Path Tracing Rendering from Scratch: ACG Final Report

Yingxi Lu 2023011435
lu-yx23@mails.tsinghua.edu.cn
Haidian Qu, Beijing Shi, China

ACM Reference Format:
Yingxi Lu 2023011435. 2025. Path Tracing Rendering from Scratch: ACG Final Report. In . ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This report details the implementation of a path tracing renderer from scratch using C++, with roughly 2500 lines of code. The renderer focuses on achieving photorealistic image rendering by simulating the interaction of light with virtual scenes. Key aspects include scene representation using geometric primitives and materials, the implementation of the core path tracing algorithm for handling diffuse and specular materials, and the development of acceleration structures like Bounding Volume Hierarchies (BVH) to optimize ray-scene intersection tests. Additionally, advanced sampling techniques such as importance sampling and Russian Roulette are employed to enhance efficiency and reduce noise. The report also explores the design of a custom material model and evaluates the challenges and trade-offs encountered during development. A comprehensive analysis of the renderer's performance, limitations, and future improvement opportunities is provided to conclude the study.

Below is the main pipeline of our project:

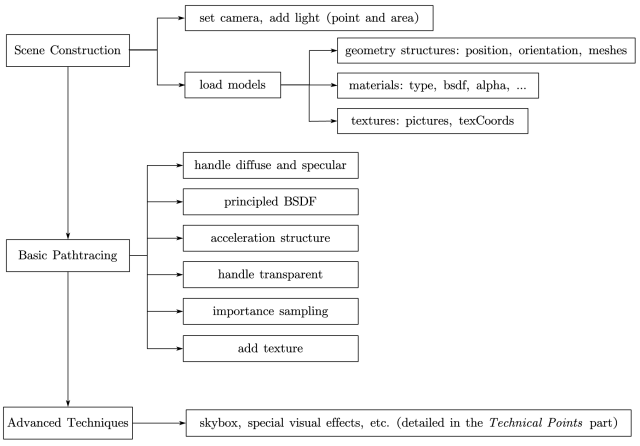

Figure 1: Main pipeline of our project

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 SCENE CONSTRUCTION

We obtained our rendering scene from the Rendering Resources website[2]. Below is the example rendering structure diagram provided on the website:

Figure 2: Example rendering structure diagram[2]

The scene consists of 284,541 triangles, carefully distributed across 60 OBJ files, all composed and controlled via a structured XML file for efficient rendering and scene management. The XML file contains the information about both the materials for each OBJ file, and the textures informations.

Materials. The materials in our scene are defined by the following parameters:

- Type: including Diffuse, Plastic, Conductor, Rough Dielectric, Rough Conductor and Rough Plastic. More specificly:
  - Diffuse: Diffuse materials scatter light uniformly in all directions, often used for surfaces like walls or fabrics. They are rendered using Lambertian reflection, which assumes that light is scattered equally, independent of the viewing angle.
  - Plastic: Plastic materials consist of a diffuse base layer combined with a specular layer on top, giving them both matte and shiny characteristics. The reflectance of the diffuse layer defines the material's color, while the specular layer produces highlights influenced by the Fresnel effect.
  - Conductor: Conductor materials, such as metals, are highly reflective and lack any diffuse scattering. Their optical properties are defined by complex refractive indices (η and k), which determine the color and intensity of reflections. Conductors produce mirror-like surfaces and are commonly used for metals, where accurate reflections are critical for realism.
  - Rough Dielectric: Rough dielectric materials are transparent surfaces with a rough microstructure, causing light

to scatter as it refracts or reflects. The roughness parameter (α) determines the degree of blurring, making these materials suitable for frosted glass or rough transparent objects.
  - Rough Conductor: Rough conductor materials represent metallic surfaces with a textured or brushed appearance. The roughness of the surface scatters reflected light, producing blurred highlights and reducing the polished look of the material. These materials are often used for surfaces like stainless steel or brushed aluminum, where subtle texture enhances realism.
  - Rough Plastic: Rough plastic materials combine a diffuse base layer with a rough specular coating, resulting in a material with less glossiness than smooth plastic. The roughness parameter controls how light is scattered by the specular layer, creating a more subdued and realistic appearance.
- Two sided: Boolean determines if the material interacts with light on both sides.
- Diffuse reflectance: Base color of the material, controlling how light scatters diffusely.
- Specular reflectance: Reflectance of the specular layer, defining the intensity of highlights.
- Alpha: Roughness parameter for rough dielectric and rough conductor materials, controlling the degree of blurring.
- Eta: Real part of the complex refractive index for conductor materials, determining the color of reflections.
- K: Imaginary part of the complex refractive index for conductor materials, influencing the intensity of reflections.
- Interior and exterior refractive index: Refractive indices for dielectric materials, affecting how light is refracted.
- Emission: Defines the color and intensity of self-emitted light from the material.

