CSE 167 (WI 2025) Final Project – Due 3/21

Submission Guidelines

This final project contains a step by step (7 steps) guide towards a basic working ray tracer. To make sure that each main tasks are implemented correctly, you are required to submit output images for each of the seven steps. So save these output images while working on these seven steps. This would constitute the **implementation** part of the final project.

In addition to implementation, you will also submit a **write-up** (pdf file). There, explain what you have implemented to demonstrate your understanding of the topic. In particular, for Task 7 about doing something new, clearly explain your algorithm and your reasoning behind the algorithm. You may include details of your learning outcomes, the challenges you encountered, and the experiments you conducted.

In the write-up, include a result section that contains a **compilation of your favorite results** that best show the power of your ray tracer.

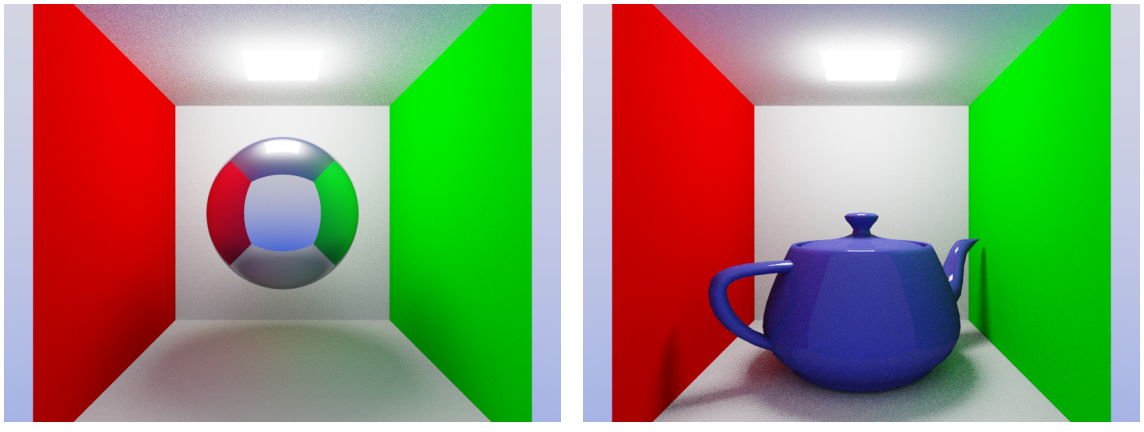
In summary:

- 7 step implementation (10% of course grade). Upload the output of each step.
- Write-up (5%).
- Quality of demonstration (result section of write-up) (5%)
- Bonus (up to 5%) for excellent final projects.

Introduction

In this final project, you will create a ray tracer. Ray tracers can generate some of the most impressive and realistic renderings by producing high-quality shadows, reflections, and more. The goals of this project are to:

- Implement ray tracing to achieve realistic effects such as shadows and soft shadows.
- Implement shading model with diffuse and specular material properties.
- Implement multiple ray bounces to achieve global illumination.
- Build and render a complex scene on your own or add some new feature.



(a) Mirror-like ball in Cornell box **(b)** A glossy Utah teapot

Figure 1 Images rendered using the Ray Tracer you are about to code

Before implementation

Make sure that you can compile and run the provided code base. On macOS and Linux, you can do the following:

```
mkdir build
cd build
cmake ..
cmake --build .
../bin/RayTracer 1 1 1
```

On Windows, compile and run using Visual Studio in a similar way.
This should produce an image with a completely blue screen, as shown in Figure 2.

Figure 2 Initial image generated when the code is run without any changes

The generated `RayTracer` executable takes three command line arguments:

```
../bin/RayTracer <int: samples_per_pixel> <int: max_ray_bounces> <int: scene_id>
```

We will understand each of these arguments in the coming sections.

Task 1: Ray creation (3 pts)

The first task is to complete the implementation of the function:

```
Ray RayTracer::ray_thru_pixel()
```

which is located in `src/RayTracer.cpp`.

- Make sure you properly compute the direction of the ray from the camera through the pixel.
- Use the perspective projection model discussed in the lecture, ensuring that the aspect ratio matches the screen dimensions.
- For now, the ray must go through the center of the pixel. We will change this later in Task 3.

If this part is implemented correctly, the output image should resemble Figure 3, showing a gradient transitioning from white at the top to blue at the bottom. Refer to Debugging for debugging-related assistance.. This gradient represents non-intersecting rays hitting the sky. Mathematically, the color of these rays is determined by a linear interpolation between white and blue based on the ray's y axis component.

Figure 3 Expected result of correctly implemented ray creation

Task 2: Ray Intersections (10 pts)

Next, you will implement ray-geometry intersection. Two geometry classes, **Sphere** and **Triangle**, are provided in the code base. By combining these two shapes, we can construct a wide variety of models.

Task 2.1: Ray-Sphere Intersection (5 pts)

A basic scene has already been set up with a sphere inside a box. However, because the ray-sphere and ray-triangle intersection logic is missing, the ray tracer cannot detect any geometry yet. Your job for **Task 2.1** is to implement the ray-sphere intersection in:

```
vector<Intersection> GeomSphere::intersect(Ray &ray)
```

which is found in `src/geometries/GeomSphere.cpp`. This function returns a `vector<Intersection>` of the given ray's intersections with the sphere. The `Intersection` struct (in `include/Intersection.h`) holds details about a ray-geometry intersection, such as the intersection point and normal.

- Recall that a sphere intersection can be found by solving the quadratic equation

$$\|\mathbf{r_o} + t\mathbf{r_d} - \mathbf{c}\|^2 = R^2,$$

  where $\mathbf{r_o}$ and $\mathbf{r_d}$ are the ray origin and direction, respectively, $\mathbf{c}$ is the sphere center, and R is its radius.
- Only include the intersection(s) where $t \geq 0$, since negative t values lie behind the camera.

