Laboration 2: Raytracing

Danilo Catalan Canales

Daniloc@kth.se

DH2320 Introduction to Computer Graphics and Visualization

Content

Introduction to the Lab

In this lab our goal was first to learn about to represent 3D scenes using triangular surfaces, trace the ray of each pixel in the camera image into the scene, compute ray-triangle intersections to find out which surface a ray hit. For the second part we will extend the program by adding camera motion, simulating light sources and reflecting light for diffuse surfaces.

During the first part of the lab we're given a short introduction of the data structure of a triangle, which we worked on throughout the lab. We were also given a function which computes and stores data, that created a figure which we worked with throughout the lab.

Intersection of Ray and Triangle

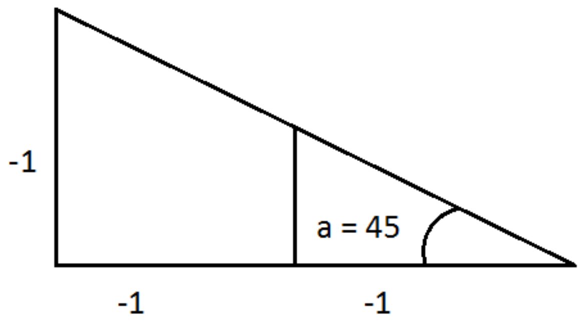
For this part we're given a introduction on how three 3D vectors represent vertices of the triangles. We are introduced with the equation of a plane as well as a line, and how to use these to compute the intersection between them. The equations were already familiar since before, and implementing them throughout the function was just putting them in and figuring out how to set the limits. The function that was in focus here was ClosestIntersection. The function took a start vector, a direction vector, an array of triangles and a structure to store the intersection values. In the function I check for intersections between a ray against the triangle's and return information about the closest intersection, but the primary function was giving out a true or false statement if an intersection was made between a line and the triangles.

In the instructions we're asked to set the camera and focal length and so for my choice of camera position and focal length I took the first lab into consideration, with a focal length of H/2 where I obtained a 90 degree angle for the Vertical and Horizontal field of view.

$$H = 2arctan(tan(\frac{V}{2}) * \frac{w}{h})$$
$$V = 2\ arctan(tan(\frac{H}{2}) * \frac{h}{w})$$

I kept the same Focal Length at H/2 with a Vertical and Horizontal Field of View at 90 Degrees (angle of view). With Focal length set, I had to decide a camera position. I had to consider the room dimensions and from there work out a logical strategy:

$$-1 \leq x \leq 1$$
$$-1 \leq y \leq 1$$
$$-1 \leq z \leq 1$$



I took a trigonometric approach where we can see that by putting the camera distance at -2 from the center of the cube we cover all aspects, giving us a full view of the room.

After implementing the function ClosestIntersection with all the vectors and variables that we had, the figure started to take shape.

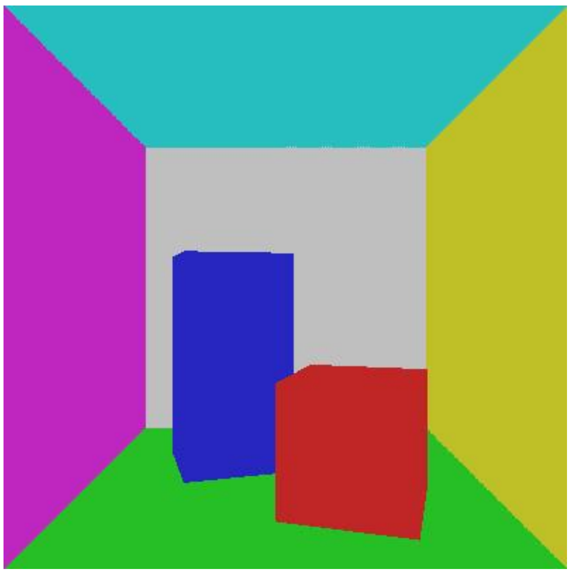


Figure 1: yay!

I had a couple of small faults of occasional misspelled variables or that the program crashed due to insecure use of C++ and the extensive library. But that made the re-runs of the code extra important in order to get confident in my calculations and learn about the different functions that SDL and GLM libraries have to offer. Some small issues that I had was to figure out that the limits for the scalars to the plane and line wasn't just as the instruction showed, since the edges of the triangle also count. The difference was that there was a line drawn between the figures because the limit excluded the edges of the triangles.

Soon enough i had movement up and going. To move the camera along the z-axis I just updated the camera position vector by adding or subtracting a constant to the z-value. I had to look for the rotation matrix that made the camera spin around its own y-axis. After obtaining and setting up a function that updated the global R-matrix value to the correct rotation matrix, I multiplied the rotation matrix with the camera position vector. After implementing the rotation to the camera, I added rotation to the image by multiplying the triangles, vertices and vectors, with a separate rotation matrix.

