CMP143 - Computer Graphics: Programming Assignment N 1

Rendering Polygonal Meshes with OpenGL and GLSL shaders

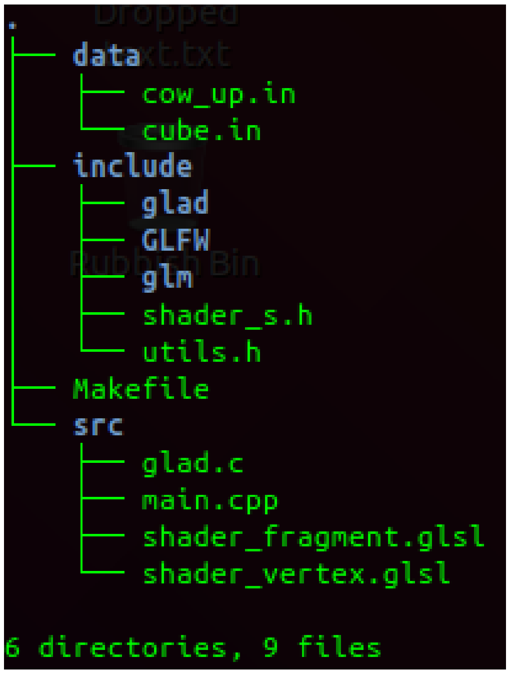
Objective: The main objective of this assignment is to write a program capable to render polygonal meshes (triangles) using OpenGL and GLSL shaders. There will be functionalities needed that are not in the scope of this assignment but without them the program will look incomplete such as: graphical interface to perform actions over the object, depth buffering, virtual camera arbitrary specification, etc. So, these complementary functionalities needs to be developed. Furthermore, this program needs to provide the following core functionalities:

- **Read and display arbitrary geometric models:** This geometric models have to be represented as triangle meshes and they are specified in text files with a fixed layout. Once the model has been read, it has to be displayed at 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.
- **RGB:** support for interactively change the colors of the models, applying a single RGB color to all triangles in the model.

In the following sections we will explain in a detailed manner how the core functionalities mentioned above were achieved:

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. Finally in the main directory, a file called Makefile which contains the instructions to the program to be compiled and executed.



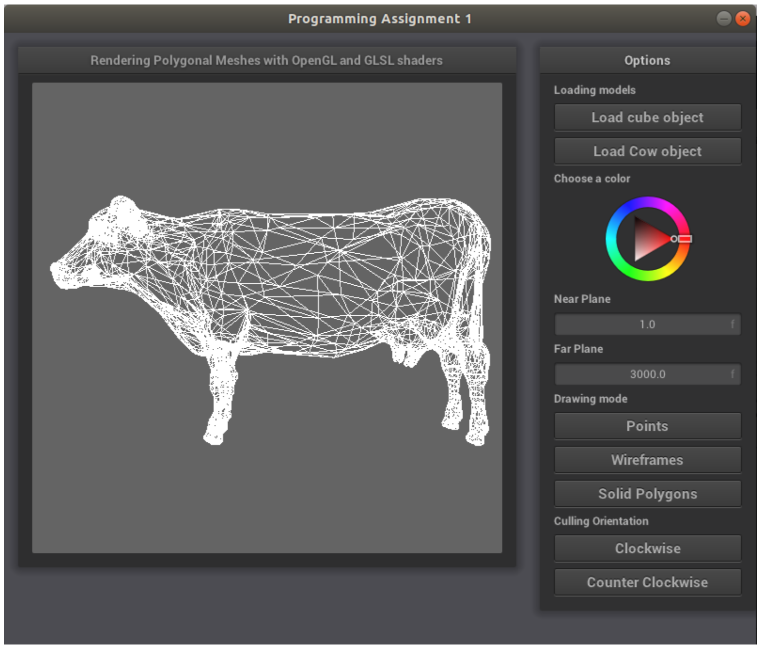
```
.
├── data
│   ├── cow_up.in
│   └── cube.in
├── include
│   ├── glad
│   ├── GLFW
│   ├── glm
│   ├── shader_s.h
│   └── utils.h
├── Makefile
└── src
    ├── glad.c
    ├── main.cpp
    ├── shader_fragment.glsl
    └── shader_vertex.glsl

6 directories, 9 files
```

Figure 1: Screenshot of how program looks like with NanoGUI

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 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.

