Alban PUECH 17/05/2022

Computer Graphics Project 1: Raytracer

This project is the implementation from scratch and in C++ of a ray-tracer.

1.1 Diffuse surfaces and Mirror surfaces (Lab 1 / pages 17--25 of the lecture notes):

To render diffuse surfaces - in our case, spheres - we compute the intersection of the rays with the scene (constituted of multiple spheres) and take into account the contribution from our (unique) light source.

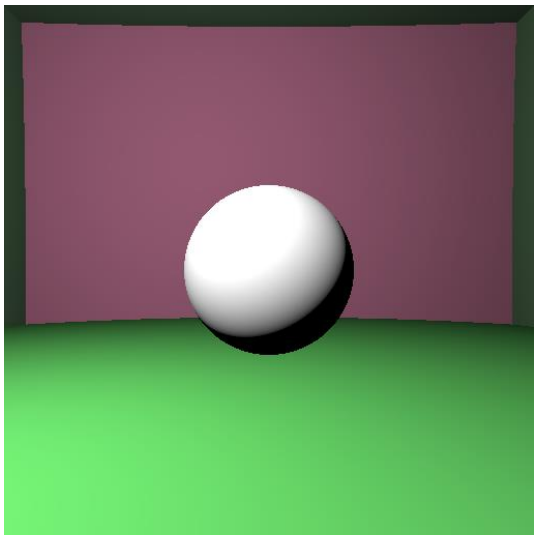

Figure 1: Scene with a white sphere. We use gamma correction. The code runs in 35ms without parallelization for a 512x512 image

1.2 Mirror surfaces (Lab 1 / pages 17--25 of the lecture notes):

To render mirror surfaces, we need to compute light paths to the camera. This is done by computing recursively the amount of light arriving to the camera. Indeed, the light arriving to a camera is no longer the reflection of our point light source but is the result of reflected lights coming from all directions.

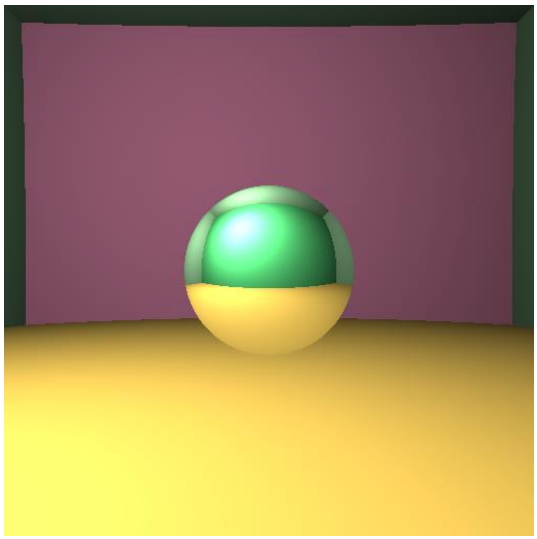

Figure 2: A scene with a sphere with reflection. The code runs in 65ms for a 512x512 image without parallelization

1.3. Shadows (Lab 1 / pages 13-19) :

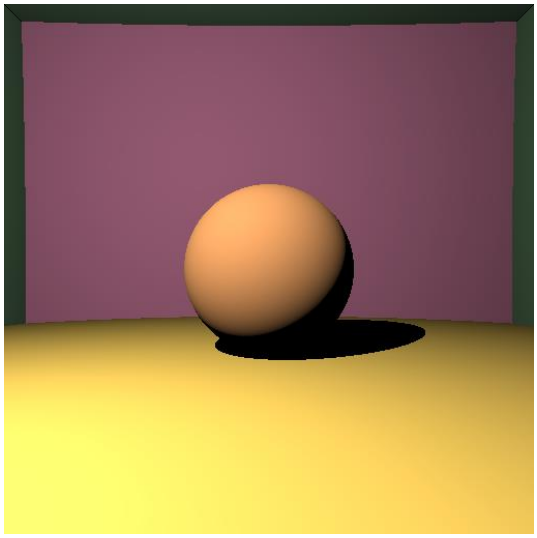

Figure 3: A scene with a sphere and its shadow. The code runs in 55ms for a 512x512 image without parallelization

2.1 Indirect lighting for point light sources (Lab 2 / pages 25--31) without Russian roulette and Antialiasing

To add indirect light to our ray-tracer, we need to perform Monte-Carlo integration to evaluate the rendering equation. To do this, we generate random vectors and use the pdf derived in (2.10). To reduce the noise, we shoot multiple rays for each pixel and average-out the result to obtain its colour.

