scattered ray is sampled using a Phase Function, which models the angular distribution of scattered light within the material. The anisotropy factor (g) influences this sampling, allowing the simulation of materials that scatter light preferentially in forward or backward directions:

$$\texttt{scatteredDir} = \text{ttsamplePhaseFunction}(\texttt{hitRecord.normal}, g)$$

This function ensures that the scattered direction adheres to the physical properties of the material, enhancing realism in the rendered output.

A new ray is then constructed from the scattered origin in the sampled direction:

$$\texttt{Ray scatteredRay(scatteredOrigin + scatteredDir} \times \texttt{0.001, scatteredDir)}$$

The slight offset (0.001 units) along the scattered direction prevents self-intersection artifacts due to floating-point precision limitations.

The incoming radiance along the scattered ray is computed recursively which allows the renderer to accumulate light contributions from multiple scattering events.

Finally, the contribution of the scattered radiance to the final color is computed by combining the absorption factor with the scattering probability. This was intended originally to be implemented simulate the soft light a light bulb could cast on the lamp shade. The effect is shown in Figure 17 and 18. Obviously, to achieve this effect, each mesh forming the light bulb is treated as an area light source, emitting light similar to a standard light source.

Finally, Figure 18 showcases all the discussed effects, including fundamental ray tracing features, texture mapping, glossy materials, pixel sampling, emission materials, soft shadows, transmission filters, surface scattering, and more. Unfortunately, numerous emissive meshes that form the light bulb significantly increases the complexity of light reflections in the scene, making rendering extremely inefficient. Due to time constraints, the final image was rendered with 32 pixel samples and 16 light samples.

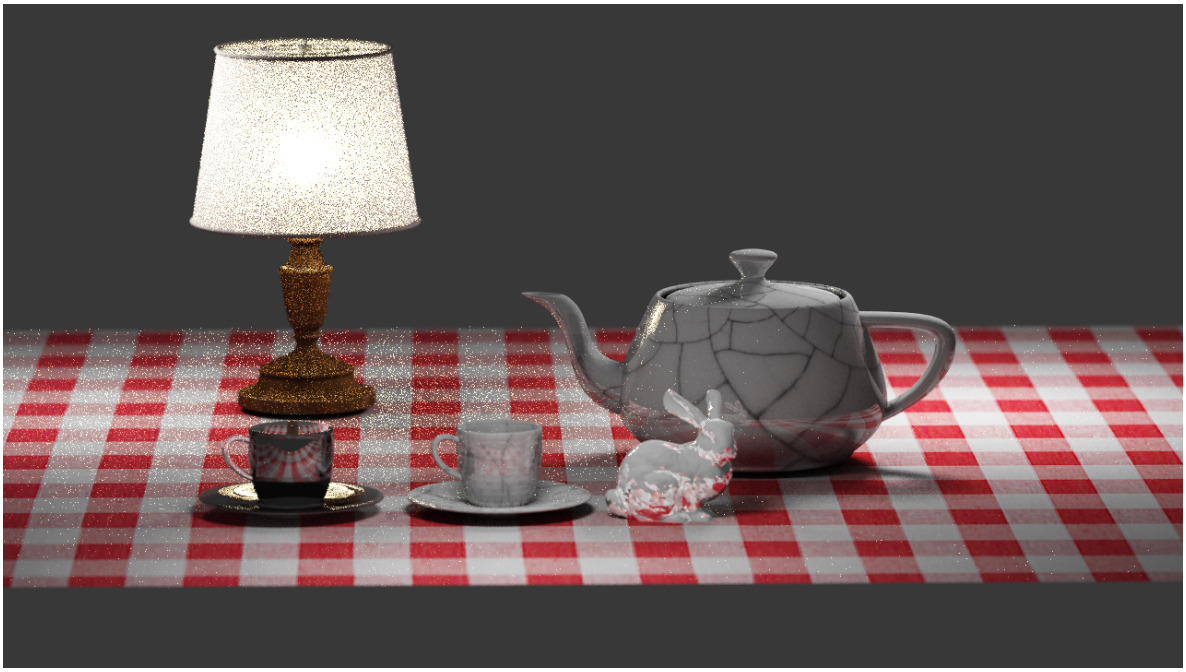


Figure 18: Model in Figure 17 with material

References

[MT97] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[RSSF23] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. *Photographic Tone Reproduction for Digital Images*. Association for Computing Machinery, New York, NY, USA, 1 edition, 2023.