Stockholms universitet

Laboration 3: Rasterization

Danilo Catalan Canales

Daniloc@kth.se

DH2320 Introduction to Computer Graphics and Visualization

Stockholms universitet

Content

Rasterization Intro

In this lab I used some of what I did in lab 1 and implemented it to the scene I used in lab 2. By implementing rasterization I will get a faster computing algorithm. I will project and draw the scene by implementing Interpolation for the most part of the lab.

Drawing Points

This was the first part of the lab. I had to compute and transform the 3D coordinates into 2D coordinates. When done, I drew out the vertices from our scene into the frame. For this lab I had our camera and focal length pre-decided for us. With a camera position vector set to (0,0,-(3.001)) and a focal length of the frame height I had double the vertical and horizontal Field of view at 180 degrees. Using the same methods I used in Lab 2, one could actually calculate that the whole room will be displayed. I got the following results:

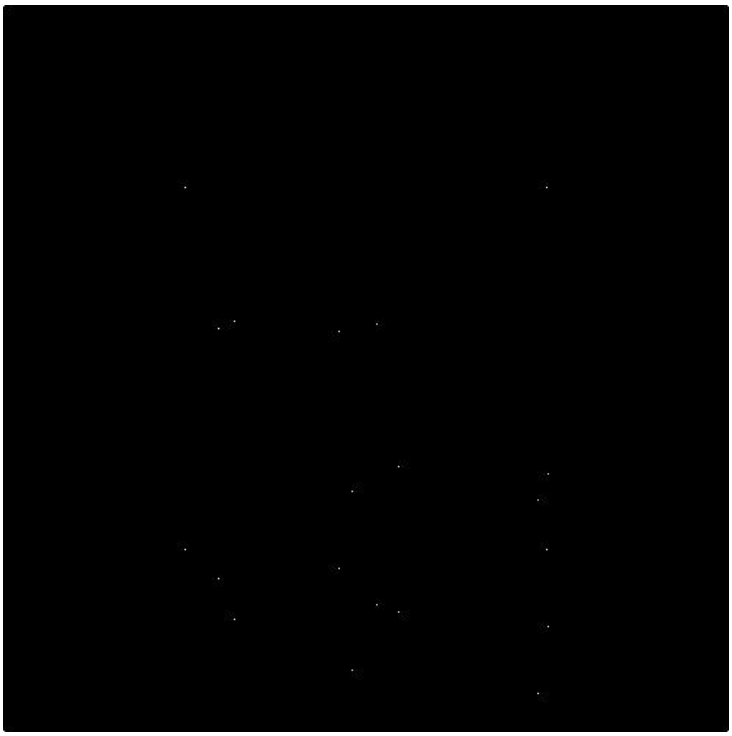


Figure 1: “You can almost see the two boxes”

Drawing Edges

I was given a function which handled interpolation between two 2D vectors. This part was very straight forward. By using the interpolation function, create a function that creates lines. From there, create a function that draws out the edges of a polygon by computing it from the vertices of the polygon. I was also given a lot of hints and coding which gave a good push in the right direction. By this point I also added movement for the camera and rotation for both the camera and the object. The difference between this lab and the previous one was noticed quite instantly. There was no delay during the update as I moved the camera and object around:

