CMP143 - Computer Graphics: Programming Assignment N 2

Close2GL without Rasterization: Transforming and projecting vertices

Objective: The main objective of this assignment is to learn how to implement some of the major steps of the geometry-based rendering pipeline . In this assignment, a substantial portion of OpenGL will be implemented in software, and we will call it Close2GL. In the following paragraphs a brief but detailed explanation about what is expected for this assignment is shown:

- **Read and display arbitrary geometric models:** represented as triangle meshes. Once you read the objects, these should be displayed in the center of the window.
- **Translate the virtual camera:** along its own axes (not along the world coordinate system axes).
- **Translate the virtual camera while looking at the center of the object:** in the same way that the previous functionality the translation should be along its own axes (of the virtual camera).
- **Rotate the virtual camera:** in the same way along the axes of the virtual camera.
- **Reset the camera to its original position:** it means centered inside the window.
- **Culling orientation:** support for rendering the objects using its vertices oriented in Clockwise and Counter Clockwise direction.
- **Near and far clipping planes:** Support for changing the near and far clipping planes and support to change vertical and horizontal field of view.
- **RGB:** support for interactively change the colors of the models, applying a single RGB color to all triangles in the model.

Additionally to the Close2GL features mentioned above (and already implemented for OpenGL), some improvements will be developed for OpenGL rendering, specifically we will add some shading capabilities:

- Gouraud Shading (ambient + diffuse)
- Gouraud Shading (ambient + diffuse + specular)
- Phong Shading(ambient + diffuse + specular)

Some extra points will be provided if the implementation in vertex and fragment shaders are provided avoiding creation of multiple shaders or using if statements. Finally, in the following sections we will explain in a detailed manner how the core functionalities mentioned above were achieved.

CMP143 - Computer Graphics: Programming Assignment N 2

Project Structure

In the image below, we show the project structure which basically consists of a directory called data which contains the input text files with the object vertex information. Another directory called include which contains some of the libraries needed: GLFW, glad and GLM (for matrix operations). Additionally a directory called src which contains the actual implementation in the main.cpp file and the GLSL shaders definition (specific vertex and fragment shaders for OpenGL and Close2GL). Finally in the main directory, a file called Makefile which contains the instructions to the program to be compiled and executed.

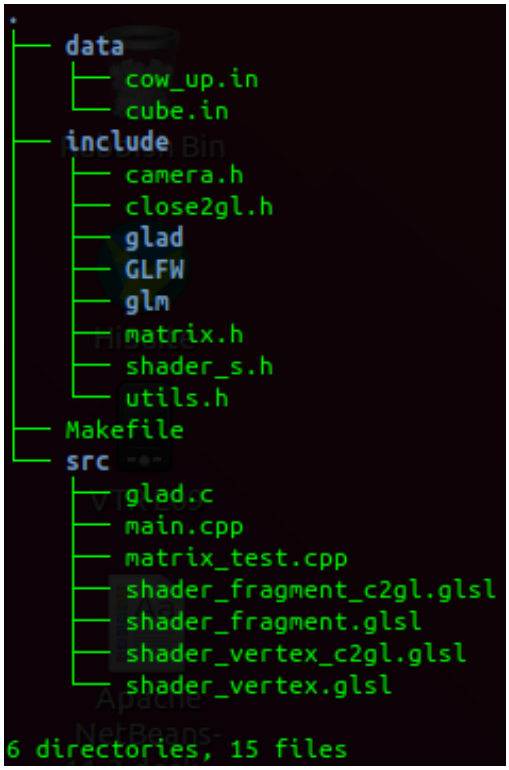

Figure 1: Project structure: directories and files

Graphical User Interface

In order to achieve a program in which the user can change parameters and see the result interactively a graphical user interface is needed. **The Nanogui widget** [1] was choosen for this task due to his minimalistic approach and also because it is oriented to work with OpenGL. In the figure below an screenshot of the program running is shown, as we could

[1] https://github.com/wjakob/nanogui

see the graphical interface allow us to change some parameters using buttons, color pickers and text. Additionally, NanoGUI provides some callbacks in order to handle every single keyboard and Mouse motion events.

Figure 2: Screenshot of how program looks like with NanoGUI

NanoGUI Installation

In order to install NanoGUI we need to perform some steps which are listed in the following lines:

1. We need to install cmake since nanogui uses it to compile its source code and additionally we need to install the mesa libraries. In order to accomplish that we need to execute the following command:

```
apt-get install cmake xorg-dev libglu1-mesa-dev
```

2. We need to clone the git repository of the widget with the recursive option, since it needs of some other components such as Eigen (to perform matrix operations, altough we will use glm for matrix operations), glfw, glad, etc. In order to do that we need to execute:

```
git clone --recursive https://github.com/wjakob/nanogui.git
```

3. Perform source code compilation with cmake according to the nanogui documentation[2]
4. Perform make and make install according to the following lines:

```
sudo make
sudo make install
```

5. Finally, use the command listed below in order to load the recently created library into the system (another option is to reboot the computer)

```
sudo ldconfig
```

Matrix Operations

- **OpenGL** In the case of OpenGL, the matrix operations will be calculated using the OpenGl Mathematics library (GLM)[3] according to the recommendations for the first assignment. Since GLM is a header only C++ mathematics library we only need to download the source code and put it in the includes directory of our project in order to use it in our main.cpp file of source code. Finally, all other packages needed to implement the following work are defined in the First Assignment when we render the same object using OpenGL.
- **Close2GL** In the case of the Close2GL implementation, we used glm only as a container for vector and matrices datatype but all the matrix methods needed for the assignment were implemented manually and can be found in the header file called matrix.h inside include directory. Chosing glm::vec and glm::mat as datatypes for vectors and matrices have no relevance for manual implementation but this decision makes more intuitive the coding process.

Implementation

1. **Reading and display arbitrary geometric models:**
    - **OpenGL**: As was reported in the assignment 1 report, a function called **readFile** was implemented. This function receives the name of the file to be read and returns the ID of the VAO created to store the VBO with the file information in it. Additionally, this file updated some global variables to manage min and max values for each of the three dimensions x,y,z. Previous shaders used for OpenGL were modified in order to perform shading and will be presented in the shading section.

[2] https://nanogui.readthedocs.io/en/latest/compilation.html#compilation
[3] https://glm.g-truc.net/0.9.9/index.html

- **Close2GL**: In the same way that OpenGL, a function called **readFile_close2gl** was implemented. The only difference with the previous mentioned function is that this one returns an std::vector of **Triangle_c2gl** (a custom struct created to save information about vertexs, normals, color,etc of the object). Additionally, specific vertex and fragment shaders were created for Close2GL called **shader_vertex_c2gl.glsl** and **shader_fragment_c2gl.glsl**, these files are shown in Code Snippet 1 and 2.

