Assignment 2 - INF01009
Implementing Camera View, Projection and Model Transformations for *Close2GL*

Guilherme G. Haetinger - 00274702

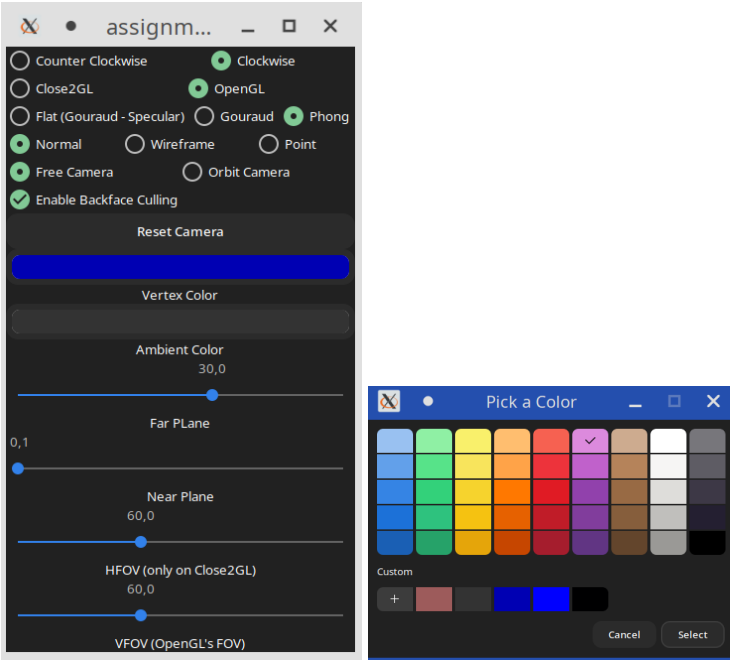

Figure 1: Options (left), Color picker (right).

1 Implementing a new UI

As a new challange to the *Close2GL* project, we need a decent UI. To do so, I chose to use the **GTK** library, more specifically, **GTKMM**. I designed, thus, a somewhat of a state handler class *Options* and two files that interact with it by fetching (`main.cpp`) and setting (`option_window.cpp`) data. I used a concurrent environment to make sure both the window handlers and the OpenGL iterations would run together and be interactive.

As a miscellaneous note, as we had to enable the interaction for both *hfov* and *vfov*, I set the *vfov* slider to be correspondent to the regular field of view in **OpenGL**. The reason for it is that **OpenGL**'s projection matrix is created with a single angle input.

The final layout is in fig. 1.

2 Applying Transformations

In order to rasterize the model in the screen, we need a camera definition. I kept the one from the previous assignment, including movement. The difference right now is that we need these attributes to:

1. Read data into triangle objects (instead of loose vertices);
2. Produce the matrices;
3. Apply these matrices;
4. Cull unecessary data;
5. Write in the Vertex Buffer;

We describe these operations in the following subsections:

2.1 Reading into Triangles

The first thing that I did was to create a `Triangle` class that contains a number of operations implemented statefully. Once we have the same vertex data as we had in the previous assignment, instead of sending it to the vertex buffer, we can just go triangle by triangle and deciding whether or not to discard them. This is where the class mentioned comes in handy.

2.2 Matricial Operations

Described in the assignment specification, the matrices are a core subject in this project. I defined a namespace `mtxs` in which I define creation functions for the *View*, *Projection* and *Model* (Translation and Scaling) matrices.

We apply the matrices as $PVM * t_{a,b,c}$, where t is the triangle.