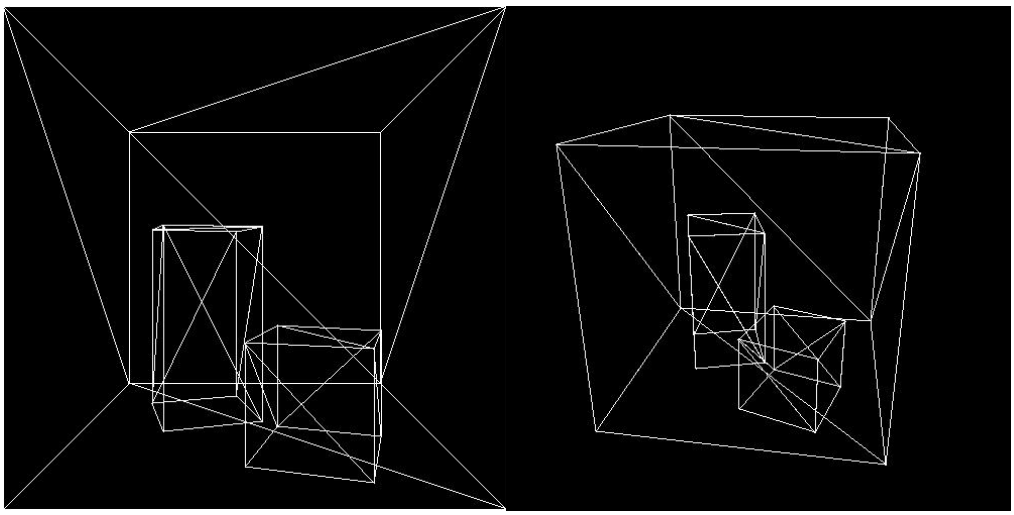
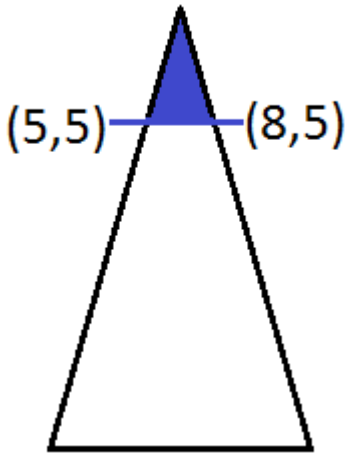


Figure 2(left) and Figure 3(right): After implementing Polygon edges and movement/rotation

Filling in the Triangles!

When it was time to fill in the triangles, the instructions provided less practicalities and got more informative. By filling the triangle row by row, one could completely fill out the triangle. First I had to implement the function ComputePolygonRows. By computing on the rows of the polygon and filling two arrays, which had coordinates for the left and right side of the triangle, I could store the coordinates in a well sorted way. I computed around the triangle to get out all the coordinates by interpolating and then comparing the values and matching them up with every row. Once that was set I implemented DrawRows which was basically what was done in the first lab, but instead of one side of the frame to the other, I interpolated from one side of a triangle to the other.



Visualization of how the triangle is supposed to be filled out.

For this to work I had to make a Structure chart to see what went where. This took some time but it made it all much easier to understand. Having things organized kept it all from getting messy too and after applying the two functions to the drawpolygon function I was able to fill out the triangles:

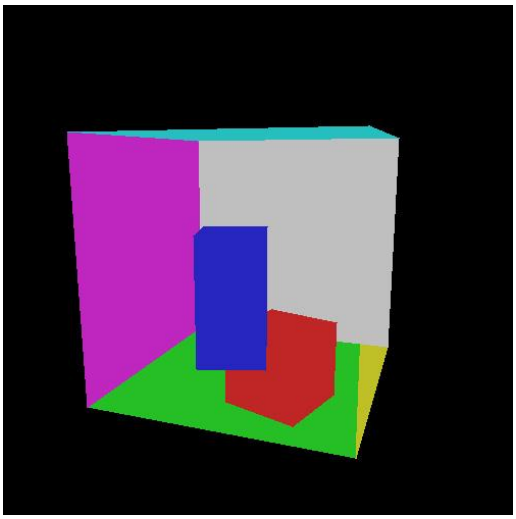


Figure 4: It should definitely not look like this.