To correct aliasing (colour discontinuity between adjacent pixel), we use again Monte-Carlo methods and produce samples ray that are often in the middle of each pixel.

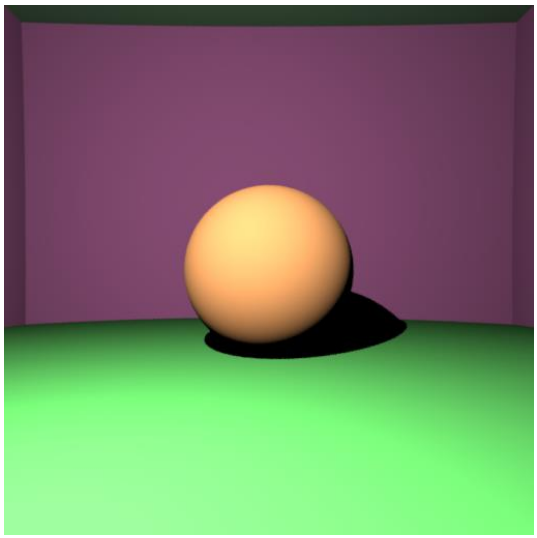

Figure 4: A scene without indirect light but with antialiasing and shadows. The code takes 65ms to run for a 512x512 image without parallelization

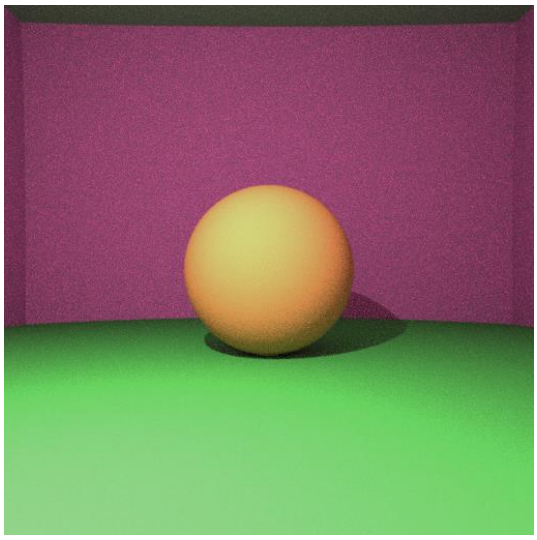

Figure 5 : A scene with a sphere. We added indirect light and used 32 rays. We set a maximum of 3 bounces. The code takes 1.1 seconds to run for a 512x512 image with parallelization

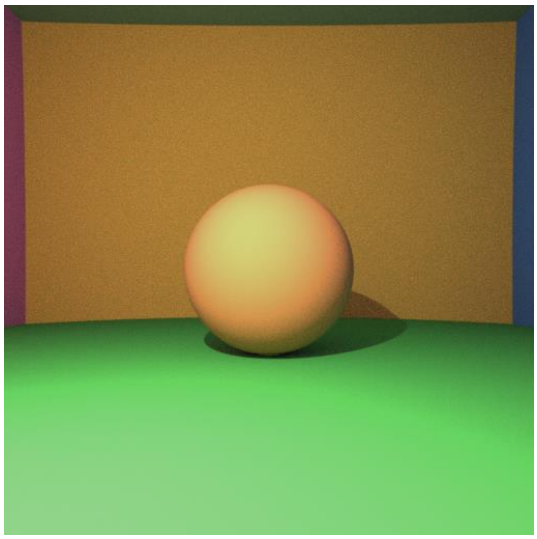

Figure 6 : A scene with a sphere. We added indirect light and used 64 rays. We set a maximum of 3 bounces. The code takes 1.5 seconds to run for a 512x512 image with parallelization

2.2 Refraction:

During lab 2, I also implemented refraction. To do this, we compute the direction of the transmitted ray using the Snell-Descartes law and call the function get_color recursively on this refracted ray.