Note: One important thing to keep in mind is that the ray is already transformed into the model's coordinate system. Additionally, the inverse transformations are handled by the wrapper function calls in the model's `intersect(..)` method. This means you can apply deforming transformations to your models without worrying about adjusting normal vectors, because the process is handled automatically. Check out the implementation of `ModelBase::intersect(..)` in the `include/ModelBase.h` file.

If this part is correctly implemented, the output image should look like Figure 4. By default, our ray tracer uses normal shading, so you will see a sphere shaded by its normals. At this point, the sphere is correctly detected, but the ray tracer still cannot detect triangles.

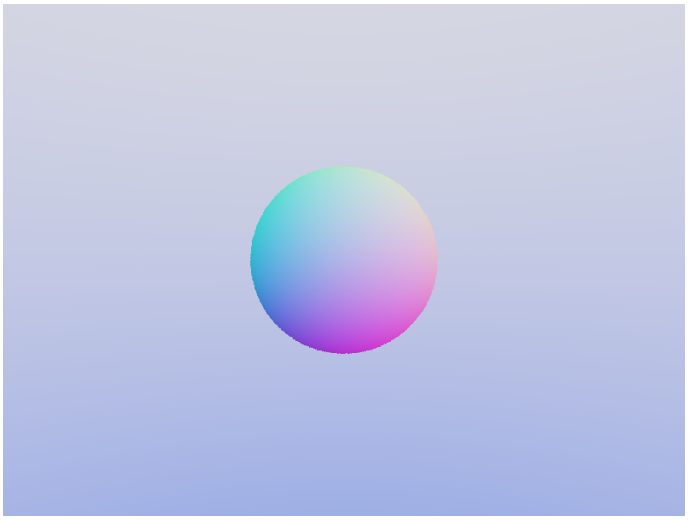


Figure 4 Sphere detected by the ray-sphere intersection code

Task 2.2: Ray-Triangle Intersection (5 pts)

The scene also includes walls of a box made from triangles. In order to see these walls, you must implement the ray-triangle intersection. For **Task 2.2**, complete:

```
vector<Intersection> GeomTriangle::intersect(Ray &ray)
```

in `src/geometries/GeomTriangle.cpp`. Similar to the sphere intersection, this returns a `vector <Intersection>` of the ray's intersections with the triangle.

- You can use the Möller–Trumbore algorithm or a standard barycentric coordinate approach.
- Remember to check that the barycentric coordinates lie within $[0, 1]$ and sum up to 1 (or very close to it, within floating-point error).
- Also ensure $t \geq 0$ for a valid intersection in front of the camera.

If implemented correctly, the result should resemble Figure 5, where the walls of the box are visible.

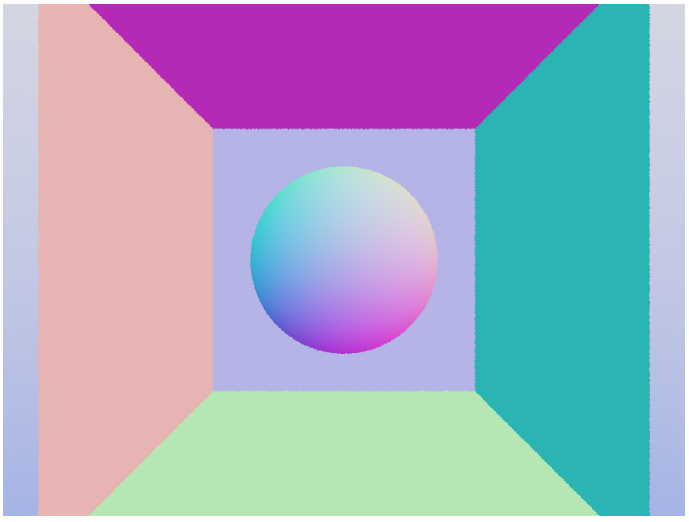


Figure 5 Walls of the box detected once ray-triangle intersection is implemented

Task 3: Anti Aliasing (3 pts)

You can save your scene as an image by pressing `O`. This saves `test.png` in your working directory. If you open this image and zoom in around object edges (Figure 6a), you will notice jagged boundaries. So far, we have been shooting only one ray per pixel (through the pixel center). At edges, a pixel might be only partially covered by an object, so sending multiple rays per pixel and averaging their contributions helps produce smooth edges.

```
../bin/RayTracer 1 1 1
```

Until now, we used only one sample per pixel (spp). This can be changed by increasing the first command-line argument:

```
../bin/RayTracer 10 1 1
```

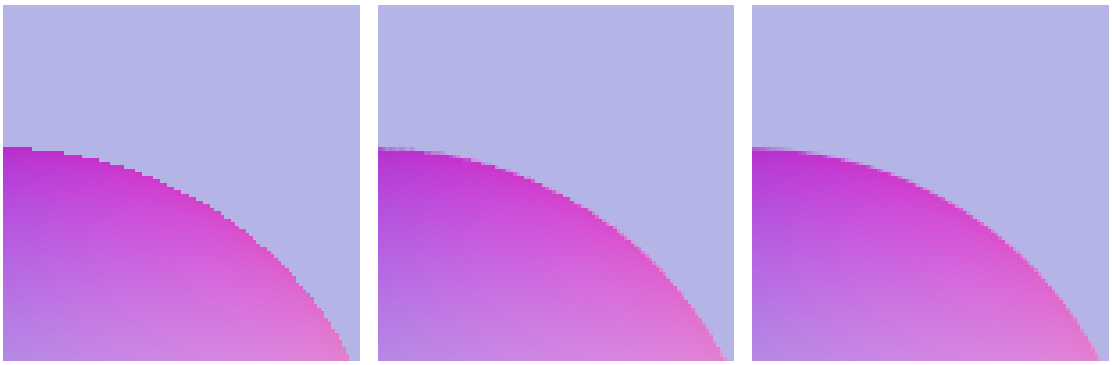
However, simply increasing the number of samples per pixel will not achieve anti-aliasing yet, because all rays still pass through the same center point. For **Task 3**, modify

`RayTracer::ray_thru_pixel()`

in `src/RayTracer.cpp` so that it randomly samples a point inside each pixel. Then, if `RayTracer::ray_thru_pixel()` is called multiple times for a single pixel, each call should yield a different random position within that pixel.