Deep Buffer

So I've been able to fill out the triangles, but as one can see, the triangles are drawn on top of each other. To approach this I've been instructed to create a deep buffer and a pixel data structure to be able to draw out everything in the right order.

$$float\ \ depthBuffer[SCREEN_H][SCREEN_W]$$

I treated the depth buffer like a net, more or less, to be able to compare the depth of the image. The pixel data structure stores the x, y and the inverse depth, 1/z. By comparing the z inverse of every pixel to the net depth value, I can decide what pixel is put in front. The value closest to the camera is replaced in the net and therefore if any value is closer it is drawn in front of the previous one. After being able to put all the ranges in place and tweaking the other functions after the pixel data structure, I was able to get the following results:

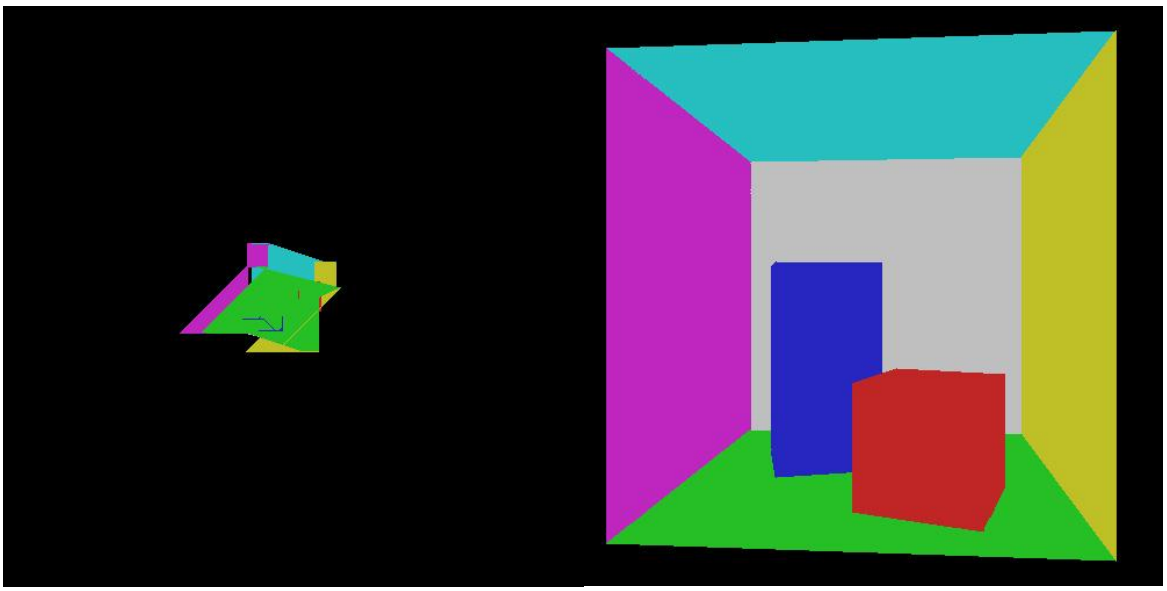


Figure 5.1:First got the ranges a bit sloppy after a tired day programming
Figure 5.2 Rotated so I have an even better look!

The most time consuming task must have been to modify the functions after the data structure. There was always some function that was forgotten or mismatched and I even got a nice picasso drawing after implementing the ranges a bit sloppy.

Illumination

So for this part there were two approaches, either compute the illumination for every vertex or compute for every pixel. As instructed I approached the task computing it for every vertex. Though it is much faster than ray tracing it's less accurate. As in the previous lab, I used the same Equation, but with an addition to the vertex and the pixel data structure. By defining the reflectance and the vertex normal, I could compute and store the illumination for every vertex.