Aesthetic Appeal. This scene presents a modern bathroom design that is both elegant and well-structured. It combines natural wood textures with clean white surfaces, creating a warm and simple aesthetic. The main features include two stylish freestanding washbasins placed on a wooden counter, paired with wall-mounted faucets to maintain a minimalist look. A built-in bathtub, surrounded by matching wooden panels, adds a sense of luxury to the space.

The room is thoughtfully decorated with small niches that hold sculptures and other subtle details, contributing to a visually balanced layout. Soft lighting enhances the room's clean and welcoming atmosphere, tying the design together seamlessly.

3 METHODS AND TECHNICAL DETAILS

In this section, I will detail the core algorithms and techniques employed in our project to enhance the realism and efficiency of our rendering. The discussion will primarily focus on the components I contributed to, while excluding those implemented by my teammate, due to the report's length considering. Specifically, the following topics will be covered:

- The basic path tracing algorithm, addressing diffuse and specular materials.
- Anti-aliasing techniques.
- Implementation of point light and area light resources.
- Acceleration structure with BVH, SAH and multi-thread.
- Integration of color textures.
- Importance sampling with Russian Roulette and multiple importance sampling.
- Motion blur effects.
- Environment lighting with HDR.

P.S. . Although handling transmissive materials is outlined as a separate basic task in the project documentation, I have integrated its handling method with those of diffuse and specular materials in the following explanations.

3.1 Basic Path Tracing

The basic path tracing algorithm simulates global illumination by recursively tracing rays as they interact with surfaces in the scene. The algorithm emits rays from the camera and traces them until they intersect with a light source or escape the scene bounds.
For diffuse materials, it samples random directions on the hemisphere above the surface and calculates the contribution of incoming light based on the Lambertian reflection model. For specular materials, it computes the precise reflection direction and continues tracing the ray.
By incorporating these processes, the path tracing algorithm effectively models complex light transport phenomena, such as indirect illumination and reflections, to produce physically realistic images. Here is the pseudocode for the basic path tracing algorithm:

Algorithm 1: Path Tracing Algorithm

```
Function: Shade(p, wo);
Input: Point p, outgoing direction wo;
Output: Radiance L;
Trace a ray r(p, wo);
if r hits a light source then
    return Emitted radiance L_i;
if r hits a surface at q then
    Let w_i be the output light direction /* Check the
       material type at q                                  */
    if material is diffuse then
        w_i = Sample a random direction on the hemisphere
          above the surface normal;
    if material is specular then
        w_i = Reflect(wo, n);
    if material is transmissive then
        if not total internal reflection then
            w_i = Refract(wo, n);
    return Shade(q, wi) · f_r · cos θ / pdf(wi);
return 0 /* Ray escapes the scene bounds                  */
```

3.2 Anti-aliasing Techniques

The anti-aliasing method generates multiple random sample points within each pixel and calculates the radiance for each sample ray.

The results are then averaged to reduce aliasing effects and image noise.
Specifically, for each pixel, multiple random sample points are generated and mapped to the view frustum to determine ray directions. Each ray is traced until it intersects with an object, a light source, or escapes the scene. The contributions of all samples are accumulated and averaged to compute the final radiance value for the pixel.
This approach improves rendering precision and ensures smoother and more visually appealing results by performing multiple samples per pixel.