- Use `glm::linearRand()` to generate random x and y offsets in $[0, 1]$.
- Make sure to `#include <glm/gtc/random.hpp>`.
- After introducing this random sampling, edges will appear smoother.

Figures 6 illustrate the improvement from increasing samples per pixel.



(a) Single ray per pixel **(b)** 10 rays per pixel **(c)** 100 rays per pixel

Figure 6 Zoomed-in patches of the rendered image under different samples-per-pixel values

Note on performance: Increasing *spp* significantly raises the total number of traced rays. For example, 10 *spp* on an 800×600 resolution entails $10 \times 800 \times 600 = 4{,}800{,}000$ rays, which can be slow. For debugging, it is wise to use smaller *spp* values. By default, our ray tracer begins with normal shading (which is much faster). Moving forward, it is unnecessary to apply multiple samples per pixel in normal shading mode. Once Task 3 is complete, you can modify the value of `active_samples_per_pixel` in `RayTracer::draw()` (found in `src/RayTracer.cpp`) to `1` for normal shading, and only use higher *spp* values for actual ray tracing.

Shading (17 pts)

Up to this point, we have used normal shading. If you press space to switch to actual ray tracing, you will see that, apart from the light source, everything is black (Figure 7). This happens because no shading model is implemented yet. In **Task 4**, you will implement a shading model to produce more realistic appearances, including soft shadows, color bleeding, and mirror reflections.

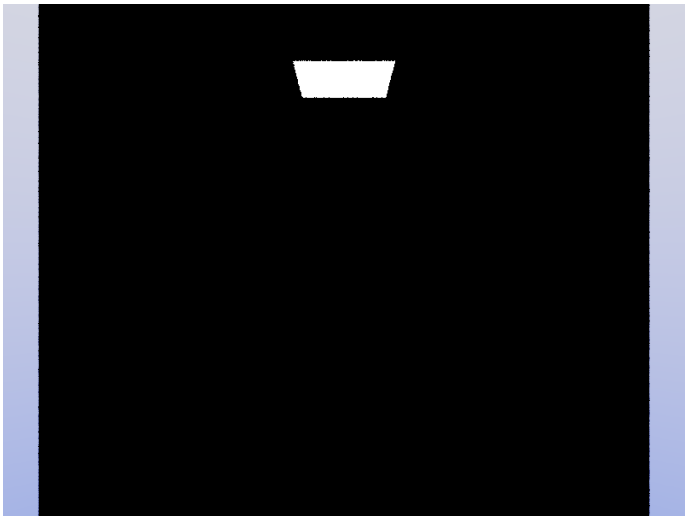


Figure 7 Scene render before implementing Task 4.1

We can express the recursive shading model from the ray tracing lecture as:

$$\mathbf{L}_{\text{seen}} = \begin{cases} W_{\text{specular}}\, \mathbf{L}_{(\mathbf{p},\mathbf{r})} & \text{with probability } \sigma, \\ W_{\text{diffuse}}\, \mathbf{L}_{(\mathbf{p},\mathbf{d})} & \text{with probability } (1-\sigma). \end{cases} \tag{1}$$

Here,

$$W_{\text{diffuse}} = C_{\text{diffuse}}(n \cdot d), \quad W_{\text{specular}} = C_{\text{specular}}.$$

Also, for N concrete ray bounces, we have:

$$\text{Color}_N(\text{RayAtPixel}_{(i,j)}) = W_1 W_2 W_3 \cdots W_{N-1}\, \text{Color}(\text{LastBounce}), \tag{2}$$

where each W_i is either W_{specular} or W_{diffuse} based on σ. The color of the last bounce is computed via direct diffuse lighting, just like in Phong's local illumination:

$$\text{Color}(\text{LastBounce}) = C_{\text{diffuse}} \sum_{\ell \in \text{Lights}} (\text{visibility to light}_\ell) \frac{L_\ell}{\text{attenuation}_\ell} \max(\langle \mathbf{l}_\ell, \mathbf{n} \rangle, 0). \tag{3}$$

Task 4: Single Ray Bounce (5 pts)

From Equation (2), for $N = 1$, we get pure Phong-like diffuse shading (i.e., a single bounce). Thus, for **Task 4**, you will implement single-bounce ray tracing:

$$\text{Color}_1(\text{RayAtPixel}_{(i,j)}) = \text{Color}(\text{LastBounce}). \tag{4}$$

Effectively, at each ray-geometry intersection, you only need to evaluate Equation (3). To compute it, we must consider the term visibility to light$_\ell$. In ray tracing, checking visibility is straightforward: simply shoot a *shadow ray* from the intersection point to the light. If the closest intersection of this shadow ray is not the light source, the light is blocked.

Rewriting Equation (3) in a more compact form:

$$\text{Color(LastBounce)} = C_{\text{diffuse}} \underbrace{\sum_{\ell \in \text{Lights}} (\text{visibility to light}_\ell) \frac{L_\ell}{\text{attenuation}_\ell} \max(\langle \mathbf{l}_\ell, \mathbf{n} \rangle, 0)}_{L_{\text{direct}}} = C_{\text{diffuse}} L_{\text{direct}}.$$

For this task, you need to calculate L_{direct} at a given intersection. Complete the implementation of:

```
vec3 GlossyMaterial::get_direct_lighting(Intersection &intersection, Scene const
                                         &scene)
```

in `src/materials/GlossyMaterial.cpp`, which computes the contribution of all light sources at the given intersection. Once you implement this, the next step is already done for you inside `GlossyMaterial::color_of_last_bounce(..)`. Your scene render at this stage should resemble Figure 8.

