

Application of Graphics Principles Assignment - By Mateusz

by Mateusz Jadasz

Submission date: 18-Nov-2022 07:05PM (UTC+0000)

Submission ID: 189492152

File name: 17845_Mateusz_Jadasz_Application_of_Graphics_Principles_Assignment_-
_By_Mateusz_467856_1167421200.docx (89.18K)

Word count: 2991

Character count: 17466

APPLICATION OF GRAPHICS PRINCIPLES

CONTENTS

Introduction	3
The Graphics Pipeline	3
Vertex Processing	3
Clipping and Primitive Assembly	3
Rasterization	3
Fragment Processing	4
Optimisation Methods	4
Code Testing	5
Object Conversion – From Object space to view Volume	6
Translation	6
Rotation	7
Viewspace Conversion	8
Camera U Axis Rotation	8
Camera V Axis Rotation	9
Camera N Axis Rotation	10
Viewspace Conversion	11
Perspective Projection	13
Calculating the Culling Plane	15
Bibliography	16
Appendix	17

INTRODUCTION

OpenGL, or any other graphics programming interface, relies almost entirely on applying existing mathematical methods and processes, to produce things such as a render of an image, or for calculating things such as object movement and rotation. Although these aren't the only things OpenGL can do, in fact saying that this is even a quarter of what the interface is capable of would be doing it a massive injustice. However, the focus of this report is precisely what was just mentioned and more. Using matrices, angles, and vectors, we can calculate the original position of an object as well as its position after transforming it, rotating it, and even converting it into view space and after applying a perspective projection.

THE GRAPHICS PIPELINE

The graphics pipeline is what allows an object to be rendered and therefore seen on a screen. The pipeline itself, comprises of four major steps, which allow it to carry out the imaging process.

VERTEX PROCESSING

The purpose of the first stage of the graphics pipeline is to calculate the appropriate coordinate transformations and then to determine the colour of every vertex. Calculating the transformations is done using 4x4 matrices, which are used to represent all the different coordinates, those being the X, Y, Z and the homogenous coordinate (often represented as W). The difficulty of applying colour to each vertex is directly impacted by the complexity of the object that we would like to colour. For example, if we want to display a simple primitive, with a solid colour, then colour application would be as simple as providing each vector of the primitive, with a normalised RGB value. On the other hand, if we were to do something more complex such as colouring in a pool of water in a 3D game, then the process becomes much more difficult, as the colour would have to be calculated while also taking into consideration things like lighting, and possibly surface properties like reflecting the surrounding environment.

CLIPPING AND PRIMITIVE ASSEMBLY

During the second stage of the graphics pipeline, the vertices are clipped accordingly. This is purely because, there is no way for the synthetic camera to capture and display the entire world, in a 360° field of view around the player, and even if it could, it would be very suboptimal as it would mean that everything must be rendered, all at the same time. This would become problematic even further when the player starts to move and rotate, as it would mean that the entire world would have to start scaling, translating and possibly even rotating all at the same time, while also keeping objects on screen rendered. This problem is solved through clipping, which essentially de-renders everything outside of the camera frustum, which in turn reduces the load on the GPU by a significant amount. Clipping is done on a primitive-by-primitive basis, which means that instead of clipping every vertex one by one, the vertices must be assembled into primitives like polygons and lines, for them to be clipped. This part of the pipeline process stage is the primitive assembly.

RASTERIZATION

The third stage of the graphics pipeline is the rasterization of the different pixels that are present within a primitive. Those pixels (fragments) are coloured in accordingly based on whether they find themselves within a primitive shape, which is all calculated by the rasterizer. The rasterizer examines a shape that has not been clipped, such as a triangle by locating its three vertices, and then determining which pixels are within the zone of the three vertices, and finally by assigning a colour to each fragment, so that it can be converted into the frame buffer.

FRAGMENT PROCESSING

The fourth and final stage of the graphics pipeline is the fragment processing. This last stage simply updates the fragments that were generated during the rasterization stage, in the frame buffer. However, if the surfaces that are being processed are three dimensional, it may mean that some surfaces may not be visible as there are things that affect it such as perspective. Certain surfaces may be behind others, also some parts of the surface may be lighter/darker than others because of lighting as well as bump and texture mapping.

