CSE306 Report for project 1

Olavi Äikäs

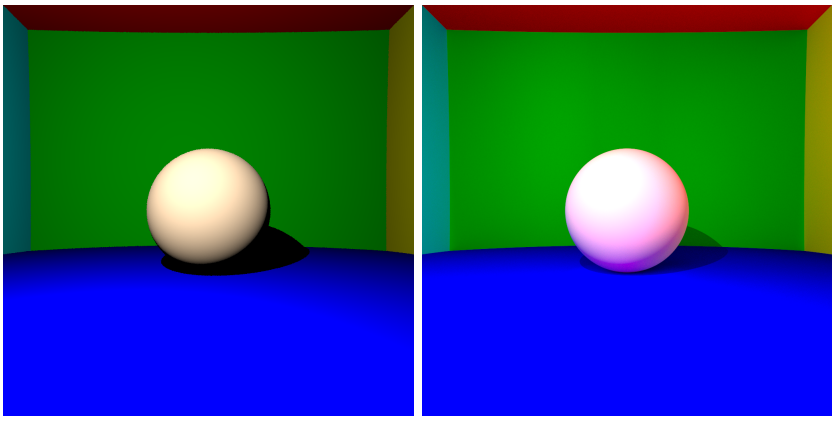
May 2021

The code discussed in this report is found at "https://github.com/OlaviAikas/CSE306Raytracing".

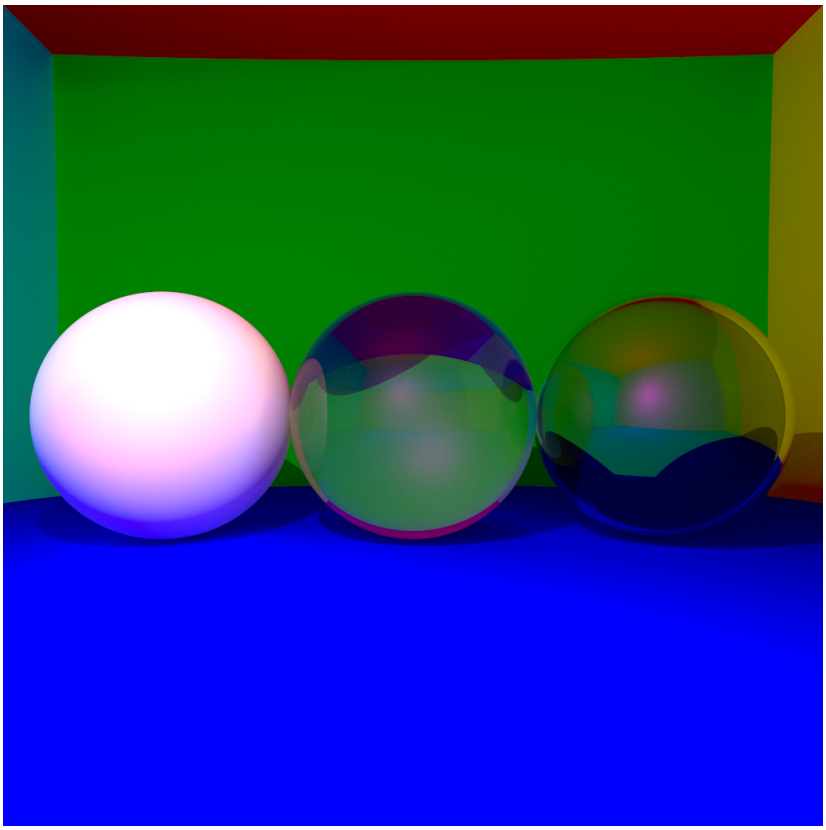
The program uses the convention from the course notes that the camera is centered at (0,0,55) and looking in the $-z$ direction. I did not implement any way to move the camera. The program implements the following path-tracing features:

- gamma correction
- balls and meshes with diffuse surfaces
- balls with perfectly mirroring surfaces and refractive transparent balls
- Fresnel reflection for transparent balls
- indirect lighting for point light sources
- anti-aliasing
- ray-mesh intersection, with a broken implementation of BVH