Figure 8 Scene render after implementing Task 4.1

Just with single-bounce diffuse shading, we can already achieve dynamic shadows and lighting. However, it still does not feel very realistic because we are missing soft shadows and global illumination.

Task 5: Soft Shadows (3 pts)

In Figure 8, even though our light source is an area light, the shadow boundaries are unnaturally sharp. In reality, objects illuminated by area lights should have blurry shadow edges (soft shadows). Currently, the function `get_surface_point()` in `Scene::get_direct_lighting()` always returns the *center* of the square light, effectively treating it as a point light. We need to fix this by uniformly sampling a point over the entire area of the square.

Inside the `Square` class, the area light is defined using two orthonormal basis vectors (`tangent` and `bitangent`), and a side length s. Let these basis vectors be $\vec{t}$ and $\vec{b}$, the center be $\underline{c}$, and u, v be two independent random numbers in $\left[-\frac{1}{2}, \frac{1}{2}\right]$. Then a uniformly sampled point $\underline{p}$ on the square is:

$$\underline{p} = \underline{c} + u \cdot s \cdot \vec{t} + v \cdot s \cdot \vec{b}, \qquad u, v \in \left[-\frac{1}{2}, \frac{1}{2}\right]. \tag{5}$$

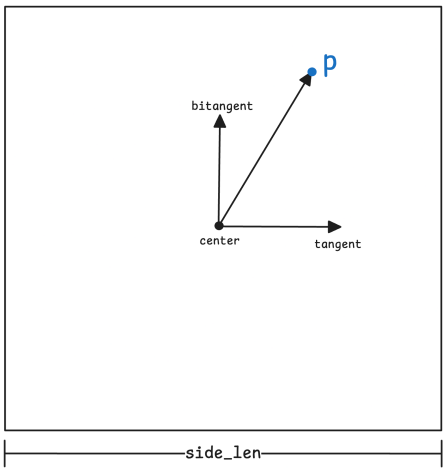


Figure 9 Sampling a point uniformly on a square

Your **Task 5** is to implement Equation (5) in:

```
vec3 Square::get_surface_point()
```

found in `src/models/Square.cpp`. Once done, the scene rendered in ray tracing mode should look like Figure 10, showing softer (though initially noisier) shadows. Increasing *spp* values will reduce the noise, as in Figure 10b.

Remark 1

After implementing sampling for the light surface position, the rendered image became noisy. What do you think is the source of this noise, and why does increasing the *spp* value reduce it?

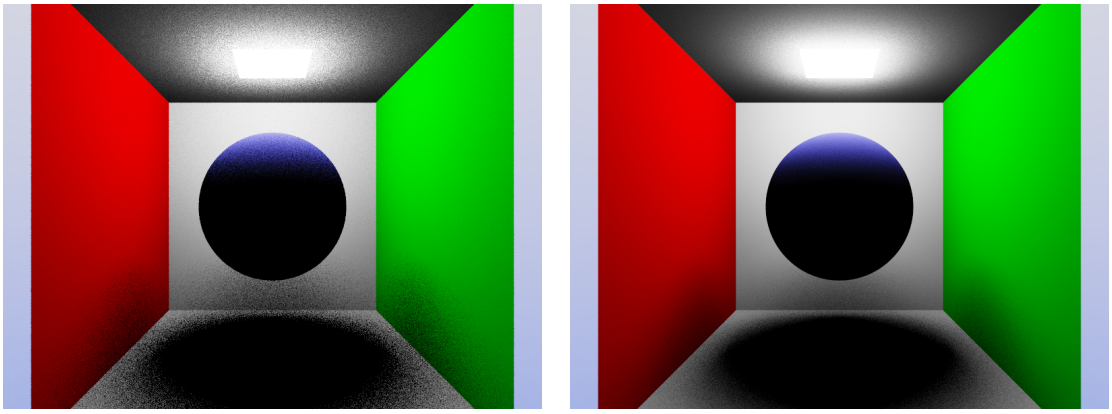
Task 6: Multiple Ray Bounces (9 pts)

Previously, we implemented Equation (2) for $N = 1$. Now, we will generalize it for $N \geq 1$. For N concrete ray bounces, the final color of our pixel is:

$$\text{PixelColor}^{N}_{(i,j)} = \sum_{k=1}^{N} \text{Color}_k(\text{RayAtPixel})_{(i,j)}. \tag{6}$$

Expanding Equation (6),

$$\text{PixelColor}^{N}_{(i,j)} = \text{Color}_1(\text{RayAtPixel})_{(i,j)} + \text{Color}_2(\text{RayAtPixel})_{(i,j)} + \cdots + \text{Color}_N(\text{RayAtPixel})_{(i,j)}.$$



(a) 10 samples per pixel **(b)** 100 samples per pixel

Figure 10 Scene rendering with soft shadows

Further expanding $\text{Color}_k(\text{RayAtPixel})$:

$$\begin{aligned}\text{PixelColor}^N_{(i,j)} = \text{Color}(\text{LastBounce}_1) + \\ W_1\text{Color}(\text{LastBounce}_2) + \\ W_1W_2\text{Color}(\text{LastBounce}_3) + \\ W_1W_2W_3\text{Color}(\text{LastBounce}_4) \ + \ \ldots\end{aligned}$$

If you observe carefully, there's a nice iterative pattern emerging in above expression. We can express it in following pseudo-code:

```
pixel_color =  vec3(0.0)
W_wip = vec3(1.0) // wip: Work In Progress

for i in 1...N:
    pixel_color = pixel_color + W_wip * last_bounce_color[i]
    W_wip = W_wip * W[i]
```

A similar logic is employed by our Ray Tracer. Rather than sequentially handling N bounces for each ray, we handle them in parallel, storing `W_wip` and `color` in each `Ray` instance (see `include/Ray.h`). The `Scene::intersect(..)` function is called for each bounce. At the end of every bounce, we must update `W_wip` and `color`. From the previous task, the function `GlossyMaterial::color_of_last_bounce(..)` already computes the last-bounce color. The remaining part is to choose the next bounce direction and update the radiance (`W_wip`) based on the shading model in Equation (1).

