Computer Graphics & Interaction
Lab Assignment #3: Rasterization

Enrique Perez Soler

1 Transformations

Basically this section teaches the tools and theories we are gonna use to create a scene and rasterize it. We saw two approaches in the first labs that could be used to make a 2D image of a 3D world, nevertheless we will use projection from now on.
We situate the camera position and axis orientation the same way we did on lab 2 then the projection of point in a 3 dimensional space to a 2 dimensional space follows this translation:

$$x = f\,X/Z \quad (1) \qquad y = f\,Y/Z \quad (2)$$

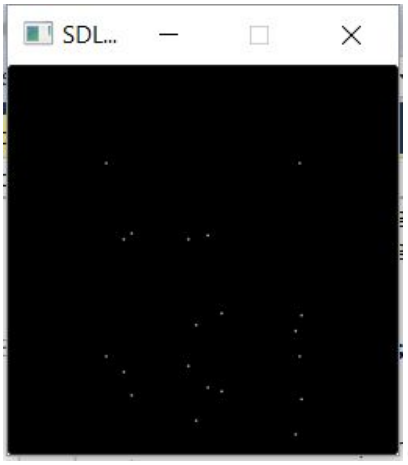
If the image has width W and height H the projection can then be written:

$$x = f\,X/Z + W2 \quad (3) \qquad y = f\,Y/Z + H2 \quad (4)$$

This formulas we will keep in mind as we might use them early on. The next section explains the movement of the camera using the same principles as projecting equations (3) and (4). Regarding the rotation, for the transformation we will use a 3x3 matrix and the vector-matrix product to do so. So in conclusion, once the rotation and translation are implemented the way to transform a point from world coordinate system to camera coordinate system takes two steps: 1. Transform it from world space to a coordinate system that is centered at the camera and 2. Transform it from the coordinate system that is centered at the camera to a system that is also rotated as the camera. (Following the equations i haven't included in this document as it was not necessary to explain)

2 Drawing points

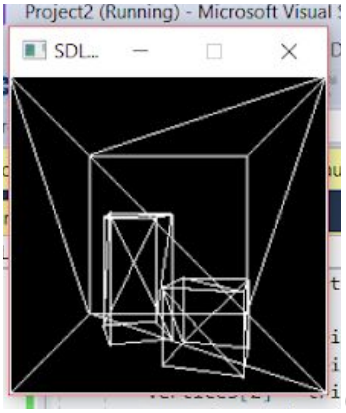
First task is to project the vertices of the scene using the same principle of the starfield on lab1. The Draw function loops through all triangles and all their vertices after which it calls the function *VertexShader*() function. This one takes the position of a vertice *v* and calculates its 2d representation saving it in the 2d integer vector. This function was implemented by following and using the formulas 3, 4 and 5 principles; and by using a fixed position for the camera with global variable *cameraPos.*



(200x200)

3 Drawing edges

Next step was drawing the edges, further stepping to creating the real thing. This is accomplished by joining the vertices through 2d lines created by linear interpolation. So I proceeded to review the interpolation function in lab 1 to get refresh the principles and ideas behind it and comprehend better what use we are going to give it in this lab.
Before drawing the lines, because really drawing a line is interpolating the values between two vertices and coloring them leaving no space therefore creating a line, we have to count the amount of pixels the line should have. Then, we loop through all of them calling *PutPixelSDL()* function to draw. But the real magic happens when implementing the *DrawPolygonEdges()* function as the result was most successful and complete. The way it worked was that by taking the vertices of the triangles it could draw the lines of each edge using *DrawLineSDL()*. The output result follows the same structure as figure 3 in the documentation:

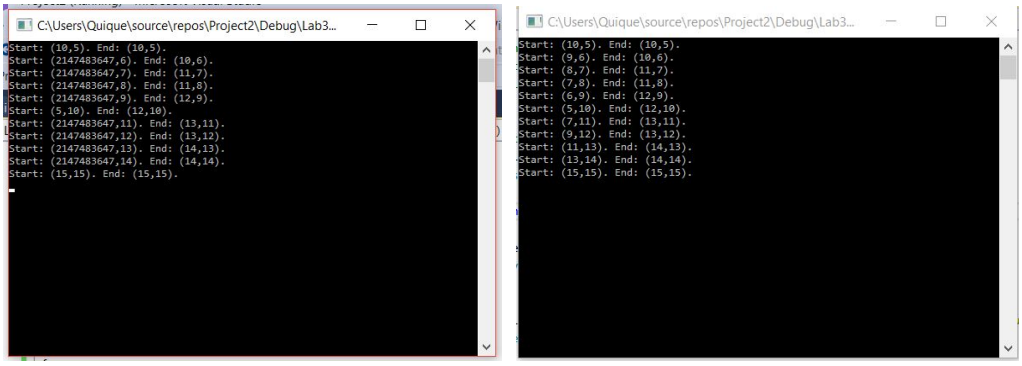


(200x200)

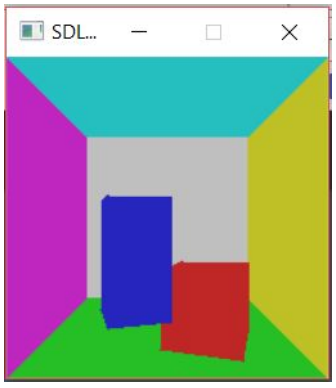
Last thing due in this section is update the Update() function so it rotates in the y-axis when the usr presses arrow keys just like in lab 2.

4 Filled Triangles

Next step is coloring up the triangles to make solid surfaces. To do so the procedure is drawing the triangle row by row, two arrays store each starting and ending positions of the triangle for each row. the left and right of that row.
I followed the instructions and steps described in the documentation making use of the knowledge of rows and the interpolating function at the time of implementing the *ComputePolygonRows()* function.
To try out the function I made use of the examples provided and at first was unsuccessful with the result, I miscalculated the computing of the rows (slight condition error in the “if statements”), here I show two screenshots of the main program output, one with the error and one finally solved:



Proceeding to implementing now the draw function for these rows, I had some trouble following the calls because I think the function name has an error in the documentation. When showing how to define the function it says “DrawRows(...)” and scrolling through the next documentation there is no clue of such function but rather one called “DrawPolygonRows(..)” with the exact same arguments and it was fairly obvious that it must have been a misunderstanding. After that small confusion, I went to implement it by making use of a call to PutPixelSDL for each pixel between the start and end for each row.
Another reason why i realised the two functions were the same was that this one is necessary for the nex function *DrawPolygon()* which will project the vertices and will be called from Draw-function instead of *DrawPolygonEdges()*. After creating the new variable currentColor and setting it to current color of the triangle in the Draw() function to use it in *DrawPolygonRows()*, the result that happened appeared as follows, resembling figure 4 in the documentation:

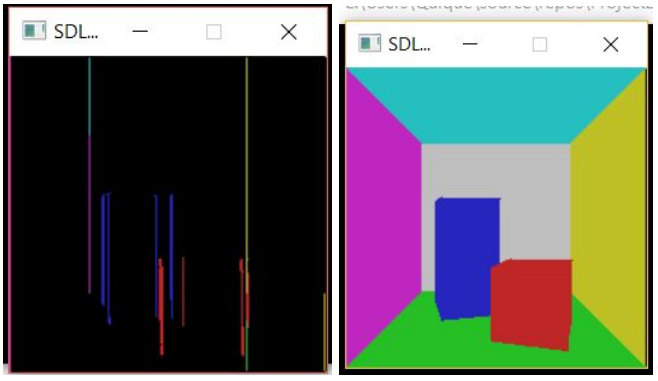


(200x200)