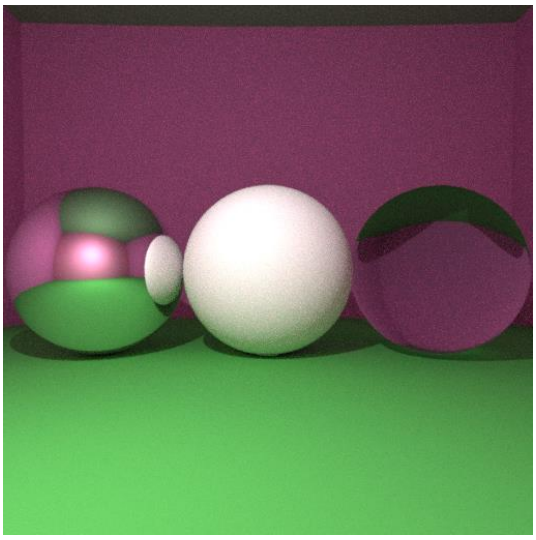

Figure 7: Scene with a white sphere, a sphere with reflection, and a sphere with refraction (from left to right). We used 32 rays and a maximum of 5 bounces. The code runs in 2.2s for a 512x512 image with parallelization

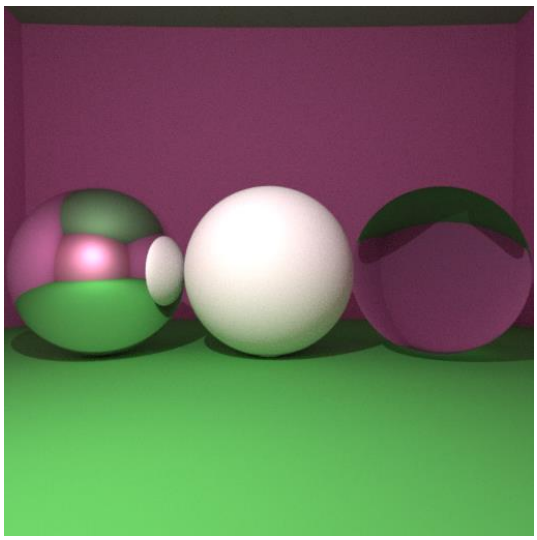

Figure 8: Scene with a white sphere, a sphere with reflection, and a sphere with refraction (from left to right). We used 128 rays and a maximum of 5 bounces. The code runs in 10.1s for a 512x512 image with parallelization

3.0 Ray mesh intersection (Lab 3 / pages 42-44) :

During lab 3, we implemented the support of triangle meshes. Triangle meshes are sets of triangles that form an object. As we did previously to compute the intersection of a ray with the scene, the intersection with a mesh is computed by traversing all triangles of the mesh and returning the intersection that is the closest to the camera. To accelerate this, we can use a bounding box that allows us to check the intersection with the triangles of the mesh only if a given ray intersects the bounding box. However, the computation remains rather slow.

