Klaara Huima

Ray Tracer

For this project, I have implemented a ray tracer that visualises a scene of spheres and triangle meshes. The ray tracer supports the following features:

- diffuse, mirror, and refraction surfaces (without Fresnel's law)
- direct and indirect light from point light sources with shadows
- antialiasing
- ray-triangle mesh interaction
- Phong interpolation
- a simple bounding box
- BVH acceleration

The project is run on my personal computer (2 years old, i7 processor with 14 cores). All presented scenes below are of a 512x512 image with light intensity 1E7 - 1E10, field of view 60 degrees, and camera position and scene as described in the textbook.

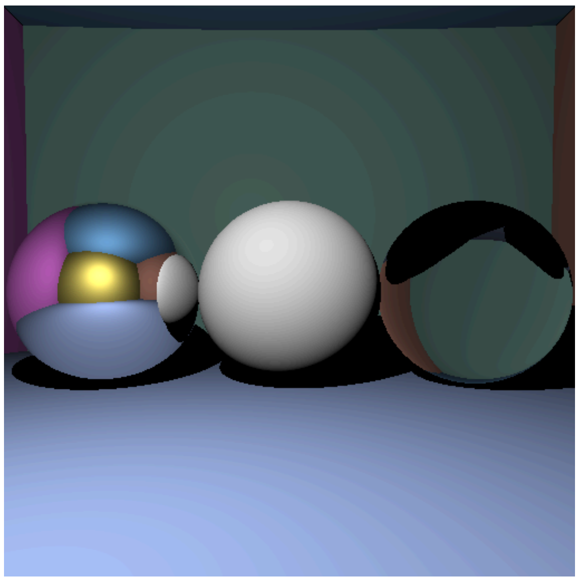
Diffuse, mirror, and refraction surfaces (direct lighting)

The first version of the ray tracer works by sending light rays from a camera with a fixed location and angle towards a scene of spheres and finding the first sphere each ray intersects with. The light intensity and properties of the intersected sphere determine the color of the pixel. Additionally, if there is another object between the found intersection point and the light source, the pixel is in the shadow, in which case the color of the pixel is set to black.

Mirror surfaces are implemented by sending a ray from the intersection point on the surface on the sphere in the direction of reflection, and coloring the pixel based on the color the new, mirrored ray finds.

Refraction surfaces are implemented by changing the direction of the light ray according to Snell's law.

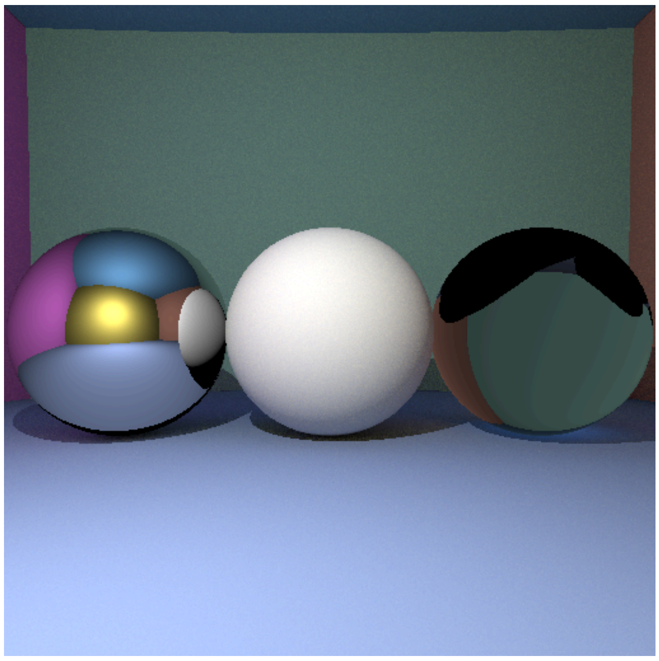
From left to right: mirror surface, diffuse surface, refraction surface (<1 second)



Indirect lighting

We add indirect lighting to the scene by generating light rays in random directions from every initial intersection.

