Lab 3: Rasterisation

DH2323 (DGI17)
KTH Royal Institute of Technology

Mikael Forsberg <miforsb@kth.se>
Robin Gunning <rgunning@kth.se>

May 27, 2017

Overview

We completed all the given tasks and added camera pitch, mouse control and backface culling.

Controls

- Mouse motion: camera pitch and yaw
- Arrow keys: camera pitch and yaw
- WASD+(Q,E): camera movement
- 1,2,3,4,5,6: light movement

Transformations

As we've already done the pinhole camera in labs 1 and 2 it was relatively easy to implement again. The transformations were easy to implement as all the required formulas were presented in an easy-to-understand manner.

Drawing Points

Drawing the points was essentially the same problem that we had already dealt with in the starfield part of lab 1.

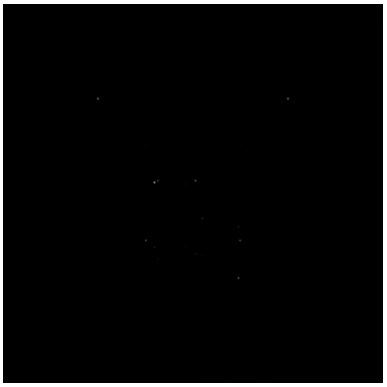


Figure 1: The vertices of the Cornell box

You can "clearly" see that the points in the picture represent the Cornell box.

FOV in theory and practice

This section aims to answer the question regarding FOV posed in the lab instructions.

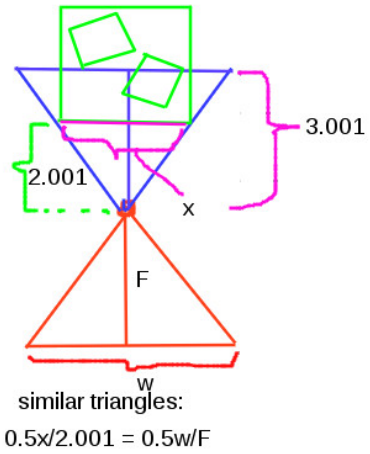


Figure 2: FOV calculation diagram

Let F, w and h be the same number, say $F = w = h = 320$. Let the camera be positioned at $(0, 0, -3)$ and have all angles of rotation at zero. A point on the vertical midpoint of the front edge of the left wall of the Cornell box is located at $(-1, 0, -1)$. The corresponding right point is located at $(1, 0, -1)$. Translating to camera space the left point is located at $(-1, 0, -1) - (0, 0, -3) = (-1, 0, 2)$ and the right point at $(1, 0, -1) - (0, 0, -3) = (1, 0, 2)$ Projecting the left point onto screen space we get $x' = F * x/z + w/2 = 320 * (-1/2) + 320/2 = 0$ and for the right point we get $x' = F * x/z + w/2 = 320 * (1/2) + 320/2 = 320$ meaning the two points are situated at the exact horizontal extremes of the screen. An analogous result can be shown for a pair of points on the midpoints of the front edges of the floor and roof of the box.

From the diagram (Figure 2) we see that at a distance of 2.001 from the pinhole of the camera the maximum difference in x that can fit on the screen (assuming $F = w$) is precisely 2.001, which means that the entire Cornell box will be visible since it happens to have a side width of 2.

Drawing Edges

To draw the edges we used the linear interpolation learned from lab 1. To fix the issue with vertices being behind the camera we added a form of "fake" clipping by looking at the camera space z-coordinates of triangle vertices and skipping the entire triangle as soon as any vertex is found located at $z < 1$. This is not a good solution since it causes triangles to "pop out of existence" when one or more of its vertices ends up too close to the camera, but we believe it is better than nothing.

Implementing camera movement and rotation was easy as we had already done so for lab 2.

