Assignment 3 - INF01009
Rasterizing and Shading Triangles

Guilherme G. Haetinger

2021-09-29

---

1 Introduction

We were assigned to finish the rendering pipeline for our **Close2GL** system. In the last assignment, we had to implement Model-View-Projection transformations along with clipping and culling. Now, we had to take the remaining points of the triangle set to transform them to fit the window space and rasterize them. These two processes along the CPU based shading is described in the next sessions.

2 Additional Transformations

By the end of the last assignment, we had triangles with points in the homogenous clipping space ($[-w, w]$). Now, the first thing we needed to do was, after setting them to the NDC (Normalized Device Coordinates $[-1, 1]$), is to align them with window coordinates, *i.e.* transform them so they have their coordinates between 0 and $width$ or $height$.

This is done by applying a Viewport matrix to the coordinates:

$$V = \begin{bmatrix} \frac{width}{2} & 0 & 0 & \frac{width}{2} \\ 0 & \frac{-height}{2} & 0 & \frac{height}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3 Scan Conversion Process

3.1 Loading and Rendering Color/Z Buffer

The first step to scan conversion is the implementation of a texture buffer to fill during the process. To do so, I call `glTexImage2D` repeatedly for every frame with the color array. The latter is allocated with the following size:

$$width * height * 4 * \text{sizeof(GLfloat)}$$

I also allocate a *zbuffer* array with one value per pixel.

The only modification I had to do in the shaders was to receive an *uniform* texture and rasterize based on the texture coordinates, which is passed in the same buffer I fill with *normals* in the OpenGL iterations.

3.2 Scan Conversion

The scan conversion process starts with window coordinates for the vertices as their attributes divided by previous homogenous coordinate, including $\frac{1}{w}$. I sort them by descending y value (closest to the top first). Then, I define three different types of triangles. Each of them have specific requirements to fill their pixels:

- Horizontal edge on top $dx_1 \leq 0.9$;

  Needs to render a line between `vtxs[0]` and `vtxs[1]`. Defines the active edges as `vtxs[0]`, `vtxs[2]` and `vtxs[1]`, `vtxs[2]`;

- Horizontal edge on bottom $dx_3 \leq 0.9$;

  Needs to render a line between `vtxs[1]` and `vtxs[2]`. Defines the active edges as `vtxs[0]`, `vtxs[1]` and `vtxs[0]`, `vtxs[2]`;

- The rest;

  Needs to switch active edges once y goes past the second vertex. Defines the initial active edges as `vtxs[0]`, `vtxs[1]` and `vtxs[0]`, `vtxs[2]`;

Apart from these differences, their essential operations are the same:

1. Initialize a y iterator;
2. Calculate the incrementation for x, z along the active edges;
3. For every y inside the triangle ($y \leq dy_2$), starting at each edge start:
   (a) Increment the current x, y by the incrementation in each active edge;
   (b) Define the enter point to be the one with the smaller x;
   (c) Go from `enter.pos.x` to `exit.pos.x`, interpolating their values to find the current pixel value. For each pixel computed, divide all its attributes by its interpolated $\frac{1}{w}$, check the *z-buffer* for other pixels in front of it and fill it alongside the color buffer if not.

By the end, we get a filled color buffer and send it to the fragment shader.

4 Rendering & Shading modes

This shading process is just replicating what I had in the previous assignment. I compute the color for every vertex after applying the Model matrix for Gouraud Shading and compute the color for every painted pixel in the process "c)" above for Phong shading.

The *Wireframe* render option is done for all shading modes by limiting the pricess in "c" to only render the `enter` and `exit` pixels. This seems to differ from the OpenGL proposal, but it gave me a very good intuition on where my software could go wrong.

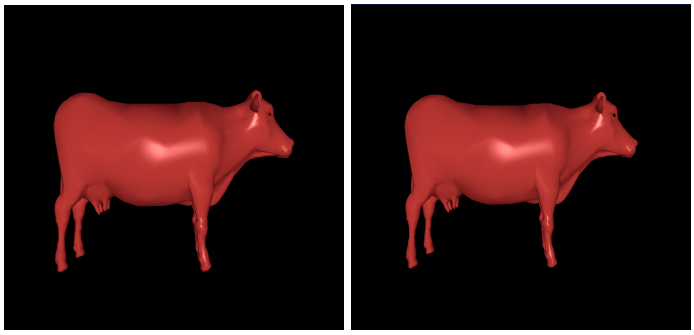
5 Performance & Conclusion

Of course, we get a huge performance decrease from the last assignment. Now, we can see the observed FPS value in the following table:

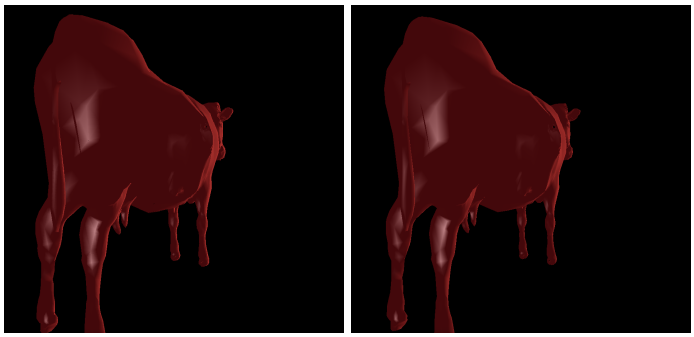
	OpenGL	Close2GL
No Shading	65	10
Gouraud AD	65	8
Gouraud ADS	65	8
Phong	60	7

In summary, this assignment was by far the hardest one. Debugging was very hard as it went from analyzing vertices to analyzing pixels, which is a very high quantity bump. I went through a very hard problem of approximation because I was accounting continuous vertex positions when computing the α for bilinear interpolation. This led to black "scratches" across the model. I also didn't implement the Wireframe mode with the highest fidelity possible. There are also numerous optimization options to increase framerate that I didn't have the time to put effort into, *i.e.* using `SubTexImage` instead of `TexImage` and make sure to delete the texture everytime the width and height changes (not necessary with `TexImage`).

Here are a few screenshots comparing my result to OpenGL's (Left: Close2GL, Right: OpenGL):

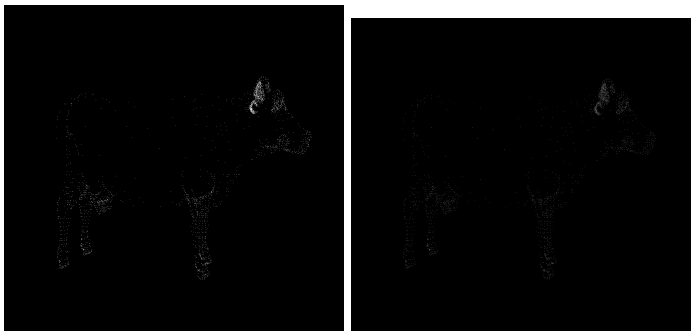


A red Pong shaded cow



A wrong winded cow using Gouraud Shading

Wireframe cow



Rendered points (not compatible with Phong shading on Close2GL)