Code Snippet 1: Source code for Vertex Shader - Close2GL

```
#version 330 core

layout (location = 0) in vec2 vert;

void main(){
    gl_Position = vec4(vert, 0.0f, 1.0f);
}
```

Code Snippet 2: Source code for Fragment Shader - Close2GL

```
#version 330 core

uniform vec4 rasterizer_color;
out vec4 FragColor;

void main(){
        FragColor = rasterizer_color;
}
```

In the case of Close2GL, since all the processing before rasterization is implemented in software, we only to pass the tranformed vertex x and y coordinates to the vertex shader making it extremely simple. On the other hand, for OpenGL, we need to pass the model, view and projection matrix to the shader as uniform variables. In both cases, we pass the color to the fragment shader using another uniform variable. For both cases, in order to show the object centered in the window, we perform a translation of the model matrix to the center of the object and then in the view matrix we set the camera position in a point that allows us to see the object (difference between the maximum and minimum coordinate of all three axes). Finally, we set the projection matrix with an initial angle of field of view of 45.0f and set the near plane to 1.0f and the far plane to 3000.f as requested in the assignment definition. The image below shows the cow object centered on the window after loading it using the button **Load Cow Object** for both OpenGL and Close2GL implementation:

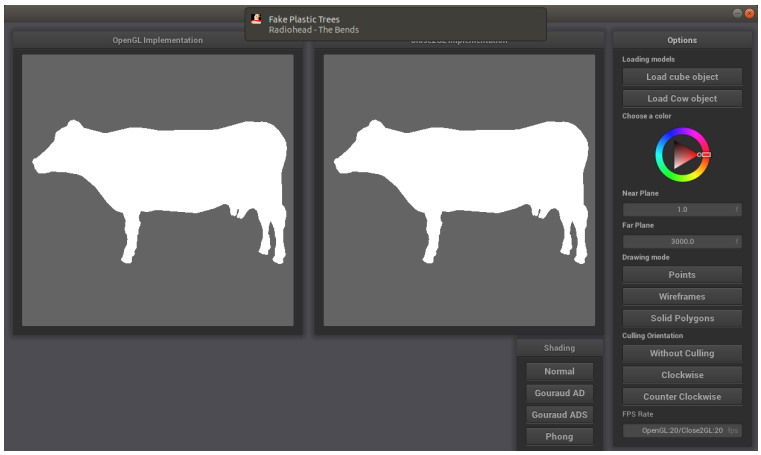

Figure 3: Loading object at the centre of the window.

2. **Translate the virtual camera along its own axes:** For this part of the implementation we reused the keyboar inputs implemented in the first assignment. We used the following keys to perform translations in an specific direction:

   - **W key:** In order to perform camera Translation along the n axis simulating a close up operation to the object.
   - **S key:** In order to perform camera Translation along the n axis simulating that the camera is going far away the object.
   - **D key:** In order to perform camera Translation along the u axis going to the right.
   - **A key:** In order to perform camera Translation along the u axis going to the left.

   In the following images, we show screenshots of the program running after the execution of the camera translation operations listed above:

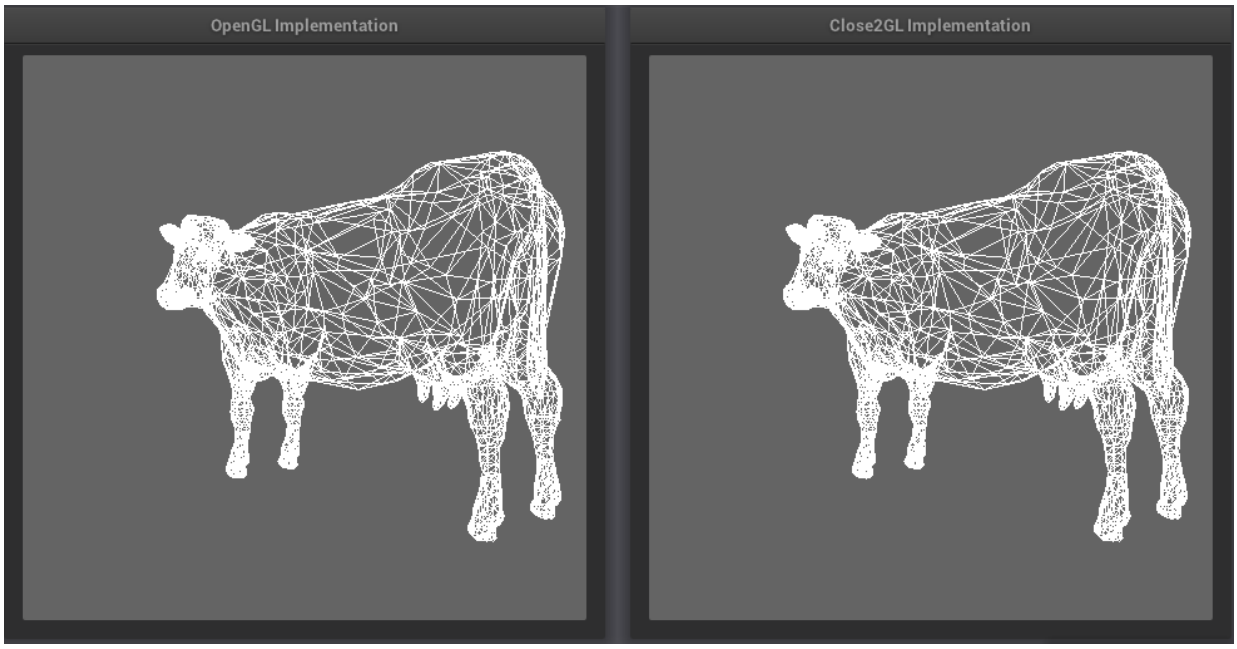

Figure 4: Camera translated in the x axis towards right using keys A or D.

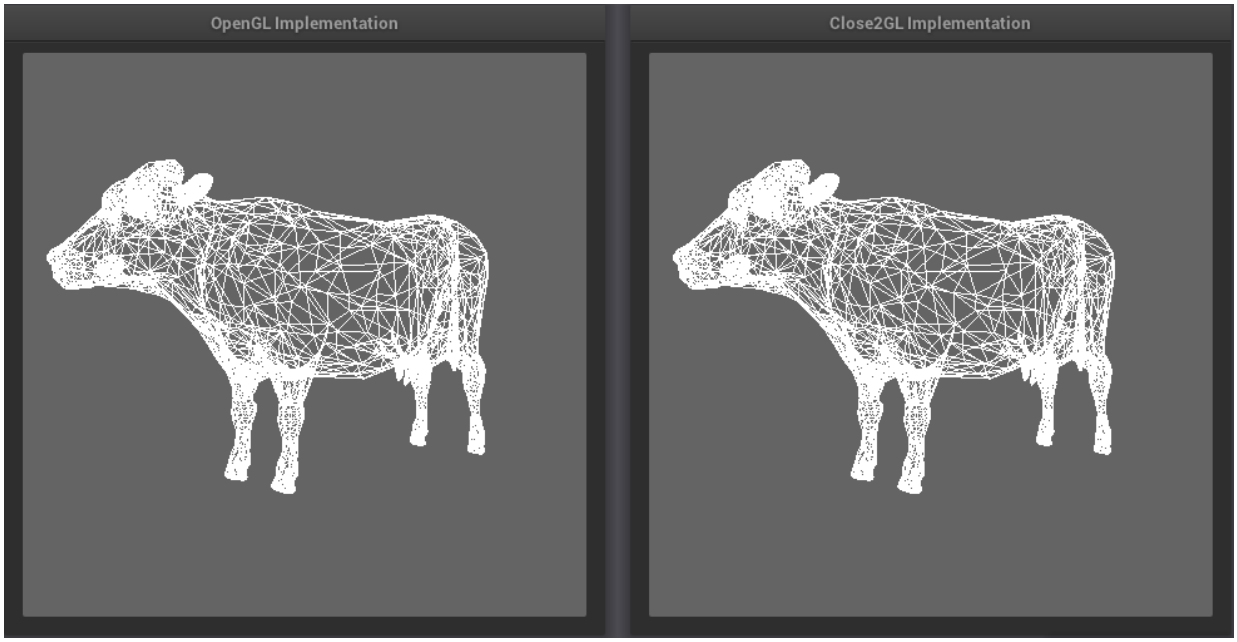

Figure 5: Camera translated in the x axis towards left using keys A or D.

In order to implement the functionalities exposed above we initialize the u, v, and n vectors as unit vectors. We create two header files named camera.h (camera class for OpenGL) and close2gl.h (camera class implementation for Close2GL). These classes implement similar functions in order keyboard input, as well as other core functions like getting LookAt, projection and viewport matrix.