In **Task 6**, complete:

```
Ray GlossyMaterial::sample_ray_and_update_radiance(Ray &ray, Intersection &
                                intersection)
```

in `src/materials/GlossyMaterial.cpp`. The material model can behave either diffusely or like a mirror, according to a probability σ. The material's diffuse and specular parameters (`C_diffuse`, `C_specular`) were set during scene creation. We split this into two subtasks:

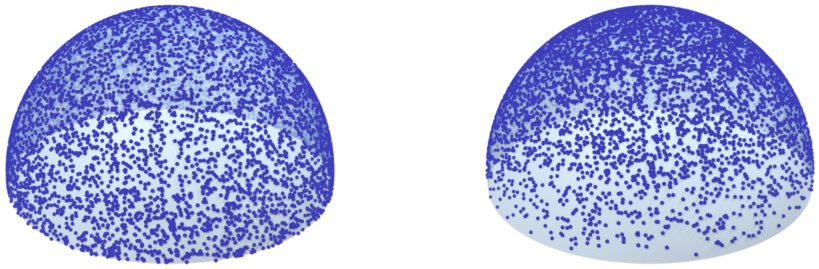


Figure 11 Left: Uniform hemisphere. Right: Cosine-weighted hemisphere.

Task 6.1: Diffuse Scattering (6 pts)

In diffuse scattering, we choose a random direction over the hemisphere above the surface.

- A straightforward approach is to sample uniformly in the hemisphere.
- A more efficient approach is to perform *cosine-weighted* sampling, which biases samples toward the normal direction (Figure 11).

If s, t are two uniform random numbers in $[0, 1]$, then a common way to generate a hemisphere direction is:

$$u = 2\pi s, \quad v = \sqrt{1 - t^2}, \quad \mathbf{d} = \begin{bmatrix} v\cos(u) \\ t \\ v\sin(u) \end{bmatrix}. \tag{7}$$

A cosine weighted hemisphere sampling of $\mathbf{d}$ is given by

$$u = 2\pi s, \quad v = \sqrt{1 - t}, \quad \mathbf{d} = \begin{bmatrix} v\cos(u) \\ \sqrt{t} \\ v\sin(u) \end{bmatrix}. \tag{8}$$

Steps for this subtask:

- Implement the new ray direction $\mathbf{d}$ using Equation (8).
- Transform $\mathbf{d}$ to the local coordinate system so that $\mathbf{d}$ is oriented above the surface normal.
- Update the ray's `W_wip` by multiplying it with $W_{\text{diffuse}} = C_{\text{diffuse}} \max(n \cdot d, 0)$.

In **Task 6.1** you need to complete implementation of diffuse reflection part for function

```
Ray GlossyMaterial::sample_ray_and_update_radiance(Ray &ray, Intersection &
                                intersection)
```

in `src/materials/GlossyMaterial.cpp`. Increase the `max_ray_bounces` parameter in the command line to see indirect lighting effects.

```
../bin/RayTracer 10 3 1
```

The scene render should look like Figure 12 after implementing diffuse light scattering. You can now notice effects of color bleeding due to secondary ray bounces. The image look more realistic than just Phong shading.

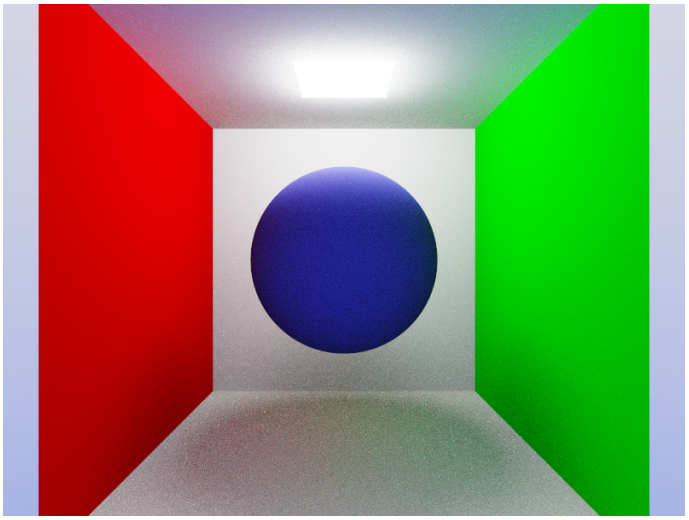


Figure 12 Scene rendering after task 6.1 with 3 bounces and 100 spp

Task 6.2: Specular Reflections (3 pts)

Our rendering is already looking realistic, it's able to model complex phenomenon like, shadows, soft-shadows and color bleeding due to indirect illumination. But till this point all materials are purely diffuse, for **Task 6.2** you'll be implementing specular part of our glossy material shading model. For specular reflections:

- The reflection direction is $\mathbf{r} = 2(\mathbf{n} \cdot \mathbf{v})\mathbf{n} - \mathbf{v}$, where $\mathbf{v}$ is the incoming direction (from the intersection point back to the camera), and $\mathbf{n}$ is the normal.
- Update the ray's `W_wip` by multiplying it with $W_{\text{specular}} = C_{\text{specular}}$.
- Remember that σ controls the probability of reflecting vs. diffusing.