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



Near Plane 1.0
Far Plane 3000.0

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]

[2]https://nanogui.readthedocs.io/en/latest/compilation.html#compilation

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

In the present work, the matrix operations will be calculated using the OpenGl Mathematics library (GLM)[3] according to the recommendations in the definition of this 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 Complementary Assignment when we render a single triangle.

Implementation

1. **Reading and display arbitrary geometric models:** In order to implement this functionality we need to read the vertex information for *.in files placed in data directory. In order to do that, a function called **readFile** was implemented based on the example provided in the assignment definition which could be found in the file main.cpp . This function receives the name of the file to be read as input parameters and returns the ID of the VAO created to store the VBO generated with the file information. This file is read line by line and update some global variables standing up for min and max values in the three dimensions x, y and z.

   Additionally, the vertex and fragment shaders needs to be loaded, in this case these shaders are loaded as an attribute for the Class which manage the actual rendering (MyGLCanvas). A header file called shader_s.h located under the include directory was created in order to use it for loading shaders and performs compilation checks. The source code of the vertex and fragment shaders are shown below:

[3]https://glm.g-truc.net/0.9.9/index.html

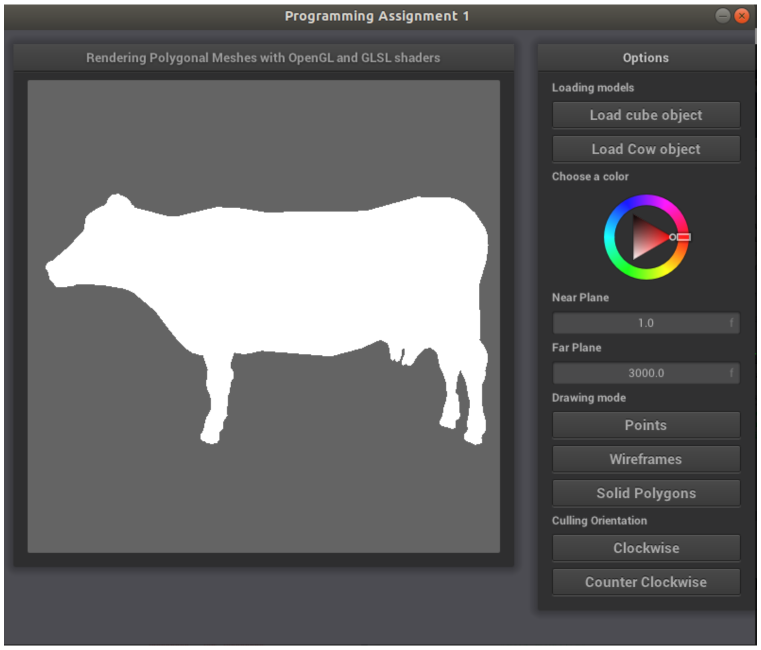
Code Snippet 1: Source code for Vertex Shader

```
#version 330 core

layout (location = 0) in vec3 vert;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

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

Code Snippet 2: Source code for Fragment Shader

```
#version 330 core

uniform vec4 rasterizer_color;
out vec4 FragColor;

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

As can be observed, we pass the model, view and projection matrix to the shader as uniform variables. In the same way, we pass the color to the fragment shader using another uniform variable. So, 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**:



Near Plane 1.0
Far Plane 3000.0

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

2. **Translate the virtual camera along its own axes:** We will explain what have done to achieve this objective and also the other objective which request to translate the camera while looking at the centre of the object since we could not separate it. For this part of the implementation we used the keyboard in order to accomplish that. 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.
   - **Q key:** In order to perform camera Translation along the v axis going to the up direction.
   - **Z key:** In order to perform camera Translation along the u axis going to the down direction.

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

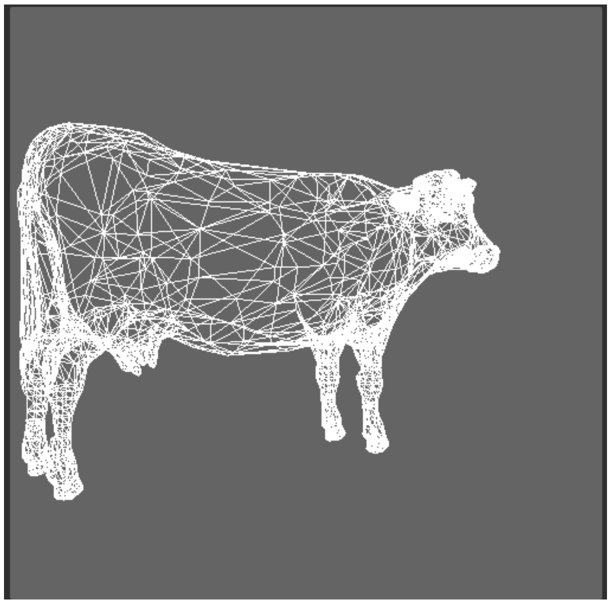


Figure 4: Camera translated in the x axis to the left direction using A key.

