

Assignment 1

by Nathan Willis

Submission date: 30-Nov-2024 02:27PM (UTC+0000)

Submission ID: 244195037

File name: 41436_Nathan_Willis_Assignment_1_628112_691873614.docx (199.01K)

Word count: 2493

Character count: 15612



Graphics and Games Programming

Assignment 1

Nathan Willis (2103532)

Table of Contents

Task 1	2
Overview of graphics pipeline	2
Application Stage	2
Vertex Shader	2
Tessellation	2
Geometry Shader	3
Rasterization	3
Fragment Shader	3
Pixel Operations	3
Example	3
Task 2	7
Task 3	8
References	10
Appendix	11
testCamera.cpp	11

Task 1

Overview of graphics pipeline

Application Stage

The first step of the graphics pipeline is the application stage, which is entirely handled by the CPU. At this stage, the developer is responsible for implementing the logic and preparing data for rendering, relying on graphics APIs like OpenGL to send commands to the GPU [1][2]. This stage processes tasks such as handling user input, managing animations, resolving collisions, and preparing transformation matrices[3]. It acts as the "brain" behind the scenes, setting up the structures and logic that later stages will use. Another key process in this stage is vertex specification, where the developer sets up vertex data that defines the geometry of objects. This includes creating vertex buffers, configuring layouts, and specifying how primitives like triangles or lines are constructed [4]. Because this stage is fully CPU-driven, its efficiency depends on the developer's understanding of the pipeline. Poorly optimized logic can lead to significant bottlenecks in rendering performance, making careful design and testing essential but it does not rely entirely on the developer for optimisation, this stage is executed in parallel across multiple CPU cores in a superscalar construction[2].

Vertex Shader

The vertex shader processes individual vertices in parallel on the GPU[4]. It transforms vertices from model-space to clip-space, calculates attributes like texture coordinates and normals, and passes this data to later stages[5][6][3]. Vertex shaders are necessary for producing realistic scenes, as they determine the effect of light on each vertex based on its position, texture, and colour. Usually, the vertex lighting is handled by the fragment shader however it is still a valid approach for simpler or less performance-intensive scenes to handle lighting here. Vertex shaders are user-defined, meaning developers can write custom shader programs to specify their behaviour, and they can define outputs to pass specific data (e.g., colours, normals, or custom attributes) to subsequent stages. Bottlenecks occur when vertex shaders perform excessive computations, but modern GPUs mitigate this with parallelization, where thousands of vertices are processed simultaneously. Optimization strategies include simplifying vertex calculations, reducing vertex counts through Level of Detail (LOD), and precomputing complex transformations when possible.

Tessellation

Tessellation dynamically subdivides geometry to add detail based on the current scene. It involves two main stages: the Tessellation Control Shader (TCS), which determines how much tessellation to apply to a primitive and ensures proper connectivity between adjacent tessellated primitives, and the Tessellation Evaluation Shader (TES), which interpolates the newly created vertices or performs other operations for the tessellated primitives[7]. While powerful, tessellation can become a bottleneck if excessive detail is added. Optimizing tessellation involves controlling the level of detail based on the camera's view and distance from objects[8]. Modern GPUs handle tessellation efficiently with parallelism, processing each tessellated patch independently, which helps maintain high performance[3].

Geometry Shader

The geometry shader processes primitives (points, lines, or triangles) and can add or modify geometry. It enables effects like dynamic mesh generation, particle systems, or procedural geometry creation. While powerful, geometry shaders are computationally expensive. Bottlenecks occur when the shader generates too much geometry, resulting in an increased workload for subsequent stages like rasterization. Optimizing this stage involves limiting the output geometry or moving tasks like particle generation to other stages of the pipeline. GPUs parallelize this stage by processing each primitive in parallel, assigning different primitives to separate threads or shader cores, allowing efficient handling of geometry-based effects[9].

