Lab 3: Rasterization

DH2323 Computer Graphics and Interaction

Author: Mohammad Javad Ahmadi

April 16,2017

1. Drawing vertices of triangles of the model.
2. Camera positioning.
3. Drawing triangle edges.
4. Filled triangles.
5. Depth Buffer.
6. Vertex Illumination.
7. Per Pixel Illumination.
8. Summary.

1. Drawing vertices of triangles of the model.

Drawing the vertices of the loaded triangular model was done by defining the VertexShader function which translates the vertices of triangles from 3d to 2d and by drawing the 2d points we get the following image. The function was implemented using the following equations.

$$x = f\frac{x}{y} + \frac{W}{2}$$

$$y = f\frac{y}{z} + \frac{H}{2}$$

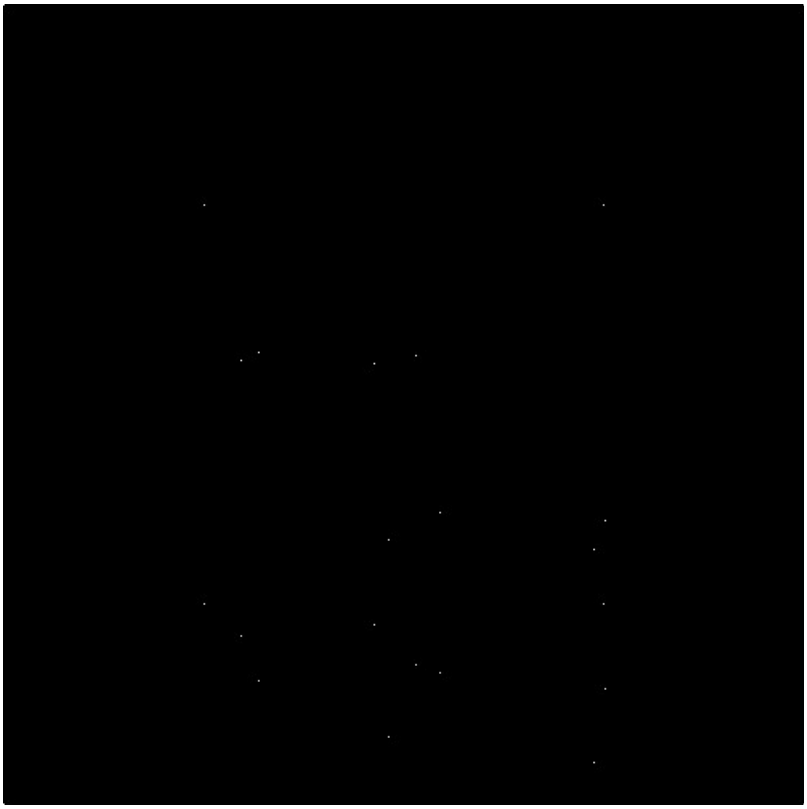

Figure 1: Drawing points projected by a pinhole camera.

2. Camera positioning.

The the starting position of the camera is (0, 0, -3.001) and we use quadratic width and height for the screen therefor in order for the camera to cover the whole screen we use the a focal length as the same value of our dimension.

3. Drawing triangle edges.

Next we draw each edge of the triangles using interpolation along the vertices. And using the rotation matrix along different axis we implement the rotation about x,y,z axis and the translation of the camera position. The rotation was implement similar to the previous lab.

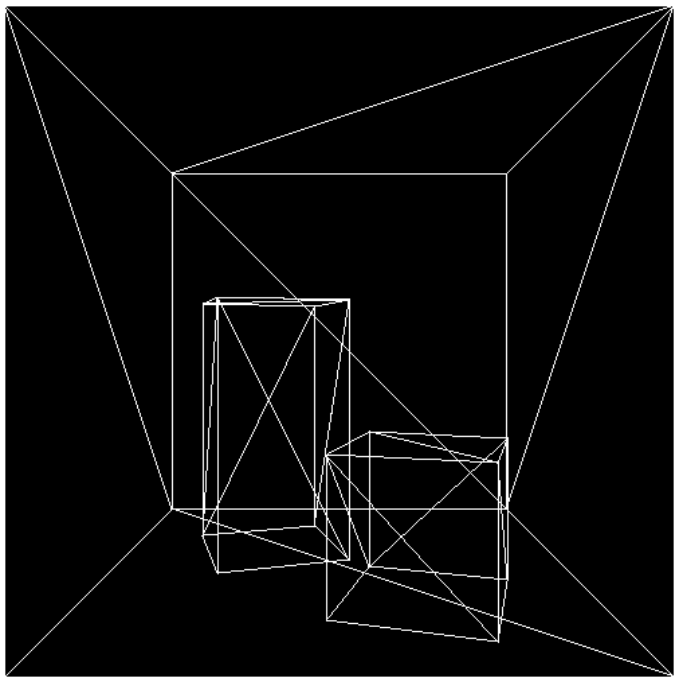

Figure 2: Drawing edges of the triangles.