In the following code snippet we will show how the new position is calculated when press the movement keys in the case of Close2GL class using manually implemented matrix functions, keep in mind that the classics u,v and n vector were replaced by right, up and front respectively:

Code Snippet 3: Source code for updating cameraPos when pressed the W key

```
void processRotation(camera_movement movement_direction){
        if (movement_direction == FORWARD){
                position -= movementSpeed_FB * front;

                //Updating distance Projection Sphere
                distanceProjSphere = matrix.lengthv3(position);
        }else if (movement_direction == BACKWARD){
                position += movementSpeed_FB * front;

                //Updating distance Projection Sphere
                distanceProjSphere = matrix.lengthv3(position);
        }else if (movement_direction == RIGHT){
                //Calculating update for right
                right = matrix.normalizev3(matrix.crossProduct(up, front));

                //Calculating new Camera position
                position -=  right * movementSpeed;

                //Updating front
                front = matrix.normalizev3(position);

        }else if (movement_direction == LEFT){
                //Calculating update for camera right
                right = matrix.normalizev3(matrix.crossProduct(up, front));

                //Calculating new Camera position
                position += right * movementSpeed;

                //Updating front
                front = matrix.normalizev3(position);

        }
}
```

As we can see what is basically is done when updating the cameraPos variable is to calculate the direction in which we want to update the camera Position (taking a

normalized cross product of the other two axis values) and scale this cameraPosition by a determined factor. Since we not scale the objects when they are rendered, we will need different scale factors for each object in order to make the movement seems natural. Additionally, as can be observed we always update the cameraFront in the negative direction of the cameraPos, this is done in order to maintain the camera looking at the centre of the object while translating. Finally when we translate in the z axis, we do not need to update the cameraFront but as we can observed when we move forward or backward we update a variable called **distanceProjSPhere** to the length of the cameraPos vector, this variable is used to maintain the camera in a fixed distance to the centre of the object (ratio of an imaginary sphere when the object is set) while translating. Keep in mind that this offer additional functionalities since these movements could be achieved also in the next section using the mouse.

3. **Rotate the virtual camera:** In order to perform the rotation of the camera along its own axes, we use the mouse movement as an input of the direction of the movement. Basically we calculate pitch and yaw angle based on the current and last position of the mouse pointer and apply some arbitrary **sensitivity** value in order to make the movement faster (if value is greater) or slower (if the value is minor). The following code snippet shows how to the pitch and yaw angle are calculated.

Code Snippet 4: Calc of new pitch and yaw angles after a mouse movement

```
virtual bool mouseMotionEvent(const Vector2i &p, const Vector2i &rel,
int button, int modifiers){
    //If the mouse is moved for the very first time
    if(firstMouse){
        lastX = p.x();
        lastY = p.y();
        firstMouse = false;
    }
    float xoffset = p.x() - lastX;
    float yoffset = lastY - p.y(); // reversed since y go from bottom to top
    lastX = p.x();
    lastY = p.y();

    float sensitivity = 0.01f; // change this value to your liking
    xoffset *= sensitivity;
    yoffset *= sensitivity;
    yaw += xoffset;
    pitch += yoffset;

    // make sure that when pitch is out of bounds, screen does not get clipped
    if (pitch > 89.0f)
        pitch = 89.0f;
    if (pitch < -89.0f)
        pitch = -89.0f;

    //Setting new yaw and pitch values for opengl canvas
```

```
    mCanvasObject->camera.yaw = yaw;
    mCanvasObject->camera.pitch = pitch;

    //Setting new yaw and pitch values for close2gl canvas
    mCanvasObjectC2GL->close2gl.yaw = yaw;
    mCanvasObjectC2GL->close2gl.pitch = pitch;

    return true;

}
```

The code shown above is executed every time the mouse is moved in any direction and updates the yaw and pitch angles for both classes, Camera and Close2GL and finally recalculates the view matrix by applying rotation in the u and v axis, the rotation performed is shown below for both cases OpenGL and Close2GL:

Code Snippet 5: Performing actual rotation over the view matrix - OpenGL

```
view = glm::rotate(view, pitch, glm::vec3(-1.0f, 0.0f, 0.0f));
view = glm::rotate(view, yaw, glm::vec3(0.0f, 1.0f, 0.0f));
```

Code Snippet 6: Performing actual rotation over the view matrix - Close2GL

```
//Performing rotation
view = matrix.rotate(view, close2gl.pitch, glm::vec3(-1.0f, 0.0f, 0.0f));
view = matrix.rotate(view, close2gl.yaw, glm::vec3(0.0f, 1.0f, 0.0f));
```

In the following pictures we will show two screenshots showing the program performing rotation operations:

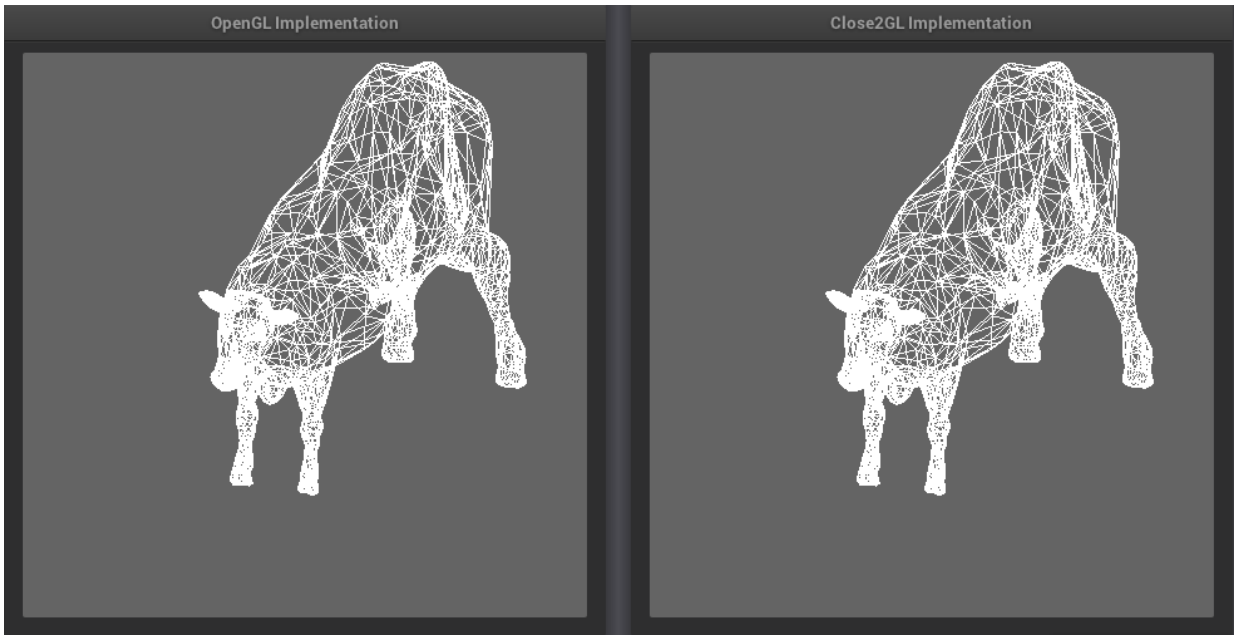

Figure 6: Example of camera rotation showing the cow doing some anger movements observed from the back.

