Coms 336 - Raytracer Report

By: Joseph Schmidt

Intro:

This report summarizes the raytracer I developed for Coms 336. The project was both challenging and rewarding, offering a valuable learning experience. I implemented things such as triangle meshes, different materials, lights, and textures. I hope to be able to add more to it in the future.

Body:

The first thing I did was work on being able to render a red sphere which appeared to be a red circle on my camera from a lack of ray bouncing. Initially, the rendered sphere appeared pixelated because the raytracer sent only one ray per pixel. I solved this by implementing anti-aliasing**.** This is a technique where you randomly choose locations in the area of the current pixel and shoot multiple rays through that pixel averaging the color to get a more fluid picture. Moving on, I added triangles to my scene. To calculate whether a ray intersects a sphere, the ray equation is substituted into the sphere's equation. Solving this quadratic equation yields 0, 1, or 2 roots. If you have 0 roots you missed your sphere. Having 1 means you grazed the sphere. And lastly, 2 roots means that your ray has passed through the sphere. In the last case, you can take the root that is closer to the camera as unless you are glass you should only see the one side of the sphere. Triangles were calculated in a different way using a technique called barycentric coordinates. This involved creating three sub-triangles by connecting the intersection point (where the ray meets the triangle's plane) to each pair of the triangle's vertices. If any of these sub-triangles have a negative area, it indicates that the intersection point lies outside the triangle.

Next, I moved onto implementing normals. Normals are vectors that point outward from the surface of an object, perpendicular to its surface at the point of intersection. For example, any point on a sphere has its normal facing from the center of the sphere into that point going outwards. This allows you to start calculating how rays should interact with the objects. This moves into the next section which is materials.

Diffuse materials produce a matte-like finish by scattering light uniformly in all directions. This was achieved by taking the normal at the point of intersection and forming a unit sphere around it. A random vector was then selected from this unit sphere to determine the scattered ray's direction. For a material like metal if you want a perfect reflection you can just take the normal as the new vector and add the color of the metal. If you want to simulate a slightly smudged or imperfect metal surface, you can introduce a fuzziness factor. This adds small random deviations to the reflected vector, making the reflections appear less sharp. For glass, light interaction involves a combination of refraction and reflection. This was calculated by randomly choosing if a ray should choose to reflect or go through the object.

In my raytracer, the initial implementation checked every object in the scene to determine if a ray intersected with it. This brute-force approach was computationally expensive, especially for scenes with many objects. To optimize this, I implemented bounding volume hierarchies (BVH). A BVH works by dividing the scene into a hierarchical structure of bounding boxes, each encompassing a subset of objects. Rays are first tested against these bounding boxes, quickly eliminating large groups of objects that cannot be intersected. If a ray intersects a bounding box, only the objects inside that box are tested for intersection. This process significantly reduces the number of objects that need detailed checks, as each step narrows down the potential candidates. By organizing the scene into a tree-like structure, the algorithm reduces the complexity of object intersection checks from linear to logarithmic, making the raytracer much more efficient for complex scenes.

I also configured a movable camera, allowing adjustments to both its position and the point it is directed at. This required only minor changes to the math, as the rays were calculated relative to the new starting position.

Next, was loading triangle meshes into the raytracer. I did this via a third party library called: tiny_obj_loader.h. All I had to do was pass it in a pointer to a vector for each of the following: vertices, uv_coords, indices, and normals. With these I was able to iterate through each of these vectors and create triangles from each of these points.

From there I moved onto textures. This involved having two coordinates a u and a v. These dictate how the texture is mapped onto the 3D object. I used another third party library in order to load the images and make the mapping process easier. This library was stb_image.h.

The last required component for the raytracer was lights. Lights were implemented to enhance the brightness of rays that intersect with them, adding intensity rather than subtracting values. The issue with this is if you have a small light a lot of your rays bouncing around the scene are not going to hit this light causing a lot of noise in your scene. This is a difficult problem that requires additional logic to solve.

Additional Features:

HDR

The third party library that I mentioned earlier for importing lights made it very easy to import hdr images into the project. I basically had to change my data variable for my color variable from just a char to a float. This preserved standard image values while allowing HDR images to store additional data for enhanced visual effects, such as varying exposure levels.

Volume Rendering

This feature enabled my raytracer to simulate smoke or cloud-like objects. As a ray traveled through the volume, its probability of scattering increased with distance, causing it to change direction. Adjusting the scattering probability allowed for control over the density and appearance of the smoke or cloud.

Quads

Quads are a parallelogram object in space. It is made by having a corner point Q that is a reference for all of the other corners. It then has two vectors that are used to dictate the next two points. These are a u and a v vector. So point two is Q + u, and point three is Q + v. Lastly, point four is Q + u + v giving you a quad. You can then find ray intersections with it by solving for the plane equation plugging in your ray values.

Motion Blur

This was achieved by assigning a time variable to each ray, indicating when it was sampled. When a ray checks for intersections, it accounts for the object's position at that specific time. This allows objects to have motion, with their positions varying over time based on a defined speed. As a result, different rays can interact with the same object at different points along its path, simulating motion blur and dynamic movement in the scene.

Defocus / depth of field

This worked by having a focus plane that was perpendicular to the camera’s view direction and had a focus distance from the camera. Any object at the focus distance that appears perfectly in focus. I then had a defocus disk. This basically meant instead of all rays starting from a single point in my camera they were shot from random points in a circular aperture. The larger the aperture the more blur that was introduced. This is because multiple rays originating from different points are hitting them at different locations.