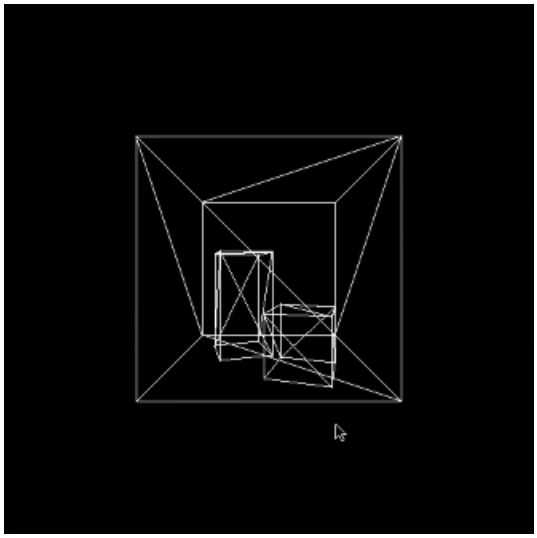
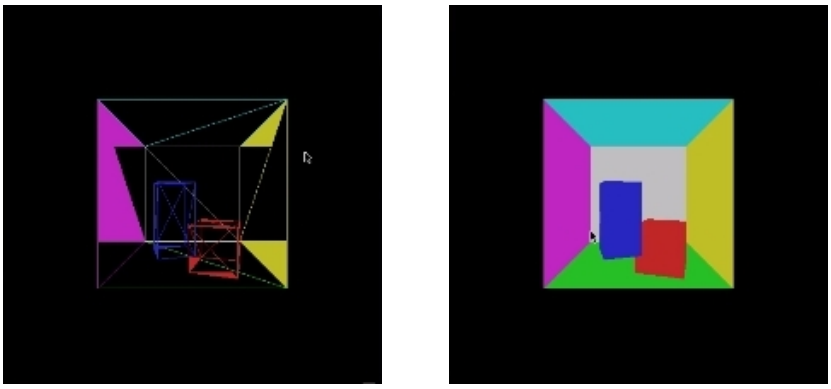


Figure 3: Wireframe image of the Cornell box

Filled Triangles

We first computed how tall the triangle will be (Δy) to get the number of rows needed. Then we walked each edge (or so we thought) of the triangle computing x_{min} and x_{max} for every row. Finally we drew the rows to the screen in the color of said triangle.

As hinted to we had an issue with missing one of the edges (one of the vertex pairs) in the loop that was supposed to walk all the edges.



(a) Missing an edge (b) All edges included

Figure 4: Drawing filled triangles

Depth Buffer

The depth buffer was pretty straightforward to understand and implement. However, implementing the new struct Pixel and changing almost every function we had already created was time consuming and frustrating. The reason for using $1/z$ is still pretty mysterious, as also mentioned below.

Illumination

The per-vertex illumination was similar to the bilinear interpolation done in lab 1. The effect of per-vertex illumination was itself quite reminiscent of the way games tended to look running on early 3D-acceleration hardware in the late 1990's. The switch to per-pixel illumination mainly consisted of adding even more elements to the interpolation function and not forgetting to copy them in when computing the rows for drawing.

We did have an art-generating issue at some stage in implementing lighting (see Figure 5), but neither of us remembers what caused it.

Making the per-pixel lighting be perspective-correct again involved the somewhat mysterious maths trick of premultiplying values by $1/z$ in order to be able to interpolate them in screen-space and later restore them by dividing by $1/z$. We still don't really have an intuitive grasp on why this works.

Figure 5: Some problem with lighting

Result

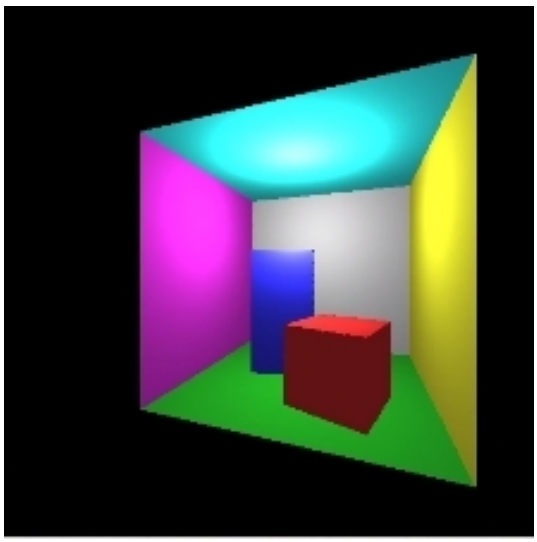


Figure 6: Final result

Bonus additions

Camera pitch

We added support for camera pitch by extending the camera code from lab 2 using the standard matrix for rotation about the x-axis. The two rotation matrices are multiplied together to create a final rotation matrix. We do not allow the camera heading to be affected by the pitch, and so we extract the heading before performing the matrix multiplication.

Camera control with mouse

We added support for controlling the camera pitch and yaw using the mouse, using the suggested `SDL_GetRelativeMouseState` function. This was fairly straightforward except for an issue where the function returned very large readings immediately at the start of the program. We fixed this by ignoring the mouse during the first 500 ticks (by `SDL_GetTicks`) of runtime.

Backface culling

We implemented the simplest possible form of backface culling by skipping any triangles where the angle between the surface normal of the triangle and a vector from the triangle to the camera position exceeds 90 degrees. Since we don't take the camera heading and FOV into account we are not able to cull triangles that are behind the camera.

Contributions

All of the hard requirements were completed as a joint effort. Mikael implemented the bonus additions mentioned above (camera pitch, mouse controls and backface culling).