Algorithm 2: Ray Generation for Path Tracing

Function: ray_generation(camPos, pixel);
Uniformly sample N positions $\{s_1, s_2, \ldots, s_N\}$ within pixel;
pixel_radiance $\leftarrow 0$;
for *each sample* s_i *in* $\{s_1, s_2, \ldots, s_N\}$ **do**
  Compute ray direction:
  dir $\leftarrow$ get_ray_dir(camPos, s_i);
  Trace ray r(camPos, dir);
  if r *hits the scene at point* p **then**
    pixel_radiance+ = $\frac{1}{N} \cdot$ shade(p, dir);
return pixel_radiance;

3.3 Point Light and Area Light Resources

The handling of point lights and area lights is achieved through explicit calculations of their contributions to direct and indirect illumination. Direct illumination is computed by tracing rays from the surface to the light source, while indirect illumination is calculated using the path tracing algorithm, i.e., interatively tracing the scattered/reflected light.
In this section, I will provide a detailed explanation of how point lights and area lights are implemented in our path tracer, covering their respective sampling strategies and lighting contributions.

3.3.1 ***Point Light.*** Point lights are idealized light sources that emit light uniformly in all directions from a single point in space. Unlike area lights, point lights do not have a spatial extent, and their illumination contribution depends on the inverse square of the distance.

In the implementation, point lights are treated as singular entities, and their contribution is computed directly without the need for sampling over a surface. This simplifies the computation compared to area lights.

Direct Illumination. Direct illumination from point lights is computed by checking the direct visibility of the light source from the shading point. This involves:

- Direction and distance calculation: The light direction is computed as the vector from the shading point to the point light's position. The light distance is the Euclidean distance between the shading point and the point light.
- Shadow testing: A shadow ray is cast from the shading point slightly offset in the normal direction to the light source. If the shadow ray intersects any object before reaching the light, the point light is considered occluded, and its contribution is ignored.
- Light attenuation: The intensity of the point light is attenuated by the inverse square of the distance to the shading point.
- Contribution computing: The diffuse component is calculated using Lambert's cosine law, which accounts for the angle between the surface normal and the light direction. The contribution is scaled by the material's diffuse reflectance.

diffuseColor+ = brdf $\times$ lightIntensity $\times$ lightColor $\times$ cosTheta $\times$ attenuation.

For specular materials, the reflection direction is calculated, and the Phong or GGX model is used to compute the specular contribution, where Phong model is simpler and less computationally expensive, and is used when the surface is relatively smooth, and the highlights can be approximated by the specular exponent; while the GGX model is more physically realistic and more computationally costly, and is used when accurate light reflection behavior is necessary, e.g., metals, rough glass, or other microfacet-based surfaces.
For conductor, we use Phong model to compute:

$$L_{specular} = k_s \cdot I \cdot (\max(0, R \cdot V))^{\alpha},$$

where k_s is the specular reflectance, I is the light intensity, R is the reflection direction, V is the view direction, and α is the shininess factor.
And for other specular material, we use GGX model to compute:

$$L_{specular} = \frac{D(h) \cdot G(l, v) \cdot F(v, h)}{4(N \cdot L)(N \cdot V)},$$

where $D(h)$ is the normal distribution function, $G(l, v)$ is the geometry function, and $F(v, h)$ is the Fresnel function.
The contribution from refraction is calculated using Fresnel equations. The Fresnel term determines the proportion of light refracted based on the relative indices of refraction (η) and the incident angle. For refraction, the contribution is scaled by (1 − fresnelTerm), with the remaining light accounted for in reflection.

$$L_{refraction} = (1 - \text{fresnelTerm}) \cdot I_{light} \cdot attenuation$$

Indirect Illumination. Indirect illumination due to point lights is handled through recursive path tracing: For reflective or refractive materials, the recursive tracePath function handles the light contribution by spawning secondary rays in the reflection or refraction direction. The point light's contribution is implicitly included in the computation when the secondary ray intersects geometry that can "see" the point light.

3.3.2 ***Area Light.*** Area lights are light sources with a finite spatial extent, which is represent by a rectangular consist of two triangles in our project (since we only store triangles). Unlike point lights, area lights require sampling to compute their contribution accurately, as light is emitted from the entire surface of the area light.
We also store the position of the light, two spanning vectors u and v, and a sampling resolution to determine the number of points sampled on the surface of the light.

Direct Illumination. The direct illumination pipeline and contribution calculator for area lights are similar to those for point lights, with only adding a sampling step. In the basic implementation, we uniformly sample points on the area light surface, compute the contribution from each sampled point and then average the results. In more advanced implementation, we can use importance sampling to sample points on the area light surface, and multiple importance sampling to couple with indirect contributions.