OPTIMISATION METHODS

Modern GPU's such as the relatively recent 3000 series Nvidia processors, or even the even more recent 4000 series, will contain hundreds or small processing cores, which allows them to work incredibly fast, especially when compared to the common way that CPUs are constructed, where they usually have around 6-8 cores. Even though some of the newer CPUs such as the AMD Ryzen 9 3900x have 12 cores, it still does not compare to a GPU, such as the RTX 3080, when it comes to processing graphics, although admittedly, that is partially because the 'X' series of the AMD processors do not have integrated graphics, but also partially because graphics processing units have hundreds more, smaller cores which allow it to parallelise the different graphics data. Parallelisation is the process of splitting up larger, more demanding tasks into smaller, easier to handle tasks, to achieve the overarching, larger, more demanding tasks at a much more efficient rate.

CODE TESTING

Test	Screenshot	Description
Calculation process without rounding numbers	<pre>-----[Task 1]----- Task 1 where none of the numbers were rounded Translated vector: (3.5, 2, -3, 1) Rotated vector: (1.53109, 2, -4.34888, 1) Viewspace Converted Vector: (3.66747, -4.33494, -0.517949, 1) Perspective Projection Vector: (20.564, -24.3066, -4.86453, 1)</pre>	The numbers that were calculated without rounding any values, have turned out to have differed from, the values that I have calculated by hand. Most of them however, did not differ from my own results by anything higher than 0.02. The exception to this is of course the perspective projection vector. However, I believe this is purely because of the differences in values that occurred, the further down the process I went.
Calculation process while rounding all the numbers	<pre>Task 1 where all of the numbers were rounded to two decimal places Translated vector: (3.5, 2, -3, 1) Rotated vector: (1.55, 2, -4.36, 1) Viewspace Converted Vector: (3.68, -4.3, -0.5, 1) Perspective Projection Vector: (21.34, -24.94, -5.08, 1)</pre>	If every single value that exceeded two decimal places, was rounded, every single number seemed to have been equal to everything I have calculated
Culling Plane Calculation	<pre>-----[Task 2]----- Distance from culling plane: 4.5 Therefore, the startingVector is above the culling plane and will not be culled</pre>	Just like the object conversion from object space to view space calculations, the culling plane calculation has also seemed to agree with the number that I have personally concluded.

Vertex	Rotation Angle	Translation	Viewpoint	Up Vector	Look At Point	Field of view	Near plane distance	Far plane distance	Aspect Ratio
(1,2,-3)	30.0°	(2.5, 0, -1)	(0, 4, -1)	(0, 1, 0)	(0.5, 0, 0)	38.0°	1.5	80	1:1

OBJECT CONVERSION – FROM OBJECT SPACE TO VIEW VOLUME

TRANSLATION

Where V is the desired vector:

$$\vec{V} = \begin{bmatrix} 1 & 0 & 0 & 2.5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} 1 \\ 2 \\ -3 \\ 1 \end{pmatrix}$$

$$\vec{V}_x = [1 * 1 + 0 * 2 + 0 * -3 + 2.5 * 1]$$

$$\vec{V}_x = [1 + 0 + 0 + 2.5]$$

$$\vec{V}_x = 3.5$$

$$\vec{V}_y = [0 * 1 + 1 * 2 + 0 * -3 + 0 * 1]$$

$$\vec{V}_y = [0 + 2 + 0 + 0]$$

$$\vec{V}_y = 2$$

$$\vec{V}_z = [0 * 1 + 0 * 2 + 1 * -3 + 0 * 1]$$

$$\vec{V}_z = [0 + 0 + -3 + 0]$$

$$\vec{V}_z = -3$$

$$\vec{V}_w = [0 * 1 + 0 * 2 + 0 * -3 + 1 * 1]$$

$$\vec{V}_w = [0 + 0 + 0 + 1]$$

$$\vec{V}_w = 1$$

$$\vec{V} = \begin{pmatrix} 3.5 \\ 2 \\ -3 \\ 1 \end{pmatrix}$$

ROTATION

$$\vec{V} = (3.5, 2, -3, 1)$$

$$R = \begin{bmatrix} \cos 30 & 0 & \sin 30 & 0 \\ 0 & 1 & 0 & 0 \\ -\sin 30 & 0 & \cos 30 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} 3.5 \\ 2 \\ -3 \\ 1 \end{pmatrix}$$