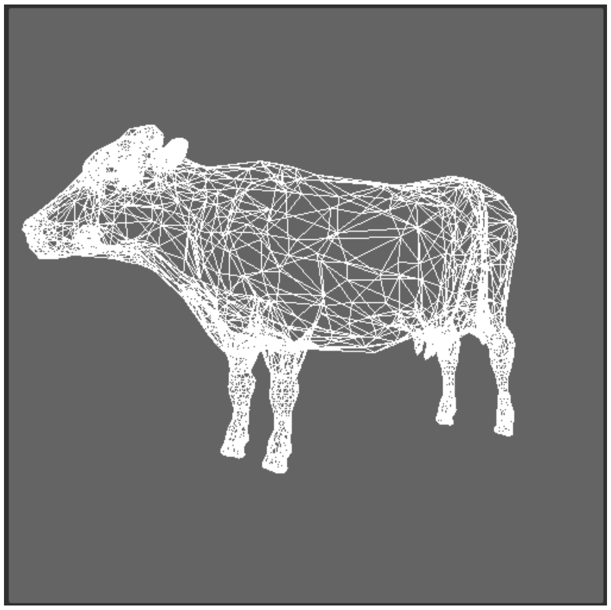


Figure 5: Camera translated in the x axis to the right direction using D key.

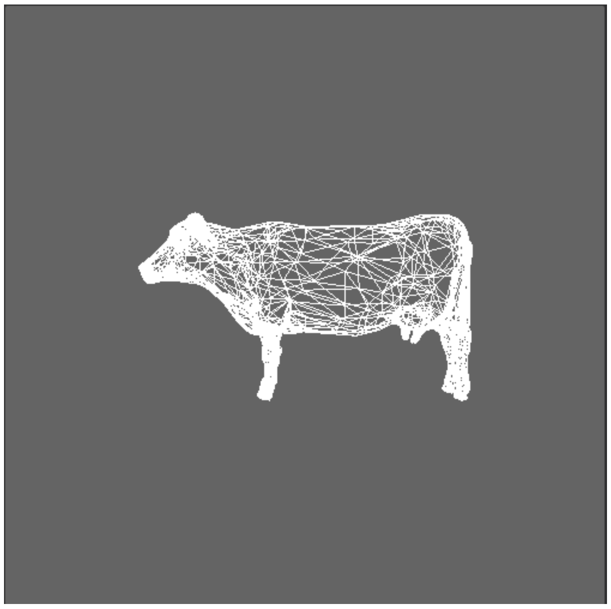


Figure 6: Camera translated in the n axis giving an impression of moving the camera away from the object using S key.

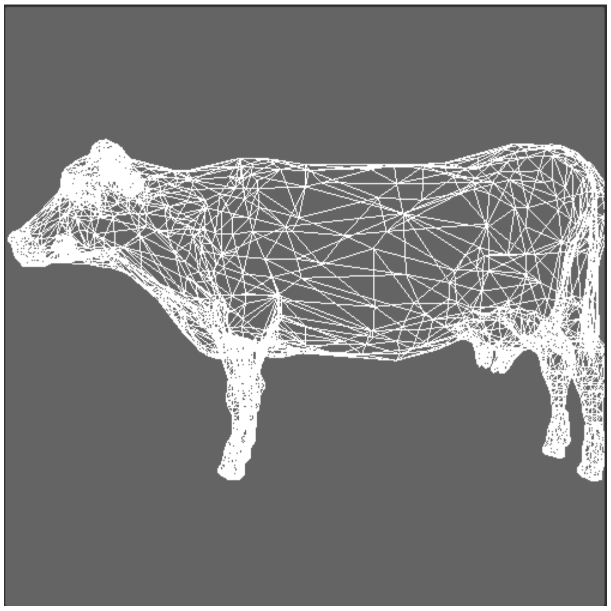


Figure 7: Camera translated in the n axis giving an impression of bringing the camera closer to the object using W key.