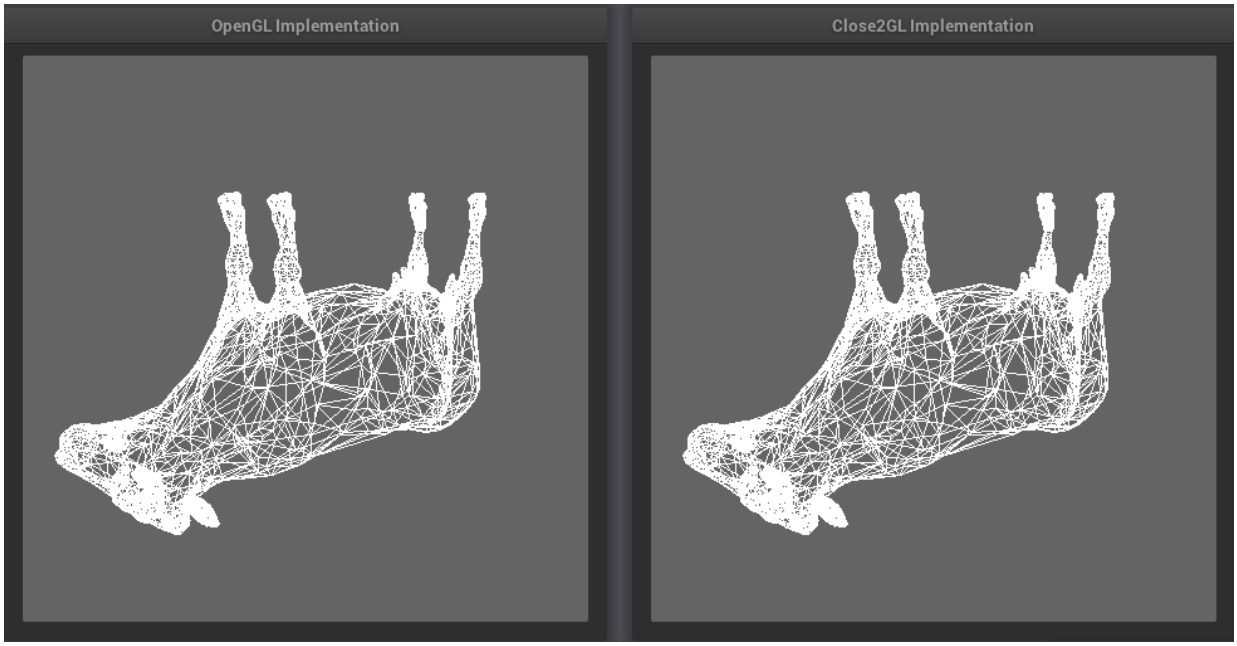

Figure 7: Example of camera rotation showing the cow a little bit more relaxed lying on his back.

4. **Reset the camera to its original position:** In this case, we reset both cameras

(OpenGL and Close2GL) to its original position by clicking on the buttons **Load cube object** and **Load Cow object** since they performed the initial operations indicated in the item 1 of this section. After button is pressed, it loads the corresponding object at the centre of the window, just like at the beginning.

5. **Culling orientation:** The GUI provides two buttons to change culling orientation (clockwise or counter clockwise). This feature is achieved in the **OpenGL** parts by using the glFrontFace OpenGL method by setting the constants GL_CW for clockwise and GL_CCW for counter clockwise. For the case of **Close2GL**, we perform backface culling verification after we have performed perspective division, i.e. we perform backface culling verification in NDC coordinates. To do backface culling verification, we get the first vertex of each triangle, and get the edges (vectors) of the triangle that share this vertex and calculate the cross product between them. Then, we choose what triangles we are going to draw based on the z component value of this cross product (perpendicular vector to our triangle) and the culling orientation chosen. In the following code snippet we will show the part of the code that implement backface culling verification on Close2GL:

Code Snippet 7: Performing backface culling verification on Close2GL

```
//Performing backface culling
temp.clear();
for(int i=0; i<clipped_triangles.size(); i++){
    glm::vec3 vecBA = clipped_triangles[i].v1 - clipped_triangles[i].v0;
    glm::vec3 vecCA = clipped_triangles[i].v2 - clipped_triangles[i].v0;

    glm::vec3 bfvec = matrix.crossProduct(vecBA, vecCA);

    if(culling_orientation == 1){
        //clockwise
        if(bfvec.z < 0){
            temp.push_back(clipped_triangles[i]);
        }
    } else if (culling_orientation == 2){
        if(bfvec.z > 0) {
            //counter clockwise
            temp.push_back(clipped_triangles[i]);
        }
    } else {
        temp.push_back(clipped_triangles[i]);
    }

}

clipped_triangles = temp;
```

In the following figures we will show the graphical difference while chosing Clockwise or Counter Clockwise Orientation.

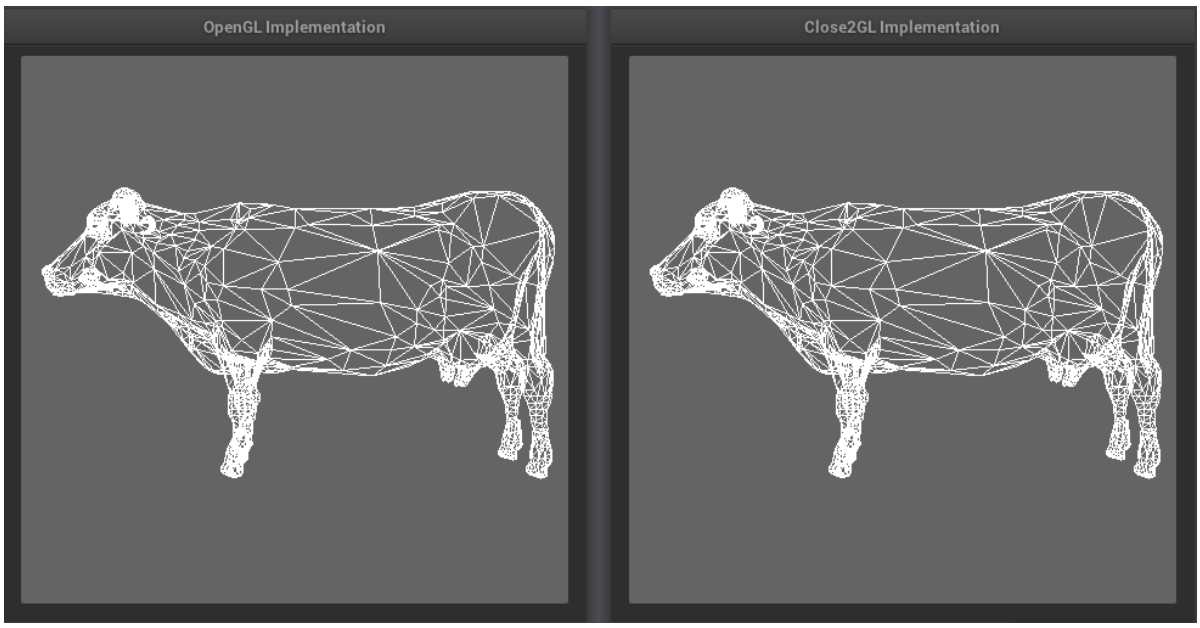

Figure 8: Example of the loaded object using clockwise orintation in both implementation OpenGL and Close2GL.

