CMP143 - Computer Graphics: Programming Assignment N 3

Extending Close2GL to support rasterization

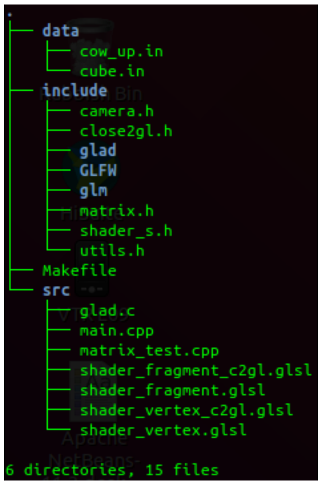
Objective: The main objective of this assignment is to implement shading and rasterization for our Close2GL implementation. In the following paragraphs a brief but detailed explanation about what is expected for this assignment is shown:

- **Rasterization of the projected and shaded triangles:** it is expected to achieve rasterization for each triangles of our example models in **solid mode** (full triangles) and in **wireframe mode** (triangle edges only).
- **Implement Phong Lighting model**.
- **Implement Gouraud Shading:** for solid and wiframe modes.
- **Handle visibility:** by implementing z-buffering.
- Perform same operations allowed in previous implementations.

In this assignment, only features for our Close2GL implementation are required, no new features were developed for OpenGL implementation.

Project Structure

In the image below, we show the project structure which is basically the same for the assignment 2, no new files were added, only modifications to the previous existent files were made.



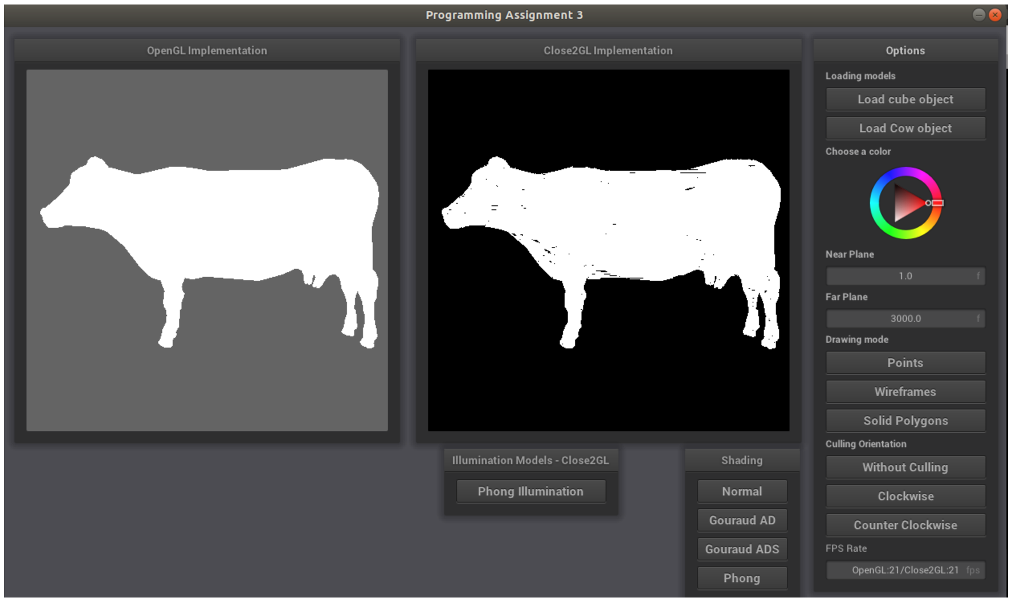
```
.
├── data
│   ├── cow_up.in
│   └── cube.in
├── include
│   ├── camera.h
│   ├── close2gl.h
│   ├── glad
│   ├── GLFW
│   ├── glm
│   ├── matrix.h
│   ├── shader_s.h
│   └── utils.h
├── Makefile
└── src
    ├── glad.c
    ├── main.cpp
    ├── matrix_test.cpp
    ├── shader_fragment_c2gl.glsl
    ├── shader_fragment.glsl
    ├── shader_vertex_c2gl.glsl
    └── shader_vertex.glsl

6 directories, 15 files
```

Figure 1: Project structure: directories and files

CMP143 - Computer Graphics: Programming Assignment N 3

Graphical User Interface

As was mentioned in the previous assignments we are using **The Nanogui widget** [1] for generating a Graphical Interface 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, a button was added to allow us to enable Phong Lighting model, 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.



Near Plane 1.0

Far Plane 3000.0

FPS Rate OpenGL:21/Close2GL:21 fps

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
```

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

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. In this implementation, a function called **distanceV** was added in order to calculate distance for two vectors in homogeneus coordinates based only in the information for x, y and z axis.

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

Implementation

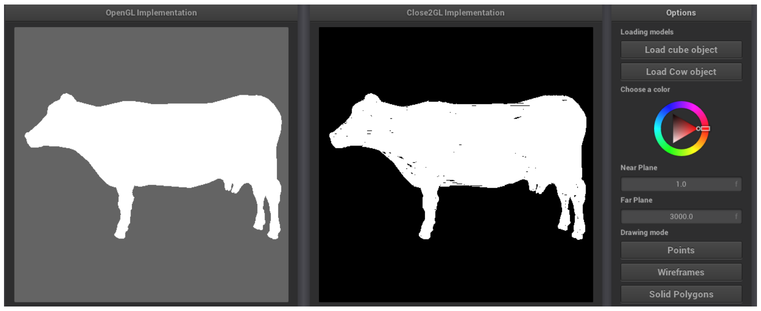
The differences between the implementations in OpenGL and Close2GL were explained in detail in the previous assignment, since in this assignment features were developed only for the Close2GL implementation, we will focus only in new features developed.

1. **Reading and display arbitrary geometric models:**

   Close2GL: As we mentioned in previous assignment a function called **readFile_close2gl** was implemented. This functions returns an std::vector of **Triangle_c2gl** (a custom struct created to save information about vertexs, normals, color,etc of the object). Additionally, as mentioned in previous assignments, specific vertex and fragment shaders were created for Close2GL called **shader_vertex_c2gl.glsl** and **shader_fragment_c2gl.glsl**, these files will be shown when we talk about triangle rasterization.

   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. Additionally, we pass the color to the fragment shader using a uniform variable. For both cases(OpenGL and Close2GL), 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:

CMP143 - Computer Graphics: Programming Assignment N 3



Near Plane 1.0
Far Plane 3000.0

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

2. **Translations, Rotations and others:** All the previous functionalities such as translations along the camera axes, rotations, GUI options, keyboard and mouse control, zooming effects, drawing mode and backface culling for both implementations continue working properly and were no modified in this assignment implementation, for simplicity we will skip detailed explanation about this since it was explained in previous assignments.

3. **Viewport Matrix**: This assignment implementation begins exactly where the previous one finished. At the end of the previous assignment we have transformed vectors (by model view projection matrix) and pass the x and y coordinates (obtained from each vertex) to the shaders in order to they mapping these vertex to the viewport. In this case, we apply a vector transformation using the ViewPort Matrix (shown below in the code snippet 1) over each triangle vertex in order to obtain the projected position x and y in the screen for every triangle vertex.

   Code Snippet 1: Vertex shader for OpenGL implementation

```
glm::mat4 getViewPortMatrix(float lv, float rv, float bv, float tv){
glm::mat4 v = glm::mat4(0.0f);

//Setting viewport matrix

v[0][0] = (rv-lv)/2;v[1][0] = 0.0f;v[2][0] = 0.0f;v[3][0] = (rv+lv)/2;
v[0][1] = 0.0f; v[1][1] = (tv-bv)/2;v[2][1] = 0.0f;v[3][1] = (tv+bv)/2;
v[0][2] = 0.0f; v[1][2] = 0.0f; v[2][2] = 1.0f; v[3][2] = 0.0f;
v[0][3] = 0.0f; v[1][3] = 0.0f; v[2][3] = 0.0f;v[3][3] = 1.0f;

return v;
}
```

   After getting the correct x and y position in the screen for every vertex, we are ready to perfom triangle rasterization.

4. **Close2GL Rasterization:** As suggested in the assignment definition we implement a function called rasterize_triangle in order to process every single triangle to be rendered. The steps followed in order to achieve triangle rasterization is listed below:

- Order the vertex from top to bottom, it means comparing only his Y component. This step is performed in order to know in which way the rasterization will be achieved.
- Evaluate the type of triangle to be rasterized based on vertex position. We identified three types of triangles, upright triangles (two vertexs with the same y coordinate at the bottom), inverted triangle (two vertexs with the same y coordinates at top) and generic triangles (every vertex with different y coordinate). In the following figures, we show the kinds of triangles just mentioned:

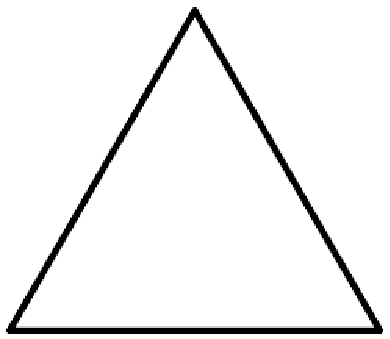


Figure 4: Example of upright triangle.

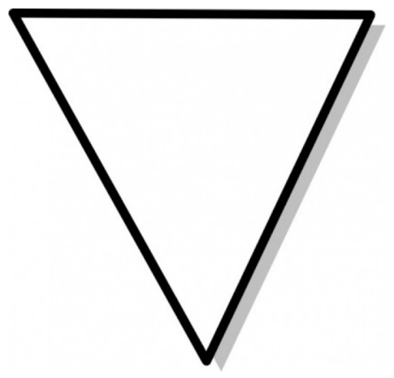


Figure 5: Example of inverted triangle.

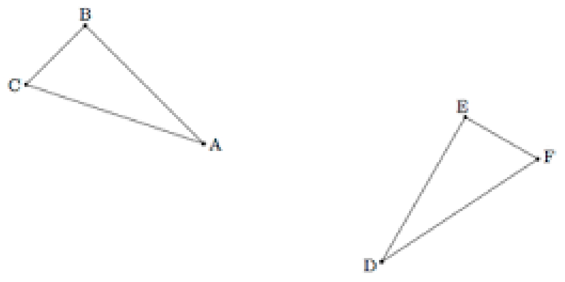


Figure 6: Examples of generic triangles.

- Based on the previous classification, we choose the first vertex (the one at the top, or one of them in case there are two) test it against our z-buffer and if it pass the test we put it in our color buffer. Then, we define our edges joining our first vertex with the other two vertexs, calculate the needed information (**dx** and **dy**) and perform color and depth interpolation just based in the distances between the two edges and the point in which the interpolation is performed at the moment. We choose to perform rasterization from the top vertex towards the most bottom vertex iterating over each scanline between them.
- Rasterization in upright and inverted triangles is performed in a pretty similar way as was just mentioned above. In the case of a generic triangle, a rasterization is achieved in two steps, th first one considering a triangle only from the first vertex and the second one and his projection on the third vertex. After the rasterization of this first part of the triangle, the second part is rasterized, this time considering an inverted triangle, since vertex two and the projection overt the third vertex have the same y component.
- **Drawing mode:** Since rasterization is being implemented manually, drawing mode: POINTS, WIREFRAMES or SOLID POLYGONS needs to be managed at this point. In order to draw the object in POINTS drawing mode, we stop the iteration at the first one value iterated (i.e. a first pixel being evaluated since points is no other thing that pixels in screen) for each triangle. SOLID POLYGONS are achieved by perform the complete iteration over each triangle (i.e. evaluating every single pixel of the screen). Finally, in the case of the WIREFRAMES drawing mode, we use the Bresenham algorithm to draw a very close approximation to a line between two vertex points in the screen being rasterized. In this case, we use the Bresenham algorithm to produce lines between the three vertexs of our triangles (edges), i.e. a line joining the vertexs V1-V2, V1-V3, V2-V3, making a triangle. In the Bresenham algorithm implementation, we also pass the color of the pixel that will be rasterized in order to use it to draw the corresponding line. In the following code snippet we show how this algorithm was implemented:

Code Snippet 2: Vertex shader for OpenGL implementation

```
void bresenham(int x1,int y1,int x2,int y2,RGBA_color average_color) {
        int x,y,dx,dy,dx1,dy1,px,py,xe,ye,i;
        dx=x2-x1;
        dy=y2-y1;
        dx1=fabs(dx);
        dy1=fabs(dy);
        px=2*dy1-dx1;
        py=2*dx1-dy1;

        if(dy1<=dx1){
            if(dx>=0){
                x=x1;
                y=y1;
                xe=x2;
            } else {
                x=x2;
                y=y2;
                xe=x1;
            }

            set_to_color_buffer(x,y, average_color);
            for(i=0;x<xe;i++){
                x=x+1;
                if(px<0){
                    px=px+2*dy1;
                } else {
                    if((dx<0 && dy<0) || (dx>0 && dy>0)){
                        y=y+1;
                    }else{
                        y=y-1;
                    }

                px=px+2*(dy1-dx1);
                }

                set_to_color_buffer(x,y, average_color);
            }
        } else {
            if(dy>=0) {
                x=x1;
                y=y1;
                ye=y2;
            } else {
                x=x2;
                y=y2;
                ye=y1;
            }

            set_to_color_buffer(x,y, average_color);
```

```
                for(i=0;y<ye;i++){
                    y=y+1;

                    if(py<=0){
                        py=py+2*dx1;
                    } else {
                        if((dx<0 && dy<0) || (dx>0 && dy>0)) {
                            x=x+1;
                        } else {
                            x=x-1;
                        }

                        py=py+2*(dx1-dy1);
                    }

                    set_to_color_buffer(x,y, average_color);
                }
            }
        }
```

The code for the triangle rasterization could be found in the file main.cpp (between lines 681 and 1066), it was developed as function of the class MyGLCanvasC2GL, which also contains some other helper functions used to achieve the rasterization such as:

- **clear_buffers:** To clear both buffers (color and depth buffer) in every frame iteration.
- **allocate_buffers:** To allocate memory for both buffers (color and depth buffers).
- **set_to_color_buffer:** To set a RGBA_color (a custom struct created to save information about the three color channels and the homogeneus coordinate) in a desired position x, y.
- **interpolate_colors:** given two colors and a given position, performs color interpolation.
- **average_color:** Given three colors calculate the average color just by calculate the average of the color intensities in each channel.
- **interpolate_depths:** in the same way than color interpolation, depth interpolation is achieved.

5. **Close2GL Texture Generation:** As a final step for our Close2GL rasterization we need to generate a texture based on the information we collected in our color buffer. We can not pass this information directly as it comes to the shader since it will try to perform rasterization as a part of the automatic graphic pipeline. Then, we generate a texture based on the color buffer and apply it to a two triangles that cover the

entire screen achieving in this way the final result of our manually rasterization. In the following code snippet we will show the code that we are in the drawing function to achieve rasterization, this include vertex transformations, triangle rasterization and texture generation:

Code Snippet 3: Added code to drawing function for Close2GL in main.cpp

```
//Performing viewport transformation
glm::mat4 viewportMatrix = close2gl.getViewPortMatrix(0.0f, float(WINDOW_WIDTH),
                                        float(WINDOW_HEIGHT), 0.0f);

for(int i=0; i<clipped_triangles.size(); i++){
    clipped_triangles[i].v0 =
    matrix.transform_vector(clipped_triangles[i].v0, viewportMatrix);
    clipped_triangles[i].v1 =
    matrix.transform_vector(clipped_triangles[i].v1, viewportMatrix);
    clipped_triangles[i].v2 =
    matrix.transform_vector(clipped_triangles[i].v2, viewportMatrix);

    rasterize_triangle(clipped_triangles[i]);
}

float vertex_data[24] = {-1.0f, -1.0f, 0.0f, 1.0f,
                          1.0f, -1.0f, 1.0f, 1.0f,
                          1.0f, 1.0f, 1.0f, 0.0f,
                          1.0f, 1.0f, 1.0f, 0.0f,
                          -1.0f, 1.0f, 0.0f, 0.0f,
                          -1.0f, -1.0f, 0.0f, 1.0f};

unsigned int VBO, VAO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);

//binding VAO
glBindVertexArray(VAO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertex_data), vertex_data, GL_STATIC_DRAW);
//Vertex coordinates information
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 4*sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
//Texture coordinates information
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 4*sizeof(float),
(void*)(2*sizeof(float)));
glEnableVertexAttribArray(1);

//Creating texture
unsigned int texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);

//Teture creation/manipulations
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, WINDOW_WIDTH, WINDOW_HEIGHT,
0, GL_RGBA, GL_FLOAT, color_buffer);

//Actual drawing
glDrawArrays(GL_TRIANGLES, 0, 6);


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


// Passing only color as uniform to the fragment shader.
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 positions for the texture were defined in a flipped way, since texture are generated flipped when rendered. One of the most importante lines in the code shown above is when using glTexImage2D, note that we pass the color_buffer as a parameter to generate the desired texture. In the following figures, we will show the rasterization of the object using the process that was recently described in the the drawing modes:

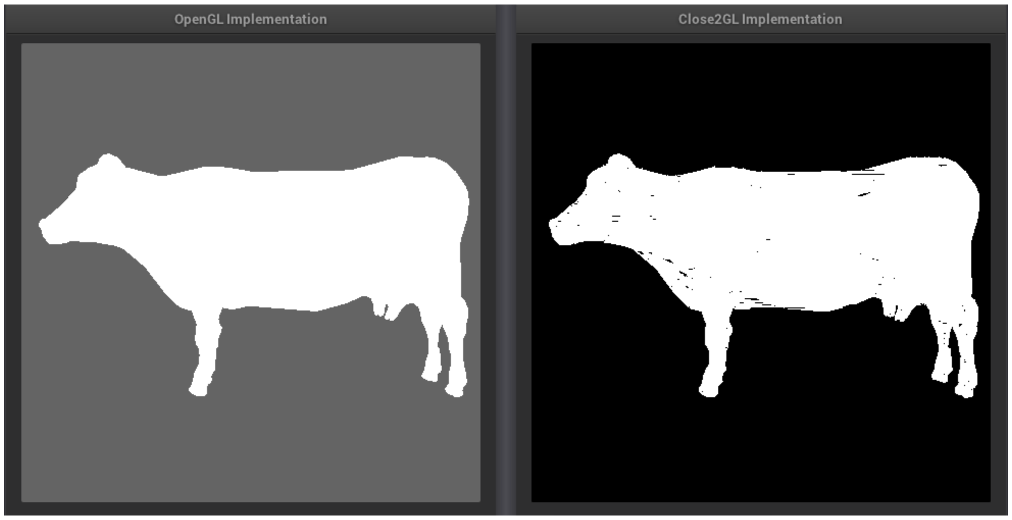


Figure 7: Figure showing OpenGL and Close2GL Implementation side by side rendering cow object in SOLID POLYGONS drawing mode.

Note that the Close2GL implementation not achieve perfect results, we believe that this artifacts generated are produced by the precision of the computations performed to achieved this, since when we draw in WIREFRAME or POINTS drawing mode, this problems are not present.

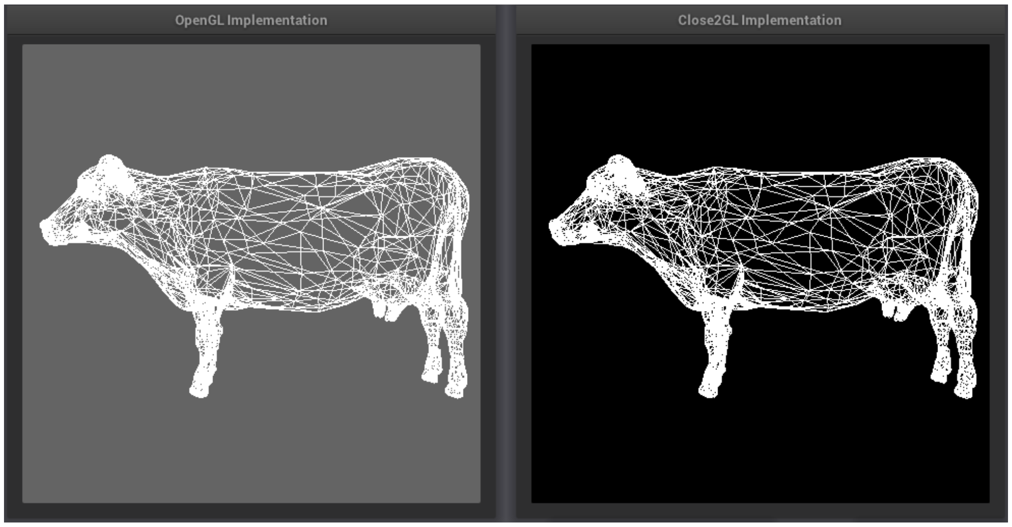


Figure 8: Figure showing OpenGL and Close2GL Implementation side by side rendering cow object in WIREFRAMES drawing mode.

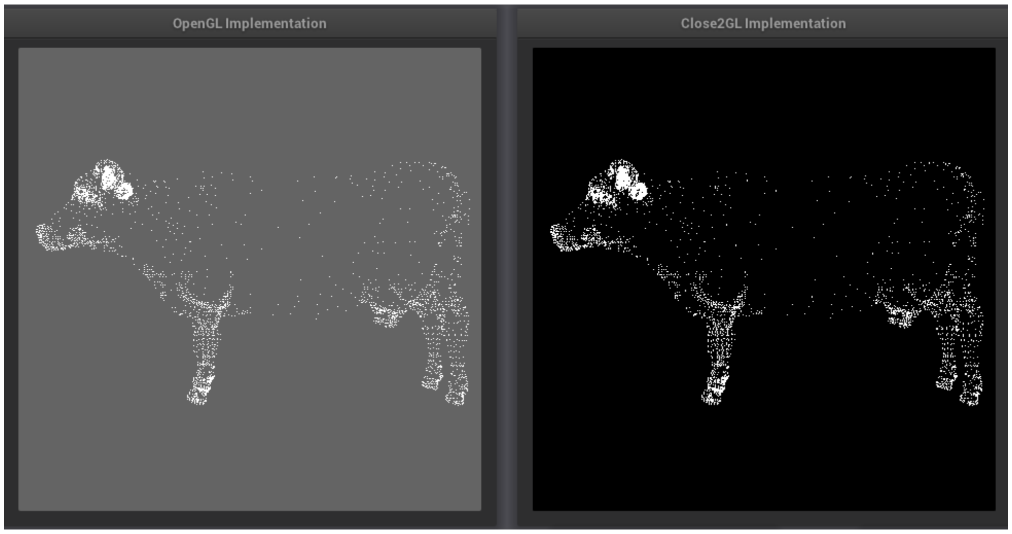


Figure 9: Figure showing OpenGL and Close2GL Implementation side by side rendering cow object in POINTS drawing mode.

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, projection and viewport 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).
- Apply vertexs transformation using the viewport matrix.
- Rasterize every single triangle that is alive at this point, in order to get the color for every single pixel of the screen.
- Generate texture based on the color buffer obtained after triangle rasterization.
- Actually drawing the object in shaders loading the texture generated over two triangles that covers the entire screen.

In the following code snippets, the code for the vertex and fragment shader is shown:

Code Snippet 4: Vertex shader for Close2GL

```
#version 330 core

layout (location = 0) in vec2 vert;
layout (location = 1) in vec2 texCoord;

out vec2 TexCoord;

void main(){
    gl_Position = vec4(vert, 0.0f, 1.0f);
    TexCoord = vec2(texCoord.x, texCoord.y);
}
```

Code Snippet 5: Fragment shader for Close2GL

```
#version 330 core

uniform vec4 rasterizer_color;
uniform sampler2D texture1;
out vec4 FragColor;
in vec2 TexCoord;

void main(){
        FragColor = texture(texture1, TexCoord)*rasterizer_color;
}
```

6. **Phong Illumination Model:** The Phong illumination model was implemented as a function of the class MyGLCanvasC2GL, it is applied over each vertex (if activated) after the model view projection transformation. It considers ambient, difusse and specular components of light and receives as parameters the vertex itself and his corresponding normal. In the following code snippet we will show his source code implementation:

Code Snippet 6: Phong Illumination Model implementation

```
glm::vec3 phong_illumination_model(glm::vec4 v, glm::vec3 v_normal) {
    double I, att, distance, kd;
    glm::vec4 N = glm::vec4(v_normal, 1.0f);

    glm::vec3 vcolor; // vertex color

    N = glm::normalize(N);
    for (int channel = 0; channel < 3; channel++) {
        I = 0;
        I = light.ambient[channel] * g_MatDiffuse[channel];

        if (g_LightsOn) {
            static glm::vec4 L, R, V;
            static glm::vec3 view;
            static double NL_dot, RV_dot;
            // ambient part
            I += light.ambient[channel] * g_MatDiffuse[channel];
            // attenuation
            distance = matrix.distanceV(light.direction, v);
            att = 1.0 / (light.att[0] + distance*light.att[1] +
            distance*distance*light.att[2]);
            if (att > 1)
                att = 1;
            // diffuse part
            L = light.direction - v;
            L = glm::normalize(L);
            NL_dot = matrix.dotProduct(N, L);
            if (NL_dot > 0) {
                I += att * NL_dot * light.diffuse[channel] * g_MatDiffuse[channel];
                // specular part if specular element present
                R = 2 * matrix.dotProduct(N, L)*N - L;
                R = glm::normalize(R);
                view = close2gl.position - glm::vec3(v);
                view = glm::normalize(view);
                RV_dot = matrix.dotProduct(view, glm::vec3(R));
                if (RV_dot > 0)
                    I += att * light.specular[channel] * g_MatDiffuse[channel]
                    * pow(RV_dot, g_shine[0]);
            }
        }
        vcolor[channel] = I;
    }
    return vcolor;
}
```

In the following image we will show the rasterized object using Close2GL implementation, we could think as this functionality as turning on and off a light source.

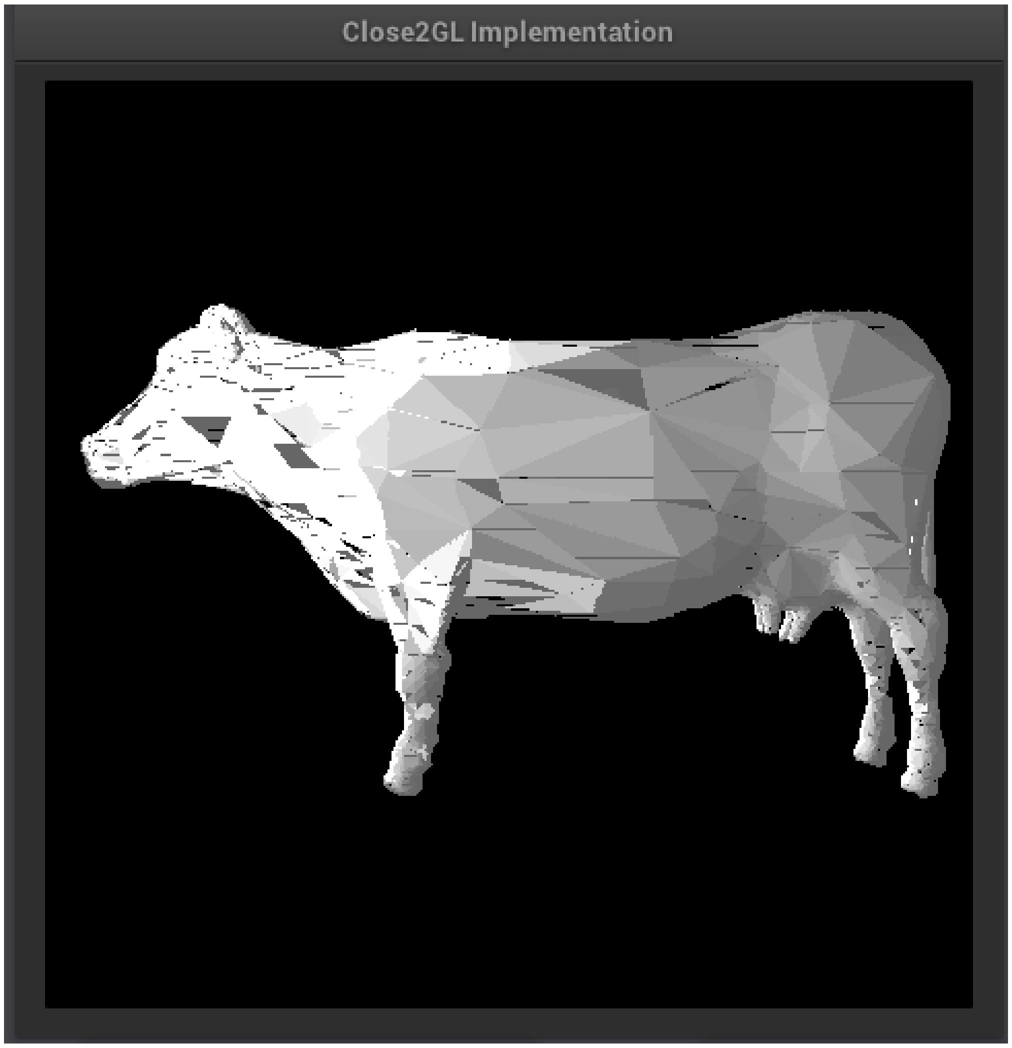


Figure 10: Figure showing Close2GL implementation of the Phong Lighting Model.

As we could see the artifacts initially generated in Close2GL rasterization are propagated and magnified when we applied the Phong Lighting Model but on the other hand, we could observe that turning on the phong lighting model the cow object takes a more realistic appearance as expected.