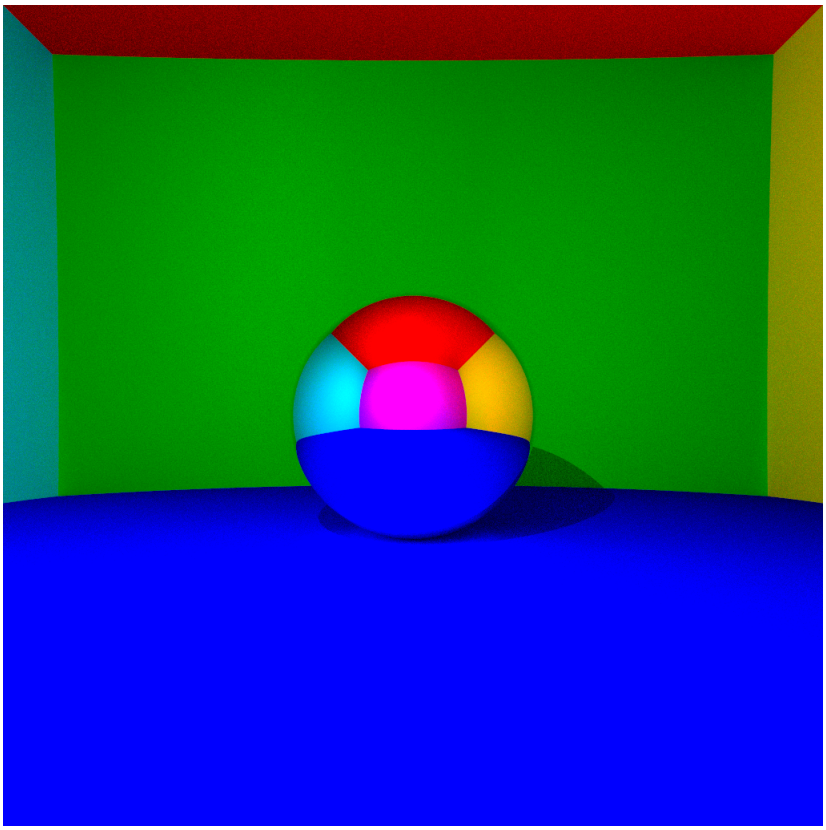
Below I will illustrate these features with images. In all of the images, the scene has a point light source at $(-10, 20, 40)$ with an intensity of $2 * 10^{10}$. The pictures objects are enclosed in a "room" composed of 6 massive balls with radii 940 each, located 1000 units away from the origin, except for the blue ball which is lifted up by 60 such that it acts like a floor at the origin. Finally all the pictures have a $1024 * 1024$ pixel resolution, and the scene is pictured with a field of view of 1.1 radians. The pictures are scaled at a factor of 0.3 in order to fit the page, except for the first two with a scale of 1.5 to fit them side-by-side.



