Shaders, Colour and Viewing 3D Shapes 4

4.1 Normalized Device Coordinates

In the previous lesson you were probably surprised by how triangles with coordinates with such a small values occupied the whole window. In fact we were just using screen coordinates which are defined as follows: X axis points to your right, Y point up, and Z points towards you (Remember the right hand rule?). The center of this coordinate system is at the middle of your window (Fig:4.1, and ranges in each direction from -1 to 1. You may think of a cube centred on the origin whose vertices have coordinates which are either -1 or 1, for each of the 3 components, and we are seeing it looking along the -Z direction. When objects are brought to this stage we say that they are in NDC (normalized device coordinates). One particular

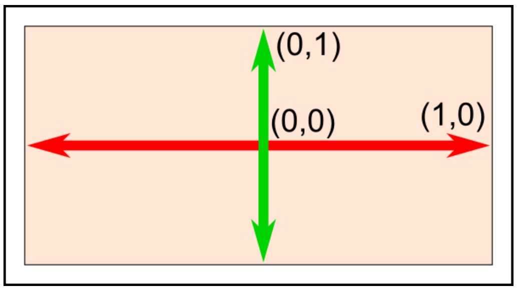

Figure 4.1: Screen coordinates

aspect of OpenGL is that the projection process used does not reduce the dimensions of the 3D scenario, but instead just deforms it [1]. In this case

[1]Question: does this goes against the definition of projection?

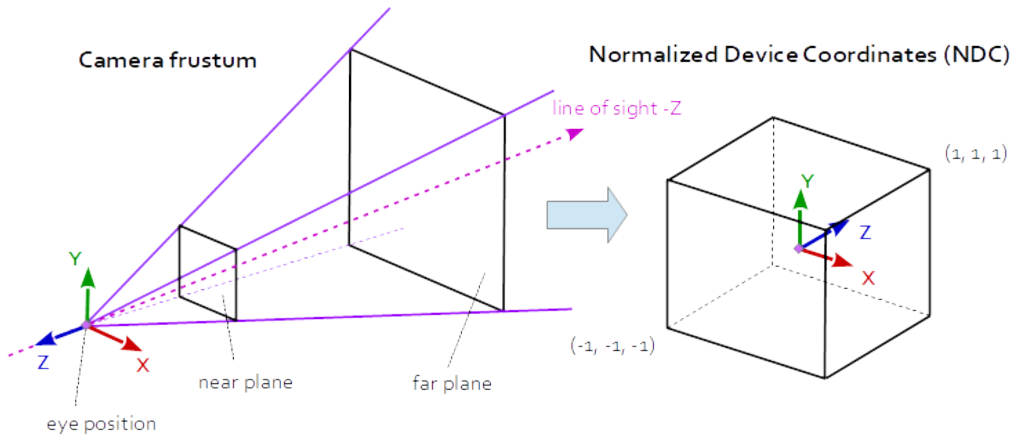

Figure 4.2: OpenGL projection model: trnasforming a frustum onto a cube

the space and its contained geometric structures that lie inside the virtual camera frustum are all deformed so that this frustum is transformed in the above referred cube, see figure 4.2.

Exercise 4.1
Run program lines, analyse its code and explain the difference between it and the triangles code studied on the previous lesson. Do not forget to search on Khronos group site what each function does, so you can explain it.

Exercise 4.2
Change the drawing function to draw a line strip, a triangle strip or a triangle fan, instead of line loop. Analise the results.

4.2 Model, View and Projection Matrices

We have already studied the model and view transformations, where the model transformations are associated with the movements of the object models or parts of it in space, and view transformations with camera movements.