Rasterization

Rasterization is the process of converting primitives into fragments and determining their screen positions. This stage includes primitive assembly and interpolation of vertex attributes across the primitive. Rasterization can become a bottleneck when there is high overdraw, meaning that fragments from multiple primitives overlap in the same screen area. To optimize, techniques like depth pre-pass and culling (removing objects not in view) can reduce unnecessary fragment processing. Modern GPUs handle rasterization efficiently through parallelism, as thousands of fragments can be processed simultaneously across many cores, ensuring high throughput even with complex scenes[10][3].

Fragment Shader

The fragment shader computes the colour and other attributes for each fragment, such as textures, lighting, and shading. This is one of the most computationally expensive stages because each fragment can involve complex calculations like per-pixel lighting, multi-texturing, and anti-aliasing. Bottlenecks often arise when handling high-resolution textures, complex lighting models, or large numbers of fragments in high-resolution scenes. To optimize this stage, developers can use techniques like texture compression, simplifying shading models, or performing light culling to avoid unnecessary calculations for distant lights. Modern GPUs leverage massive parallelism in the fragment shader, processing each fragment independently in thousands of parallel threads, which helps mitigate performance issues[11][12].

Pixel Operations

The final stage, pixel operations, determines the output for each pixel in the framebuffer, including depth testing, blending, and writing to the framebuffer. This stage can become a bottleneck if there are excessive memory operations, particularly with complex blending modes or high-resolution output. Optimizing this stage often involves using hardware-accelerated depth testing and blending features, ensuring efficient framebuffer management, and minimizing unnecessary framebuffer reads/writes[6].

Example

1. The vertex given is (1,3,-2) the translation is (1.5,0,-1) and the rotation angle is 70°

The vertex is to be translated along the x axis by the translation vector which is (1.5,0,-1) to do this only the x component so the resultant vector is (2.5,3,-2). The resultant vector is then transformed further by rotating by 70° around the x axis of the local coordinate frame, because this is a rotation around a single coordinate, the simple x axis rotation matrix can be used[13].



$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Where θ is the angle of rotation in radians. 70° in radians is $70 \times \frac{\pi}{180}$ or $\frac{7\pi}{18}$.

So, the rotation matrix can be written as

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\frac{7\pi}{18} & -\sin\frac{7\pi}{18} \\ 0 & \sin\frac{7\pi}{18} & \cos\frac{7\pi}{18} \end{bmatrix}$$

To get the resultant vector this rotation matrix must be multiplied by the original vector (2.5,3,-2)

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\frac{7\pi}{18} & -\sin\frac{7\pi}{18} \\ 0 & \sin\frac{7\pi}{18} & \cos\frac{7\pi}{18} \end{bmatrix} \cdot \begin{bmatrix} 2.5 \\ 3 \\ -2 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\begin{aligned} 2.5 \cdot 1 + 3 \cdot 0 + -2 \cdot 0 &= 2.5 \\ 2.5 \cdot 0 + 3 \cdot \cos\frac{7\pi}{18} + -2 \cdot -\sin\frac{7\pi}{18} &= 2.91 \\ 2.5 \cdot 0 + 3 \cdot \sin\frac{7\pi}{18} + -2 \cdot \cos\frac{7\pi}{18} &= 2.14 \end{aligned}$$

The resultant vector is (2.5,2.91,2.14)

All these stages would be done at the application stage of the pipeline, the developer would be responsible for doing all the calculations and all the logic would be handled by the CPU. The vector is then handed to the next step, that being the conversion of the vectors coordinate from world coordinates into clip coordinates.

2.

The first step in converting a vector into view space is the construction of the coordinate system around the camera with the camera being at (0,0,0) and facing down the negative z axis. Then the calculation of the view matrix to get the required matrix translation for the vertex into the new coordinate system, the formula for this matrix is

$$\begin{bmatrix} right_x & up_x & forward_x & position_x \\ right_y & up_y & forward_y & position_y \\ right_z & up_z & forward_z & right_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = M$$

Where M is the view matrix, the right up and forward vectors define the orientation of the camera in space and the position vector is the adjustments to the camera to the origin[14].