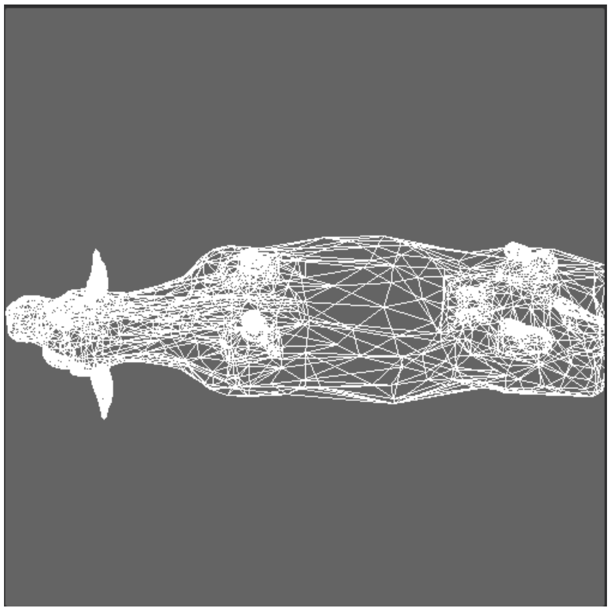


Figure 8: Camera translated in the v axis following the up direction using Q key. Note that in this case we are showing the cow loin from the top.

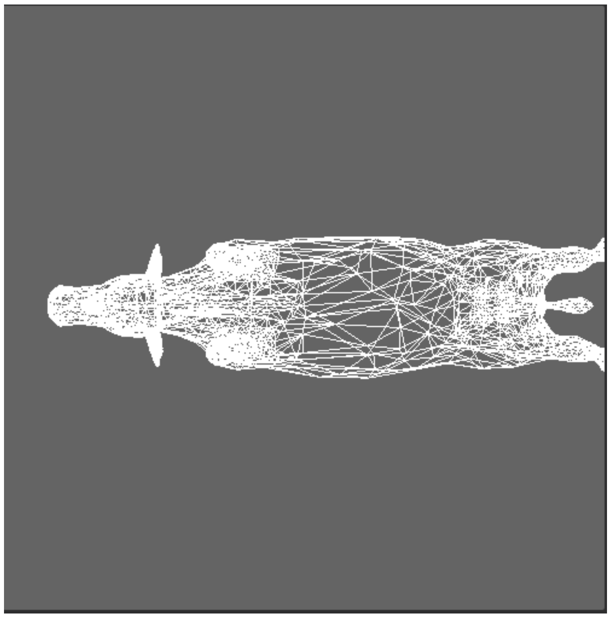


Figure 9: Camera translated in the v axis following the down direction using Z key. Note that in this case we are showing the low part of the cow from the bottom.

In order to implement the functionalities exposed above we initialize the u, v, and n vectors as unit vectors. In our code we renamed these vectors as **cameraRight, cameraUp and cameraFront**. In the following code snippets we could see how the new position is calculated when press the movement keys:

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

```
if (key == GLFW_KEY_W){
    while(action == GLFW_REPEAT || action == GLFW_PRESS){
        if(this->mCanvasObject->model_used == 1)
            this->mCanvasObject->cameraPos
            += 0.25f*camera_speed * this->mCanvasObject->cameraFront;
        else
            this->mCanvasObject->cameraPos
            += 15.0f*camera_speed * this->mCanvasObject->cameraFront;

        //Updating distance Projection Sphere
        this->mCanvasObject->distanceProjSphere
        = glm::length(this->mCanvasObject->cameraPos);

        return true;
    }
}
```

Code Snippet 4: Source code for updating cameraPos when pressed the D key

```
if (key == GLFW_KEY_D){
    while(action == GLFW_REPEAT || action == GLFW_PRESS){

        this->mCanvasObject->cameraRight
        = glm::normalize(-glm::cross(this->mCanvasObject->cameraFront,
                                this->mCanvasObject->cameraUp));

        if(this->mCanvasObject->model_used == 1)
            this->mCanvasObject->cameraPos
            +=  this->mCanvasObject->cameraRight * camera_speed;
        else
            this->mCanvasObject->cameraPos
            += this->mCanvasObject->cameraRight * 60.0f*camera_speed;

        this->mCanvasObject->cameraFront
        = glm::normalize(-this->mCanvasObject->cameraPos);

        return true;
    }
}
```