$$\sin(30) = 0.5$$

$$-\sin(30) = -0.5$$

$$\cos(30) = 0.87$$

$$R_x = [0.87 * 3.5 + 0 * 2 + 0.5 * -3 + 0 * 1]$$

$$R_x = [3.05 + 0 + -1.5 + 0]$$

$$R_x = 1.55$$

$$R_y = [0 * 3.5 + 1 * 2 + 0 * -3 + 0 * 1]$$

$$R_y = [0 + 2 + 0 + 0]$$

$$R_y = 2$$

$$R_z = [-0.5 * 3.5 + 0 * 2 + 0.87 * -3 + 0 * 1]$$

$$R_z = [-1.75 + 0 + -2.61 + 0]$$

$$R_z = -4.36$$

$$R_w = [0 * 3.5 + 0 * 2 + 0 * -3 + 1 * 1]$$

$$R_w = [0 + 0 + 0 + 1]$$

$$R_w = 1$$

$$R = \begin{pmatrix} 1.55 \\ 2 \\ -4.36 \\ 1 \end{pmatrix}$$

VIEWSPACE CONVERSION

$$\sin(-30) = -0.5$$

$$-\sin(-30) = 0.5$$

$$\cos(-30) = 0.87$$

CAMERA U AXIS ROTATION

$$C = (0, 4, -1, 1)$$

$$U = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos -30 & -\sin -30 & 0 \\ 0 & \sin -30 & \cos -30 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} 0 \\ 4 \\ -1 \\ 1 \end{pmatrix}$$

$$U_x = [1 * 0 + 0 * 4 + 0 * -1 + 0 * 1]$$

$$U_x = [0 + 0 + 0 + 0]$$

$$U_x = 0$$

$$U_y = [0 * 0 + 0.87 * 4 + 0.5 * -1 + 0 * 1]$$

$$U_y = [0 + 3.48 + -0.5 + 0]$$

$$U_y = 2.98$$

$$U_z = [0 * 0 + -0.5 * 4 + 0.87 * -1 + 0 * 1]$$

$$U_z = [0 + -2 + -0.87 + 0]$$

$$U_z = -2.87$$

$$U_w = [0 * 0 + 0 * 4 + 0 * -1 + 1 * 1]$$

$$U_w = [0 + 0 + 0 + 1]$$

$$U_w = 1$$

$$U = \begin{pmatrix} 0 \\ 2.98 \\ -2.87 \\ 1 \end{pmatrix}$$

CAMERA V AXIS ROTATION

$$C = (0, 4, -1, 1)$$

$$V = \begin{bmatrix} \cos -30 & 0 & \sin -30 & 0 \\ 0 & 1 & 0 & 0 \\ -\sin -30 & 0 & \cos -30 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} 0 \\ 4 \\ -1 \\ 1 \end{pmatrix}$$

$$V_x = [0.87 * 0 + 0 * 4 + -0.5 * -1 + 0 * 1]$$

$$V_x = [0 + 0 + 0.5 + 0]$$

$$V_x = 0.5$$

$$V_y = [0 * 0 + 1 * 4 + 0 * -1 + 0 * 1]$$

$$V_y = [0 + 4 + 0 + 0]$$

$$V_y = 4$$

$$V_z = [0.5 * 0 + 0 * 4 + 0.87 * -1 + 0 * 1]$$

$$V_z = [0 + 0 + -0.87 + 0]$$

$$V_z = -0.87$$

$$V_w = [0 * 0 + 0 * 4 + 0 * -1 + 1 * 1]$$

$$V_w = [0 + 0 + 0 + 1]$$

$$V_w = 1$$

$$V = \begin{pmatrix} 0.5 \\ 4 \\ -0.87 \\ 1 \end{pmatrix}$$

CAMERA N AXIS ROTATION

$$C = (0, 4, -1, 1)$$

$$N = \begin{bmatrix} \cos -30 & -\sin -30 & 0 & 0 \\ \sin -30 & \cos -30 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} 0 \\ 4 \\ -1 \\ 1 \end{pmatrix}$$