The forward vector is found by subtracting the look at point from the viewpoint

$$\text{Forward} = (0,6,-1) - (1.5,0,0)$$

$$\text{Forward} = (-1.5,6,-1)$$

Then normalisation so

$$||\text{Forward}|| = \sqrt{(-1.5)^2 + (6)^2 + (-1)^2}$$

$$||\text{Forward}|| = \sqrt{40.25}$$

$$\text{Forward} = \frac{(-1.5,6,-1)}{\sqrt{40.25}}$$

$$\text{Forward} = (-0.24,0.96,-0.16)$$

From this the right vector is calculated from the cross product of the Up and Forward vectors

The Up vector was given as (0,1,0)

$$\text{Right} = (0,1,0) \times (-0.24,0.96,-0.16)$$

$$\text{Right} = (-0.16,0,0.24)$$

Then normalisation

$$\text{Right} = (-0.55,0,0.83)$$

Then the Up vector is calculated as the cross product of Right and Forward vectors

$$\text{Up} = \text{Forward} \times \text{Right}$$

$$\text{Up} = (0.8,0.29,0.53)$$

Then normalisation

Because the magnitude is so close to 1 the vector ends up the same to 2dp

$$\text{Up} = (0.80,0.29,0.53)$$

The next step is the calculation of the translations for the position vector

This is calculated by taking the negative of the dot product of the viewpoint vector (the cameras position) with the right up and forward vectors

The viewpoint was given as (0, 6,-1)

Position_x :

$$\text{Position}_x = -(\text{Right} \cdot \text{Viewpoint})$$

$$\text{Position}_x = -((-0.55,0,0.83) \cdot (0,6,-1))$$

$$\text{Position}_x = 0.83$$

Position_y :

$$Position_y = -(Up \cdot Viewpoint)$$

$$Position_y = -((0.80, 0.29, 0.53) \cdot (0, 6, -1))$$

$$Position_y = -1.21$$

$$Position_z:$$

$$Position_z = -(Forward \cdot Viewpoint)$$

$$Position_z = -((-0.24, 0.96, -0.16) \cdot (0, 6, -1))$$

$$Position_z = -5.92$$

So, the resultant matrix is

$$\begin{bmatrix} -0.55 & 0.8 & -0.24 & 0.83 \\ 0 & 0.29 & 0.96 & -1.21 \\ 0.83 & 0.53 & -0.16 & -5.92 \\ 0 & 0 & 0 & 1 \end{bmatrix} = M$$

The formula for calculating the position in views pace is

$$V^1 = M \cdot V$$

Where V is the original vertex position and V¹ is the remapped vertex position.

$$V^1 = \begin{bmatrix} -0.55 & 0.8 & -0.24 & 0.83 \\ 0 & 0.29 & 0.96 & -1.21 \\ 0.83 & 0.53 & -0.16 & -5.92 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2.5 \\ 2.91 \\ 2.14 \\ 1 \end{bmatrix}$$

$$V^1 = \begin{bmatrix} 1.27 \\ 1.67 \\ -2.64 \\ 1 \end{bmatrix}$$

This stage is handled by the vertex shader stage, at this point all vertexes are being mapped sequentially and in parallel. It will not go through the trouble of working out the view matrix for every vertex however as it is constant until the camera moves.

3.

At this point, the 3d map is converted into 2d space using perspective projection matrix as shown here

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{1}{\text{aspect} \cdot \tan \frac{fov}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan \frac{fov}{2}} & 0 & 0 \\ 0 & 0 & -\frac{far + near}{far - near} & -\frac{2far \cdot near}{far - near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

All the variables here have been given except aspect as such it will be assumed to be the standard 16:9.