Indirect Illumination. The indirect ilumination for area lignt is also similar to the point light, where the indirect contribution is computed by recursively tracing rays from the shading point. The only difference is that, if the ray intersects the area light, add the emission of the area light to the indirect contribution.

3.4 Acceleration Structure

In path tracing rendering, the complexity of the scene directly correlates with the exponential increase in intersection calculations between rays and geometry. When dealing with highly complex scenes containing numerous objects, brute-force ray-geometry intersection testing becomes prohibitively inefficient. To address this bottleneck, acceleration structures are introduced, significantly reducing the number of intersection tests required and optimizing rendering performance.

This section discusses the implementation of a Bounding Volume Hierarchy (BVH) optimized with the Surface Area Heuristic (SAH) to organize and partition the scene efficiently. Additionally, multithreading techniques are employed to further enhance computation speed, enabling parallel processing of ray intersections and ensuring scalability for high-resolution images and complex scenes.

3.4.1 *Bounding Volume Hierarchy (BVH).*

Tree-like BVH. . The tree-like BVH structure organizes the nodes into a hierarchical manner, where each interior node stores the pointer to its child nodes, and a bounding box that encloses its child nodes. The leaf nodes contain a small subset of triangles in the scene. We use a recursive function to build the BVH, at each node, if it contains less than 50 triangles, or the iteration depth is greater than 32, we mark it as a leaf node; otherwise, we compute the bounding box for the node, sort the triangles by the longest axis of the bouding box, and split them into two equal halves at the median point. Then, we iteratively build the BVH for the two halves.

In the result part, we will see that, this technique can significantly reduce the running time of the path tracing rendering.

However, this technique has some limitations. A common limitation of tree-like BVH structures is that intersection tests can be slow because they rely on pointer-based traversal. Accessing nodes through pointers takes extra time and reduces efficiency. To solve this problem, the BVH can be stored in a flat array. This method makes memory access faster and improves the overall performance of the traversal process.

Flatten BVH. . In the improved flatten BVH structure, we store the BVH flatten node in an array[3]. Specifically, we fit the BVH tree into a flatten array in the following way:

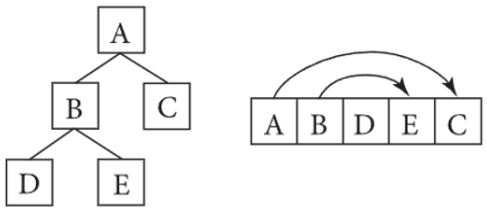

Figure 3: Flatten BVH structure

For each internal node in the BVH, we assign its left child to be the next node in the array and record the index of its right child. Additionally, we maintain an initially empty array to store ordered triangles. For leaf nodes, we append the triangles they contain to this ordered triangles array. Instead of storing the index of a right child, leaf nodes store the starting index of their triangles in the ordered array. Each node also contains metadata, such as its bounding box and the number of triangles it represents.

This approach allows for efficient traversal and intersection tests, as the nodes can be accessed sequentially in memory, reducing cache misses and improving performance. We will see in the result part, this trick reduces the running time by 80%, addressed the primary bottleneck in our performance optimization.

SAH (Surface Area Heuristic). Unlike traditional tree-like BVH construction methods, which might use simple strategies like splitting based on the median, SAH uses a heuristic splitting strategy, which focuses on minimizing the traversal and intersection costs during ray tracing, resulting in a more efficient hierarchy. Specifically, SAH evaluates potential splits along each axis using a cost function based on the surface area of the child bounding volumes and the number of triangles within each volume. This ensures that splits minimize the expected cost of ray traversal, rather than relying on geometric or arbitrary heuristics. The main pipelien for SAH splitting is as follows:

- Bucketized sorting: Triangles are sorted into buckets based on their centroid positions, simplifying the calculation of bounding volumes for child nodes.
- Surface area calculation: The algorithm computes the surface area of the bounding volumes for the left and right child nodes at each potential split. The cost function weights the surface area by the number of triangles in each volume.

  $$\text{cost} = SA_{left} \cdot N_{left} + SA_{right} \cdot N_{right},$$

  where SA is the surface area of the bounding volume, and N is the number of triangles in the volume.
- Split selection: Select the split axis and position that minimizes the cost function, ensuring that the resulting child nodes are balanced in terms of triangle distribution and surface area.
- Recursive Construction: Repeat the process for each node until reach the leaf node condition (too small triangles or too deep).