In **Task 6.2** you need to complete the remaining implementation for specular reflection part of function

```
Ray GlossyMaterial::sample_ray_and_update_radiance(Ray &ray, Intersection &
                                    intersection)
```

in `src/materials/GlossyMaterial.cpp`. By this point we have a complete implementation of a basic ray tracer and the rendered scene will look like Figure 17

Remark 2

In the current scene, the sphere inside the box is primarily diffuse with a small specular component. You can refer to the exact material property values in `src/scenes/sphere_in_box.inl`. Experimenting with the shininess, as well as the specular and diffuse properties of the material, can yield interesting visual results.

An important observation is that making objects near the light source highly specular often produces numerous "fireflies" (random bright pixels), which remain even when using higher samples per pixel (Figure 14). Why does this happen, and how can it be mitigated or minimized?

Task 7: Do Something New (10 pts)

Up to this point, you have primarily been filling in code for pre-defined implementations. We hope this has helped you gain a better understanding of how the ray tracer is implemented and how its code is structured.

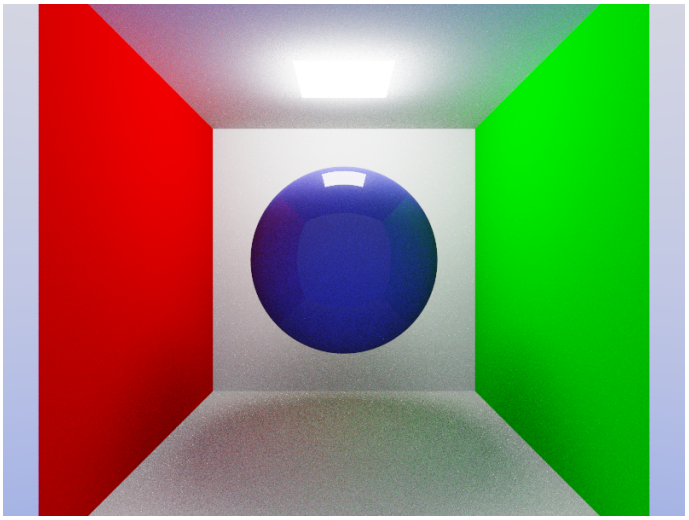


Figure 13 Scene rendering after task 6.2 with 3 bounces and 100 spp

For this task, you will build upon the existing implementation to introduce a new feature or create an interesting scene. Here are a few potential ideas:

- Make the Glossy Material shading model more realistic by accounting for the spread of specular highlights.
- Implement the Russian Roulette algorithm to probabilistically terminate rays, rather than using a fixed number of bounces.
- Implement the fix for specular fireflies discussed in Remark 2.

Along with the answers to the questions provided above, you are required to write a report documenting your approaches and results for Task 7. This is an open-ended task, and you will be graded primarily on your effort rather than on completion alone. Do not hesitate to attempt something challenging, even if it does not fully succeed or you cannot complete it. You will still earn credit based on your efforts.

Design a Custom Scene

You can design a custom scene using either Models or simple `.obj` files. Similar to HW3's scene graph, our ray tracer also uses a scene tree. Since each model in the ray tracer needs have a distinct geometry objects, they are organized in this tree structure. A few example scene implementations are provided in the `src/scenes/` folder, and you can select among them using the `scene_id` argument:

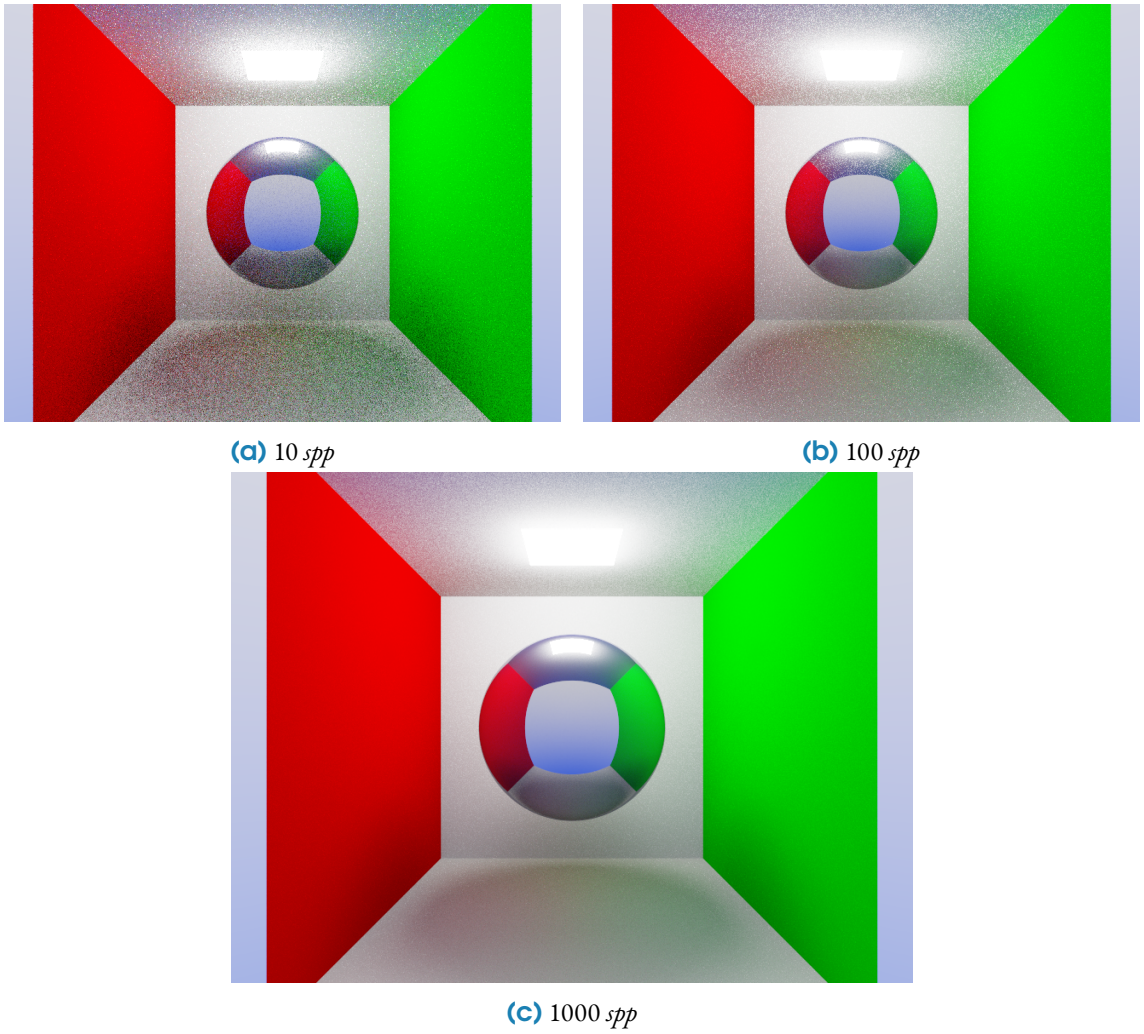
```
../bin/RayTracer 10 3 1
```

To see which scene corresponds to a particular ID, refer to the implementation of `RayTracer::init(..)` in `src/RayTracer.cpp`.

Each node in the scene tree is defined as follows:

```
struct Node {
    std::vector<std::unique_ptr<Node>> childnodes;
    std::vector<glm::mat4> childtransforms;
    std::unique_ptr<ModelBase> model;  // Only leaf node would have this set
};
```

Note that `childnodes` and `childtransforms` have a one-to-one correspondence: for an index `idx`, `childtransforms[idx]` is for `childnodes[idx]`. You can use GLM's 3D transformations,



(a) 10 *spp* **(b)** 100 *spp*

(c) 1000 *spp*

Figure 14 Rendered scene featuring a mirror-like sphere, with varying *spp* values and 3 ray bounces.

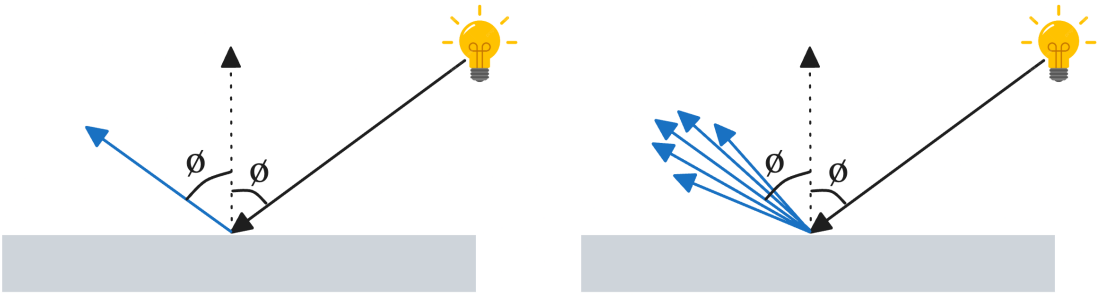
chaining them as needed to achieve the desired model placements in the scene. Refer to the provided examples for sample implementations.

For starters, you can try to implement the scene from HW3. However, remember that if you add complex models with a large number of geometric primitives (such as the Stanford Bunny and the Utah Teapot), the rendering time will increase significantly, since our ray tracer does not include any acceleration structures. In fact, implementing an acceleration structure like Uniform Grids, K-d Trees or Bounding Volume Hierarchy (BVH) would be an excellent Task 7.

Create New Material

The `GlossyMaterial` class implements the shading model discussed in Equation (1). You can create a new material class to implement a custom shading model. For example, you might modify the existing `GlossyMaterial` class by making the specular component combination of mirror-like and diffuse (Figure 15), thereby more accurately modeling real-world glossy surfaces. Another intriguing approach is to implement a shading model for dielectric materials, such as glass, which requires handling refraction using Snell's law and managing total internal reflec-

tion.



(a) Pure reflection (b) Diffuse in reflection direction

Figure 15 More realistic glossy reflections

Russian Roulette

Instead of truncating the path by setting an artificial choice of maximal recursion depth, the Russian Roulette method is a practical method that sums all the terms in the infinite series (6).

First of all, introduce a parameter $0 < \lambda < 1$. Now modify the expression tautologically at first:

$$\begin{aligned}\text{PixelColor}_{ij} = &\frac{(1-\lambda)\,\lambda}{(1-\lambda)\,\lambda}\,\text{PixelColor}^{1}_{(i,j)} \\ &+ \frac{(1-\lambda)^2\,\lambda}{(1-\lambda)^2\,\lambda}\,\text{PixelColor}^{2}_{(i,j)} \\ &+ \frac{(1-\lambda)^3\,\lambda}{(1-\lambda)^3\,\lambda}\,\text{PixelColor}^{3}_{(i,j)} \\ &+ \cdots \end{aligned} \tag{9}$$

Then, instead of picking a fixed path length n and sampling paths with that length, we will let the ray bounce indefinitely. But, at every bounce, we toss a coin with probability λ to decide whether we want to terminate the path tracing. In that way, we have a probability $(1-\lambda)^{n-1}\lambda$ to obtain a path of n bounces (to survive $(n-1)$ bounces and to be killed at the n-th bounce). By looking at each term of (9), we learn that we should re-weight the resulting color from an n-bounce path by a factor of $\frac{1}{(1-\lambda)^{n-1}\lambda}$. By doing so, the expectation value of the resulting color will unfold into the formula (9), which is the same as what we want in (6).