Aspect ratio = 16/9

Fov = $34^\circ = 0.593412r$

Far = 100

Near = 1.5

$\tan \frac{fov}{2} = 0.3$

$$\begin{bmatrix} 1.88 & 0 & 0 & 0 \\ 0 & 3.32 & 0 & 0 \\ 0 & 0 & -1.03 & -3.09 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Then applying the matrix to the vertex

$$\begin{bmatrix} 1.88 & 0 & 0 & 0 \\ 0 & 3.32 & 0 & 0 \\ 0 & 0 & -1.03 & -3.09 \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1.27 \\ 1.67 \\ -2.64 \\ 1 \end{bmatrix}$$

This gives the vector (2.39,5.54,-0.37,2.64)

But there is another step, next is conversion to normalised device coordinates, this step is simple the x,y,z coordinates are divided by the w number

This gives the vertex (0.9,2.1,-0.14)

4.

This point is not visible on screen as its y coordinates is 2.1 too large and its z coordinates is below 0 meaning it is both above and behind the camera.

Task 2

Using the equation given and substituting the values provided gives

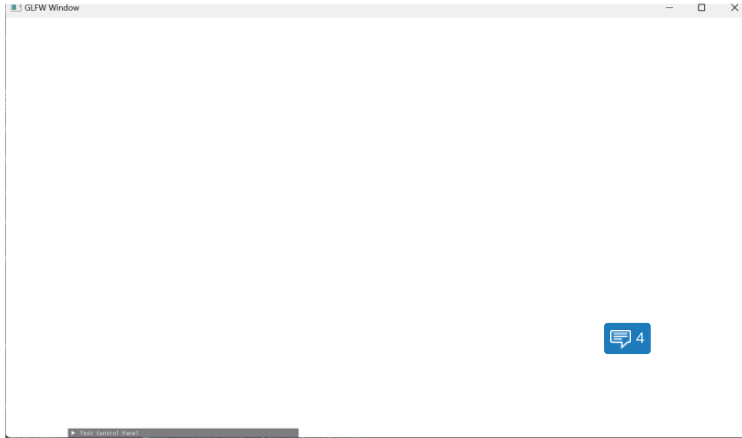
$$\begin{bmatrix} 1 \\ 3 \\ -2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} + \begin{bmatrix} 2 \\ 2.5 \\ 3 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} = 14$$

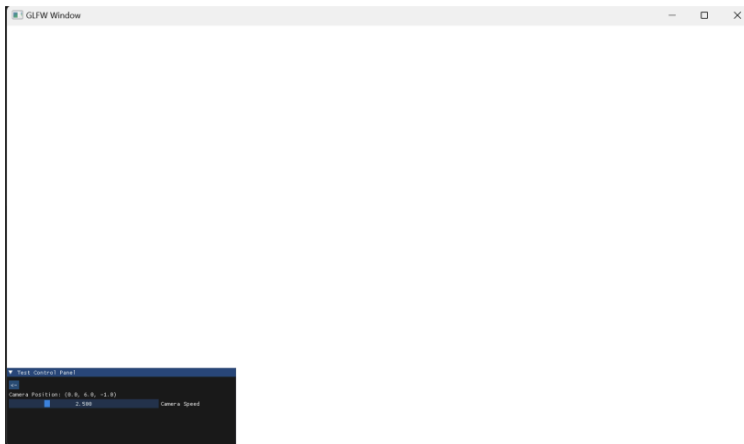
This lies 14 units on the correct side of the culling plane and as such will not be culled. This is predicated on the assumption that it lies on the correct side of all other planes and not

obstructed from the camera as this would result in it being culled either by being outside of the frustum or being subject to back face culling.

Task 3

With an aspect ratio of 16:9 the vertex was not visible to the camera post rotation





The vertex becomes visible if the camera is moved backwards showing that it is in fact behind the frame however it is shown that it is below the frame of the camera not above as I calculated earlier, it does however lie within the x axis of the frame which was according to the calculations. If I had to speculate I missed a negative sign somewhere and flipped the y axis.