4. Filled triangles.

In this part of the lab we draw solid colors in our triangles filling them row by row. We use our arrays which are holding start position and end position of each row and then we draw a line of color from start to end looping through until we fill the whole triangle. The problem which is visible in the below image is that we have not dealt with occlusion making the blue box to appear on top of the red box just because it was rendered after the red box was already rendered.

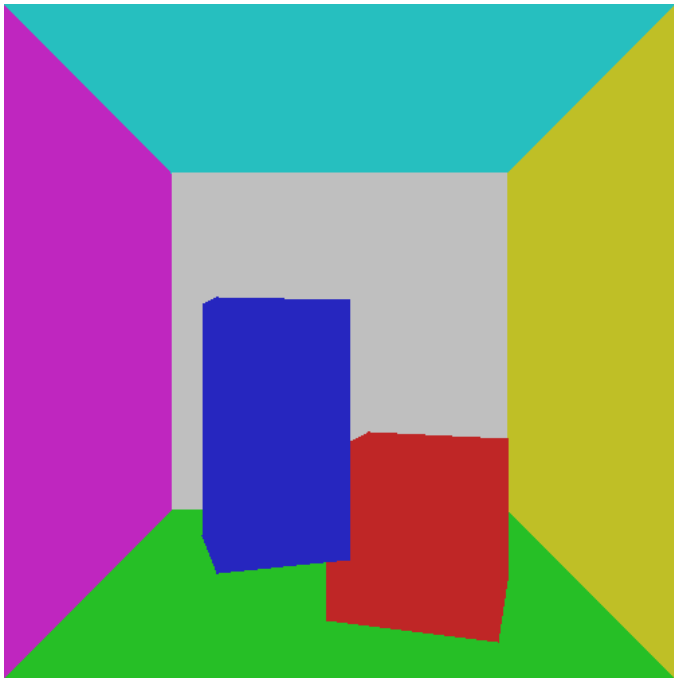

Figure 3: Filled triangles without occlusion.

5. Depth Buffer.

Using depth buffer we fix the above issue by recognizing which surface is closer to the camera using the triangles z inverse. A bug for camera movement was introduced at this point which was fixed by some conditioning to check for the bounds of the screen.

```
if (line[i].x >= 0 && line[i].x < SCREEN_WIDTH && line[i].y >= 0 && line[i].y < SCREEN_HEIGHT)
{
  PixelShader(line[i]);
}
```

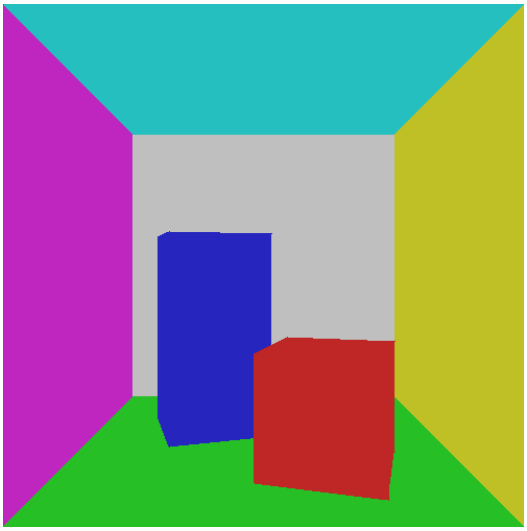

Figure 4: Filled triangles with occlusion.

6. Vertex Illumination.

At this part we add illumination to our program by computing the illumination for each vertex in our vetrexShader function and also interpolating the result of that interpolation across the polygon. We use the following equation to compute the direct illumination from the light.

$$D=\left(P\,max\left(\hat{r}.\hat{n},0\right)\right)/4\pi r^2$$

And then we add our indirect light using the following equation.

