UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

DIVISIÓN DE INGENIERÍA ELÉCTRICA

INGENIERÍA EN COMPUTACIÓN

LABORATORIO DE COMPUTACIÓN GRÁFICA e INTERACCIÓN HUMANO COMPUTADORA

REPORTE DE PRÁCTICA Nº 02

NOMBRE COMPLETO: Diego Aldair García Hernández

Nº de Cuenta: 31902404-5

GRUPO DE LABORATORIO: 02

GRUPO DE TEORÍA: 04

SEMESTRE 2025-2

FECHA DE ENTREGA LÍMITE: 22 de febrero de 2025

CALIFICACIÓN: _____

Practica 2: Proyecciones y puertos de vista. Transformaciones Geométricas

Introducción

En esta práctica se desarrollarán modelos gráficos en OpenGL, utilizando mallas, índices y shaders para representar objetos en 3D de manera eficiente. Se trabajará con proyecciones y transformaciones geométricas para construir figuras más complejas y optimizar el renderizado. La práctica se divide en dos ejercicios: el primero consiste en dibujar las iniciales de mi nombre "D", "A", "G" y "H", cada una con un color diferente, utilizando triángulos como base para su construcción. El segundo ejercicio implica la creación de una casa compuesta por cubos y pirámides, donde se implementarán shaders de colores personalizados en lugar de los valores predeterminados. A lo largo del desarrollo, se explorará el uso de shaders, la manipulación de matrices de transformación y la administración de mallas para lograr representaciones gráficas precisas.

Ejecución de los ejercicios

Ejercicio 1: Creación de letras

En este ejercicio se implementó la generación de letras utilizando mallas de colores (MeshColor). Cada letra se representa mediante una serie de triángulos que definen su forma en coordenadas 3D.

Primero, se declaran los vértices con sus respectivas coordenadas en el espacio 3D, incluyendo los valores RGB. Luego, se crea una instancia de MeshColor y se asignan los vértices para generar la forma. Finalmente, la malla se almacena en una lista para su posterior renderizado.

Por ejemplo, la figura que crea una parte de la letra D:

```
//////////////// LETRA D /////////////////
// figura 1_D
GLfloat vertices_figuraUnoD[] = {
    //X         Y           Z           R       G       B
    -0.8f,  0.9f,       0.5f,           0.0f,   0.0f,   1.0f,
    -0.6f,  0.9f,       0.5f,           0.0f,   0.0f,   1.0f,
    -0.8f,  0.1f,       0.5f,           0.0f,   0.0f,   1.0f,
    -0.6f,  0.9f,       0.5f,           0.0f,   0.0f,   1.0f,
    -0.8f,  0.1f,       0.5f,           0.0f,   0.0f,   1.0f,
    -0.6f,  0.1f,       0.5f,           0.0f,   0.0f,   1.0f,

};
MeshColor* figuraUnoD = new MeshColor();
figuraUnoD->CreateMeshColor(vertices_figuraUnoD, 36);
meshColorList.push_back(figuraUnoD);
```

Cada letra se generó utilizando el mismo patrón, cambiando las coordenadas y los colores.

Salida:

La salida del Ejercicio 1 muestra las iniciales D, A, G y H dibujadas en una ventana con fondo blanco. Cada letra se ha construido utilizando triángulos y colores diferentes para resaltar cada una de ellas.

Cada letra se generó mediante una malla de vértices con coordenadas en 3D, definiendo su forma con triángulos. Los colores fueron aplicados usando un shader específico que permite asignar valores RGB a los vértices. Además, cada letra fue escalada y posicionada adecuadamente en la pantalla usando transformaciones geométricas con GLM, lo que permitió alinearlas correctamente en una cuadrícula de 2x2.

Ejercicio 2: Creación de casa con figuras 3D (Cubo y Pirámide)

En este segundo ejercicio, se implementaron dos funciones principales para la creación de figuras tridimensionales: la pirámide y el cubo.