$$N_x = [0.87 * 0 + 0.5 * 4 + 0 * -1 + 0 * 1]$$

$$N_x = [0 + 2 + 0 + 0]$$

$$N_x = 2$$

$$N_y = [-0.5 * 0 + 0.87 * 4 + 0 * -1 + 0 * 1]$$

$$N_y = [0 + 3.48 + 0 + 0]$$

$$N_y = 3.48$$

$$N_z = [0 * 0 + 0 * 4 + 1 * -1 + 0 * 1]$$

$$N_z = [0 + 0 + -1 + 0]$$

$$N_z = -1$$

$$N_w = [0 * 0 + 0 * 4 + 0 * -1 + 1 * 1]$$

$$N_w = [0 + 0 + 0 + 1]$$

$$N_w = 1$$

$$N = \begin{pmatrix} 2 \\ 3.48 \\ -1 \\ 1 \end{pmatrix}$$

VIEWSPACE CONVERSION

MODEL VIEW MATRIX

$$MV = \begin{bmatrix} U_x & U_y & U_z & 0 \\ V_x & V_y & V_z & 0 \\ N_x & N_y & N_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -C_x \\ 0 & 1 & 0 & -C_y \\ 0 & 0 & 1 & -C_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$MV = \begin{bmatrix} 0 & 2.98 & -2.87 & 0 \\ 0.5 & 4 & -0.87 & 0 \\ 2 & 3.48 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -4 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$MV = \begin{bmatrix} 0*1+2.98*0+-2.87*0+0*0 & 0*0+2.98*1+-2.87*0+0*0 & 0*0+2.98*0+-2.87*1+0*0 & 0*0+2.98*-4+-2.87*1+0*1 \\ 0.5*1+4*0+-0.87*0+0*0 & 0.5*0+4*1+-0.87*0+0*0 & 0.5*0+4*0+-0.87*1+0*0 & 0.5*0+4*-4+-0.87*1+0*1 \\ 2*1+3.48*0+-1*0+0*0 & 2*0+3.48*1+-1*0+0*0 & 2*0+3.48*0+-1*1+0*0 & 2*0+3.48*-4+-1*1+0*1 \\ 0*1+0*0+0*0+1*0 & 0*0+0*1+0*0+1*0 & 0*0+0*0+0*0+1*0 & 0*0+0*-4+0*1+1*1 \end{bmatrix}$$

$$MV = \begin{bmatrix} 0+0+0+0 & 0+2.98+0+0 & 0+0+-2.87+0 & 0+-11.92+-2.87+0 \\ 0.5+0+0+0 & 0+4+0+0 & 0+0+-0.87+0 & 0+-16+-0.87+0 \\ 2+0+0+0 & 0+3.48+0+0 & 0+0+-1+0 & 0+-13.92+-1+0 \\ 0+0+0+0 & 0+0+0+0 & 0+0+0+0 & 0+0+0+1 \end{bmatrix}$$

$$MV = \begin{bmatrix} 0 & 2.98 & -2.87 & -14.79 \\ 0.5 & 4 & -0.87 & -16.87 \\ 2 & 3.48 & -1 & -14.92 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

POINT IN VIEWSPACE

$$PC = MV * R$$

$$PC = \begin{bmatrix} 0 & 2.98 & -2.87 & -14.79 \\ 0.5 & 4 & -0.87 & -16.87 \\ 2 & 3.48 & -1 & -14.92 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} 1.55 \\ 2 \\ -4.36 \\ 1 \end{pmatrix}$$

$$PC_x = [0 * 1.55 + 2.98 * 2 + -2.87 * -4.36 + -14.79 * 1]$$

$$PC_x = [0 + 5.96 + 12.51 + -14.79]$$

$$PC_x = 3.68$$

$$PC_y = [0.5 * 1.55 + 4 * 2 + -0.87 * -4.36 + -16.87 * 1]$$

$$PC_y = [0.78 + 8 + 3.79 + -16.87]$$

$$PC_y = -4.3$$

$$PC_z = [2 * 1.55 + 3.48 * 2 + -1 * -4.36 + -14.92 * 1]$$

$$PC_z = [3.1 + 6.96 + 4.36 + -14.92]$$

$$PC_z = -0.5$$

$$PC_w = [0 * 1.55 + 0 * 2 + 0 * -4.36 + 1 * 1]$$

$$PC_w = [0 + 0 + 0 + 1]$$

$$PC_w = 1$$

$$PC = \begin{pmatrix} 3.68 \\ -4.3 \\ -0.5 \\ 1 \end{pmatrix}$$

PERSPECTIVE PROJECTION

$$PP = P_m * PC$$

$$P_m = \begin{bmatrix} A & 0 & 0 & 0 \\ 0 & q & 0 & 0 \\ 0 & 0 & B & C \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$q = \frac{1}{\tan\left(\frac{FieldOfView}{2}\right)}$$