Working with Illumination

Illumination was a bit harder. I had to calculate it first through paper and then apply it to the program. Getting confused with what goes in what direction and new functions that were needed made this part a bit frustrating in the beginning. When the directions were set and functions were put in the right place things started to shape up again. So first up I organized which parts I had and what parts were missing. I had the Power of light, I needed the direction vector from the light source to the surface and the radius from the light source was also needed. The triangle structure already provided the normal from the surface and so the calculation was slowly filled with values. When implemented I was welcomed by rewarding results.

$$D = P\, max(\hat{r}.\hat{n})/4\pi r^2$$

Equation 1: Direct Light

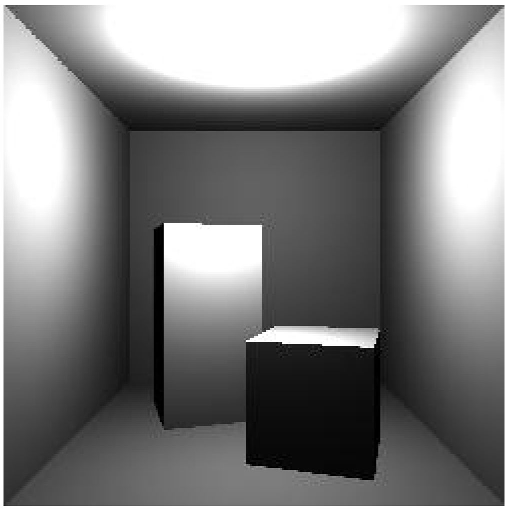


FIgure 2: Let there be light!

$$R = \rho * D = (\rho * P\, max(\hat{r}.\hat{n})/4\pi r^2$$

Equation 2: Direct light reflected by a diffuse surface.

By multiplying diffuse surface to the equation, we describe the fraction of the incoming light that gets reflected by the diffuse surface for each color component. After applying it to the equation we got the results we see in the figure below:

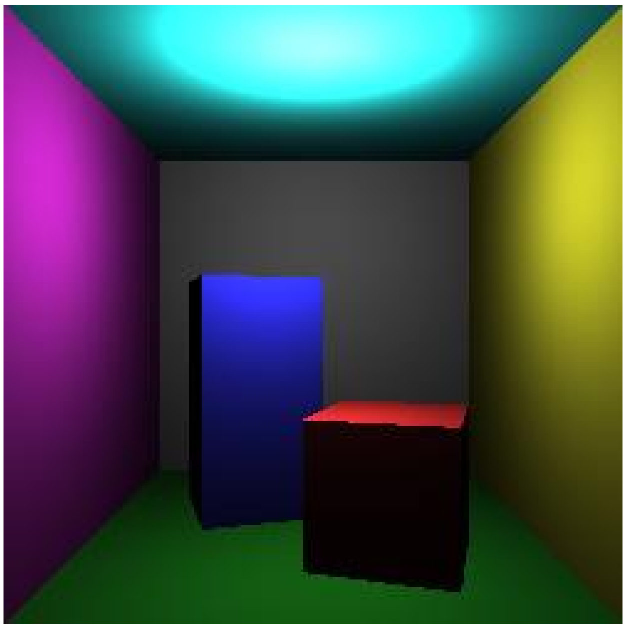


Figure 3: Direct Illumination * Diffuse Surface

We are missing an element to the scene, shadows. As a last task we're asked to implement the shadows to the scene. The idea is to cast a shadow over the surface where an obstacle gets in the way of the line of the light ray. To compute this I used the function closestIntersection to compare two distances, as instructed. The distances I compared was the distance from a surface to a potential intersecting surface and the distance from the light source to the same potential surface. If the distance from the surface to the obstacle put in front of the light ray was smaller than the distance from the light source to the same surface, then that surface was in the shadow and returned a black color vector.

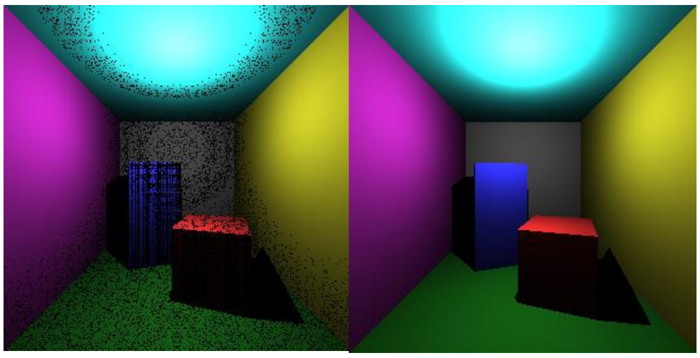


Figure 4 (left), Figure 5 (middle) and Figure 6 (right)

After putting in the condition in my directlight function I got a weird picture at first. The image was rendered in black dots(Figure 4). It was hard to actually find a way to approach this behaviour. Were there black dots because there was an obstacle? After a lot of research I found out that there is an offset in comparing distances. So in theory we compare 2 distances and if there is an intersection that's closer than the distance to the light source, then that surface is in the shadow. So by adding the offset it will always choose to display the first object if the second one isn't closer by at least the offset value. After implementing the offset I got Figure 5.
Lastly we were instructed to add indirect illumination:

$$T = D + N$$

$$R = \rho * T = \rho * (D + N)$$

Where D is the power of light and N is the power per surface area, constant approximation of indirect illumination. By simply adding this constant to the power of light, and then multiplying the diffuse surface I got a very illuminated scene, Figure 6.

This concludes lab 2 which, in overall, was very time consuming when learning how to implement the different aspects. When I got stuck there was always some part of the lab instructions that might have not been included in the equation or limit. There was also some research for either physical or mathematical equations that I was less familiar with or errors that had me stuck for far too long. Overall the best solution was to organize the bits that we were given by the instructions to see what elements were missing or were excluded and needed to be implemented. Finishing off I found myself enjoying working out most parts and getting exact results after everything was put in place.