Code Snippet 5: Source code for updating cameraPos when pressed the Q key

```
if (key == GLFW_KEY_Q){
    while(action == GLFW_REPEAT || action == GLFW_PRESS){

        this->mCanvasObject->cameraUp
        = glm::normalize(-glm::cross(this->mCanvasObject->cameraFront,
                                        this->mCanvasObject->cameraRight));

        if(this->mCanvasObject->model_used == 1)
            this->mCanvasObject->cameraPos
            += this->mCanvasObject->cameraUp * camera_speed;
        else
            this->mCanvasObject->cameraPos
            += this->mCanvasObject->cameraUp * 60.0f*camera_speed;


        this->mCanvasObject->cameraFront
        = glm::normalize(-this->mCanvasObject->cameraPos);

        return true;
    }
}
```

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 in snippet 3 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.

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 6: 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;

    return true;

}
```

The code shown above is executed every time the mouse is moved in any direction and recalculates the view matrix by applying rotation in the u and v axis, the rotation performed is shown below:

Code Snippet 7: Performing actual rotation over the view matrix

```
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));
```

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

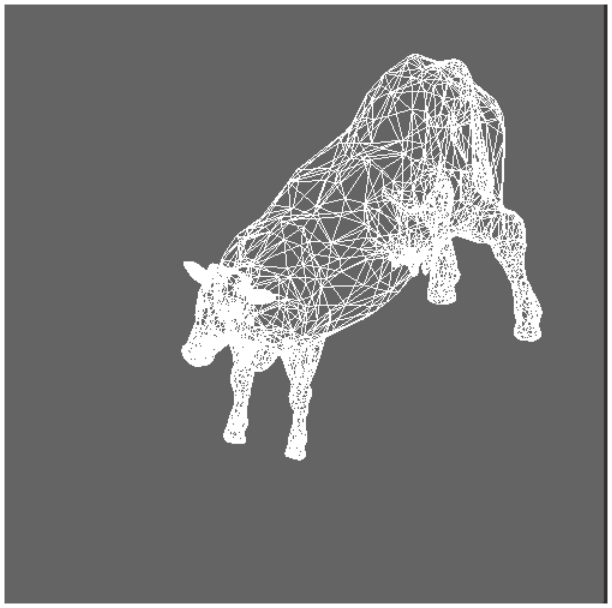


Figure 10: Example of camera rotation showing the cow doing some anger movements observed from the back.

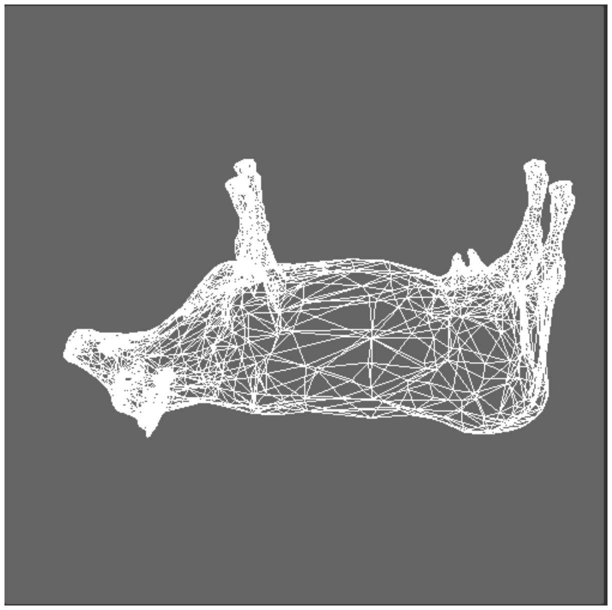


Figure 11: Example of camera rotation showing the cow a little bit more relaxed lying on his back.

4. **Reset the camera to its original position:** In this case, we reset the camera 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 by using the glFrontFace OpenGL method by setting the constants GL_CW for clockwise and GL_CCW for counter clockwise. In the following figures we will show the graphical difference while setting one option or another.

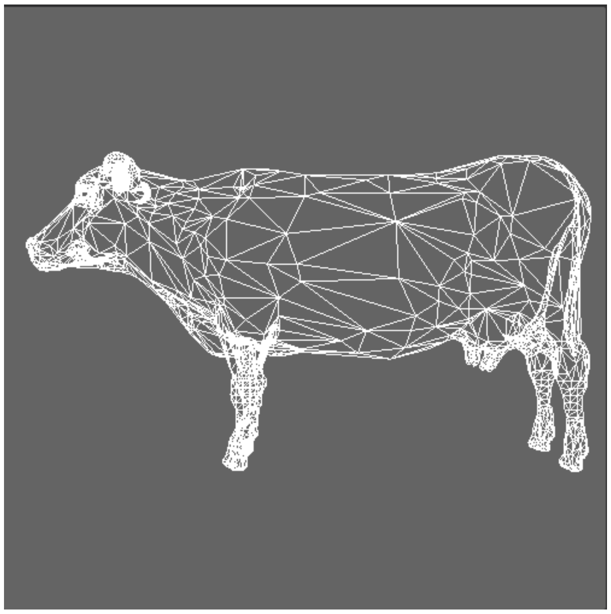


Figure 12: Example of the loaded object using clockwise orintation.

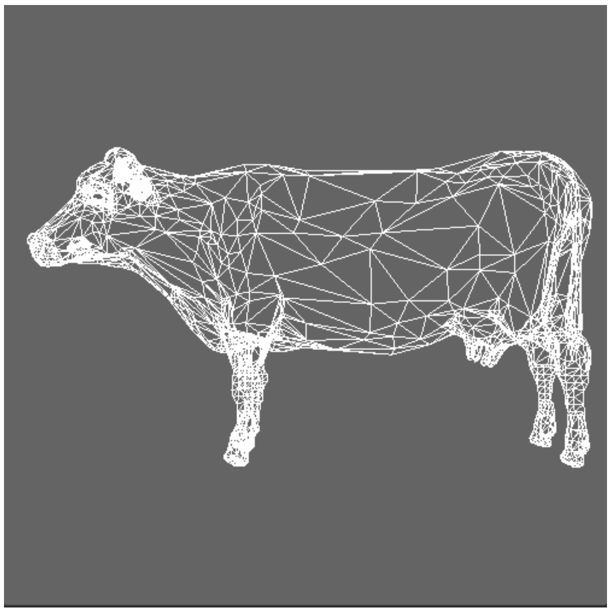
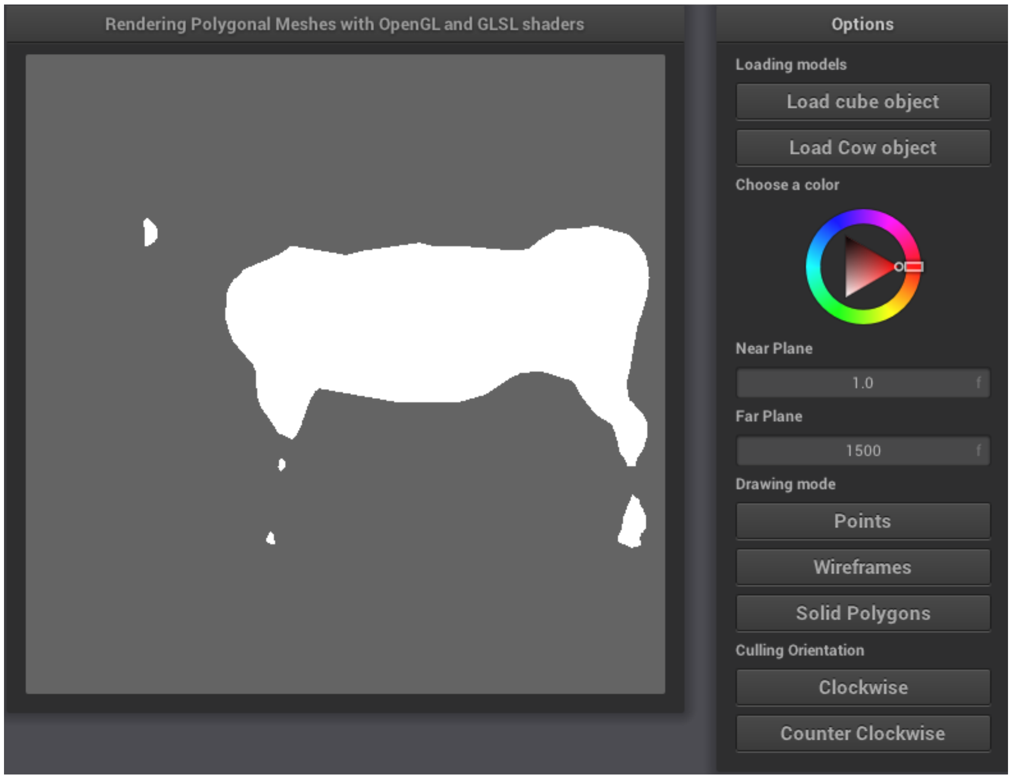


Figure 13: Example of the loaded object using counter clockwise orientation.