$$q = \frac{1}{\tan\left(\frac{38}{2}\right)}$$

$$q = 2.90$$

$$A = \frac{q}{AspectRatio}$$

$$A = \frac{2.90}{1}$$

$$A = 2.90$$

$$B = \frac{NearPlane_z + FarPlane_z}{NearPlane_z - FarPlane_z}$$

$$B = \frac{1.5 + 80}{1.5 - 80}$$

$$B = -1.04$$

$$C = \frac{2 * (NearPlane_z * FarPlane_z)}{NearPlane_z - FarPlane_z}$$

$$C = \frac{2 * (1.5 * 80)}{1.5 - 80}$$

$$C = -3.06$$

$$PP = \begin{bmatrix} 2.90 & 0 & 0 & 0 \\ 0 & 2.90 & 0 & 0 \\ 0 & 0 & -1.04 & -3.06 \\ 0 & 0 & -1 & 0 \end{bmatrix} * \begin{pmatrix} 3.68 \\ -4.3 \\ -0.5 \\ 1 \end{pmatrix}$$

$$PP_x = 2.90 * 3.68 + 0 * -4.3 + 0 * -0.5 + 0 * 1$$

$$PP_x = 10.67 + 0 + 0 + 0$$

$$PP_x = 10.67$$

$$PP_y = 0 * 3.68 + 2.90 * -4.3 + 0 * -0.5 + 0 * 1$$

$$PP_y = 0 + -12.47 + 0 + 0$$

$$PP_y = -12.47$$

$$PP_z = 0 * 3.68 + 0 * -4.3 + -1.04 * -0.5 + -3.06 * 1$$

$$PP_z = 0 + 0 + 0.52 + -3.06$$

$$PP_z = -2.54$$

$$PP_w = 0 * 3.68 + 0 * -4.3 + -1 * -0.5 + 0 * 1$$

$$PP_w = 0 + 0 + 0.5 + 0$$

$$PP_w = 0.5$$

$$PP = \frac{\begin{pmatrix} 10.67 \\ -12.47 \\ -2.54 \\ 0.5 \end{pmatrix}}{0.5}$$

$$PP = \begin{pmatrix} 21.34 \\ -24.94 \\ -5.08 \\ 1 \end{pmatrix}$$

CALCULATING THE CULLING PLANE

$$x \cdot n + P_0 \cdot n = 0$$

$$x \cdot n = (1, 2, -3) \cdot (-1, 2, 0)$$

$$x \cdot n = (1 * -1) + (2 * 2) + (-3 * 0)$$

$$x \cdot n = -1 + 4 + 0$$

$$x \cdot n = 3$$

$$P_0 \cdot n = (2.5, 2, 4) \cdot (-1, 2, 0)$$

$$P_0 \cdot n = (2.5 * -1) + (2 * 2) + (4 * 0)$$

$$P_0 \cdot n = -2.5 + 4 + 0$$

$$P_0 \cdot n = 1.5$$

Where D is the distance from the culling plane:

$$D = x \cdot n + P_0 \cdot n$$

$$D = 3 + 1.5$$

$$D = 4.5$$

Therefore, vector x is not on the culling plane, as its distance from the plane is equal to 4.5, which means that it is positioned above the culling plane, and therefore, it will not be culled.

BIBLIOGRAPHY

- [1] E. Angel and D. Shreiner, Interactive Computer Graphics: A Top-Down Approach With Shader-Based OpenGL, 6th ed., pp. 35-37.
- [2] G. Shakah, M. Alkhasawneh, V. Krasnoproshin and D. Mazouka, "Graphics Pipeline Evolution: Problems and Solutions," *Journal of Computer Science*.
- [3] D. Shreiner, "OpenGL Programming Guide," 2013. [Online]. Available: <http://www.glprogramming.com/red/index.html>. [Accessed 11 November 2022].