By integrating SAH into BVH construction, the resulting acceleration structure is more efficient for ray tracing. Rays traverse fewer nodes on average, and intersection tests are concentrated in regions with higher probabilities of intersections. In the result part we will

see that, this technique improves about 50% of the performance, compared to flatten BVH.

3.5 Color Texture

The implementation for color texture is trivial. We simply adjust the diffuse reflectance of the material by the color of the texture.

3.6 Importance Sampling

3.6.1 Basic Importance Sampling (IS). For basic importance sampling, we independently sample light and BRDF.

Light sampling. Let $\cos\theta_1$ be the dot of the surface noraml and the ray direction, $\cos\theta_2$ be the dot of area light normal and the inverse of ray direction, A be the surface area of the light, dist be the distance from the intersection point to the sampled light position, f_r be the BRDF function at this point. Then, we have:

$$\text{pdf} = \frac{1}{A},$$
$$L_{dir} = \frac{L_i \cdot f_r \cos\theta_1 \cos\theta_2}{\text{dist}^2 \cdot \text{A}}.$$

BRDF sampling. Let $\cos\theta$ be the dot of the surface normal and the ray direction, similarly, we have:

$$L_{indir} = \frac{L_{recursive} \cdot f_r \cos\theta}{\text{pdf}},$$

where

$$f_r = \frac{reflectance}{\pi},$$
$$\text{pdf} = \frac{\cos\theta}{\pi},$$

we have

$$L_{indir} = reflectance \cdot L_{recursive}.$$

The total contribution is the sum of the direct and indirect contributions:

$$L = L_{dir} + L_{indir}.$$

Algorithm 3: Russian Roulette for Path Tracing

```
Function: russian_roulette(ray, depth, threshold);
if depth ≥ max_depth then
    return 0;
    // Terminate if max depth is reached
Compute the probability
 p ← compute_survival_probability(ray);
Uniformly sample a random number r ∈ [0, 1];
if r > p then
    return 0;
    // Terminate the path
Scale contribution: weight ← 1/p;
return weight · trace_next_ray(ray, depth + 1);
```

Russian Roulette (RR). . In Russian Roulette, we randomly terminate the path with a probability p and continue with a probability $1 - p$.

3.6.2 Multiple Importance Sampling (MIS). In MIS, we sample both light and BRDF with a mixed PDF.
Let pdf_l be the PDF of light sampling, pdf_b be the PDF of BRDF sampling, and L_{dir} and L_{indir} be the direct and indirect contributions respectively. Use power heuristic, the weight for light sampling and BRDF sampling is:

$$w_l = \frac{pdf_l^2}{pdf_l^2 + pdf_b^2},$$
$$w_b = \frac{pdf_b^2}{pdf_l^2 + pdf_b^2}.$$

And the total contribution is:

$$L = w_l \cdot L_{dir} + w_b \cdot L_{indir}.$$

3.7 Motion Blur

In this section, we follow the tutorial presented in Ray Tracing: The Next Week [4]. To achieve motion blur effects, we extend the *Ray* structure by introducing a time attribute. For each ray sampled by the camera, we assign its time by a random number from 0 to 1. Additionally, we include moving spheres in the scene, defined by their radius, initial position at time 0, and final position at time 1. When a ray intersects a moving sphere at a specific time t, the sphere's position at that time is computed. By averaging the results from multiple random samples, we can simulat the motion blur effect.

3.8 Environment Lighting with HDR

We get our HDR resources at *https://polyhaven.com/hdris*[1]. To implement the skybox, the main pipeline is shown below:

- Load HDR texture and data with *stb_image*.
- Implement the function that map the direction of the ray to the texture coordinate, and return the corresponding color.
- Add the contribution of the skybox to the indirect illumination.