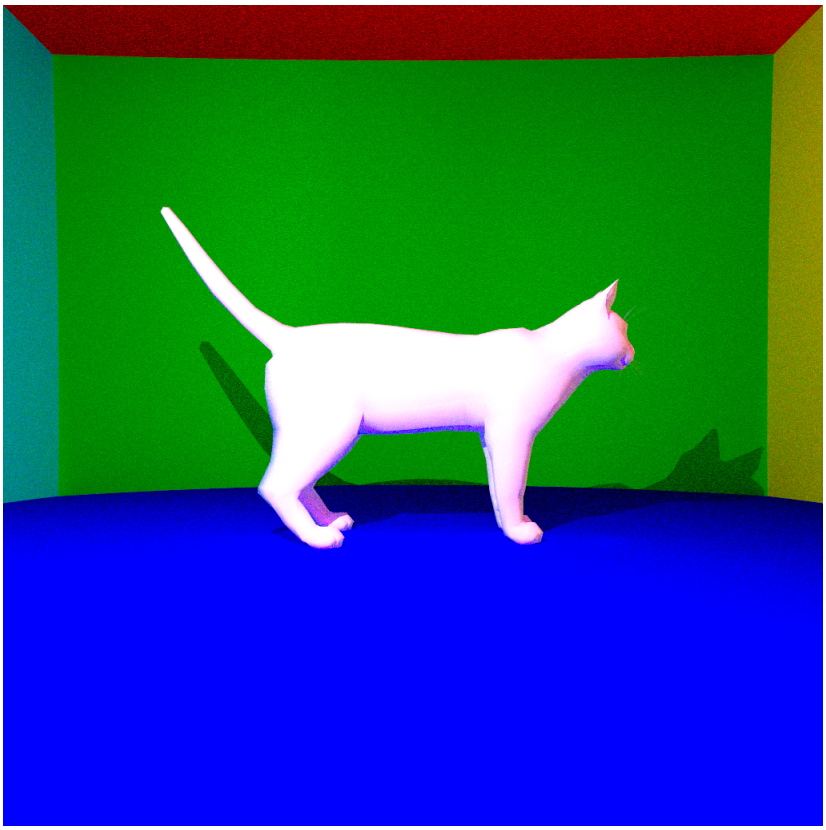
These pictures show a basic scene with a light brown ball. On the left we have the scene without indirect lighting. On the right we have the same scene, but with indirect lighting. We can see that the brown ball looks almost white, because of all of the indirect lighting. We can see that the colour of the indirect lighting changes from a red/orange hue on the right to a purple one at the bottom. This is a great example of how the random sampling for vectors favours directions that are perpendicular to the surface, and thus the blue floor is more pronounced on the bottom and the yellow/red wall and ceiling on the right. Since there are only diffuse surfaces on the left hand image, the whole image is rendered at 1 ray per pixel, and takes negligible time to render. On the other hand, the picture on the right was rendered with 1000 rays per pixel for the smooth indirect lighting, and took 22 minutes in parallel with 8 threads.



This picture demonstrates refraction with Fresnel reflections. The middle ball is solid glass with a refractive index of 1.5, whereas the right-most ball is a combination of two concentric balls, the inner with a radius of roughly 9.5 and a refractive index of 1, and the outer with radius 10 and refractive index 1.5. The picture is rendered at 1000 passes, and took around 30-40 minutes to render, multi-threaded with 8 cores. The amount of light bounces per ray was 8 for this scene.



This picture shows a ball with the perfect mirroring effect. The reasons the colours look so saturated is because of the mirror combining with the indirect lighting. This image was rendered with 100 rays per pixel and took 2 minutes.



Here we see ray-mesh intersection with the cat mesh from the course notes, scaled by a factor of 0.6 and translated by (0, -10, 0) to bring it down the the blue floor. I implemented Phong-interpolation to get the smooth surface from artist-defined normals. This was relatively easy to implement, having computed the barycentric coordinates while checking the intersection. This image is rendered with 32 rays per pixel, and took between 4 and 5 hours to render, although that was with a naive intersection algorithm that checked each triangle for each ray. I later managed to cut that down to around 20 minutes using the most simple bounding box technique where the whole mesh is contained in a big bounding box.

I had trouble implementing the rest of BVH though. After lots of work I never managed to get the quicksort-esque method of balancing the tree of bounding boxes to work, and ended up with a very unbalanced tree (2000+ triangles in a leaf). I later resorted to using std::sort with a sorting function that checks the coordinate along the major axis of the diagonal, and with this I can compute a big tree of bounding boxes. I however suspect that there is also some bug that makes the intersection algorithm visit more leaves than it needs to, because having a tree with many leaves also slows the intersection checking down by a lot. I found that having roughly 500 triangles/leaf produced the best results with that method, being able to render the cat scene in 170 seconds,

with 1 ray per pixel. That however is still quite far from the 80 seconds it took to render with 1 ray with the method that simply put the whole mesh into a simple bounding box. At this point the program is so convoluted with all of the BVH code that having a tree with just one bounding box now takes more than 300 seconds. Therefore after all of my efforts the fastest my code ever got was with the most naive bounding-box checking.