APPENDIX

```
#include "glm/glm.hpp"
#include "glm/ext.hpp"
#include "glm/gtx/string_cast.hpp"
#include <iostream>
#include <math.h>
#define _USE_MATH_DEFINES
using namespace std;

glm::vec4 startingVector = glm::vec4(1, 2, -3, 1);
glm::vec4 translationVec;
glm::vec4 rotationVec;
glm::vec4 viewspaceVec;
glm::vec4 perspectiveVec;
float rotation = 30.0f;
float fov = 38.0f;
float nearPlane = 1.5f;
float farPlane = 80.0f;

float RoundToNearest(float value) {
    float roundedValue = round(value * 100) / 100;
    return roundedValue;
}

float cosRounded(float number) {
    float value;

    value = RoundToNearest(cos(number));

    return value;
}

float degreesToRadians(float degrees) {
    double pi = 2 * acos(0);
    float rotationRadians = degrees * (pi / 180);

    return rotationRadians;
}

glm::vec4 UpdatedVector(glm::mat4 matrix, glm::vec4 vector) {
    glm::vec4 newVector = matrix * vector;
    newVector = glm::vec4(RoundToNearest(newVector.x), RoundToNearest(newVector.y),
        RoundToNearest(newVector.z), RoundToNearest(newVector.w));

    return newVector;
}

float dot(glm::vec3 vector1, glm::vec3 vector2) {
    float product = (vector1.x * vector2.x) + (vector1.y * vector2.y) + (vector1.z *
        vector2.z);

    return product;
}

void Task1() {
    glm::mat4 translationMatrix = glm::mat4(1.0, 0.0, 0.0, 0.0,
        0.0, 1.0, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        2.5, 0.0, 0.0, 1.0);

    translationVec = translationMatrix * startingVector;
```

```

        std::cout << "Translated vector: " << "(" << translationVec.x << ", " <<
translationVec.y << ", " << translationVec.z << ", " << translationVec.w << ")" << endl;

        glm::mat4 rotationMatrix = glm::mat4(cos(degreesToRadians(rotation)), 0, -
sin(degreesToRadians(rotation)), 0,
        0, 1, 0, 0,
        sin(degreesToRadians(rotation)), 0, cos(degreesToRadians(rotation)), 0,
        0, 0, 0, 1);

        rotationVec = rotationMatrix * translationVec;

        /*std::cout << "Sin: " << sin(rotationRadians) << "\n";
std::cout << "-Sin: " << -sin(rotationRadians) << "\n";
std::cout << "Cos: " << cos(rotationRadians) << "\n";
std::cout << "Rotation in Radians: " << rotationRadians << "\n";*/

        std::cout << "Rotated vector: " << "(" << rotationVec.x << ", " << rotationVec.y <<
", " << rotationVec.z << ", " << rotationVec.w << ")" << endl;

        glm::vec4 camPos = glm::vec4(0, 4, -1, 1);
        glm::mat4 xAxisRotation = glm::mat4(1, 0, 0, 0,
        0, cos(-degreesToRadians(rotation)), sin(-degreesToRadians(rotation)), 0,
        0, -sin(-degreesToRadians(rotation)), cos(-degreesToRadians(rotation)), 0,
        0, 0, 0, 1);
        glm::mat4 yAxisRotation = glm::mat4(cos(-degreesToRadians(rotation)), 0, -sin(-
degreesToRadians(rotation)), 0,
        0, 1, 0, 0,
        sin(-degreesToRadians(rotation)), 0, cos(-degreesToRadians(rotation)), 0,
        0, 0, 0, 1);
        glm::mat4 zAxisRotation = glm::mat4(cos(-degreesToRadians(rotation)), sin(-
degreesToRadians(rotation)), 0, 0,
        -sin(-degreesToRadians(rotation)), cos(-degreesToRadians(rotation)), 0, 0,
        0, 0, 1, 0,
        0, 0, 0, 1);

        glm::vec4 U = xAxisRotation * camPos;
        glm::vec4 V = yAxisRotation * camPos;
        glm::vec4 N = zAxisRotation * camPos;

        /*std::cout << "U: " << glm::to_string(U) << endl;
std::cout << "V: " << glm::to_string(V) << endl;
std::cout << "N: " << glm::to_string(N) << endl;*/

        glm::mat4 modelView = glm::mat4(U.x, V.x, N.x, 0,
        U.y, V.y, N.y, 0,
        U.z, V.z, N.z, 0,
        0, 0, 0, 1) *
        glm::mat4(1, 0, 0, 0,
        0, 1, 0, 0,
        0, 0, 1, 0,
        -camPos.x, -camPos.y, -camPos.z, 1);

        viewspaceVec = modelView * rotationVec;

        std::cout << "Viewspace Converted Vector: " << "(" << viewspaceVec.x << ", " <<
viewspaceVec.y << ", " << viewspaceVec.z << ", " << viewspaceVec.w << ")" << endl;

        //glm::mat4 perspective = glm::perspective(degreesToRadians(fov), (float)1 /
(float)1, nearPlane, farPlane);
        //std::cout << "Perspective Matrix: " << glm::to_string(perspective) << endl;

        float q = 1 / tan(degreesToRadians(fov) / 2);
        float A = q / ((float)1 / (float)1);
        float B = (1.5 + 80) / (1.5 - 80);
        float C = (2 * (1.5 * 80)) / (1.5 - 80);