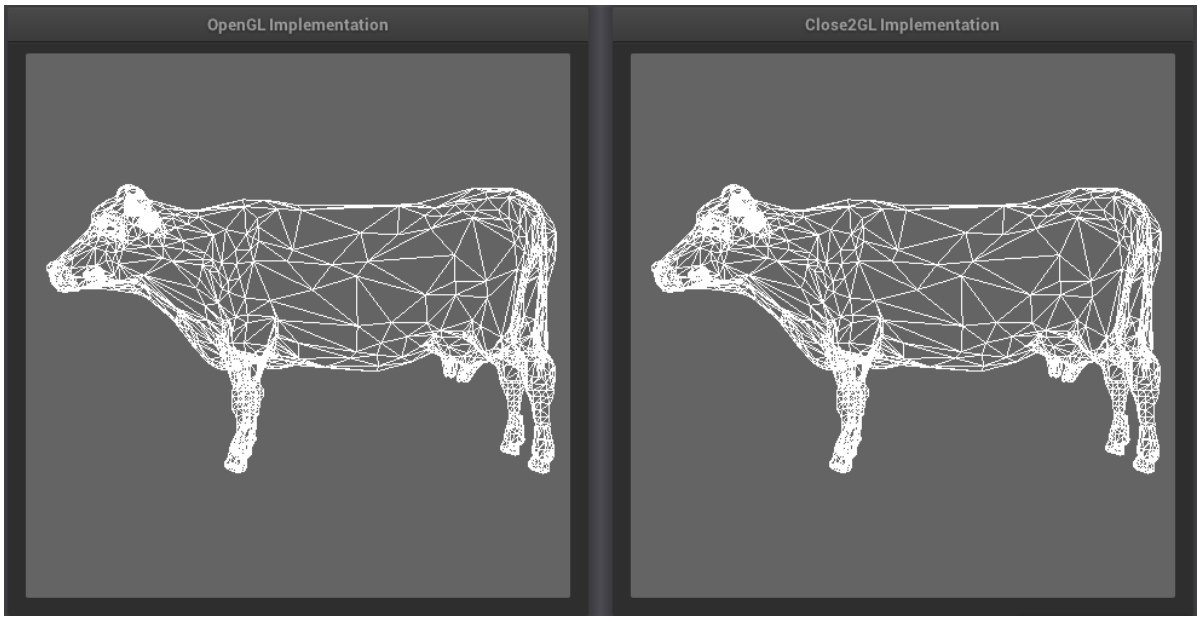

Figure 9: Example of the loaded object using clockwise orintation in both implementation OpenGL and Close2GL.

As we can see the final object being rendered in Wireframes mode (to manage to see differences and what is actually drawn) differs when we use clockwise or counter clockwise orientation. When activating backface culling on OpenGL all the faces that are not front-faces are discarded, so when we set clockwise or counter clockwise orientation we are in fact indicating if triangles would be front or back faces and then the backface culling discard different triangles when different orientation is chosen. It is important to mention that both implementations (OpenGL and Close2GL) matches perfectly when they are drawn.

6. **Near and far clipping planes:** Defining near and far clipping planes we are actually defining a parallelepiped that defines a clipping space. Everything that is inside this parallelepiped is actually drawn and all the vertices that not belongs to this space are discarded. We set the near and far plane in the projection matrix with initial values of 1.0f for near plane and 3000.0f for far plane according to the assignment specification. In the graphical user interface we could modify this values for near and far plane by modifying the values there, we will show an example of a reduced clipping space and the final object being rendered (clipped) for OpenGL and Close2GL setting the far plane in 1500.0f:

Figure 10: Object loaded with far plane set to 1500.0f

As we can see the object is rendered but it looks incomplete due to some of its vertices are not in the clipping space defined by near and far plane, if we rotate or translate the camera we probably would see other parts of the object but it would look incomplete anyway. Note that both objects in OpenGL and Close2GL looks exactly equal since the same parameters are being applied for both of them.

7. **RGB:** In our graphical user interface we provide a color picker, this color picker get the RGB values of the chosen color and pass it to the fragment shader as uniform

variable in both cases (OpenGL and Close2GL). This allows us to change the color of the object being rendered anytime we want. Shading and illumination models are performed over the RGB color chosen using this option. We provide an screenshot showing the choose color with the load object in the centre for both implementations (OpenGL and Close2GL):

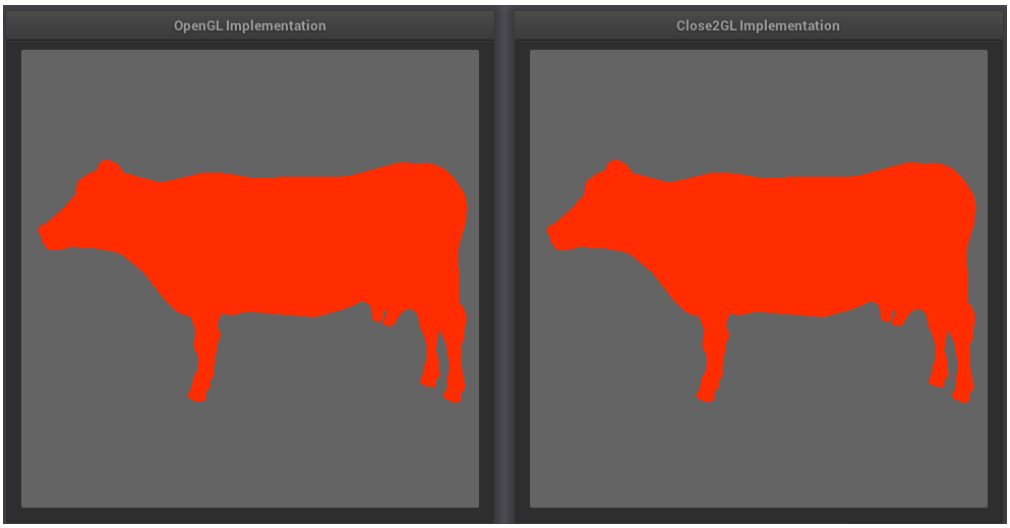

Figure 11: Example screenshot of the object being rendered in a color chosen from the color picker for both implementations (OpenGL and Close2GL).

8. **Drawing mode:** We provide in the GUI the option to change the drawing mode between POINTS, WIREFRAMES or SOLID POLYGONS which is achieved just calling the function GLDrawArrays of OpenGL with different primitives: GL_POINTS, using glPolygonMode for wireframes and finally GL_TRIANGLES. In the case of Close2GL, we perform the rendering according to the drawing mode chosen over the final clipped triangles, i.e. the remaining triangles after clipping, backface culling and perspective division. We show examples of WIREFRAMES and POINTS drawing modes in the figure below:

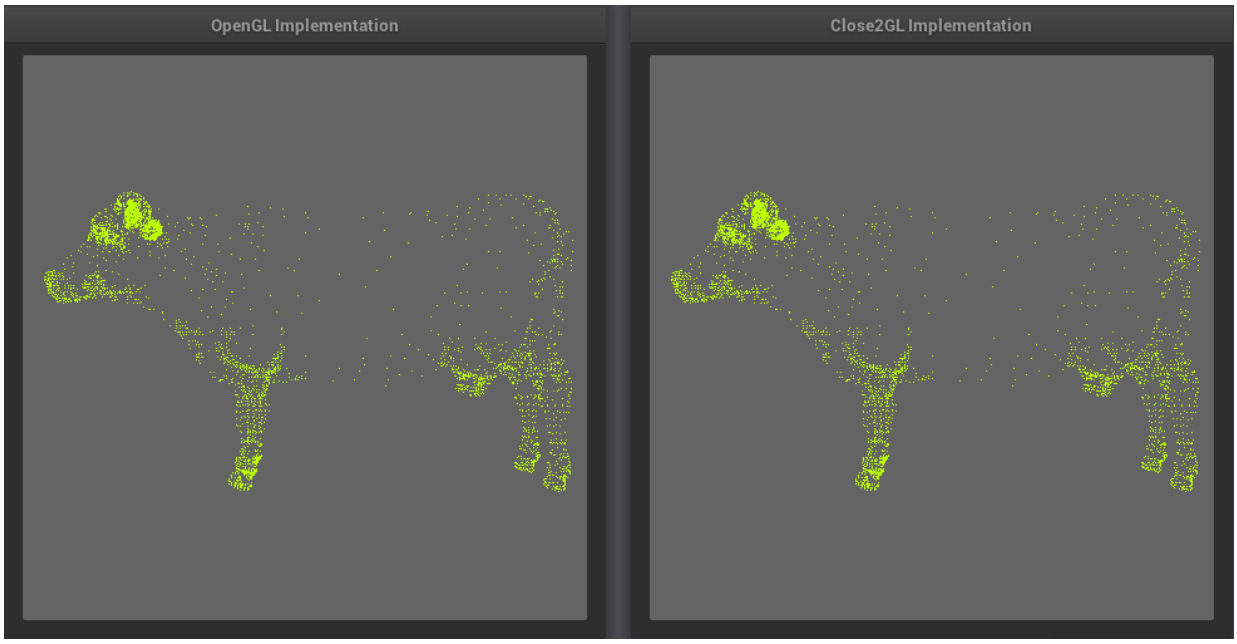

Figure 12: Object being rendered in POINTS drawing mode.