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

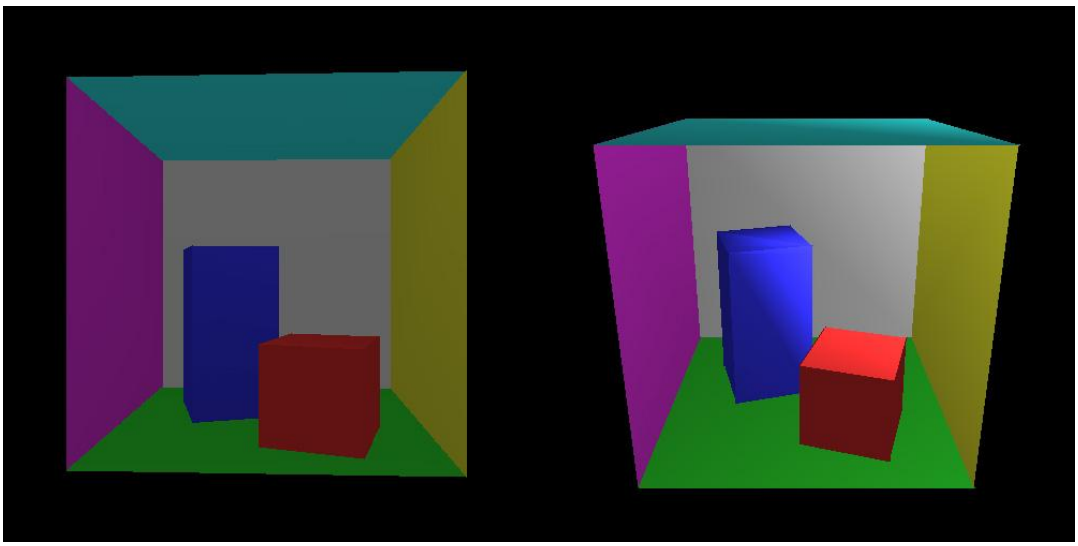


Figure 6 (left) and Figure 7(right): First figure has less power illuminating and the other has the light power increased.

After Implementing this I moved on to implement illumination for every pixel, where I moved the computation to the pixel function. I felt the biggest problem was to move around the different functions, changing them after the data structures were changed. I guess there was a well taught difference between the different approaches but very time consuming indeed.

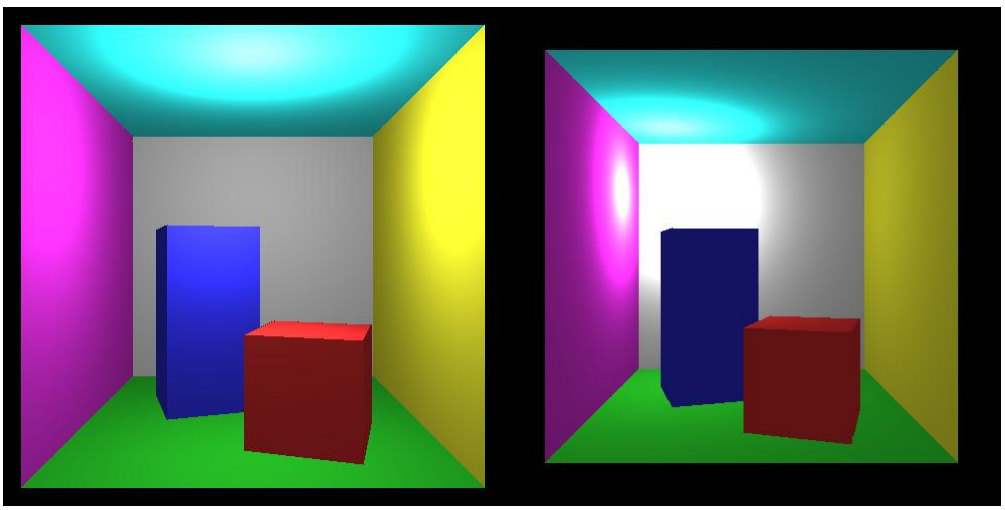


Figure 8.1 and Figure 8.2: illumination computed for every pixel, check, and added light source movement.

Conclusion

This finishes up the last of the three labs for this course. There was a lot of more computations and approaches that made a lot of difference for the ending results. Rasterization proved to be several times faster than the ray tracing that I worked with during Lab 2.