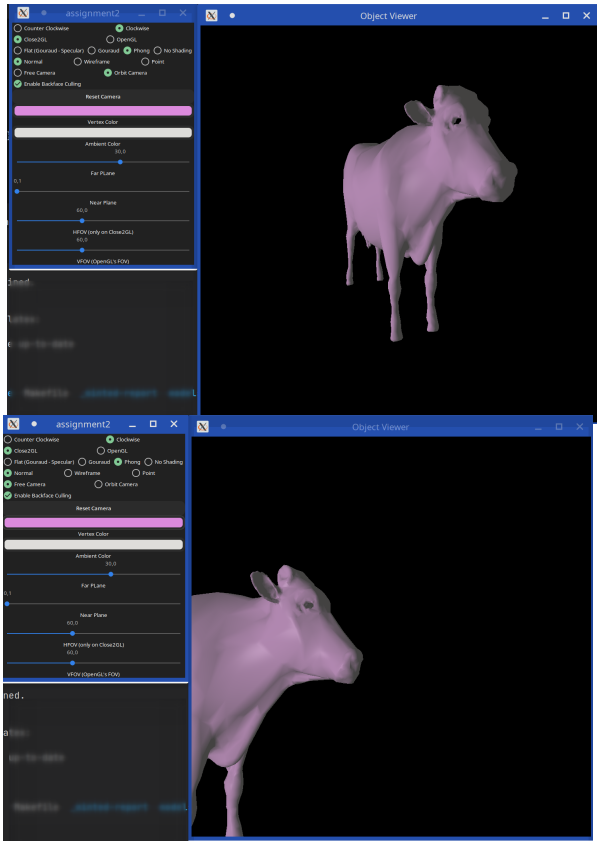

Figure 2: The two types of camera: Orbiting (above), Free (below).

2.2.1 View

For the View matrix, I only need the camera position (c), the *look at* position l and the $\vec{up}$ as input. From them, we can derive the CCS by computing cross products:

$$
\begin{aligned}
n &= l - c \\
u &= \vec{up} \times n \\
d_x &= -(u \cdot c) \\
d_y &= -(\vec{up} \cdot c) \\
d_z &= -(n \cdot c) \\
V &= \begin{bmatrix} u_x & u_y & u_z & d_x \\ \vec{up}_x & \vec{up}_y & \vec{up}_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}
$$

Of course, the process of creating an oribiting and free camera is the same as in the previous assignment. This is shown in fig. 2.

2.2.2 Projection

The Projection matrix requires the *field-of-view* parameters and the far/near planes. To calculate it the same way it is in **OpenGL**, we also need an aspect ratio, which will

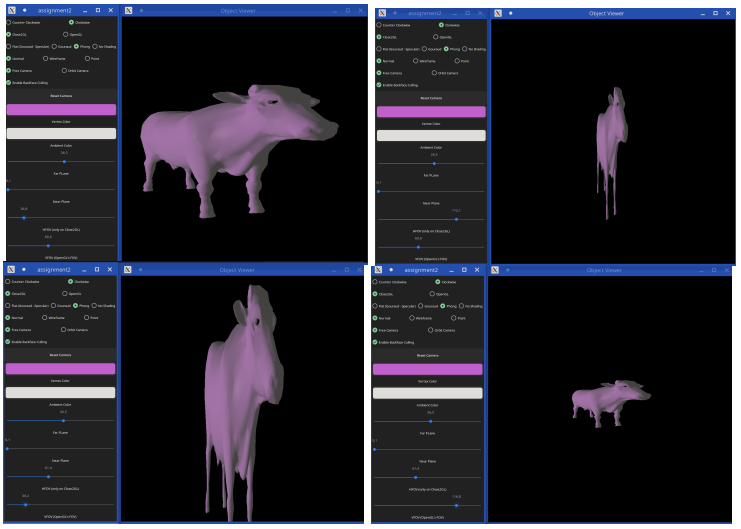

Figure 3: Low FOV (left), High (right), Hfov manipulation (above), Vfov manipulation (below).

be used in the *x-scaling* component of the matrix. The final matrix is calculated as follows [1] :

$$
\begin{aligned}
\text{aspect} &= 4/3 \\
S_h &= \frac{1}{\text{aspect} * \tan \frac{f_h}{2}} \\
S_v &= \frac{1}{\tan f_v/2} \\
P &= \begin{bmatrix} S_h & 0 & 0 & 0 \\ 0 & S_v & 0 & 0 \\ 0 & 0 & \frac{-(\text{far}+\text{near})}{\text{far}-\text{near}} & -1 \\ 0 & 0 & \frac{2*\text{far}*\text{near}}{\text{far}-\text{near}} & 0 \end{bmatrix}
\end{aligned}
$$

Modifying these attributes can generate different images, stretching and flatteining their aspect as well as changing the perspective strength fig. 3.

2.2.3 Model

I won't go into detail on the Translation T and Scaling S matrices since they are very basic. We construct the model matrix by doing $M = S * T$.

2.3 Culling and Clipping

Before finalizing the process and creating the normalized volume coordinates, we need to make sure there won't be any divisions by 0. We do so by filtering out the triangles with points behind the near plane. We do the same for the ones in the far plane fig. 4.