Object Instancing

This involved being able to create an object and move and rotate the object in space. This was done through linear transformations. You can add a vector onto all of the current vector positions practically moving it all by that vector position. Rotations are a bit more complicated but employ a similar method of applying a function to all points of an object. This time using a trig function to get this rotational effect.

Perlin Noise

This is a kind of noise that differs from white noise in the fact that it creates natural smooth looking textures and patterns. It basically creates smooth transitions between pixels through gradients and interpolating points making noise look smooth.

Cube Maps

This is basically taking a collection of 6 images and creating a background from it. This was done by creating a cube map class that took in six images and created six separate images that connected seamlessly to create the background.

Importance Sampling

When we have a small light source in an image, noise occurs because many rays miss the light, contributing little to the final result. Importance sampling addresses this by focusing more rays in directions that are likely to contribute significantly to the lighting (e.g., toward the light source). This reduces noise and improves convergence by prioritizing sampling in important areas while de-emphasizing less significant regions.

Parallelization

Raytracing is a highly parallelizable computation. Each ray is processed independently, allowing for efficient parallelization. The way the ray tracing computation is brought out is by having three loops that it iterates through. A width, height, and the anti aliasing amount. I chose to split up the computation by dividing up the rows amongst the chosen amount of threads. I used the standard C++ threads library to implement the threads. I stored the written data in a framebuffer that once all threads are done grabs the data in order from each of the framebuffers and combines them into a final render.

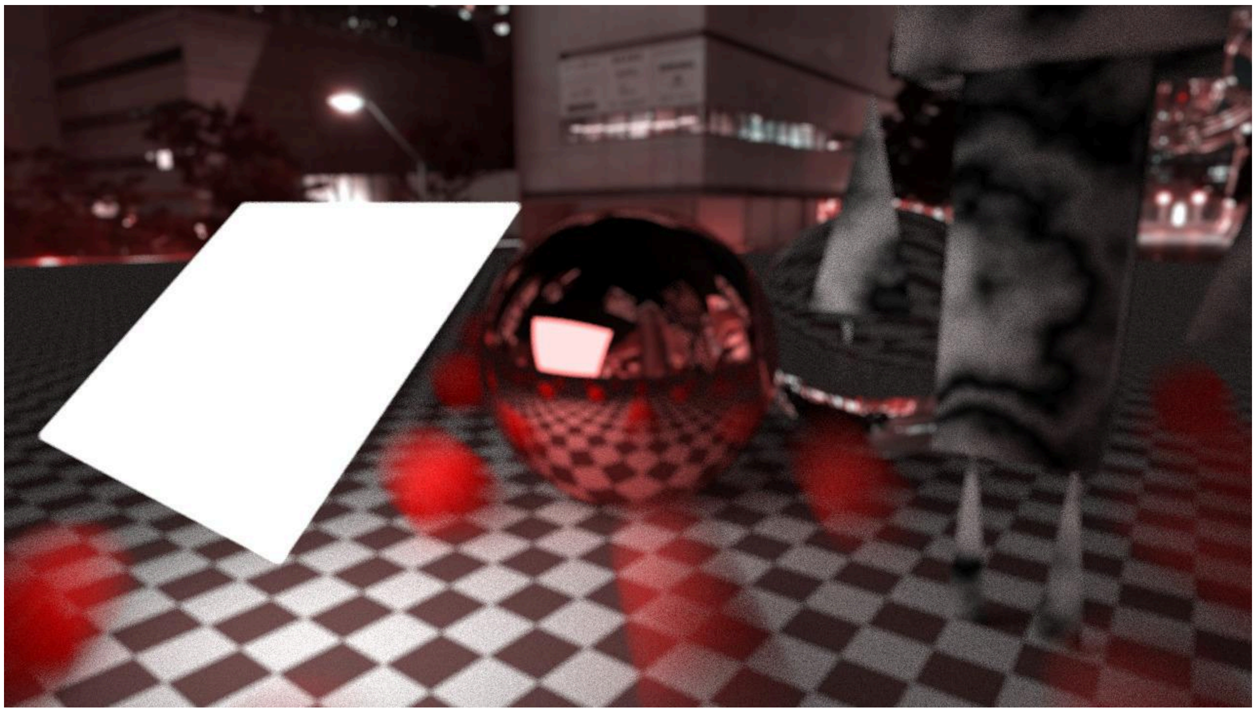
Normal Interpolation

This is a method where instead of having one normal for a triangle you have three normals for each of the points of the triangle. You then interpolate between the triangles. This allows you to get curved like surfaces instead of a flat triangle. It is helpful for triangle meshes because it helps the model appear not to be so polygony.

Conclusion:

Overall, his project was highly educational and enjoyable. This project significantly improved my C++ programming skills and deepened my understanding of computer graphics concepts. I had never done any C++ prior to this and I loved working with it. I learned so many graphics terms that are used in modern day graphics. As a gamer I appreciate being able to understand these terms now. I also have a newfound appreciation for graphic pipelines and how they work. This is one of the coolest projects I was able to work on and I hope to be able to contribute to it further.

Results:



This picture includes the following features: quads, triangle meshes, motion blur, defocus, texturing, metal material, glass materials, cube maps, hdr, anti-aliasing, configurable camera, perlin noise, and lights. For processing it employed parallelization and bounding boxes to increase the render time.