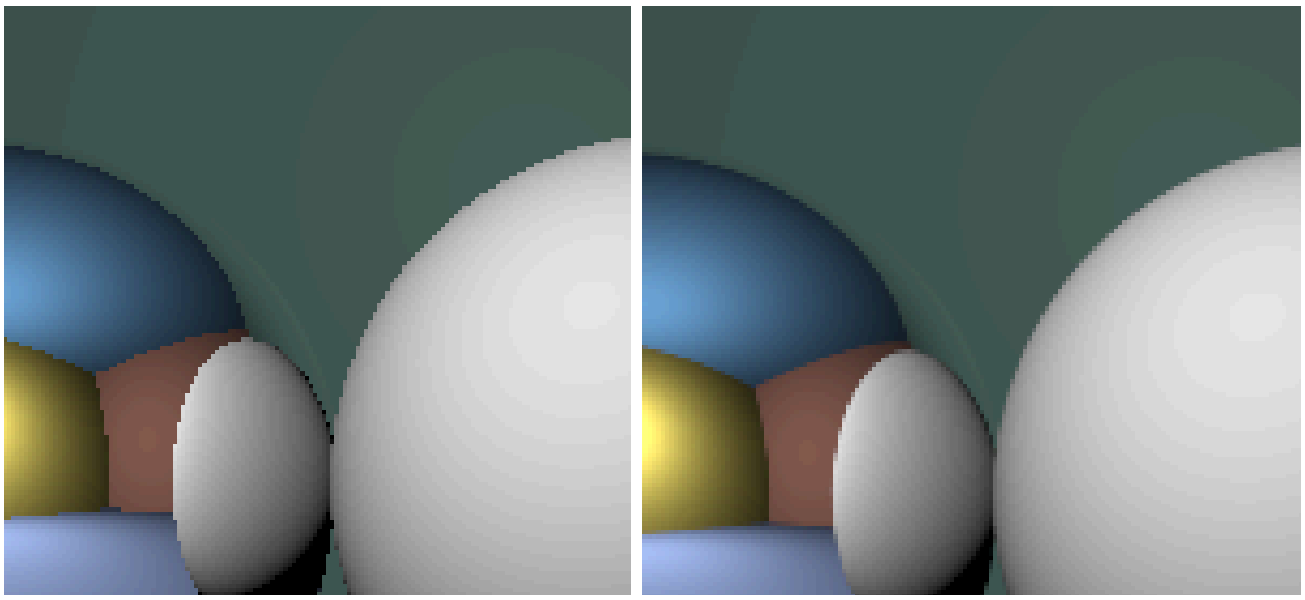
Indirect lighting, 128 rays per pixel, maximum depth 5 (2 seconds with parallelisation)



Antialiasing

To smoothen the surface of the sphere, we implement antialiasing. For every pixel, we send a number of rays with slight perturbations in the direction and take the average of their respective colors. Below, we use the same scene as in direct lighting.

Without (left) and with (right) antialiasing (64 rays per pixel) in direct lighting, maximum depth 5 (2 seconds with parallelisation)