In the usual sequence of events, we would also need to clip (recalculate the polygons that are partially outside the frustum) the triangles. Here, however, we'll just remove the triangles that are **completely** outside of the

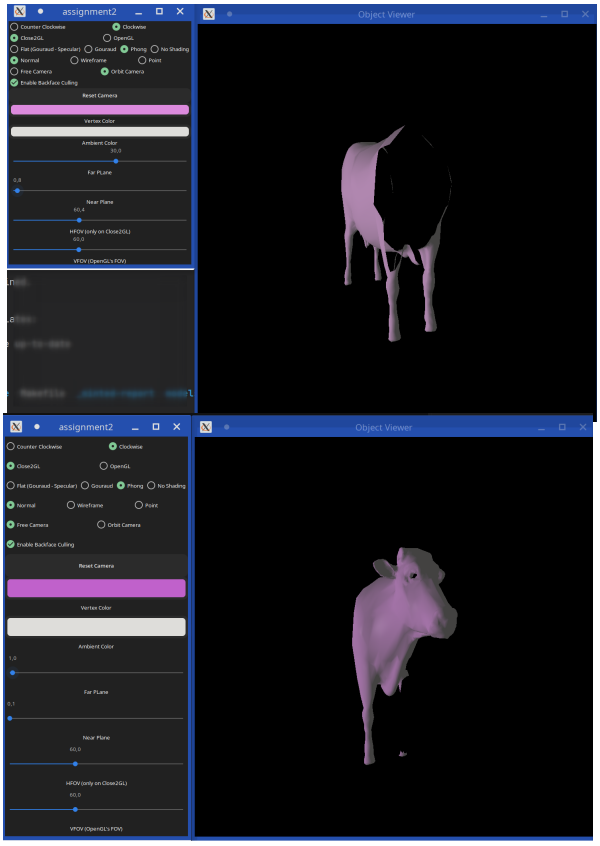

Figure 4: Near plane (above), Far plane (below).

`[-1, -1, -1] -> [1, 1, 1]` space, leaving the clipping to the GPU.

Another important part of the triangle filtering is *Backface Culling*. Here, we'll make sure that all triangles going to be rasterized are *front-facing*. To do so, we need to move the coordinates into window space, *i.e.* dividing by w and moving $[-1, 1] \rightarrow [0, 1]$ for both x and y coordinates, and to execute the simple arithmetic operation described in [2]. We can see the effects of switching winding orders in fig. 5.

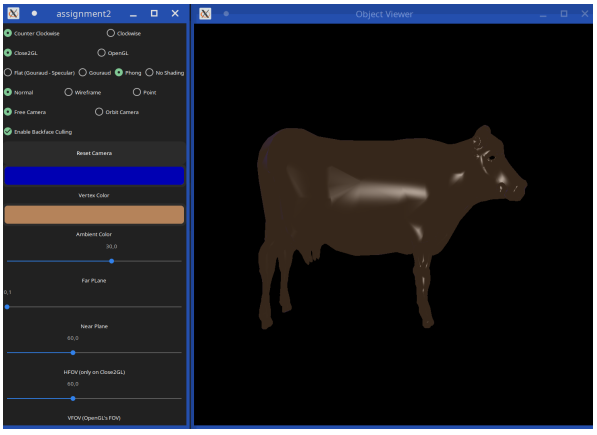

Figure 5: Rendering a cow with the incorrect winding order.

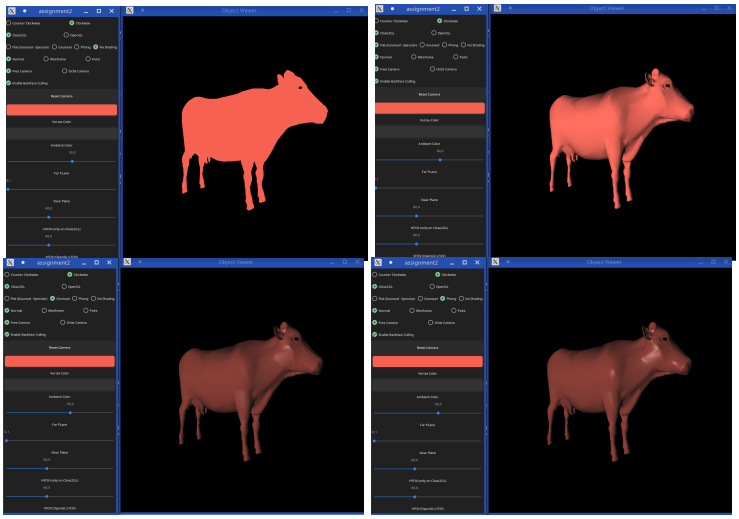

Figure 6: No shading (Top left), flat diffuse shading (Gouraud AD - Top right), Gouraud shading (Bottom left) and Phong shading (Bottom right)

2.4 Sending Data to Shaders

As we now have a set of xy coordinates that are ready to be rasterized, we can just send them over as we did to the regular positions in **OpenGL**. Here, we'll just change the amount of data that we are sending over. This makes it so that there is a possibility to send 0 bytes to the vertex buffer. It seems that this causes a `Segmentation Fault`, so if it's 0, I just clear the screen. We also filter the triangle normals in the previous step, which we send the same way as in *assignment 1*.