Demonstration of the perspective

```
// Set the projection matrix
glm::mat4 projection = glm::perspective(glm::radians(37.0f), 1600.0f / 900.0f, 1.5f, 100.0f);
m_Shader->setUniformMat4f("projection", projection);
```

Camera angle and rotations

```

m_window(window),
cameraPosition(0.0f, 6.0f, -1.0f),
cameraFront(1.5f, 0.0f, 0.0f),
cameraUp(0.0f, 1.0f, 0.0f),
cameraSpeed(2.5f)

```

Translations being applied

```

//moves the model to the initial vertex
glm::mat4 initiate = glm::translate(glm::mat4(1.0f), glm::vec3(1.0f, 3.0f, -2.0f));
//translates the vertex 2 along the x axis
glm::mat4 translation = glm::translate(glm::mat4(1.0f), glm::vec3(1.5f, 0.0f, 0.0f));
//rotates around the x axis
glm::mat4 rotation = glm::rotate(glm::mat4(1.0f), glm::radians(70.0f), glm::vec3(1.0f, 0.0f, 0.0f));
// Set the model matrix
//glm::mat4 model = glm::rotate(glm::mat4(1.0f), (float)glfwGetTime(), glm::vec3(0.5f, 1.0f, 0.0f));
glm::mat4 model;

```

```

//initialisation of the vector
model = initiate;
//applying the translation and then rotation to the vector
model = rotation * translation * model;

```

References

References

- [1] "Optimize Your Application Performance(/exp:channel_entries)."
https://www.opengl.org/pipeline/article/vol003_8 (accessed Nov 29).
- [2] T. Akenine-Möller, E. Haines and N. Hoffman, *Real-Time Rendering*, CRC Press, 2019.
[Online]. Available: https://books.google.co.uk/books?id=Y-KEDwAAQBAJ&pg=PA11&source=gbs_toc_r&cad=2#v=onepage&q&f=false
- [3] "Rendering Pipeline Overview - OpenGL Wiki."
https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview (accessed Nov 29).
- [4] J. X. Chen and J. X. Chen, "Vertex Shading, Pixel Shading, and Parallel Processing," *Guide to Graphics Software Tools*, pp.1–43, 2009.
- [5] C. Maughan and M. Wloka, "Vertex shader introduction," *NVIDIA Technical Brief*, 2001.
- [6] V. S. Gordon and J. Clevenger, *Computer graphics programming in OpenGL with C*, Walter de Gruyter GmbH & Co KG, 2024.
- [7] J. Hladky, H. Seidel and M. Steinberger, "Tessellated shading streaming," *Computer Graphics Forum*, 2019.
- [8] K. Chung, et al., "Shader-based tessellation to save memory bandwidth in a mobile multimedia processor," *Comput. Graph.*, vol. 33, no. 5, pp.625–637, 2009.

- [9] A. Valdetaro, et al., "Understanding shader model 5.0 with directx11," *IX Brazilian symposium on computer games and digital entertainment*, 2010.
- [10] S. Laine and T. Karras, "High-performance software rasterization on GPUs," *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, 2011.
- [11] K. Halladay, *Practical Shader Development: Vertex and Fragment Shaders for Game Developers*, Apress, 2019.
- [12] E. Chan, et al., "Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware," *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2002.
- [13] J. Diebel, "Representing attitude: Euler angles, unit quaternions, and rotation vectors," *Matrix*, vol. 58, no. 15-16, pp.1–35, 2006.
- [14] "Understanding the View Matrix", *3D Game Engine Programming*, <https://www.3dgep.com/understanding-the-view-matrix/> (accessed Nov 30, 2024).