```

```

    /*std::cout << "A: " << A << endl;
    std::cout << "Q: " << q << endl;
    std::cout << "B: " << B << endl;
    std::cout << "C: " << C << endl;*/

    glm::mat4 perspective = glm::mat4(A, 0, 0, 0,
        0, q, 0, 0,
        0, 0, B, -1,
        0, 0, C, 0);

    perspectiveVec = perspective * viewSpaceVec;
    perspectiveVec = glm::vec4(perspectiveVec.x / perspectiveVec.w, perspectiveVec.y /
perspectiveVec.w, perspectiveVec.z / perspectiveVec.w, perspectiveVec.w /
perspectiveVec.w);

    std::cout << "Perspective Projection Vector: " << "(" << perspectiveVec.x << ", "
<< perspectiveVec.y << ", " << perspectiveVec.z << ", " << perspectiveVec.w << ")" << endl;
}

void Task1Rounded() {

    glm::mat4 translationMatrix = glm::mat4(1.0, 0.0, 0.0, 0.0,
        0.0, 1.0, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        2.5, 0.0, 0.0, 1.0);

    translationVec = UpdatedVector(translationMatrix, startingVector);

    std::cout << "Translated vector: " << "(" << translationVec.x << ", " <<
translationVec.y << ", " << translationVec.z << ", " << translationVec.w << ")" << endl;

    glm::mat4 rotationMatrix = glm::mat4(cosRounded(degreesToRadians(rotation)), 0, -
sin(degreesToRadians(rotation)), 0,
        0, 1, 0, 0,
        sin(degreesToRadians(rotation)), 0, cosRounded(degreesToRadians(rotation)),
0,
        0, 0, 0, 1);

    rotationVec = UpdatedVector(rotationMatrix, translationVec);

    /*std::cout << "Sin: " << sin(rotationRadians) << "\n";
    std::cout << "-Sin: " << -sin(rotationRadians) << "\n";
    std::cout << "Cos: " << RoundToNearest(cos(rotationRadians)) << "\n";
    std::cout << "Rotation in Radians Unrounded: " << rotationRadians << "\n";
    std::cout << "Rotation in Radians Rounded: " << RoundToNearest(rotationRadians) <<
"\n";*/
    std::cout << "Rotated vector: " << "(" << rotationVec.x << ", " << rotationVec.y <<
", " << rotationVec.z << ", " << rotationVec.w << ")" << endl;

    glm::vec4 camPos = glm::vec4(0, 4, -1, 1);
    glm::mat4 xAxisRotation = glm::mat4(1, 0, 0, 0,
        0, cosRounded(-
degreesToRadians(rotation)), sin(-degreesToRadians(rotation)), 0,
        0, -sin(-
degreesToRadians(rotation)), cosRounded(-degreesToRadians(rotation)), 0,
        0, 0, 0, 1);

    glm::mat4 yAxisRotation = glm::mat4(cosRounded(-degreesToRadians(rotation)), 0, -
sin(-degreesToRadians(rotation)), 0,
        0, 1, 0, 0,
        sin(-
degreesToRadians(rotation)), 0, cosRounded(-degreesToRadians(rotation)), 0,