For the implementation, you need to update the `Scene::intersect(..)` function to scale the ray's color contribution and terminate the ray probabilistically.

Appendix

Code Structure

The `RayTracer` class is where everything begins. When you press Space to start ray tracing, the `RayTracer::draw()` function is called. This function, in turn, calls `Scene::intersect(..)`, which implements ray intersection and shading.

Inside `Scene::intersect(..)`, the given ray is tested for intersections with all models by calling `ModelBase::intersect(..)`. The model class then transforms the ray into the model's coordinate frame and calls `GeometryBase::intersect(..)` on its geometries to find any intersections. Polymorphism is used here to correctly invoke the intersection functions for different geometry types. All resulting intersections are stored in the `Ray.intersections` vector.

Based on these intersections, the closest intersecting model is determined, and the ray's radiance is updated according to that model's material properties. A new ray is cast and added to a processing queue, which supports parallel processing of multiple rays.

Below is a minimal illustration of the major classes, along with their key functions and attributes. Keep in mind that there is a considerable amount of additional code around this logic for handling parallel ray processing, object movement, taking screenshots, and displaying the rendered image using OpenGL.

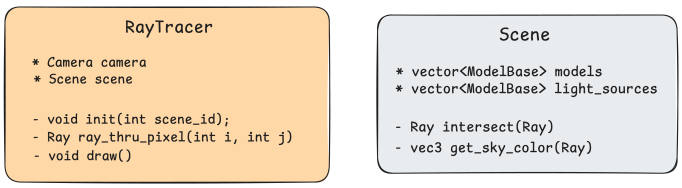


Figure 16 Ray and Scene class

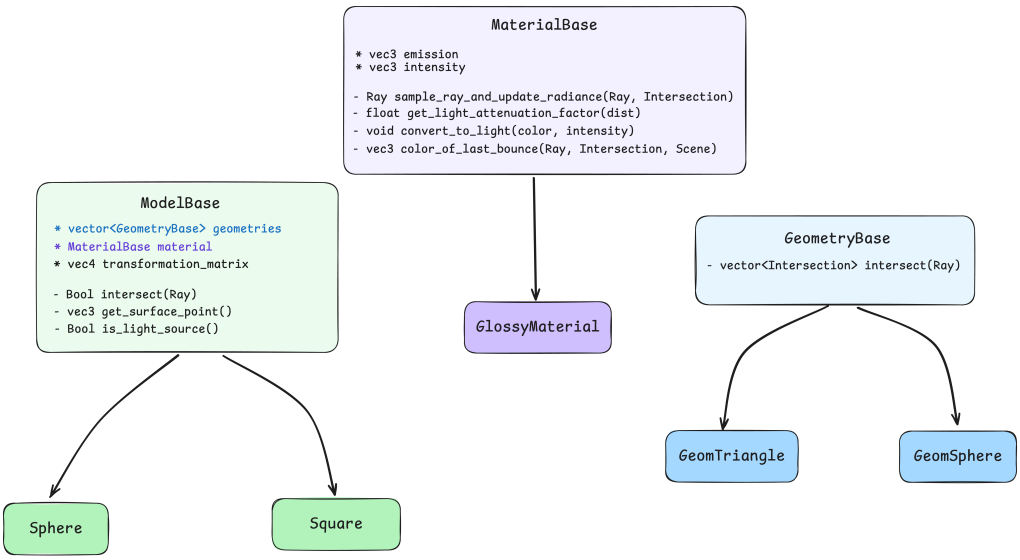


Figure 17 Class inheritance hierarchy

Debugging

While working on these tasks, you will encounter many bugs. Debugging a ray tracer is particularly challenging because simply adding print statements or using breakpoints can be cumbersome, given the sheer number of rays that need to be traced. For instance, for an 800×600 pixel image resolution, even shooting only 10 rays per pixel results in 4.8 million rays in parallel!

A more effective strategy is to encode the information you want to debug directly into the rendered image. We achieve this by embedding debug information into the pixel color. Normal shading is a good example: by looking at the image, we can infer the direction of normals from the pixel colors.

Our ray tracer implementation provides three shading modes: normal shading, ray tracing, and debug mode. As shown in the `Ray` struct (in `include/Ray.h`), each ray has both a `color` and a `debug_color` attribute. When the shading mode is set to either normal or ray tracing, `ray.color` is displayed, whereas in debug mode, `ray.debug_color` is used. You can toggle debug mode by pressing `P`. Note that even in debug mode, the full ray tracer is still running—only now you can store and visualize your debugging information in `ray.debug_color`.

For example, Figure 18a shows how you can visualize a ray's direction by assigning:

```
ray.debug_color = 0.5f * ray.dir + 0.5f
```

in the `RayTracer::ray_thru_pixel(..)` function. Meanwhile, Figure 18b demonstrates how to display the diffuse color contributed by the ray's second bounce. You can achieve this by conditionally assigning `debug_color` as:

```
if (ray.n_bounces == 1) {
    ray.debug_color = W_diffuse;
}
```

in the diffuse reflection section of `GlossyMaterial::sample_ray_and_update_radiance(..)`.

(a) Ray direction visualized (b) Color contribution of diffuse 2nd ray bounce

Figure 18 Example usage of debug shading mode

Movements

Our ray tracer dynamically calculates lighting, shadows, and reflections, allowing you to move around the scene and still render it from different perspectives. In normal shading mode, you can explore the scene and view models from various angles. However, this method is slower than the GPU-based rasterization approach because our ray tracer runs entirely on the CPU.

The controls are similar to those in an FPS game. Use W, S, A, D to move forward, backward, left, and right; Q and E to move up and down; the arrow keys to rotate around the vertical and horizontal axes; and Z and X to rotate around the view axis. You can also use + and - to zoom in and out. For the exact implementation, refer to `src/Camera.cpp`.