5 Depth buffer

To solve the depth problem in our rasterizer we make use of a structure called “depth buffer”. That is an image storing a depth value for each pixel instead of a color value.
To implement a depth buffer we create a global 2D array with the same size as the camera image:
float depthBuffer[SCREEN_HEIGHT][SCREEN_WIDTH];
In it we will store the inverse depth 1=z for each pixel. Since we now also need to interpolate the depth over the triangle, not just 2D position, we create a new struct to hold the information needed for each pixel *struct Pixel*. As said in the documentation I followed to construct these variables needed for further implementations. Afterwards I switched every gml::ivec2 to Pixel, making the correct modifications in the different functions that were altered by that switch.
As we consider the inverse of the depth (denominated *zinv*, z corresponding to depth) before we

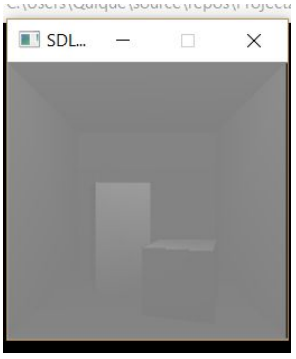
project the image in *VertexShader()* it is very important to do it correctly because if we do we'll then have access to the depth inverse of every pixel for the following *DrawPolygonRows()* function.
By doing so we are able to adjust the depth of which a pixel is with respect of each other by using the depth buffer. When adjusting the new z-inverse parameter to fit to the rest of the program and had a glitch when executing to see a result similar to figure 5. Encountering the image shown below to the left. At first i thought it had something to do with one of the drawing functions but to my surprise and frustration (after spending a good amount of quality time) it turned out i had mistakenly used a different coordinate in one of the equations on the interpolation function. In the end, I did get what i was looking for but all through wasting more time that i really have.



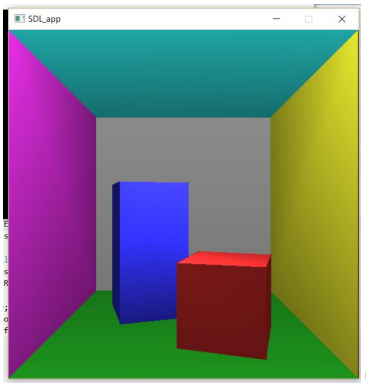
(200x200)

6 Illumination

Implementing first a function that deals with pixels *PixelShader()*, following the same principle as the VertexShader() for each vertex, we had to keep in mind that it should be called for every interpolated pixel in *DrawPolygonRows()*, but not much to be added to it than understanding the code provided for it.
My vertexshader looks a little different than the one provided but the results seem to be the same so there's nothing to worry about at first. Now that we have created a new struct *Vertex* it is just right to update the functions that deal with vec3. Just like in lab 2, at the time of illuminating the vertices we will store the parameters of the light source as global variables. According to the formula we add the *normal* and the *reflectance* at the struct of Vertex to process their illumination. At first there was some problems with the value of the reflectance, it didn't seem to add up but then i was told by a former colleague of mine that the type was incorrect and rather than vec2 it should be vec3. Once i fixed that everything went correctly. As all the changes were applied to the functions that would be affected by the new method of illumination we have just learned, i stumbled upon a "colorless" glitch such as follow:



Just to realise, stupid of me, that i didn't count for adding the *currentColor* in the *PutPixelSDL()*, i had taken it out for some reason and left as parameter only the illumination of the vertices so the 3D scene was illuminated but there was no color on it. After that the scene was correctly rasterized but for some reason the light was not right, the scene was too dark. I imagined there had to be a problem with the position of the light source so i went to change the values of the global variables and messing around realised the position it was given to us in the lab documentation was wrong. After that bump everything went fine:



(500x500)

And the best part of it was that like it said in the doc, the rasterization of the scene was way faster than the one in the raytracer.

Next task lets us create a more detailed illumination by evaluating equ 10 and 11 in PixelShader instead of VertexShader. I create the new variables, the normal and reflectance as constants and we add to the Pixel structure its position, necessary for this task. As well, I get rid of the illumination and the normal and reflectance from the VertexShader as it won't be necessary anymore. The scene we create resembles that of figure 8 in the documentation: