Assignment 1 - Rendering arbitrary geometric models using OpenGL and GLSL

Fábio de Azevedo Gomes

[1] Universidade Federal do Rio Grande do Sul (UFRGS)
INF01009 / CMP143 Computação Gráfica

fagomes@inf.ufrgs.br

Abstract. *This paper describes an implementation of a 3D triangle mesh loader in C++, with the help of OpenGL and GLSL for the rendering pipeline utilities. Here we describe how certain application requirements were developed, and the mathematical formulas and foundation behind each feature, relating them when possible to the code segments that implement it.*

1. Environment and Tools

The application was developed under a 64 bit Windows 10 operating system, with an Nvidia GeForce GTX 1050, as the GPU, DCH driver version 456.71. The program was compiled using x86_64-w64-mingw32-g++ compiler, version 9.2.0. GLFW was used as the window management library, GL3W as the OpenGL loading library and GLM as the vector and matrix utilities library.

1.1. Disclaimer

All code present in the `lib` folder is **not written by me**, and all credit goes to their respective creators. More information can be found in the references section.

2. Three-Dimensional representations

For this assignment three important entities needed to be modeled in order to present the final picture: `Model`, `Camera` and `Scene`. The following sections explain in detail the choices made for each entity representation, as well as how they interact with each other to compose the final image.

2.1. Models

Three-dimensional models are being represented by the `Model3D` class, which contains the information present in the model file. The `Triangle` structure contain information about it's vertices and normal. The `Vertex` structure contains information about the vertex, such as position, color and normal. The `Material` structure contains information about the ambient, diffuse and specular color, as well as the shine coefficient. Additionally, a `BoundingBox` structure was created for later usage on the camera framing process, and is updated as the model is read from the file.

2.2. Camera

The `Camera` class contains the definition for the camera, which includes the $\vec{u}$, $\vec{v}$ and $\vec{n}$ vectors, pitch roll and yaw values, position and look-at point, near and far plane, and finally field of view. All of the methods that implement camera movement, rotation and object framing are present there.

2.3. Scene

The `Scene` class contains the proper definition of the virtual scene - namely the camera and a list of models. For this first assignment, only one model is present in the scene at a time, but the application was written in such a way that more models can be easily added. This class also has a `PropertyManager`, which is responsible for handling the GUI and user interaction. Every frame a call to `RenderScene()` is done, where properties input by the user are applied and calls to the OpenGL rendering pipeline are made to display the frame.

3. Features

3.1. Model loading, rendering and framing

As explained in a previous section, the model information is read from a file and stored in the appropriate data structures inside the `Model3D` class. After loading the data from the file, a vertex array object is created and bound for the model, as well as 3 vertex buffer objects for position, color and normal of the vertices.

To render the model in three-dimensional perspective view we calculate the `ModelViewProjection` matrix and send it to the vertex shader, where it gets multiplied by each vertex's position in order to obtain it's position in the final scene. For this assignment there is no distinction between object coordinates and world coordinates, since we do not apply any transformations the object, therefore the `Model` matrix is always the identity matrix.

In order to frame the object in the center of the screen, the middle point from the object's bounding box is calculated, and the `x` and `y` values from it are copied to the camera position. In order to obtain the `z` coordinate, we first calculate the distance to the front of the bounding box by right triangle rules, using half the field of view as our angle. We then add the distance to half the bounding box's side length, and finally add that to the point's `z` coordinate. The process is described by formula (1), and visualized in figure 1. In the actual implementation we take the maximum value between the bounding box's front length and front height in order to make sure the entire object will be in-frame.

$$Camera_z = lookAtPoint_z + \frac{bboxSideLength}{2} + \frac{\frac{bboxFrontLength}{2}}{tan(\theta)} \tag{1}$$

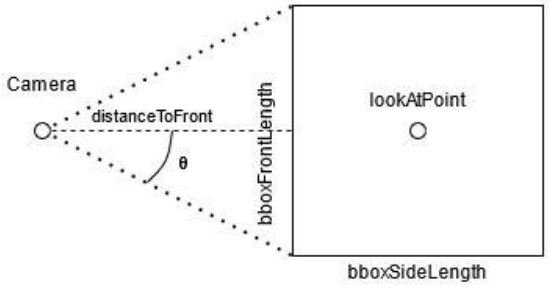

Figure 1. Object framing calculation

In figure 2 you can see an example of a model being rendered and how the camera is positioned using the method described above to keep it in frame.