3 Shading

3.1 Gouraud Shading

Gourad shading is very simple. Considering we already had *Phong* implemented, we just needed to copy the code into the *vertex shader*, which would now define the final code of all the interpolated fragments.

3.2 Retrieving data from *Close2GL* Vertex Data

The described above and all the other shading options require that we retrieve some information from the normalized volume coordinates sent over to the GPU. Thus, we also send the $P * V * M$ matrix alongside M. With both these values, we can compute the original world coordinates ($M * (P * V * M)^{-1} * c$, where c is the normalized camera coordinate) of a vertex/fragment and compute the necessary light reflections, etc.

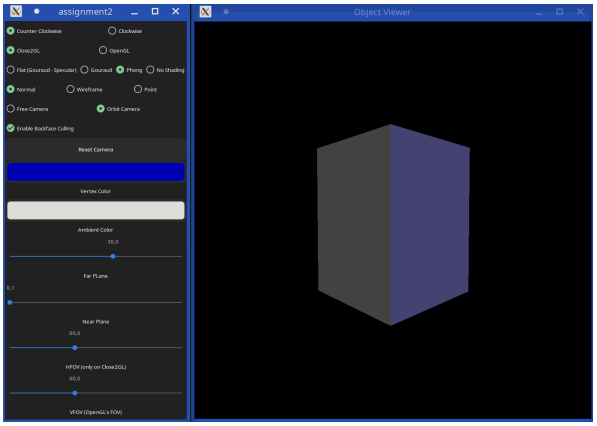

Figure 7: Rendering a cube!

4 Results & Conclusions

4.1 Efficiency

I added a Framerate display in the terminal during the execution of the application. There were a few things that can be considered from it:

- The OpenGL execution had an almost constant **60FPS** framerate, ranging in the interval $[58, 75]$.
- In contrast with the OpenGL execution, the Close2GL executed most of its operations in the CPU, which would, intuitively, lower its efficiency. However, we can get a decent framerate, ranging from **50FPS** to **60FPS**.
- It is once we disable Close2GL's *Backface Culling* that our efficiency decreases significantly, reaching **11FPS**!

It's worth mentioning that these observations were made after I optimized the execution switching, keeping an untouched vertex and normal values array so that I wouldn't have to reload the 3D object file everyframe. So believe me when I say this: It was slow!

4.2 Observations

Before concluding, I also wanted to point out that we can still:

- Load arbitrary objects (With inverted *z-axis* normals) fig. 7;
- Rescale window fig. 8

Also, all the images were generated with my implementation of *Close2GL*. There was no need to add an image with the OpenGL implementation as there didn't seem to have any difference except for efficiency/framerate purposes.

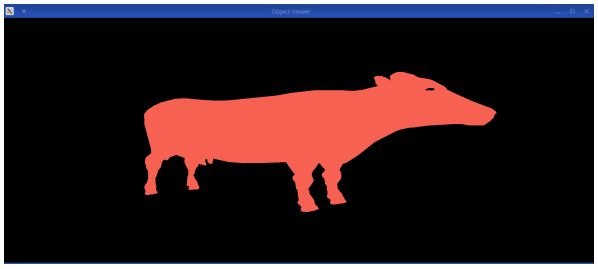

Figure 8: Rescaled a flat shaded cow!

4.2.1 Possible bugs

- If we toggle the backface culling a few times, even the OpenGL execution can get slower, which is a definite sign of memory leak.
- There are times in which no vertex is supposed to be rendered, but the vertex buffer still sends some data into the vertex shader. There is probably an edge case regarging either the frustum culling or the conditions to force clean the screen (done when we have 0 vertices to be shown).

4.3 Final Thoughts

This was a very interesting assignment! I feel like I am starting to understand how things are supposed to happen behind the curtains. Also, learning GTK and making it interact with the OpenGL environment was a very cool challenge. There are a few bugs, but they didn't really keep me from completing all the items in the specification and I intend to work them out in the next assignment.

References

[1] The perspective and orthographic projection matrix (building a basic perspective projection matrix). `https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/building-basic-perspective-projection-matrix`. (Accessed on 09/08/2021).

[2] Dave Shreiner, Bill The Khronos OpenGL ARB Working Group, et al. *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*. Pearson Education, 2009.