As we can see the final object being rendered 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.

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) in the figure below:



Near Plane: 1.0
Far Plane: 1500

Figure 14: 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.

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. This allows us to change the color of the object being rendered anytime we want. We provide an screenshot as an example of what was said here:

CMP143 - Computer Graphics: Programming Assignment N 1

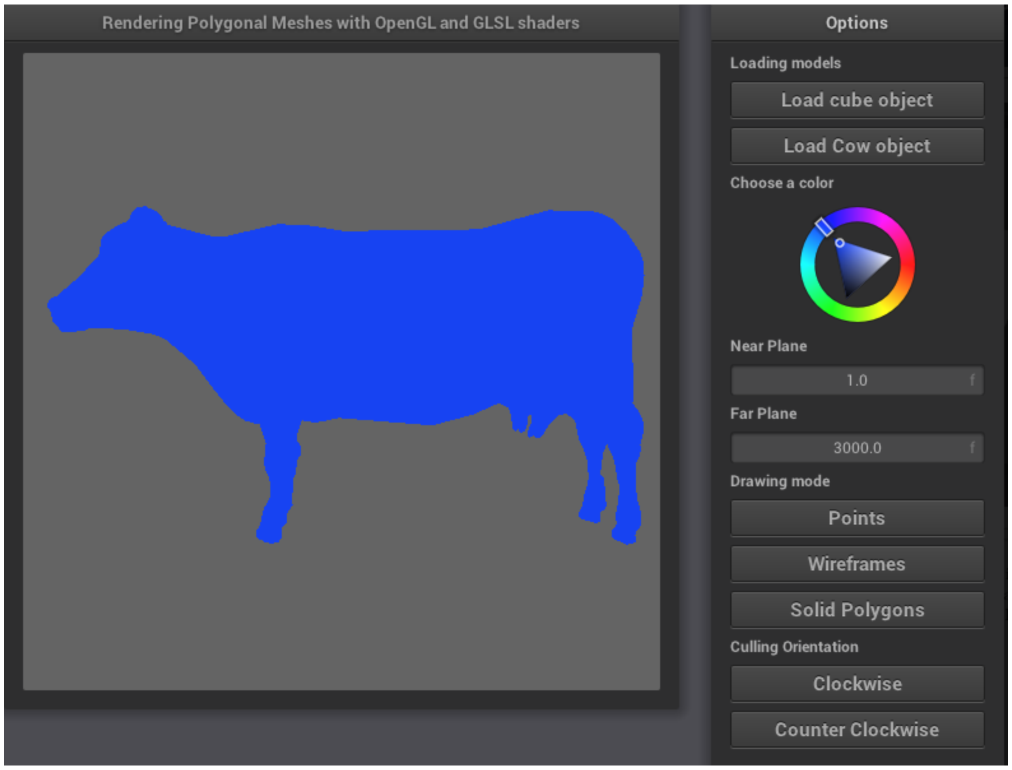


Figure 15: Example screenshot of the object being rendered in a color chosen from the color picker.

8. **Drawing mode:** We additionally provide in the GUI options 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. We show examples of WIREFRAMES and POINTS drawing modes in the figure below:

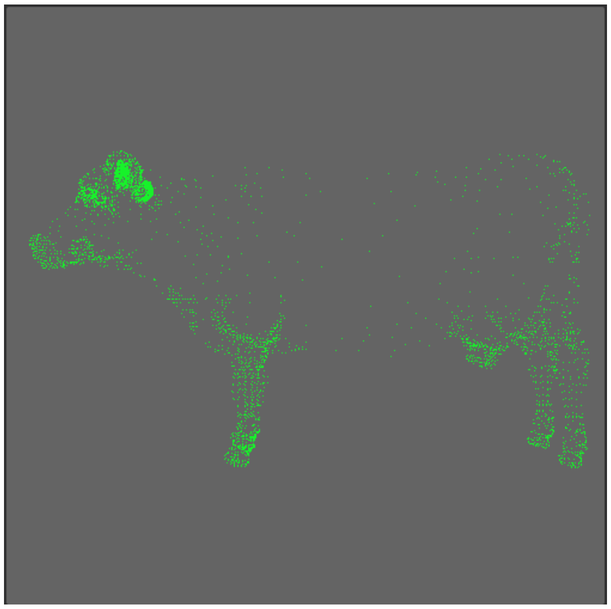


Figure 16: Object being rendered in POINTS drawing mode.

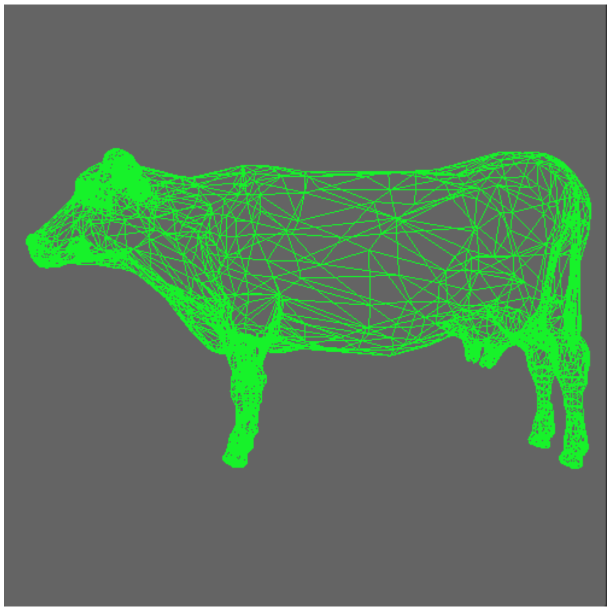


Figure 17: Object being rendered in WIREFRAMES drawing mode.

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;
    }
```