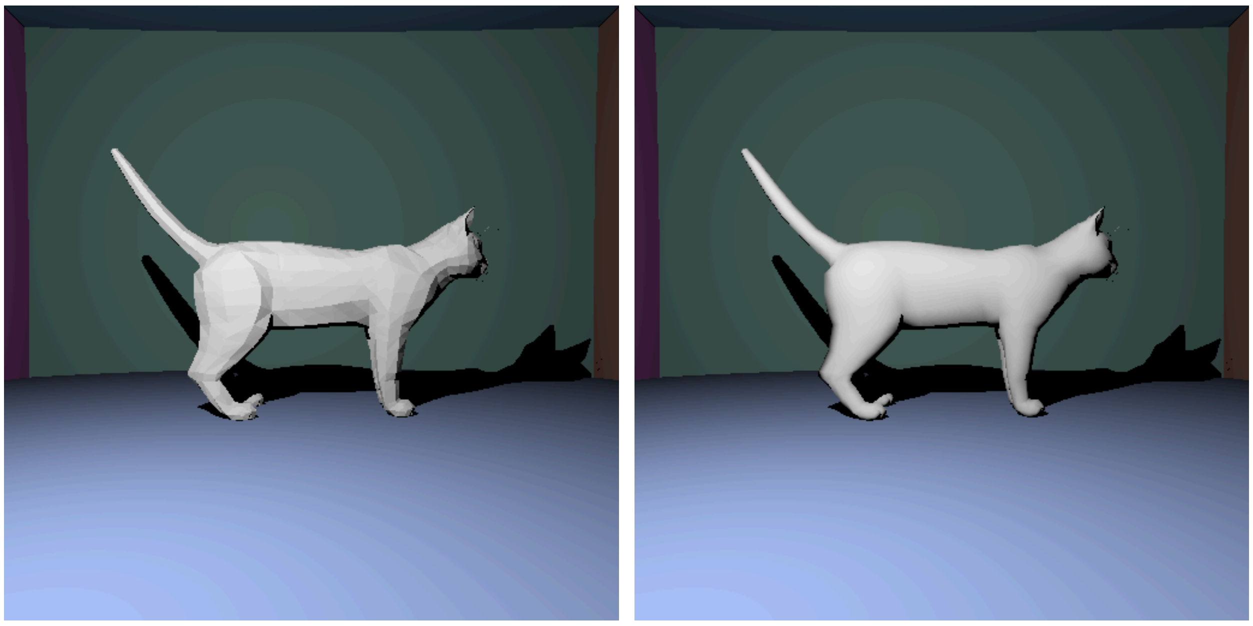
Ray-triangle mesh intersection

We can implement a larger variety of shapes by implementing triangular objects. We build a mesh of thousands of triangles, and similarly compute the intersection between the light rays and the mesh. We import a ready-made mesh from an .obj file for the project. To speed up the computation of the intersections, we implement a simple bounding box. This crops the scene in which the rays look for mesh intersections according to the extrema of the mesh.

Phong interpolation

We can further smooth out the surface of the triangle mesh with Phong interpolation.

Triangle mesh with simple bounding box in direct lighting (left) and with Phong interpolation (right) (1.5 seconds with parallelisation and bounding box)

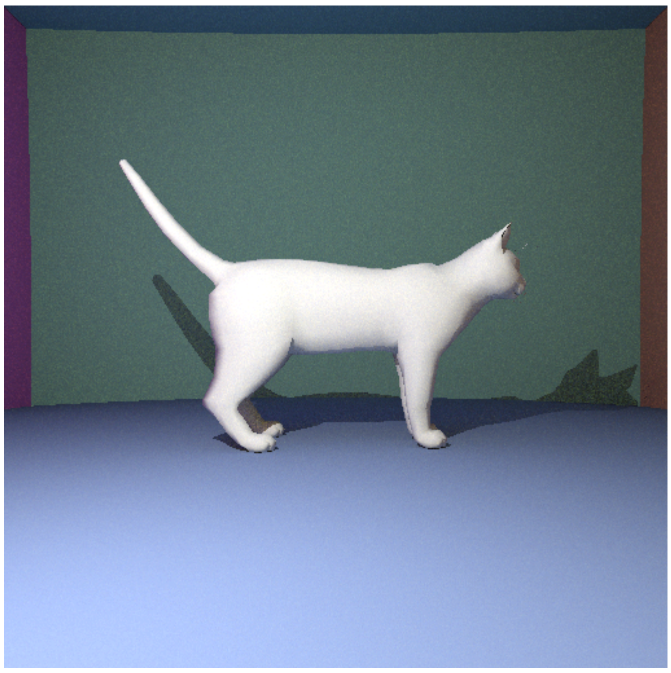


Bounding Volume Hierarchy acceleration

When we incorporate indirect lighting the computations become significantly slower. To speed up the computations, we implement a bounding volume hierarchy algorithm, splitting the mesh into smaller bounding boxes in a tree-like structure. When looking for an intersection, we perform a depth-first traversal, stopping once we find a triangle the ray has intersected with (if any). This largely speeds up the computations.

Final image

Indirect lighting (64 rays per pixel), max depth 5 (3.5 seconds with parallelisation and BVH)



The ray tracer also supports creating transparent triangle meshes by grouping the geometric objects under one parent class.

Transparent and mirror cat, 256 rays per pixel (7.5 seconds with BVH)