For the purpose of manipulating matrices we will use GLM library, that provides types and constructions that will help us to bridge between C++ of the main programs and GLSL of the shaders. As an example

```
glm::mat4 myMatrix;
glm::vec4 myVector;
//  after giving values to myMatrix and myVector
glm::vec4 transformedVector = myMatrix * myVector;
```

as we can see it is quite similar to the GLSL equivalent

```
mat4 myMatrix;
vec4 myVector;
// after giving values to myMatrix and myVector
vec4 transformedVector = myMatrix * myVector;
```

Although you can fill the values of the matrices by yourself and in many situations you want to do that, GLM also provides some help for the specifying the basic transformations, as follows

```
#include <glm/glm.hpp>
#include <glm/gtx/transform.hpp>

// Translation
glm::mat4 TranslationMat = glm::translate(glm::mat4(),
                                glm::vec3(10.0f, 0.0f, 0.0f));
glm::vec4 myVector(10.0f, 10.0f, 10.0f, 0.0f);
glm::vec4 transformedVector = TranslationMat * myVector

// Identity and scale
glm::mat4 IdentityMat = glm::mat4(1.0f);
glm::mat4 ScalingMat = glm::scale(2.0f, 2.0f ,2.0f);

// Rotations
glm::vec3 myRotAxis(a, b, c);
glm::mat4 RotMat = glm::rotate(glm::degrees(angle_in_degrees),myRotAxis);
glm::mat4 RotMat2 = glm::rotate(glm::radians(angle_in_radians),myRotAxis);

// Composition example
transformedVector = TranslationMat * RotMat * ScalingMat * MyVector;
```

4.3 View transformations

View transformations were already discussed and you may think of them as: If you want to go and see the ocean, either you go to the ocean or you bring the ocean to you. In fact positioning a camera means defining the point in space where the camera is, orient it so that it looks in the right direction (towards the object of interest) and define the upper side of the view. For the latter part think of yourself looking at something in a straight up head pose or tilting the head towards the left shoulder.

GLM provides a function for defining the view matrix given the following inputs

```
glm::mat4 ViewMatrix = glm::lookAt(
    cameraPosition, // the position of your camera, in world space
    cameraTarget,   // where you want to look at, in world space
    upVector        // probably glm::vec3(0,1,0)... or not
);
```

The first two parameters are self-explained, but the upVector may raise some doubts. Think of it as the orientation of the top of your camera when you take a photo. You place the camera somewhere in space, point the lens at the object, and in most cases you keep the top part of the camera towards the ceiling or sky. But you may want to tilt it to the left or to the right for aligning the photo with the object, for example. So the upVector indicates the direction that defines the top side of your view with respect to the world (and object in it). This function just provides a method to compute the view matrix based on the above information. This "ViewMatrix" in reality is the one that transforms world coordinates into camera coordinates.

4.4 Projection

The last missing transformation is the projection. A projection is by definition a mapping of a set (or structure) into a subset (or sub-structure). Typically we are interested in graphics and computer vision courses on the image generation from objects or models.

There are slight differences between projection matrices in computer vision and those used in OpenGL, due to the processing pipeline. In OpenGL the projection brings any point inside the view frustrum (referred in the classes of this course), which is the view volume, and a normalised cube. Here, contrary to the expected, the projected points will not loose depth. This indeed serves for visibility processing, also called hidden parts removal, but it suffices to ignore the z coordinate to get the 2D projections of the points, after being divided by the homogeneous coordinate.

GLM provides functions to generate these matrices also, and for the case of a perspective camera the function is glm::perspective as follows, where

```
glm::mat4 projectionMatrix = glm::perspective(
        glm::radians(FoV),
        aspectratio,
        nearplane,
        farplane
);
```

FoV The vertical Field of View, in radians: the amount of "zoom". Think "camera lens". Usually between 90º (extra wide) and 30º (quite zoomed in)

aspectratio Aspect Ratio. Depends on the size of your window. Notice that 4/3 = 800/600 = 1280/960

nearplane Near clipping plane distance. Keep as big as possible, or you may have precision issues.

farplane Far clipping plane distance. Keep as little as possible, for the same reason as above.

Now you may compute the transformation that is going to be applied to each vertex of a model as

```
glm::mat4 MVPmatrix = projection * view * model;
```

Exercise 4.3
Inspire yourself on the lines program and create a 3D cube as a set of triangles using only coordinates of value -1 and 1, where each vertex should have a different colour. Define a model matrix that places the cube at $(10, 10, 10)$. Define the camera and the view matrices of your choice. Do not forget that the resulting matrix must be passed as a uniform to the shader to be applied to all vertices.

Exercise 4.4
Draw multiple cubes at different positions and continuously rotating about different axes.

Exercise 4.5
Create a different pair of shaders that also vary the scale and colour along time of some of the cubes.