Pirámide

Para definir la pirámide, se utilizaron cuatro vértices y una base rectangular los cuales se representan en el siguiente bloque:

```
//Pirámide triangular regular
void CreaPiramide()
{
    unsigned int indices[] = {
        0,1,2,
        1,3,2,
        3,0,2,
        1,0,3

    };
    GLfloat vertices[] = {
        -0.5f, -0.5f,0.0f,  //0
        0.5f,-0.5f,0.0f,    //1
        0.0f,0.5f, -0.25f,  //2
        0.0f,-0.5f,-0.5f,   //3

    };
    Mesh* obj1 = new Mesh();
    obj1->CreateMesh(vertices, indices, 12, 12);
    meshList.push_back(obj1);
}
```

Cubo

Para definir el cubo, se utilizó una malla con índices que conectan los vértices para formar triángulos del cubo. Cada cara del cubo está formada por dos triángulos (6 índices por cara). Se especifican los índices para las seis caras: frontal, derecha, trasera, izquierda, inferior y superior.
Posteriormente, se crean los ocho vértices del cubo y se definen los puntos para la cara frontal y trasera. Además, se crear un objeto de tipo Mesh que representa la geometría del cubo. Finalmente, el cubo se almacena en la lista meshList para su posterior renderizado.

```
unsigned int cubo_indices[] = {
    // front
    0, 1, 2,
    2, 3, 0,
    // right
    1, 5, 6,
    6, 2, 1,
    // back
    7, 6, 5,
    5, 4, 7,
    // left
    4, 0, 3,
    3, 7, 4,
    // bottom
    4, 5, 1,
    1, 0, 4,
    // top
    3, 2, 6,
    6, 7, 3
};

GLfloat cubo_vertices[] = {
    // front
    -0.5f, -0.5f,  0.5f,
    0.5f, -0.5f,  0.5f,
    0.5f,  0.5f,  0.5f,
    -0.5f,  0.5f,  0.5f,
    // back
    -0.5f, -0.5f, -0.5f,
    0.5f, -0.5f, -0.5f,
    0.5f,  0.5f, -0.5f,
    -0.5f,  0.5f, -0.5f
};
Mesh* cubo = new Mesh();
cubo->CreateMesh(cubo_vertices, cubo_indices, 24, 36);
meshList.push_back(cubo);
}
```

Creación de Shaders

La función CreateShaders() se encarga de cargar y compilar los diferentes shaders que se utilizarán en la representación de los objetos en OpenGL. En esta práctica, se implementaron varios shaders para permitir el uso de diferentes colores en los objetos especialmente en el segundo ejercicio donde se construyó una casa con cubos y pirámides de distintos colores.

Para permitir que la casa tenga colores específicos en cada una de sus partes, se implementaron shaders individuales para cada color. Se crearon los siguientes shaders: shaderRojo para la estructura principal de la casa, shaderVerde para ventanas y puerta, shaderAzul para representar el techo, shaderVerdeF para los pinos y shaderCafe para el tronco de los mismos. Cada uno de estos shaders utiliza un Vertex Shader especializado y el mismo Fragment Shader (fShader), que se encarga de definir el color final del objeto en pantalla.

Finalmente, todos los shaders creados se almacenan en la lista shaderList. Esto permite que puedan ser utilizados posteriormente en el programa, aplicándolos a los diferentes objetos según sea necesario.

```
void CreateShaders()
{

    Shader *shader1 = new Shader(); //shader para usar índices: objetos: cubo y  pirámide
    shader1->CreateFromFiles(vShader, fShader);
    shaderList.push_back(*shader1);

    Shader* shader = new Shader();//shader para usar color como parte del VAO: letras
    shader->CreateFromFiles(vShaderColor, fShaderColor);
    shaderList.push_back(*shader);

    Shader* shaderRojo = new Shader();
    shaderRojo->CreateFromFiles(vRojoShader, fShader);
    shaderList.push_back(*shaderRojo);

    Shader* shaderVerde = new Shader();
    shaderVerde->CreateFromFiles(vVerdeShader, fShader);
    shaderList.push_back(*shaderVerde);

    Shader* shaderAzul = new Shader();
    shaderAzul->CreateFromFiles(vAzulShader, fShader);
    shaderList.push_back(*shaderAzul);

    Shader* shaderVerdeF = new Shader();
    shaderVerdeF->CreateFromFiles(vVerdeFShader, fShader);
    shaderList.push_back(*shaderVerdeF);

    Shader* shaderCafe = new Shader();
    shaderCafe->CreateFromFiles(vCafeShader, fShader);
    shaderList.push_back(*shaderCafe);
}
```

Main

Dentro de esta función, se ejecutan los dos ejercicios principales: el dibujo de las iniciales y la construcción de la casa.