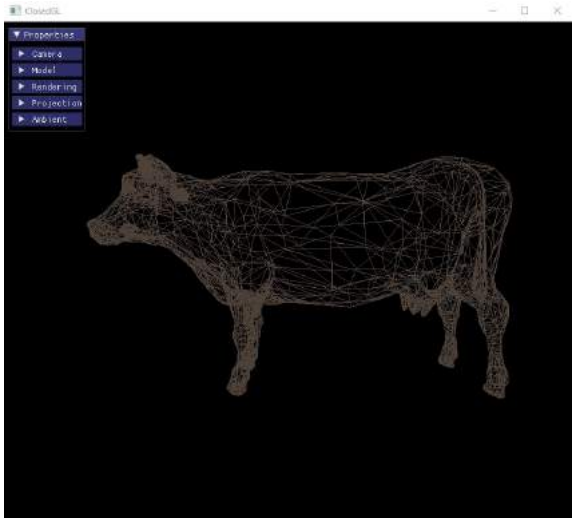

Figure 2. Camera placement upon loading cow_up.in

3.2. Camera interaction

3.2.1. Translation

Because the camera is defined by three vectors, $\vec{u}$, $\vec{v}$ and $\vec{n}$, the translation along it's own axes is a very simple process when executed individually, requiring only for it's current position to be added to the appropriate axis, scaled by the movement speed. This process can be expressed as $newPos = oldPos + movementSpeed * directionAxis$, where direction axis is $\vec{u}$, $\vec{v}$ and $\vec{n}$[1] when moving right, up and forwards, respectively, and $-\vec{u}$, $-\vec{v}$ and $-\vec{n}$[1] when moving left, down and backwards.

When forced to look at the object, we first move the camera normally, then recompute the camera vectors in the following order: First, we compute the new $\vec{n}$ vector as the difference between the point we are looking at the the current position ($lookAtPoint - cameraPosition$), then we compute the new $\vec{u}$ vector as the cross product between $\vec{n}$ and $\vec{v}$. Finally, we compute the new $\vec{v}$ vector as the cross product between the new $\vec{u}$ and $\vec{n}$ vectors.

3.2.2. Rotation

The camera rotation is performed using euler angles to obtain the resulting $\vec{u}$, $\vec{v}$ and $\vec{n}$ camera axes, following the right hand coordinate system convention. To rotate around it's $\vec{n}$ axis, we use (2) and (3).

$$\vec{newU} = \vec{u} * cos(roll) + \vec{v} * sin(roll) \tag{2}$$

$$\vec{newV} = -\vec{u} * sin(roll) + \vec{v} * cos(roll) \tag{3}$$

[1]Here we add $\vec{n}$ and not $-\vec{n}$ because the vector is defined as $\vec{n} = (0, 0, -1)$

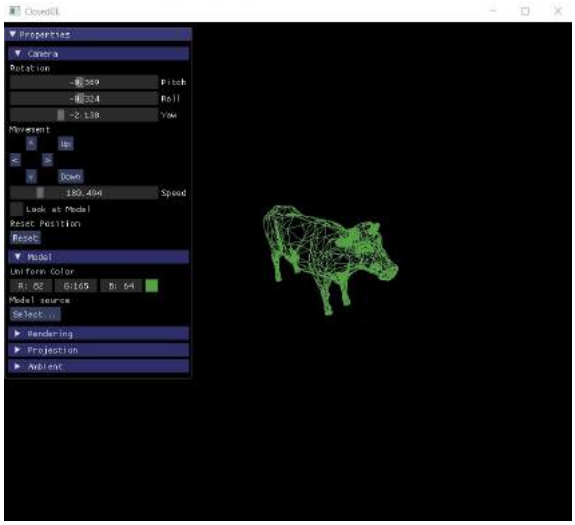

Figure 3. A virtual scene where the camera was rotated and translated to look at cow_up.in from a different position and perspective