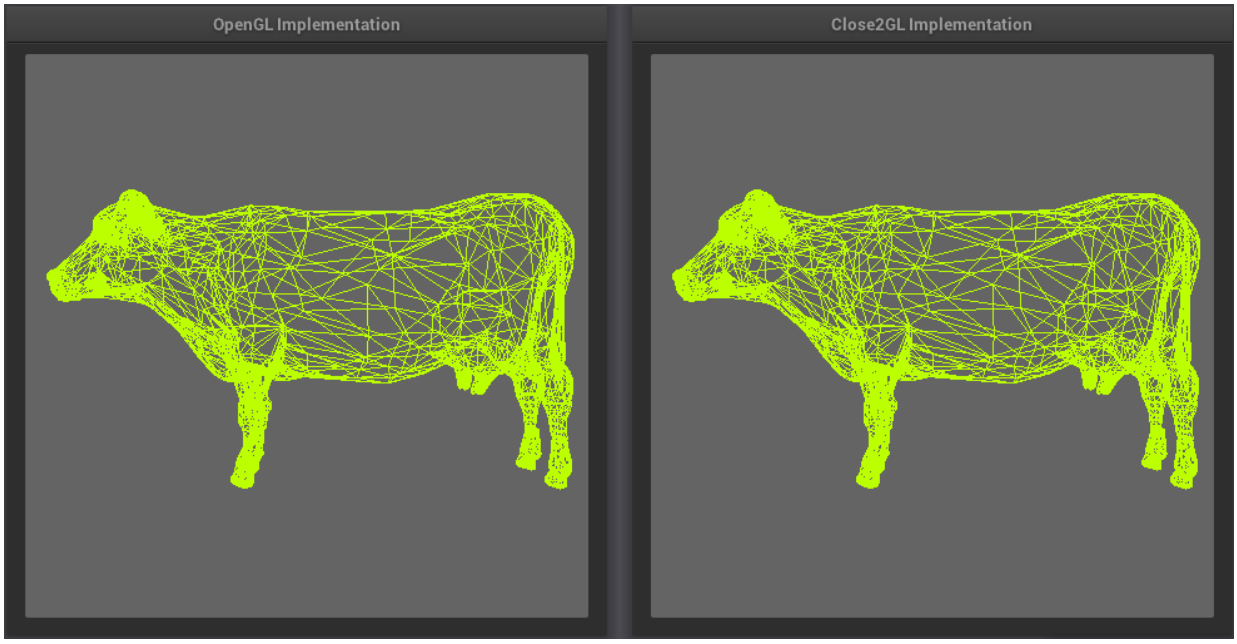

Figure 13: Object being rendered in POINTS drawing mode.

Note that both implementations (OpenGL and Close2GL) looks exactly the same for both drawing modes.

9. **Zooming effects:** We additionally implement some zooming effects that could be

activated using the scroll button of the mouse. If the scroll is activated to the front, there will be a zooming effect (it seems like object is closer) and if the scroll is activated to the back,there will be a reverse zooming effect (it seems like object is far away from the camera). This zooming effect is achieved by changing (increasing or decreasing) the field of view (the angle of the field of view in fact). This was implemented as shown in the following code snippet:

Code Snippet 8: Zooming effects on Scroll Button event

```
virtual bool scrollEvent(const Vector2i &p, const Vector2f &rel){
        if (fov >= 1.0f && fov <= 45.0f)
            fov -= rel.y();
        if (fov <= 1.0f)
            fov = 1.0f;
        if (fov >= 45.0f)
            fov = 45.0f;
    }
```

As in the previous items, this gui function provides information for both implementation. So, activating zooming will be performed over both implementations. Similar effects will be achieved using the W (for zooming) and S (for moving away) key, but in this case we are actually translating the camera.