Primero, se crea una ventana con mainWindow = Window(800, 600);, y se inicializa con mainWindow.Initialise();. Luego, se llaman las funciones CrearLetrasyFiguras(), CreaPiramide() y CrearCubo(), que generan los datos de las letras y las figuras en la escena. También se ejecuta CreateShaders() para cargar los diferentes shaders que se utilizarán en la renderización, incluso los que cree de los diferentes colores.

Para definir la vista de la escena, se configura una matriz de proyección ortográfica que permite visualizar los objetos en 2D sin perspectiva.

A continuación, se entra en el bucle principal donde se manejan los eventos de la ventana y se limpia el buffer de color y profundidad.

Posteriormente, se muestra el dibujado de la casa y letras. Para el primero, se puede ver cómo se aplican transformaciones geométricas a cubos y pirámides para formar la estructura. Se utilizan shaders de colores para asignar colores específicos a cada parte de la casa. Cada objeto es escalado y trasladado y luego se aplica para enviar las matrices de transformación a los shaders antes de renderizar los objetos.

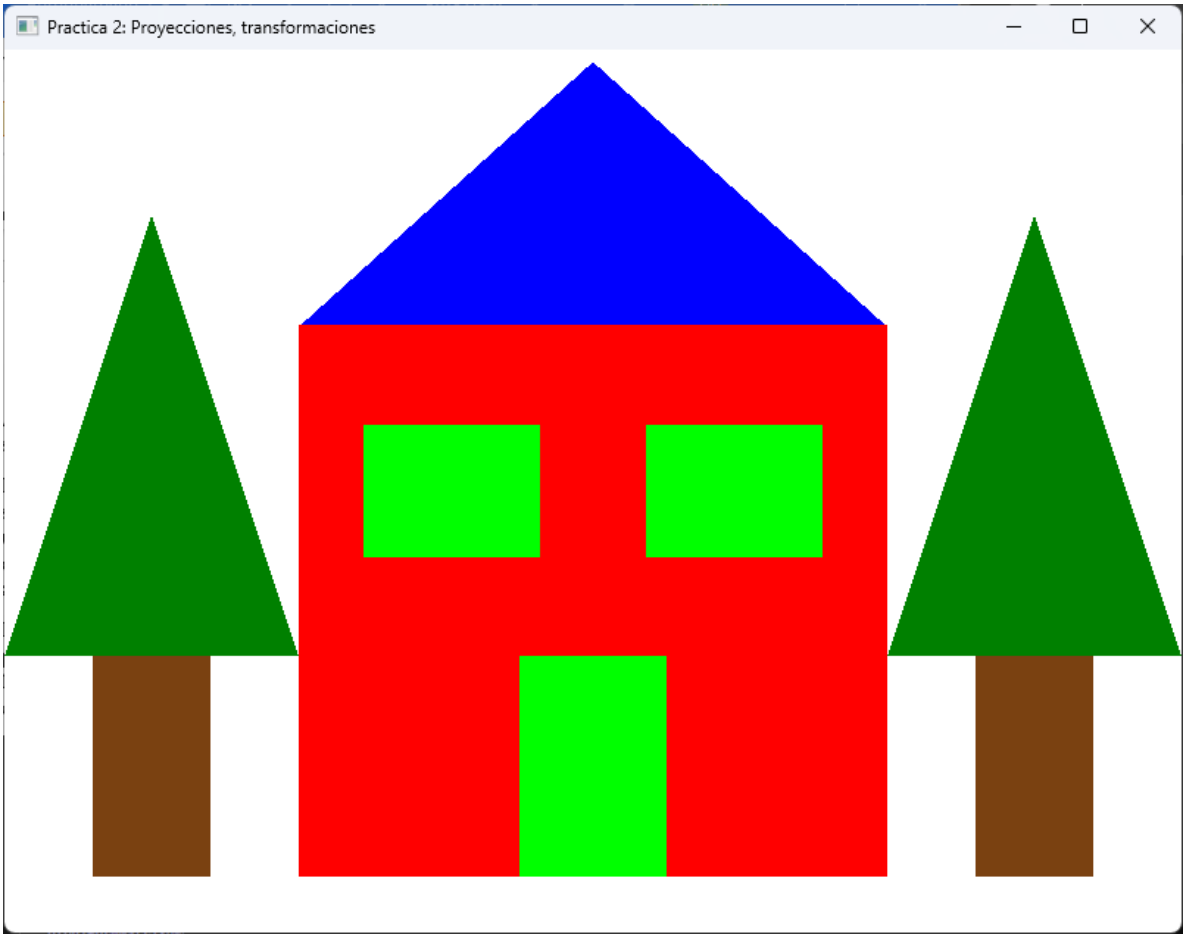
Después, se renderizan las letras de mi nombre (DAGH) utilizando shaderList[1], que permite usar colores en los vértices. Para cada letra, se asigna una transformación y se envían las matrices de transformación a la GPU con glUniformMatrix4fv(). Luego las letras se dibujan con meshColorList[i]->RenderMeshColor();.

```
/////////////// DIBUJO EJERCICIO 1 /////////////////

//Para las letras hay que usar el segundo set de shaders con indice 1 en ShaderList
shaderList[1].useShader();
uniformModel = shaderList[0].getModelLocation();
uniformProjection = shaderList[0].getProjectLocation();

/////////////// LETRA D /////////////////
// Figura1 D
model = glm::mat4(1.0);
glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));
glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
meshColorList[0]->RenderMeshColor();

// Figura1_1 D
model = glm::mat4(1.0);
model = glm::scale(model, glm::vec3(1.0f, 0.5f, 1.0f));
model = glm::translate(model, glm::vec3(0.5f, 0.5f, 0.0f));
glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));
glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
meshColorList[0]->RenderMeshColor();

// Figura2 D
model = glm::mat4(1.0);
glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));
glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
meshColorList[1]->RenderMeshColor();
```