7. **Shading**: In this case, Gouraud Shading was implemented only considered ambient and diffuse components of light. Gouraud Shading is achieved by color and depth

interpolation when perform triangle rasterization. When Gouraud Shading is activated (and Phong Lighting model too) we consider the interpolated color as the final color for the pixel, otherwise, the average color (of three vertex) is considered as the color of the pixel. In the following figure, the resulting image after applying gouraud shading is shown:

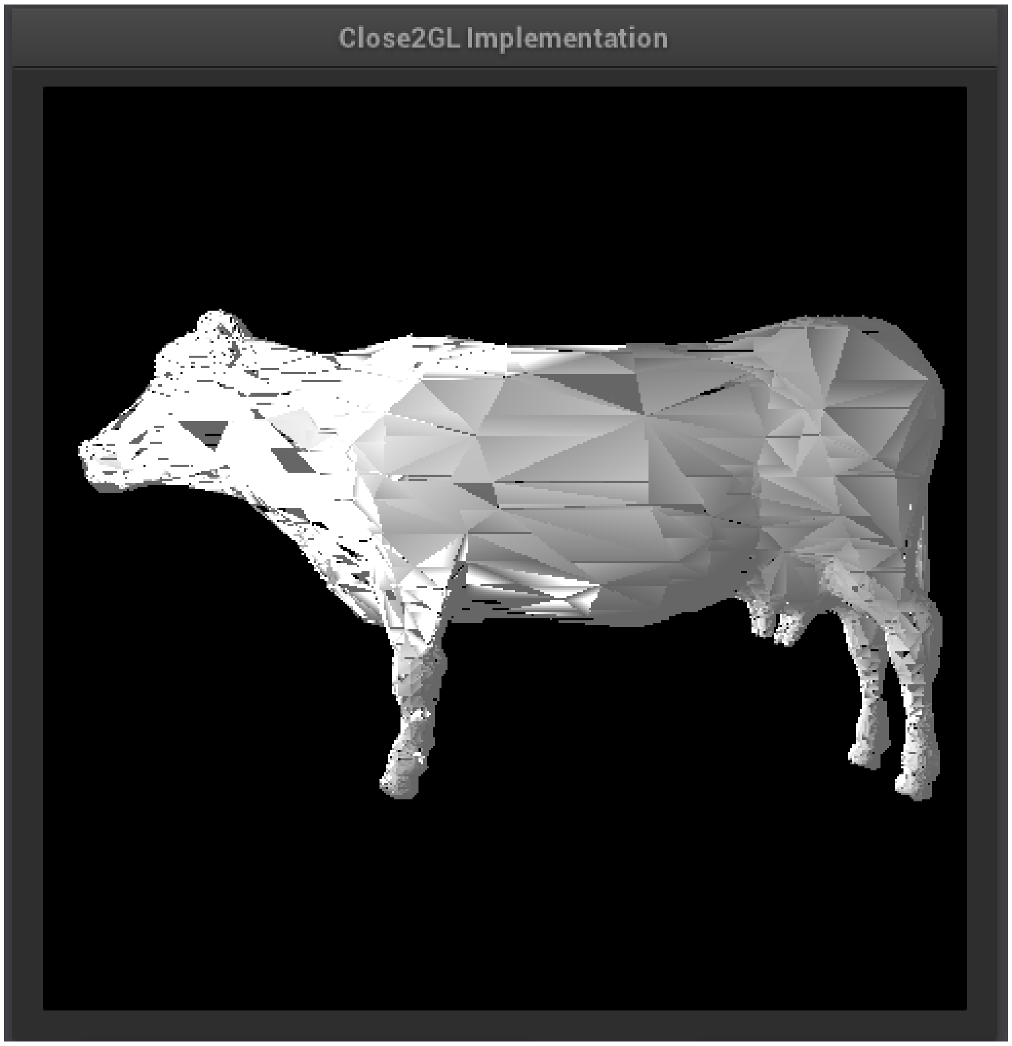


Figure 11: Figure showing Close2GL implementation of the Gouraud shading.

We expect that the images could allow to perceive the subtle differences when using Phong Lighting Model and Gouraud Shading.

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

CMP143 - Computer Graphics: Programming Assignment N 3

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, but we get some rasterization artifact that could not be removed although it does not affect the operation of the application.

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 looking at the FPS rate. In other words, Close2GL implementation is synchronized with OpenGL implementation all the time.

6. We managed to implement the complete graphics pipeline from scratch implementing even the helper and related functions (matrix and vector manipulation, camera movements, model, view, projection and viewport matrix.)

7. 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. Additionally, there could be some **precision** problems that we have to care about that were not completely managed in this implementation.

8. 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.

9. Additionally to the objectives required for this assignment, we allow the POINTS drawing mode for both implementations: OpenGL and Close2GL.

10. We manage to use z-buffer in order to handle visibility as was requested for this assignment.