10. The complete code used for Close2GL drawing implementation is shown in the following code snippet.

Code Snippet 9: Drawing function implementation for Close2GL

```
virtual void drawGL() override {

    double currentTime = glfwGetTime();
    ++framesPerSecond;

    if(currentTime - lastTime > 1.0f){
        lastTime = currentTime;
        fpsrate_close2gl = framesPerSecond;
        framesPerSecond = 0;
    }

    if (model_used == 0){
        glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);
    }else{
        //Loading the shader program
        custom_shader.use();

        //create transformations
        glm::mat4 model          = glm::mat4(1.0f);
```

```
glm::mat4 view          = glm::mat4(1.0f);
glm::mat4 projection    = glm::mat4(1.0f);

float center_x = (g_min_X+g_max_X)/2.0f;
float center_y = (g_min_Y+g_max_Y)/2.0f;
float center_z = (g_min_Z+g_max_Z)/2.0f;

//Translating the model to the origin (0,0,0) using manually implemented
//matrix class.
glm::mat4 trans = matrix.translate(glm::vec3(-center_x, -center_y,
                                                    -center_z));
model = matrix.multiply_matrix(trans, model);

//Normalizing and scaling cameraPos by distanceProjSphere
close2gl.position = matrix.normalizev3(close2gl.position)*
                                    (close2gl.distanceProjSphere);

if(firstMouse){
    view = close2gl.getLookAtMatrix();
} else {
    view = close2gl.getLookAtMatrix();

    //Performing rotation
    view = matrix.rotate(view, close2gl.pitch,
                         glm::vec3(-1.0f, 0.0f, 0.0f));
    view = matrix.rotate(view, close2gl.yaw,
                         glm::vec3(0.0f, 1.0f, 0.0f));

}

projection = close2gl.getProjectionMatrix(glm::radians(close2gl.fov),
                          ((float)this->width())/this->height(),
                            g_near_plane,
                            g_far_plane);


//Calculating the model view projection matrix
glm::mat4 modelViewProj = projection * view * model;

//assigning read_triangles to triangles vector in order to iterate over it.
triangles = read_triangles;

for (int i=0; i<triangles.size(); i++){

    triangles[i].v0 = matrix.transform_vector(triangles[i].v0,
                          modelViewProj);
    triangles[i].v1 = matrix.transform_vector(triangles[i].v1,
                          modelViewProj);
    triangles[i].v2 = matrix.transform_vector(triangles[i].v2,
                          modelViewProj);
```

```
}

//Clearing clipped_triangles
clipped_triangles.clear();

//Clipping (considering) only triangles inside the perspective volume
for(int i=0; i<triangles.size(); i++){
    //Getting w position of each vertex
    float w0 = abs(triangles[i].v0.w);
    float w1 = abs(triangles[i].v1.w);
    float w2 = abs(triangles[i].v2.w);

    if ( abs(triangles[i].v0.z) <= w0 &&
            abs(triangles[i].v1.z) <= w1 &&
            abs(triangles[i].v2.z <= w2)){
        clipped_triangles.push_back(triangles[i]);
    }

}

//Performing perspective division over the clipped triangles
and transforming them with viewport matrix
for(int i=0; i<clipped_triangles.size(); i++){
    clipped_triangles[i].v0 = clipped_triangles[i].v0 /
                                        clipped_triangles[i].v0.w;
    clipped_triangles[i].v1 = clipped_triangles[i].v1 /
                                    clipped_triangles[i].v1.w;
    clipped_triangles[i].v2 = clipped_triangles[i].v2 /
                                    clipped_triangles[i].v2.w;

}


//Performing backface culling
temp.clear();
for(int i=0; i<clipped_triangles.size(); i++){
    glm::vec3 vecBA = clipped_triangles[i].v1 - clipped_triangles[i].v0;
    glm::vec3 vecCA = clipped_triangles[i].v2 - clipped_triangles[i].v0;

    glm::vec3 bfvec = matrix.crossProduct(vecBA, vecCA);

    if(culling_orientation == 1){
        //clockwise
        if(bfvec.z < 0){
            temp.push_back(clipped_triangles[i]);
        }
    } else if (culling_orientation == 2){
        if(bfvec.z > 0) {
            //counter clockwise
```

```
                temp.push_back(clipped_triangles[i]);
            }
        } else {
            temp.push_back(clipped_triangles[i]);
        }

    }

    clipped_triangles = temp;

    //Getting vertices in a 1d array to pass it to the shader
    float vert[6*clipped_triangles.size()];

    for(int i=0; i < clipped_triangles.size(); i++){
        vert[6*i]   = clipped_triangles[i].v0.x;
        vert[6*i+1] = clipped_triangles[i].v0.y;
        vert[6*i+2] = clipped_triangles[i].v1.x;
        vert[6*i+3] = clipped_triangles[i].v1.y;
        vert[6*i+4] = clipped_triangles[i].v2.x;
        vert[6*i+5] = clipped_triangles[i].v2.y;
    }

    // Dealing with VAOs and VBOs
    unsigned int VBO, VAO;
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    // binding the Vertex Array Object.
    glBindVertexArray(VAO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    //Putting buffer data
    glBufferData(GL_ARRAY_BUFFER, sizeof(vert), vert, GL_STATIC_DRAW);
    //Position, # dimensions, data type, ##, 0, 0
    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, 0);
    //Setting 0 in position (according to the specification on the shader)
    glEnableVertexAttribArray(0);

    // ------------------ Actual Drawing ------------------ //

    if(drawing_mode == 1){
        glDrawArrays(GL_POINTS, 0, clipped_triangles.size()*3);
    } else if (drawing_mode == 2){
        glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );
        glDrawArrays(GL_TRIANGLES, 0, clipped_triangles.size()*3);
        glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );
    }
    else if (drawing_mode == 3)
        glDrawArrays(GL_TRIANGLES, 0, clipped_triangles.size()*3);

    // ------------------ Actual Drawing ------------------ //
```

```
        //Unbinding the VBO buffer
        glBindBuffer(GL_ARRAY_BUFFER, 0);


        // Passing only color as uniform to the fragment shader.
        //Retrieving the matrix uniform locations
        unsigned int colorLoc = glGetUniformLocation(custom_shader.ID,
                                    "rasterizer_color");

        // pass matrix uniform locations to the shaders
        glUniform4fv(colorLoc, 1, glm::value_ptr(this->color));

    }

}
```

Note that the implementation of the drawing function for Close2GL follows the same pipeline and structure that we used for OpenGL implementation (See report for the first assignment) but in this case the functions used for translation, rotation and matrix operations are provided by the class matrix defined in **include/matrix.h**. Additionally, setting the modelview and projection matrix is achieved using methods provided by the class Close2GL defined in **include/close2gl.h** according with the theory explained in classes. Basically the pipeline is as follows:

- Setting model, view and projection matrix with rotation updates for view matrix according to user interactions.
- Transform the vertexs using the modelViewProjection matrix.
- Clipping the triangles that are behind or in the same plane that the camera.
- Perform perspective division over the remaining triangles.
- Perform backface culling according to the orientation chosen (clockwise or counter clockwise).
- Actually drawing only the remaining triangles after backface culling.

11. **Shading**: We perform shading with the three shading types asked in the definition of this assignment only for the OpenGL part, the Close2GL implementation have no option to applied shading yet. The following shadings were implemented:

    - **Gouraud Shading (Ambient + Diffuse):** This shading is implemented in the Vertex shader since it is applied at each vertex.
    - **Gouraud Shading (Ambient + Diffuse + Specular):** This shading is also implemented in the vertex shader, and has similar implementation to the previous one but adding specular lightning calculations.

- **Phong Shading:** This shading was implement in the fragment shader since it is applied in a per fragment manner. Ambient, diffuse and specular lightning are calculated for this shading type.

In the case of both Gouraud shading types, the multiplication factor of the final color is calculated in the vertex shader and passed to the fragment shader which only multiply the choosen color (in the GUI) by this factor. In the case of the Phong shading the multiplication factor is calculated entirely in the fragment shader and also applied to the chosen color (in the GUI). In the following code snippets the updated vertex and fragment shader for OpenGL implementation are shown:

Code Snippet 10: Vertex shader for OpenGL implementation

```
#version 330 core

layout (location = 0) in vec3 vert;
layout (location = 1) in vec3 vert_normal;

out vec3 FragPos;
out vec3 Normal;
out vec3 FinalColor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform vec3 lightPos;
uniform vec3 lightColor;
uniform vec3 viewPos;

#define NORMAL 0
#define GOURAUD_AD 1
#define GOURAUD_ADS 2
#define PHONG 3

uniform int shading_type;

void main(){

        if(shading_type == NORMAL){
                gl_Position = projection * view * model * vec4(vert, 1.0);
        } else if(shading_type == GOURAUD_AD){
                vec3 position = vec3(model * vec4(vert, 1.0));
                vec3 Normal = mat3(transpose(inverse(model))) * vert_normal;

                //ambient
                float const_ambient = 0.5;
                vec3 ambient_light = vec3(const_ambient * lightColor);

                // diffuse
```

```
            vec3 norm = normalize(Normal);
            vec3 lightDir = normalize(lightPos - position);
            float diff = max(dot(norm, lightDir), 0.0);
            vec3 diffuse_light = diff * lightColor;

            FinalColor = ambient_light + diffuse_light;

            gl_Position = projection * view * model * vec4(vert, 1.0);

        } else if(shading_type == GOURAUD_ADS){
                vec3 position = vec3(model * vec4(vert, 1.0));
                vec3 Normal = mat3(transpose(inverse(model))) * vert_normal;

                //ambient
                float const_ambient = 0.5;
                vec3 ambient_light = vec3(const_ambient * lightColor);

                // diffuse
            vec3 norm = normalize(Normal);
            vec3 lightDir = normalize(lightPos - position);
            float diff = max(dot(norm, lightDir), 0.0);
            vec3 diffuse_light = diff * lightColor;

            // specular
            // this is set higher to better
            //show the effect of Gouraud shading
            float specularStrength = 1.0;
            vec3 viewDir = normalize(viewPos - position);
            vec3 reflectDir = reflect(-lightDir, norm);
            float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
            vec3 specular_light = specularStrength * spec * lightColor;

            FinalColor = ambient_light + diffuse_light + specular_light;

            gl_Position = projection * view * model * vec4(vert, 1.0);

        } else if (shading_type == PHONG){
                FragPos = vec3(model * vec4(vert, 1.0));
                Normal = vert_normal;

            gl_Position = projection * view * model * vec4(vert, 1.0);
        }
}
```