10. Finally, depth buffering is also activated with the corresponding OpenGL methods. The complete code used for actually drawing the objects is shown in the following code snippet:

Code Snippet 9: Zooming effects on Scroll Button event

```
virtual void drawGL() override {
    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)
        glm::mat4 trans = glm::translate(glm::mat4(1.0f),
                        glm::vec3(-center_x, -center_y, -center_z));
        model = trans * model;

        //Normalizing and scaling cameraPos by distanceProjSphere
        cameraPos = glm::normalize(cameraPos)*(distanceProjSphere);

        if(firstMouse){

            view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);

        } else {

            view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);

            //Performing rotation
            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));
        }

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

        // retrieve the matrix uniform locations
        unsigned int modelLoc = glGetUniformLocation(custom_shader.ID, "model");
        unsigned int viewLoc  = glGetUniformLocation(custom_shader.ID, "view");
        unsigned int projectionLoc  = glGetUniformLocation(custom_shader.ID
        , "projection");
        unsigned int colorLoc = glGetUniformLocation(custom_shader.ID
        , "rasterizer_color");

        // pass them to the shaders (3 different ways)
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
                glm::value_ptr(model));
        glUniformMatrix4fv(viewLoc, 1, GL_FALSE,
                glm::value_ptr(view));
        glUniformMatrix4fv(projectionLoc, 1, GL_FALSE,
                glm::value_ptr(projection));
        glUniform4fv(colorLoc, 1,
                glm::value_ptr(this->color));

        glBindVertexArray(VAO);

        if(culling_orientation == 1){
            glEnable(GL_CULL_FACE);
            glCullFace(GL_BACK);
            glFrontFace(GL_CW);
        }else if (culling_orientation == 2){
            glEnable(GL_CULL_FACE);
            glCullFace(GL_BACK);
            glFrontFace(GL_CCW);
        }

        glEnable(GL_DEPTH_TEST);

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

        glDisable(GL_DEPTH_TEST);
    }
}
```

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 completely.
2. It was a really nice experience to work developing this assignment because it clarifies the concepts discussed in classes.
3. Initially it was a little bit tricky to make OpenGL work with the graphical user interface, but the learning curve is not really so difficult.
4. 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.
5. NanoGUI and GLM libraries were fundamental in the developing of the present work.
6. 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 dificult to track.