$$R=\rho*\left(D+N\right)$$

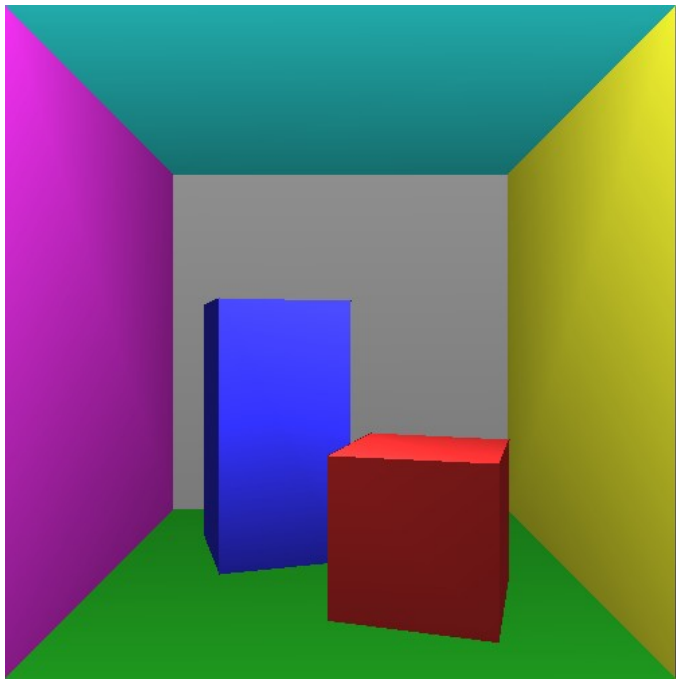

Figure 5: Vertex illumination.

7. Per Pixel Illumination.

The result of vertex illumination wasn’t so detailed therefore we try another approach as we do our calculations for each pixel and get more details. In order to achieve this we use the above equations in a new function called PixelShader instead of our VertexShader in order to run it for each pixel and for the equations to work we need to add the 3D position, normal, reflectance to each pixel which was done by following the instruction of the lab. The movement of lights were also added at the end.

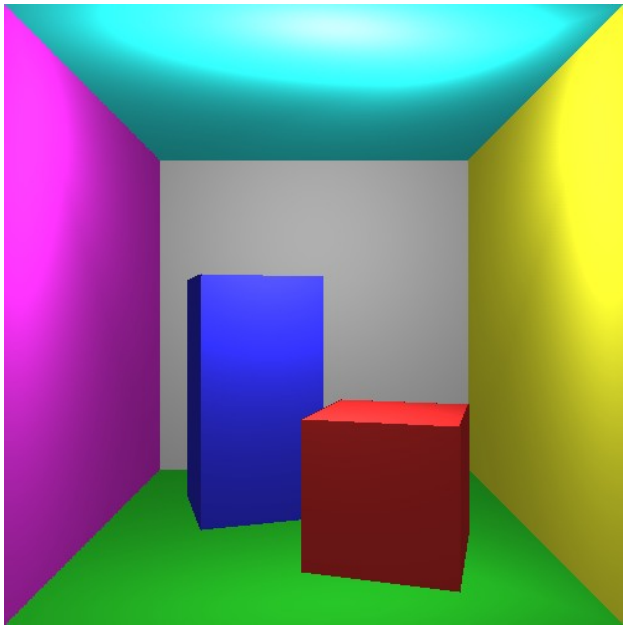

Figure 6: Per pixel illumination.

After submission of the lab I was told that the above image was without correction and after reading the instruction again I could fix the issue by dividing my the pixel's position by it's z depth and then divide it back in order for changes to be linearly done in the image. At last I ended up with the correct image as below.

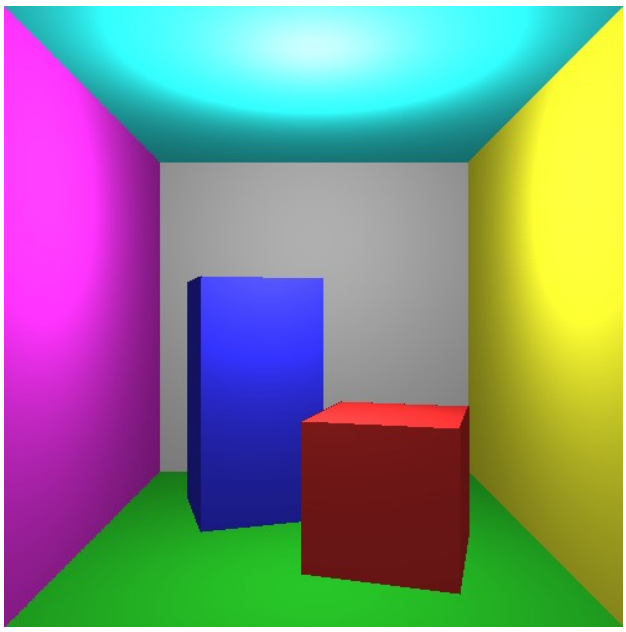

Figure 7: Per pixel illumination after correction.

8. Summary.

In this lab we created another rendering algorithm called rasterization which is way faster but less accurate and detailed in comparison to ray tracing which produces results with less details suitable for real time visualizations. We took a triangular model and converted that to an image in order to show our 3D model as a 2d image. Our rasterizer takes the stream of vertices from our model, transforms them into corresponding 2D points on our display monitor and fills in the transformed 2D triangles with appropriate colors.