4 RESULTS AND DISCUSSIONS

4.1 Point Light

Here is a demo for point light:

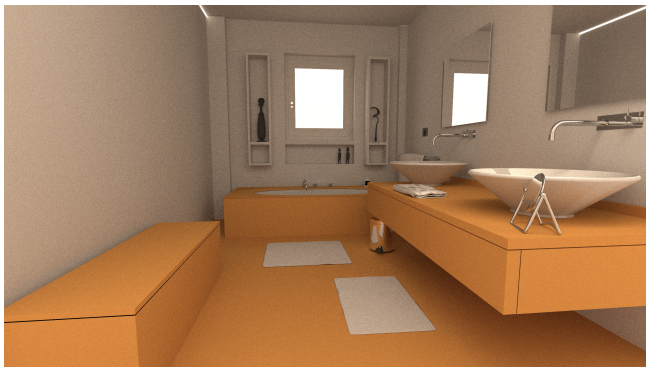

Figure 4: Point Light

Algorithm 4: HDR Image Loading and Path Tracing

Function: `loadHDR(filepath)`;
  Load HDR image file into float array (R, G, B values);
  return image data with width, height, and pixel data;
Function: `sampleEquirectangularMap(hdr, direction)`;
  Normalize direction vector;
  Compute $u \leftarrow 0.5 + \frac{\text{atan2}(z,x)}{2\pi}$;
  Compute $v \leftarrow 0.5 - \frac{\text{asin}(y)}{\pi}$;
  Map u, v to pixel coordinates x, y;
  Fetch RGB values from `hdr` data at (x, y);
  return RGB color;
Function: `tracePath(ray, scene, hdr)`;
  if `ray` *hits* `scene` **then**
    Compute surface shading;
    return surface color;
  else
    return `sampleEquirectangularMap(hdr, ray.direction)`;

Since the point light is positioned above the ceiling, its brightness is limited. Additionally, there is a gap on the upper-left side of our scene, allowing light to be visible from the outside. Moreover, our scene is enclosed on five sides, leaving the side where the camera is located open, which also allows light to escape and affects the shadows of the two washbasins. This issue is addressed in the subsequent demos by adding a large triangle behind the camera. Below demos all with area light.

4.2 Transmissive Material

To explicitly show the transmissive material, we modified the mateiral of the tub to glass. Below is a demo for transmissive material handling:

Figure 5: Transmissive Material

4.3 Acceleration Structure

We evaluate the acceleration performance by the runing time shown after "*Profile "Render xxx xspp":* in each output log shown below.

Tree-like BVH. . The running time before and after basic tree-like BVH under the same setting:

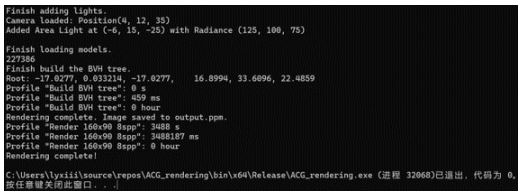

Figure 6: before BVH

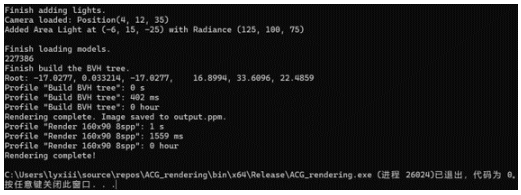

Figure 7: Basic Tree-like BVH

We can see that there is a huge improvement.

Flatten BVH. . The running time before flatten (tree-like) and after flatten under the same setting:

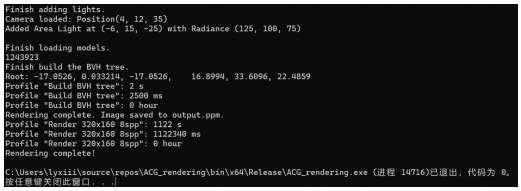

Figure 8: Before Flatten (Basic Tree-like BVH)

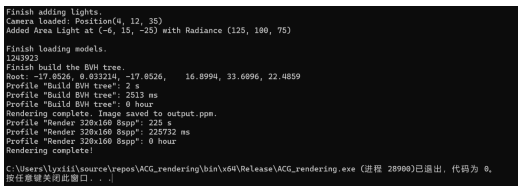

Figure 9: Flatten BVH

Thereis about 80% improvement.