To rotate around it's $\vec{u}$ axis, we use (4) and (5).

$$\vec{newV} = \vec{v} * cos(pitch) + \vec{n} * sin(pitch) \tag{4}$$

$$\vec{newN} = -\vec{v} * sin(pitch) + \vec{n} * cos(pitch) \tag{5}$$

To rotate around it's $\vec{v}$ axis, we use (6) and (7).

$$\vec{newU} = \vec{u} * cos(yaw) + \vec{n} * sin(yaw) \tag{6}$$

$$\vec{newN} = -\vec{u} * sin(yaw) + \vec{n} * cos(yaw) \tag{7}$$

These rotations are applied over the original camera coordinate system, which is aligned to the world coordinate system in order to obtain it's new coordinate system. An important side note is that, because we are using euler angles, we run into the Gimbal lock problem, where after rotating one of the axis by 90 degrees we lose a degree of freedom. The axis where this happens is dependant on the order the rotations are applied, and our case it happens after rotating pitch by 90 degrees. Figure 3 shows a virtual camera that has been translated and rotated to obtain a different viewing angle of the model.

3.3. Rendering parameters

As per the assignment requirements, the scene is being rendered based on some parameters that define variable properties, such as the near/far plane and culling procedures. As these features are very simple to define and toggle using OpenGL procedures, below is a list of the parameter and the method or variable being called to apply it to the virtual scene. Most of these are done inside the `RenderUtils` source file.

- **Model color**
  Implemented using a uniform variable in the vertex shader. This variable is updated based on the `Model3D`'s first material diffuse color.

- **Model rendering mode**
  Implemented by calling `glPolygonMode` with values `GL_POINT`, `GL_LINE` and `GL_FILL` to define the rendering primitive. Figure 4 shows these three primtives in action.

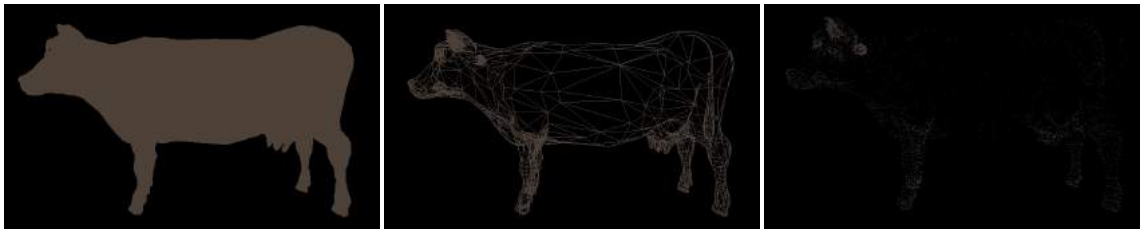

Figure 4. cow_up.in rendered using three different primitives

- **Vertex orientation**
  Implemented with calls to `glFrontFace`, using `GL_CCW` for counter-clockwise and `GL_CW` for clockwise.

  This manipulates the way OpenGL interprets the vertex position information for each triangle, and thus affects the orientation of the normal vector for each triangle. Toggling between these values with flip the normal for each triangle to the opposite direction, affecting the face culling procedure. An interesting property of this interaction is that backface culling with clockwise orientation and front-face culling with counter-clockwise orientation produce the same result, as the flip in the normals causes the opposite facing triangle to be discarded. You can see how face orientation and culling interact in figure 5, where by flipping the orientation but maintaining the backface culling method the front of the cube model is discarded instead of the back, and we can see the inside.