```

```

glm::mat4 zAxisRotation = glm::mat4(cosRounded(-degreesToRadians(rotation)), sin(-
degreesToRadians(rotation)), 0, 0,
                                -sin(-
degreesToRadians(rotation)), cosRounded(-degreesToRadians(rotation)), 0, 0,
                                0, 0, 1, 0,
                                0, 0, 0, 1);

glm::vec4 U = xAxisRotation * camPos;
glm::vec4 V = yAxisRotation * camPos;
glm::vec4 N = zAxisRotation * camPos;

/*std::cout << "U: " << glm::to_string(U) << endl;
std::cout << "V: " << glm::to_string(V) << endl;
std::cout << "N: " << glm::to_string(N) << endl;*/

glm::mat4 modelView = glm::mat4(U.x, V.x, N.x, 0,
                                U.y, V.y, N.y, 0,
                                U.z, V.z, N.z, 0,
                                0, 0, 0, 1) *
glm::mat4(1, 0, 0, 0,
            0, 1, 0, 0,
            0, 0, 1, 0,
            -camPos.x, -camPos.y, -
camPos.z, 1);

viewspaceVec = UpdatedVector(modelView, rotationVec);

std::cout << "Viewspace Converted Vector: " << "(" << viewspaceVec.x << ", " <<
viewspaceVec.y << ", " << viewspaceVec.z << ", " << viewspaceVec.w << ")" << endl;

//glm::mat4 perspective = glm::perspective(degreesToRadians(fov), (float)1 /
(float)1, nearPlane, farPlane);
//std::cout << "Perspective Matrix: " << glm::to_string(perspective) << endl;

float q = 1 / tan(degreesToRadians(fov)/2);
float A = RoundToNearest(q) / ((float)1 / (float)1);
float B = (1.5 + 80) / (1.5 - 80);
float C = (2*(1.5 * 80)) / (1.5 - 80);

/*std::cout << "A: " << RoundToNearest(A) << endl;
std::cout << "Q: " << RoundToNearest(q) << endl;
std::cout << "B: " << RoundToNearest(B) << endl;
std::cout << "C: " << RoundToNearest(C) << endl;*/

glm::mat4 perspective = glm::mat4(RoundToNearest(A), 0, 0, 0,
                                0, RoundToNearest(q), 0,
0,
                                0, 0, RoundToNearest(B),
-1,
                                0, 0, RoundToNearest(C),
0);

perspectiveVec = UpdatedVector(perspective, viewspaceVec);
perspectiveVec = glm::vec4(perspectiveVec.x / perspectiveVec.w, perspectiveVec.y /
perspectiveVec.w, perspectiveVec.z / perspectiveVec.w, perspectiveVec.w /
perspectiveVec.w);

std::cout << "Perspective Projection Vector: " << "(" << perspectiveVec.x << ", "
<< perspectiveVec.y << ", " << perspectiveVec.z << ", " << perspectiveVec.w << ")" << endl;
}

void Task2() {

```

```

glm::vec3 n = glm::vec3(-1, 2, 0);
glm::vec3 p0 = glm::vec3(2.5, 2, 4);

float distFromCullingPlane = dot(startingVector, n) + dot(p0, n);

/*std::cout << "Dot product of startingVector and n: " << dot(startingVector, n) <<
endl;
std::cout << "Dot product of p0 and n: " << dot(p0, n) << endl;*/

std::cout << "Distance from culling plane: " << distFromCullingPlane << endl;
if (distFromCullingPlane > 0) {
    std::cout << "Therefore, the startingVector is above the culling plane and
will not be culled\n" << endl;
}
else if (distFromCullingPlane < 0) {
    std::cout << "Therefore, the startingVector is below the culling plane and
will not be culled\n" << endl;
}
else {
    std::cout << "Therefore, the startingVector is on the culling plane and will
be culled\n" << endl;
}
}

void main() {
    std::cout << "-----|Task 1|-----"
    << endl;
    std::cout << "Task 1 where none of the numbers were rounded\n" << endl;
    Task1();

    std::cout << "\n\n";
    std::cout << "Task 1 where all of the numbers were rounded to two decimal places\n"
    << endl;
    Task1Rounded();

    std::cout << "\n\n";
    std::cout << "-----|Task 2|-----"
    << endl;
    Task2();
}

```

Application of Graphics Principles Assignment - By Mateusz

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

75/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6



Comment 1

you forgot the $T_z = -1$ in row 3 ,col 4.



Comment 2

-1



Comment 3

-3-1=-4

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

GRADING FORM: SMU GENERIC_GJ

MATEUSZ JADASZ

74.50

RESEARCH-INFORMED LITERATURE



You've included some references in a bibliography but please try to make use of these by references in line in the body of your report.

10

KNOWLEDGE AND UNDERSTANDING

16



A more detailed description of the object/world/camera space and projection transforms would help and include a few lines explaining the difference between clip space and the canonical view volume - z divide, change from right to left handed coordinates (by convention!), and so on.

Calculations broadly fine but error in translation propagates through and you need to include the z divide in your description. Hence some differences expected. Also not that the assignment asked for a "local" space interpretation - yours is world space as is set out in your calcs. Simply swap order of matrix mults. in model view to solve this.

ANALYSIS

22



Some good work but one or two errors as described above have crept in.

The plane eqn provided in the assignment you used. ok. Fine. no marks lost here but the + should have been a minus. No your problem - no penalty here. Correct result for eun used.

PRACTICAL APPLICATION AND DEP.

24



Code screenshots and test aschedule.

You composed the look at vector function yourself - very well done - could have used `glmlookAt` if you wanted.

Use more comments please.
Culling plane calcs correct for given equation.

SKILLS FOR PROFESSIONAL PRAC.



References used and report structured but see
comments at top regarding in line citations!

2.5