SAH. . The running time before SAH and after SAH under the same setting:

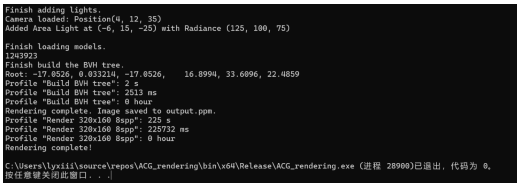

Figure 10: Before SAH (Flatten BVH)

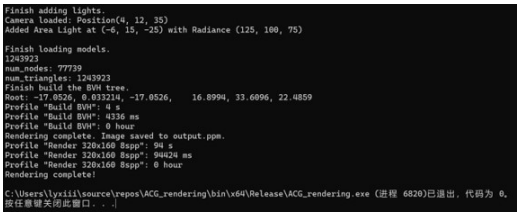

Figure 11: After SAH

4.4 Color Texture

Here is an example for our color texture:

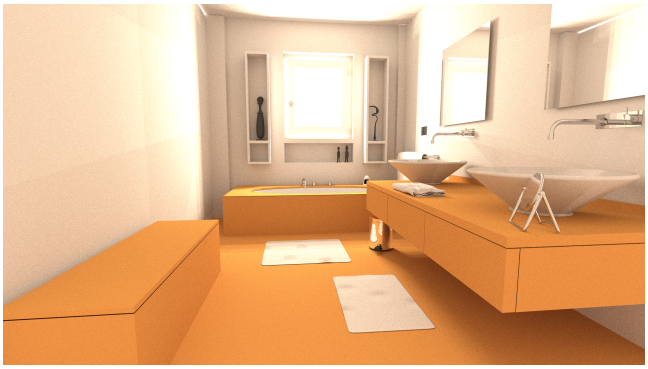

Figure 12: Color Texture Example

This is the demo we presented in the final presentation, featuring an orange color tone to match the light sources.
The following demos use a blue color tone to better align with the color of the skybox.

4.5 Importance Sampling

Apply importance sampling with Russian Roulette:

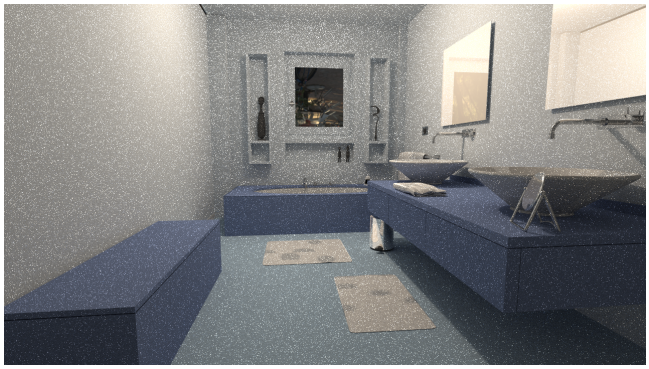

Figure 13: Importance Sampling with RR

We can see that the result is good, but still has some noise.
Apply multiple importance sampling (also modified the material, add specular component to the vanity countertop and other objects):

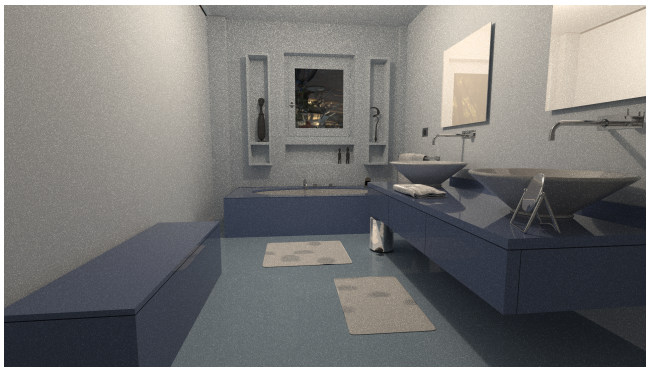

Figure 14: Multiple Importance Sampling

We can see that the rendering output is much better than the basic IS.