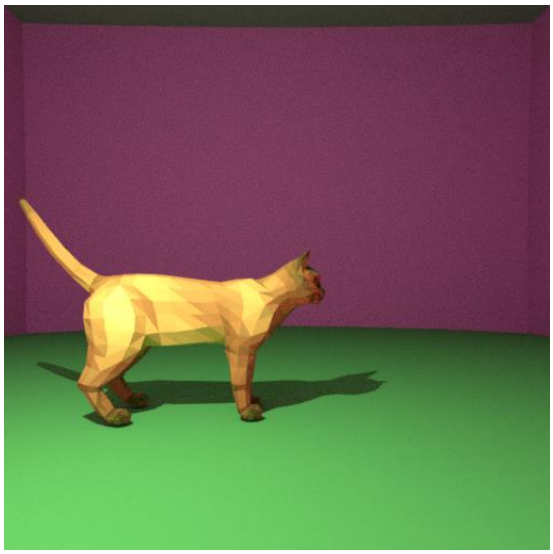

Figure 9: Scene with a cat mesh computed using the bounding box acceleration technique. The code runs in 49s using 32 rays and a maximum of 5 bounces

4.0 BVH (Lab 4 / pages 45--47):

In Lab 4, we further improved our acceleration strategy by implementing Bounding Volume Hierarchies. This strategy is performed by applying recursively the same principle as the one used in Lab 3. Starting with the entire mesh, we compute two new sets of triangles by assigning each triangle to one of the sets based on their centre point. Applying this process recursively, we obtain a BVH tree. This tree is then used to compute the intersection between the tree and the BVH. This drastically reduces the computation time.

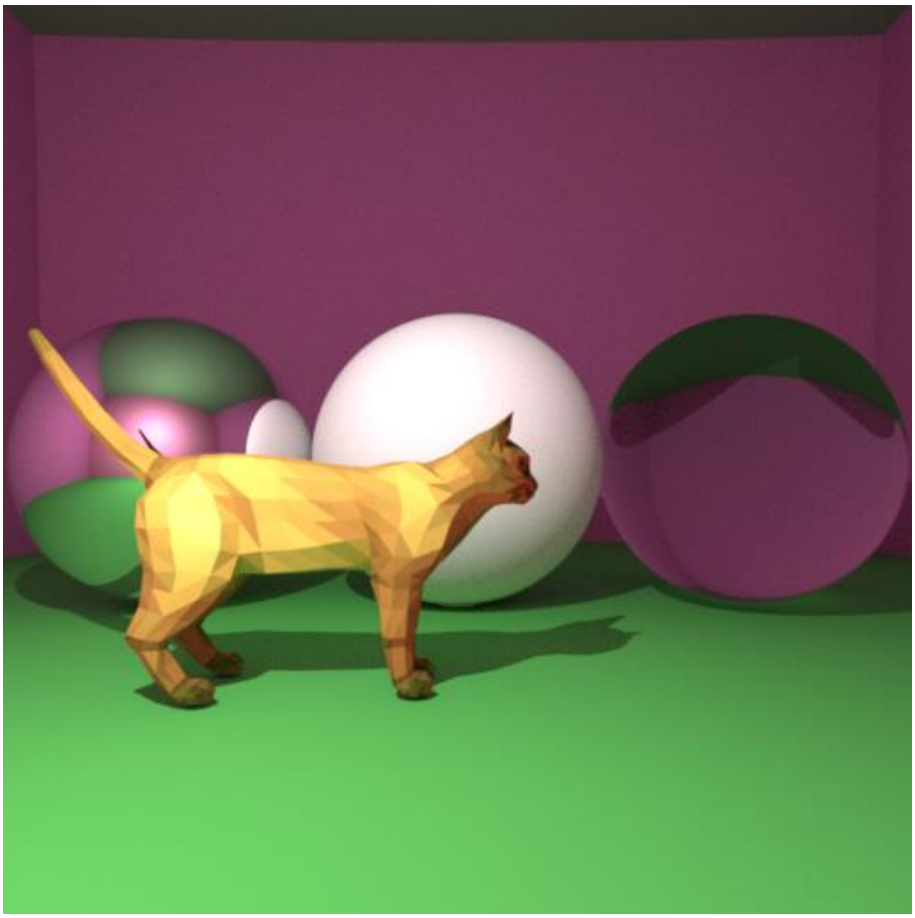

Figure 10: Scene with a cat mesh computed using the BVH acceleration technique. The code runs in 9s using 32 rays (55s using 128 rays, and 90s using 256 rays, shown here) and a maximum of 5 bounces