Code Snippet 11: Fragment Shader for OpenGL implementation

```
#version 330 core

out vec4 FragColor;

in vec3 Normal;
```

```
in vec3 FragPos;
in vec3 FinalColor;

uniform vec3 lightPos;
uniform vec3 lightColor;
uniform vec3 viewPos;
uniform vec4 objectColor;

#define NORMAL 0
#define GOURAUD_AD 1
#define GOURAUD_ADS 2
#define PHONG 3

uniform int shading_type;

void main(){

        if(shading_type == NORMAL){
                FragColor = objectColor;
        } else if(shading_type == GOURAUD_AD){
                FragColor = objectColor * vec4(FinalColor, 1.0);
        } else if(shading_type == GOURAUD_ADS){
                FragColor = objectColor * vec4(FinalColor, 1.0);
        }else if(shading_type == PHONG){
                //ambient light
                float const_ambient = 0.5;
                vec3 ambient_light = vec3(const_ambient * lightColor);

                //diffuse light
                vec3 norm = normalize(Normal);
                vec3 lightDir = normalize(lightPos - FragPos);
                float diff = max(dot(norm, lightDir), 0.0);
                vec3 diffuse_light = diff * lightColor;

                //specular light
                float specularStrength = 0.5;
                vec3 viewDir = normalize(viewPos - FragPos);
                vec3 reflectDir = reflect(-lightDir, norm);
                float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
                vec3 specular_light = specularStrength * spec * lightColor;

                FragColor = objectColor * (vec4(ambient_light, 1.0f) +
                                           vec4(diffuse_light, 1.0f) +
                                          vec4(specular_light, 1.0f));
        }
}
```

Finally we provide graphical examples of the shading types being applied to our objects in the following figures:

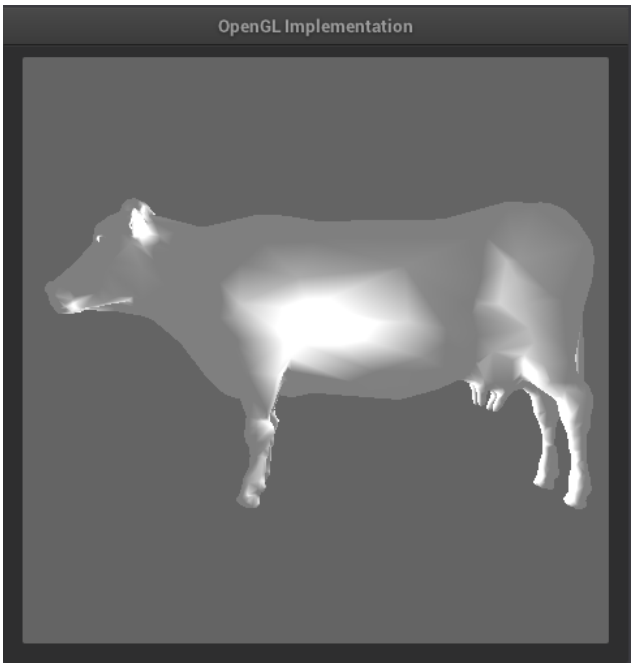

Figure 14: Gouraud Shading (Ambient + diffuse) applied.

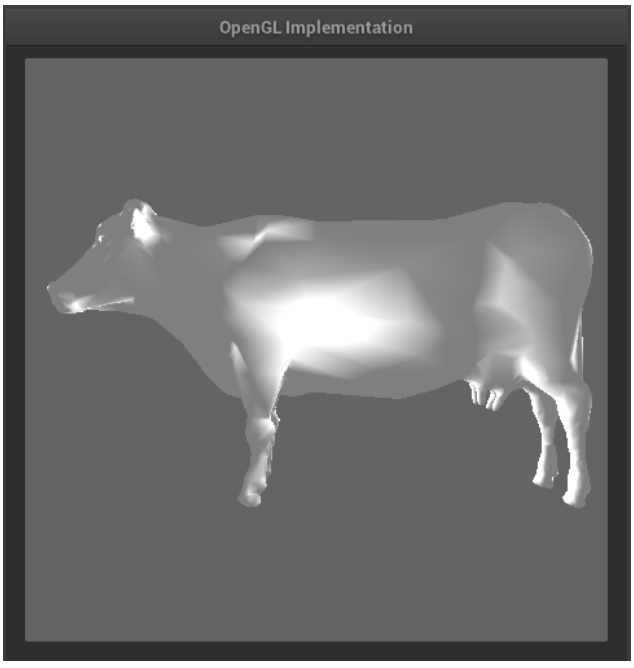

Figure 15: Gouraud Shading (Ambient + diffuse + specular) applied.

12. **FPS Rate:** We calculate the FPS rate for both implementations, and shows it at the right bottom part of the graphical interface all the time. The FPS rate remains the

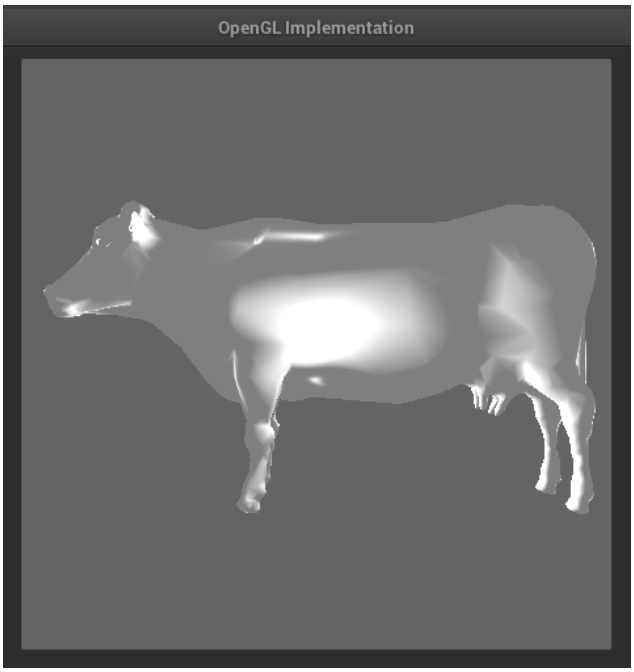

Figure 16: Phong Shading (Ambient + diffuse + specular) applied.

same for both implementations all the time, indicating that both implementations are equally efficient in computational cost.

Program Execution

In order to execute the program in a more intuitive and comfortable way a Makefile was created to compile and run the code easily. In order to compile the source code, in the main directory (which include the directories called src, bin and include) just need to execute the command **make**. If everything was executed correctly, it would not show any error message, in order to execute the just compiled code we only need to execute the command **make run**. If everything is correct we will see the program running without any error messages in the console.

Conclusions

1. The main objectives of this assignment was achieved, but we not manage to use subroutines at shaders because of time.

2. We implement a GUI that shows OpenGL and Close2GL implementation being rendered at the same time to intuitively see how these implementations works.

3. We implement separate classes for camera in **include/camera.h** for OpenGL and in **include/close2gl.h** for OpenGL in order to maintain it separately as well as different shaders for both implementations.

4. We implement a matrix class in **include/matrix.h** to performs matrix operations at the Close2GL implementation such as translation, rotation, matrix multiplication, etc that depicts to be efficient enough to match with the OpenGL implementation. Additionally, we use glm::vec3, glm::vec4 and glm::mat4 as datatypes but all the operations are implemented manually.

5. We achieve a Close2GL implementation that works as efficient of OpenGL graphically and looking at the FPS rate too. In other words, Close2GL implementation is synchronized with OpenGL implementation all the time.

6. The resulting object being rendered depends on so much variables that can be defined in different stages, so it is a good advice to try to separate as maximum as possible the code that is not OpenGL from the code that actually is.

7. We need to care about what transformations or operations we performed over the model, view and projection matrices because one single error in one of them could result in a undesirable result and even worst, a hidden bug that is really difficult to track.