4.6 Motion Blur

Below is a demo for our motion blur effect:

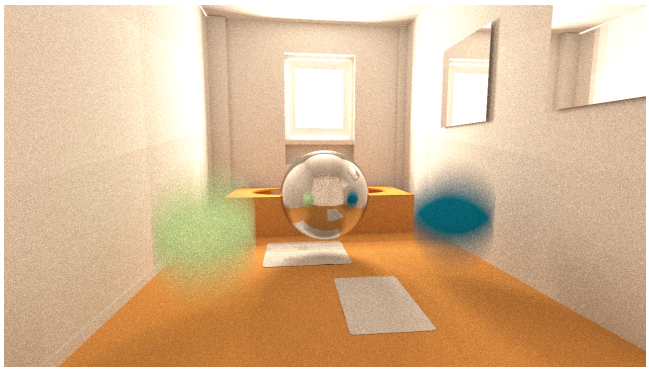

Figure 15: Motion Blur

Below are the demos showcasing our motion blur effect. The scene includes a stationary mirror sphere, with a light green moving sphere on one side and a lake blue moving sphere on the other. Moreover, as reflected in the mirror sphere, the previous issue of five enclosed sides with one open side has already been resolved.

4.7 Skybox

Here is a demo for our skybox:

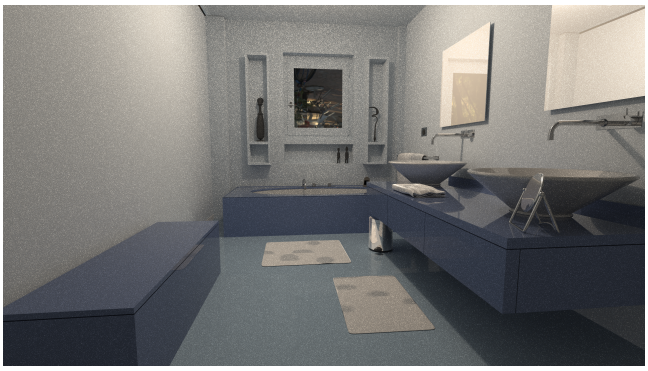

Figure 16: Skybox

In more detial:

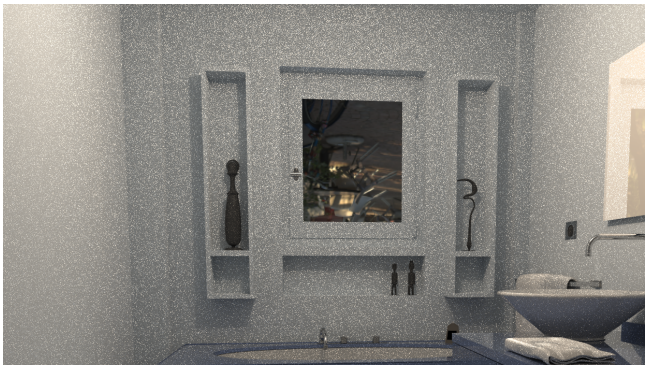

Figure 17: Skybox2

5 PERSONAL CONTRIBUTION

In this section, I will outline my contributions to the project, following the order specified in the project documentation:

- **Base:** I established the fundamental path tracing framework, which includes the basic path tracing iterations and intersection tests that sequentially evaluate every triangle in the scene.
- **Scene Creation:** I sourced the scene online, implemented the loading code to import OBJ file data into models and triangles, and set up the lighting and camera. My teammate implemented the XML file parsing, defined material structures, and loaded material data from the XML files.
- **Acceleration Structure:** I independently implemented all acceleration structures, including BVH, SAH, and multi-threading optimizations.
- **Material:** My teammate developed the structure for transmissive materials and BRDF. I resolved several bugs, particularly in handling mixed materials.
- **Texture:** I implemented the basic color texture mapping, while my teammate worked on integrating normal maps and height maps into texture mapping.
- **Importance Sampling:** I independently implemented the importance sampling module, including Russian Roulette-based importance sampling and multiple importance sampling.
- **Special Effects:** I developed the motion blur effect, while my teammate implemented depth of field and alpha shadow effects.
- **Lighting:** I independently implemented all aspects of the lighting system, including point lights, area lights, and HDR lighting.
- **Anti-aliasing:** I incorporated anti-aliasing functionality as part of the initial setup for the basic path tracing algorithm in Step 1.

REFERENCES

[1] [n. d.]. Poly Haven. https://polyhaven.com/hdris/.
[2] Benedikt Bitterli. 2016. Rendering resources. https://benedikt-bitterli.me/resources/.
[3] HeaoYe. 2024. CPU Path Tracing. https://github.com/HeaoYe/CPUPathTracing/.
[4] Peter Shirley. 2020. Ray tracing: The next week.