Appendix

testCamera.cpp

```
#include "testCamera.h"
#include "../Renderer.h"
#include "imgui.h"
#include <GL/glew.h>
#include "GLFW/glfw3.h"
#include "glm/gtc/matrix_transform.hpp"

namespace test {
    TestCamera::TestCamera(GLFWwindow* window)
    :
        m_window(window),
        cameraPosition(0.0f, 6.0f, -1.0f),
        cameraFront(1.5f, 0.0f, 0.0f),
        cameraUp(0.0f, 1.0f, 0.0f),
        cameraSpeed(2.5f)
    {
        // Vertex data for a simple cube
        const float vertices[] = {
            // Positions      // Colors
            -0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f, // Bottom-left
            0.5f, -0.5f, -0.5f,  0.0f, 1.0f, 0.0f, // Bottom-right
            0.5f, 0.5f, -0.5f,   0.0f, 0.0f, 1.0f, // Top-right
            -0.5f, 0.5f, -0.5f,   1.0f, 1.0f, 0.0f, // Top-left
            -0.5f, -0.5f, 0.5f,   0.0f, 1.0f, 1.0f, // Bottom-left (front)
            0.5f, -0.5f, 0.5f,   1.0f, 0.0f, 1.0f, // Bottom-right (front)
            0.5f, 0.5f, 0.5f,    1.0f, 1.0f, 1.0f, // Top-right (front)
            -0.5f, 0.5f, 0.5f,   0.5f, 0.5f, 0.5f // Top-left (front)
        };

        // Cube indices for drawing with Element Buffer
    }
```

```

        const unsigned int indices[] = {
            0, 1, 2, 2, 3, 0,
            4, 5, 6, 6, 7, 4,
            4, 5, 1, 1, 0, 4,
            7, 6, 2, 2, 3, 7,
            4, 7, 3, 3, 0, 4,
            1, 5, 6, 6, 2, 1
        };
        //filename to be changed
        m_Shader =
std::make_unique<Shader>(R"(src\shaders\projection.shader)");

        m_VAO = std::make_unique<VertexArray>();

        m_VBO = std::make_unique<VertexBuffer>(vertices, 48 * sizeof(float)); //
Vertex Buffer Object (VBO) to manage the vertices in GPU memory

        VertexBufferLayout layout;
        layout.Push<float>(3);
        layout.Push<float>(3);
        m_VAO->AddBuffer(*m_VBO, layout);

        m_IndexBuffer = std::make_unique<IndexBuffer>(indices, 36);

        // Enable depth testing
        glEnable(GL_DEPTH_TEST);

        // Unbind objects to prevent accidental modifications or conflicts
        m_VAO->Unbind(); // Unbind Vertex Array Object to prevent
accidental reuse
        m_Shader->Unbind(); // Unbind Shader Program to ensure clean state
        m_VBO->Unbind(); // Unbind Vertex Buffer
        m_IndexBuffer->Unbind(); // Unbind Index Buffer
        // Initialize vertex and index data (similar to your cube in
TestProjections)
        // Shader, VAO, VBO, IndexBuffer setup, depth test enabling
    }

    void TestCamera::Update(float deltaTime) {
        ProcessInput();
    }

    void TestCamera::Render() {
        Renderer render;
        render.ClearColourWhite();
        render.Clear();

        m_Shader->Bind();

        //moves the model to the initial vertex
        glm::mat4 initiate = glm::translate(glm::mat4(1.0f), glm::vec3(1.0f,
3.0f, -2.0f));
        //translates the vertex 2 along the x axis
        glm::mat4 translation = glm::translate(glm::mat4(1.0f), glm::vec3(1.5f,
0.0f, 0.0f));
        //rotates around the x axis
        glm::mat4 rotation = glm::rotate(glm::mat4(1.0f), glm::radians(70.0f),
glm::vec3(1.0f, 0.0f, 0.0f));
        // Set the model matrix