Finalmente, el programa intercambia los buffers con mainWindow.swapBuffers();, asegurando que los cambios se reflejen en la pantalla y sigue ejecutando el bucle hasta que la ventana se cierre como se ha hecho en las últimas prácticas.

Salida.

La salida de las letras en la ventana se representa tal cual se menciona en el diseño. Las letras fueron representadas de color blanco, mientras que el fondo de la ventana cambia de color cada 2 segundos.



La salida del Ejercicio 2 muestra la representación de una casa utilizando cubos y pirámides en lugar de formas básicas como triángulos y cuadrados.

Cada elemento de la casa fue construido utilizando mallas de vértices y aplicando transformaciones geométricas para ajustar su posición y tamaño. Se usaron distintos shaders para asignar colores específicos a cada parte de la estructura.

Problemas

Tuve varios problemas con los shaders de colores al hacer el Ejercicio 2. Primero, los archivos se guardaron con extensión .txt en lugar de .frag y .vert, lo que hizo que el programa no los reconociera. Lo solucioné cambiando la extensión correcta.

Luego, aparecían errores al compilar porque los fragment shaders no tenían la función main(), pero lo raro es que si lo tenían. Para arreglarlo, volví a crear los archivos.

También hubo problemas al enlazar los shaders al programa, lo que impedía su carga. La solución fue asegurarme de que cada shader tuviera tanto su vertex shader como su fragment shader y organizarlos correctamente en la función CreateShaders(). Con esto los colores se aplicaron bien a los objetos.

Conclusión

En esta práctica se logró implementar el uso de mallas, índices y shaders en OpenGL para la construcción de figuras y objetos en 3D, aplicando transformaciones geométricas y proyecciones. Se representaron correctamente las iniciales utilizando triángulos de colores y se construyó una casa con cubos y pirámides, asignando shaders específicos para cada color. A lo largo del proceso, se enfrentaron desafíos con la configuración de los shaders, los cuales fueron solucionados asegurando la correcta lectura y compilación de los archivos. Esta práctica permitió fortalecer el conocimiento sobre el manejo de shaders, buffers y matrices de transformación, sentando una base sólida para proyectos más avanzados en gráficos computacionales.

Bibliografía

- OpenGL-Tutorial. (s.f.). Tutorial 3: Matrices en OpenGL. OpenGL Tutorial. Recuperado el [fecha de acceso], de https://www.opengl-tutorial.org/es/beginners-tutorials/tutorial-3-matrices/

- Deckerix. (s.f.). Transformaciones geométricas en OpenGL. Deckerix Blog. Recuperado el [fecha de acceso], de http://deckerix.com/blog/transformaciones-geometricas-en-opengl/

- Usuario de Stack Overflow. (2013, 3 de julio). What are shaders in OpenGL and what do we need them for? Stack Overflow. Recuperado el [fecha de acceso], de https://stackoverflow.com/questions/17789575/what-are-shaders-in-opengl-and-what-do-we-need-them-for