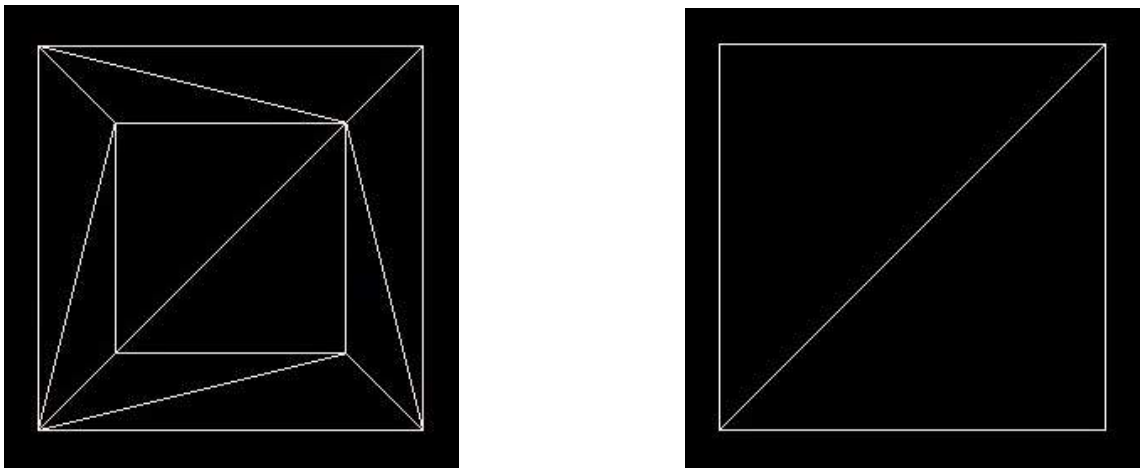

Figure 5. cube.in seen from the front with clockwise (Left) and counter-clockwise (Right) orientations, with backface culling enabled.

- **Face culling**
  Implemented with calls to `glCullFace` with `GL_BACK` for backface culling and `GL_FRONT` for frontface culling. When no culling is selected, we simply disable culling with `glDisable(GL_CULL_FACE)`. Figure 6 shows the three available options for culling.

  It is intereseting to note here that the culling procedure is performed based on the normals for each face, in a such a way that when it is facing away from the camera view (On backface culling) or towards the camera view (On front face culling), it is discarded by OpenGL and not processed any further. This is achieved by calculating the dot product between the face normal and the camera view, as in $\vec{N} \cdot \vec{n}$. On backface culling, for example, if that value is greater than 0 that means the triangle faces away from the camera, and can be safely removed from rendering.

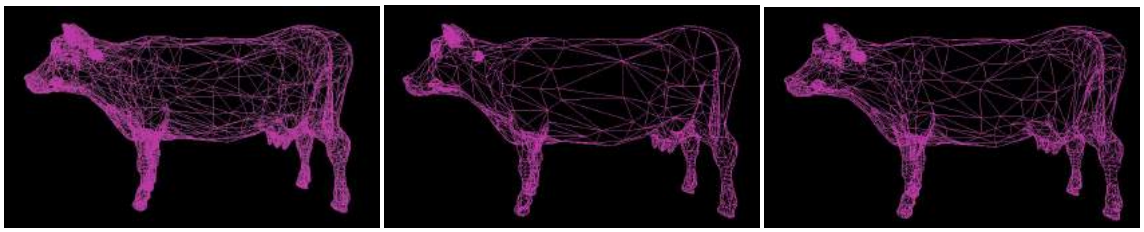

Figure 6. From left to right: No culling, Backface culling and Frontface culling

- **Field of View**
  The field of view is defined as a `float` value controllable by the user, and ranges from $-\pi$ rad to π rad.

  The greater the magnitude for the angle, the more distorted the sides of the camera view appear, as the view is widening and a bigger area is being observed. Unsurprisingly, negative values for the field of view flip the image horizontally and vertically, as the tangent for the angle becomes negative, which is used by the projection matrix.

- **Near/Far plane**
  The near and far planes are being represented by two `float` values, ranging from 0.1 to 10000. The near and far planes are used in the calculation of the projection matrix together with the field of view in order to define the minimum and maximum points that can be seen by the virtual camera. Starting at the camera, points before the near plane and points after the far plane are considered outside of the view frustum, and are discarded to conserve resources. This process is known as clipping.

References

[g truc] g truc. OpenGLMathematics.
https://github.com/g-truc/glm.

[gallickgunner] gallickgunner. ImGui-Addons.
https://github.com/gallickgunner/ImGui-Addons.

[ocornut] ocornut. ImGui.
https://github.com/ocornut/imgui.