```

```

        //glm::mat4 model = glm::rotate(glm::mat4(1.0f), (float)glfwGetTime(),
glm::vec3(0.5f, 1.0f, 0.0f));
        glm::mat4 model;

        //initialisation of the vector
model = initiate;
//applying the translation and then rotation to the vector
model = rotation * translation * model;

        m_Shader->setUniformMat4f("model", model);

        // Set the view matrix based on camera position and orientation
        glm::mat4 view = glm::lookAt(cameraPosition, cameraPosition +
cameraFront, cameraUp);
        m_Shader->setUniformMat4f("view", view);

        // Set the projection matrix
        glm::mat4 projection = glm::perspective(glm::radians(37.0f), 1600.0f /
900.0f, 1.5f, 100.0f);
        m_Shader->setUniformMat4f("projection", projection);

        m_VAO->Bind();
        renderer.Draw(*m_VAO, *m_IndexBuffer, *m_Shader);
    }

    void TestCamera::ProcessInput() {
        // Assuming we have access to the GLFW window object
        GLFWwindow* window = glfwGetCurrentContext(); // This should be
correctly initialized in your application.

        if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS) {
            cameraPosition += cameraSpeed * cameraFront;
        }
        if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS) {
            cameraPosition -= cameraSpeed * cameraFront;
        }
        if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS) {
            cameraPosition -= glm::normalize(glm::cross(cameraFront, cameraUp))
* cameraSpeed;
        }
        if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS) {
            cameraPosition += glm::normalize(glm::cross(cameraFront, cameraUp))
* cameraSpeed;
        }
    }

    void TestCamera::RenderGUI() {
        ImGui::Text("Camera Position: (%.1f, %.1f, %.1f)", cameraPosition.x,
cameraPosition.y, cameraPosition.z);
        ImGui::SliderFloat("Camera Speed", &cameraSpeed, 0.1f, 10.0f);
    }
}

```

Assignment 1

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

66/100

Engagement with Literature: 13/15

Knowledge and Understanding: 14/20

Cognitive and Intellectual Skills: 20/30

Practical Skills: 15/30

Transferable Skills: 4/5

Overall 66/100

Comment:

Overall a great attempt, the section for task 3 was a bit lackluster and would have benefitted from more of an explanation and discussion on what was happening and how it related back to your initial discussion. Do your results match etc?

In your example there were a few minor mistakes that ultimately affected your results :

- Ignoring part of the translation vector in the very first transformation.
-
-
- Mismatch in the forward vector's length (the arithmetic under the square root).
-
- Inconsistent usage of the forward vector between steps (using -0.16
-
- $vs -1.6$).
- *(the arithmetic under the square root).*

You effectively referenced a range of technical sources, including documentation, academic papers, and software engineering textbooks.

When demonstrating equations or screenshots it is best practice to label them as a numbered figure

and to provide a table of figures under the table of contents. Labelling an equation as a figure is typically done to maintain consistency with the formatting style used in the document. In some academic styles, such as in certain fields of science and engineering, equations are treated as visual elements or figures, especially when they are central to the content of the paper. By labelling an equation as a figure, it allows for clear referencing throughout the text and ensures that it is easily distinguishable from the main narrative. This approach helps readers locate and focus on key mathematical expressions more effectively.

See inline comments also

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5



Comment 1

If the translation vector is

$(1.5, 0, -1)$ adding that to

$(1, 3, -2)$ should yield

$(2.5, 3, -3)$, **not** $(2.5, 3, -2)$

PAGE 6



Comment 2

You wrote $\sqrt{40.25}$ but arithmetic suggests $\sqrt{2.25 + 36 + 1} = \sqrt{39.25}$

That difference may look minor, but it skews your normalised vector.



Comment 3

$\text{Right} = (0, 1, 0) \times (-0.24, 0.96, -1.6).$

But previously you said the forward vector's z-component was -0.16 not -1.6 drastically changing the cross product

PAGE 7

PAGE 8

PAGE 9



Comment 4

Labelling an equation as a figure is typically done to maintain consistency with the formatting style used in the document. In some academic styles, such as in certain fields of science and engineering, equations are treated as visual elements or figures, especially when they are central to the content of the paper. By labelling an equation as a figure, it allows for clear referencing throughout the text and ensures that it is easily distinguishable from the main narrative. This approach helps readers locate and focus on key mathematical expressions more effectively.

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14
