# Lab 3 report

Victor Hansjons Vegeborn
victorhv@kth.se

DH2323 Computer Graphics and Interaction
KTH Royal Institute of Technology

## 1 Transformations

In this lab, a rasterizer was implemented with a pinhole camera, just as in the first lab with the moving stars. This time the camera had to be able to move. The movement approach was similar to the one in lab 2.

The projection of 3d-points onto the 2d view plane was done with the equations 3 and 4, just like in lab 1. However, the projection had to account for the mobility of the camera. In the vertex shader, all vertices was recalculated, based on the relative position to the camera and its orientation. If a vertex position is $\mathbf{P}$, the relative transformed 3d-position of a vertex is $P'$, the camera position $\mathbf{C}$ and the rotation matrix is $\mathbf{R}$, then the new relative position is calculated by:

$$\mathbf{P}' = (\mathbf{P} - \mathbf{C})\,\mathbf{R}$$

The actual computation of camera translation and the rotation matrix is done just as in lab 2.

## 2 Drawing triangles

A surface triangle is defined with 3 vertices, a normal vector and a color vector. The rasterizer work by first transforming the triangles vertices to pixel positions, then interpolating each pixel, instead of doing a ray trace.

First, the y-component for all pixels that make up the borders of triangle is calculated. This is done by interpolating the y-coordinates of two adjacent pixels in every edge. After the border y-component have been computed, the x-coordinate can be interpolated row by row as relative to the screen orientation. The algorithm can be further examined and is implemented in the ComputePolygonRows-function.

The hardest part of implementing the algorithm was to realize that the interpolation order had to be switched when iteration over the vertex pixels reached to an end. This was solved by an if-statement that changed the ordering of interpolation.

## 3 The final implementation of interpolation

The interpolation function in this lab got extensively extended through out. As the data structure change in later stages, the interpolation had to be interpolate data which was used for the z-buffer and the lighting.

## 4 Depth Buffer

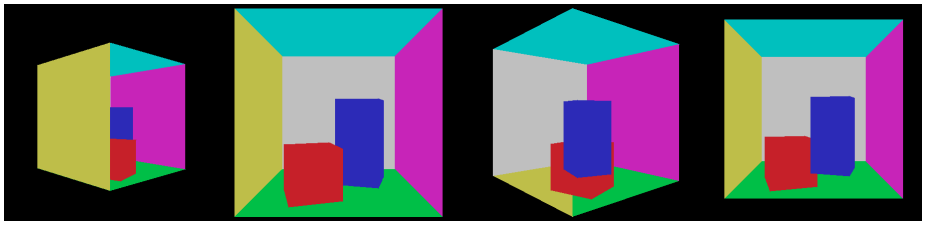

Fig. 1: Left: with z-buffer. Right: without z-buffer.

As the drawing of triangles was done in a pre-defined order (the order they got loaded into the data structure holding each triangle), the rasterizer drew triangles in an unwanted order which can be seen to the right in figure 1. The solution to this is the depth-buffer.

The depth buffer keeps track of the inverse z-component of each pixel. As pixels do not have z-components, these values are interpolated when the triangle is interpolated as stated in section 2. Before the pixel shader renders a pixel onto the screen, it checks the depth buffer of pixels that are closer to the camera. If no pixel is closer, then the shader renders the pixel.

One issue arose when implementing the depth buffer. Some lines were jagged. The solution to this was to subtract a small number (0.00001 in the implementation) from the inverse z-component before storing it in the depth buffer. Probably, this error was a consequence to the lack of precision of floating numbers.

## 5 Illumination

The illumination of pixels was also interpolated in a per-pixel approach as further explained in section 5.a. As with the inverse z-component, each pixel stored a 3d-position that was interpolated relative to the camera orientation. With this data, the possibility to model direct light from an omni light could be computed by implementing equations 10 (the same as in lab 2). The resulting direct light was then added to some constant indirect light. In the rasterizer, no rays had to be shot from a surface point, which drastically impacts the frame rate.

The last part of the lab was to correct the light problems that occurred due to linear interpolating the pixels 3d-positions. The resulting issue can be seen in the two most right images in the figure below. The solution was to multiply

the pixels interpolated 3d-position with the stored inverse z-component, before sending the pixel data to the pixel shader. The final result can be seen in the two left most images in the figure below.

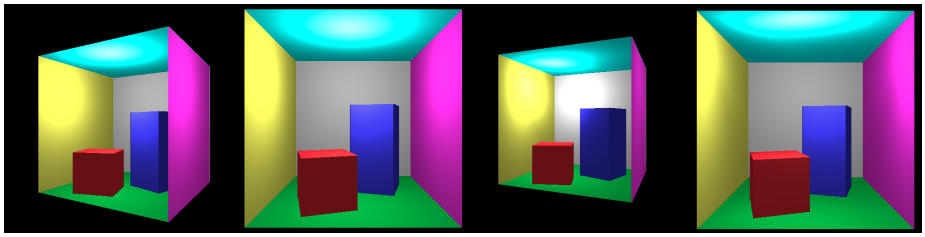

Fig. 2: The two most left: the final result. Two most right: the sewed render when not compensating for the linearly interpolated 3d-vertex postitions.

## 5.a Per-pixel vs per-vertex illumination

In this lab, the first implementation of illumination was done with a per-vertex approach. By computing the omni light equation (10) for each vertex, instead of each pixel, and interpolate the light between those values, a much faster light computation could be achieved. In a rasterizer, this is very much wanted, but the cost of computing light as with the per-pixel approach outweighs the result achived with per-vertex illumination. The result of per-vertex illumination comes with unwanted rendering of pixels.

## 6 Weird render

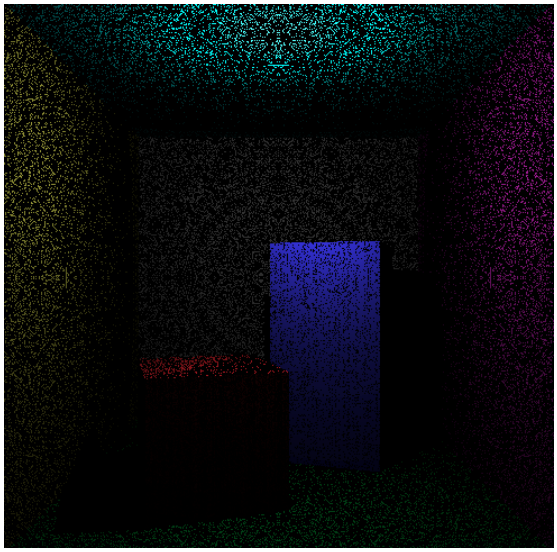

Fig. 3: A nice render which i cant explain why it happened. I just thought you should see it, becous its nice.