Reporte de Práctica de Laboratorio de Computación Gráfica e Interacción Humano-Computadora

Información de la práctica

Nombre del alumno:	Martínez Pérez Brian Erik		
Número de Cuenta:	319049792	**Práctica número:**	3
Título de la práctica:	Modelado Geométrico		
Fecha de entrega:	8 de septiembre de 2025		

Introducción: (descripción inicial sobre la práctica)

La práctica de "Modelado Geométrico" combina los principios fundamentales de las matemáticas y la geometría computacional con la implementación práctica a través de la librería OpenGL. El objetivo es crear y manipular objetos y entornos 3D. se utilizará un código que ilustra estos conceptos al construir una escena 3D simple que incluye una estructura similar a una mesa.

La base de esta práctica utiliza los conceptos de la geometría euclidiana, como puntos, líneas, planos y sólidos, los cuales se representan y manipulan mediante vectores y matrices. El código utiliza la librería GLM (OpenGL Mathematics) para llevar a cabo estas operaciones matemáticas, que son esenciales para aplicar transformaciones afines como la traslación, rotación y escalamiento a los objetos en el espacio 3D.

La representación visual de la escena es manejada por OpenGL, que utiliza primitivas gráficas como los triángulos para renderizar los objetos. El código gestiona eficientemente los datos geométricos utilizando Vertex Buffer Objects (VBOs) y Vertex Array Objects (VAOs), que almacenan la información de los vértices de las figuras.

La apariencia de estos objetos es controlada por los shaders, en particular el Vertex Shader y el Fragment Shader, que definen cómo se transforman los vértices y cómo se colorean los píxeles.

Resumen Teórico: (introducción sobre los fundamentos teóricos en los que se basa la práctica y cuáles son los objetivos teóricos que persigue.)

Esta práctica de "Modelado Geométrico" se fundamenta en tres pilares teóricos principales:

Matemáticas y Geometría Computacional, la programación con OpenGL, y la interacción con la cámara y el usuario. El objetivo es integrar estos conceptos para construir y manipular escenas 3D interactivas.

El primer pilar se centra en la *geometría euclidiana* y el *álgebra lineal*, que son esenciales para representar objetos en 2D y 3D. Esto incluye el uso de vectores para representar posiciones y direcciones, y la aplicación de operaciones como la suma, resta, producto punto y producto cruz. Las matrices 4×4 son fundamentales para aplicar transformaciones afines como la traslación, rotación y escalamiento a los objetos, permitiendo su posicionamiento y manipulación en el espacio virtual.

Se utilizan primitivas gráficas como puntos, líneas y triángulos. Los datos de los vértices de estos objetos se gestionan de manera eficiente a través de *Vertex Buffer Objects (VBOs)* y *Vertex Array Objects (VAOs)*, que optimizan la transferencia de datos entre la CPU y la GPU. Además, se emplean *shaders*, programas que se ejecutan en la GPU. El *Vertex Shader* procesa la información de cada vértice para su transformación y posicionamiento, mientras que el *Fragment Shader* determina el color de los píxeles para la visualización final del objeto.

El tercer pilar teórico aborda la visualización y el control de la cámara. La práctica busca implementar una perspectiva de visualización 3D mediante transformaciones de vista y proyección. El código utiliza funciones como glm::lookAt y glm::perspective para simular una cámara virtual que el usuario puede controlar. Esto permite al usuario interactuar con la escena a través del teclado, moviendo y rotando la cámara para explorar el entorno 3D, lo que es un componente clave para crear experiencias interactivas.

En conjunto, estos fundamentos teóricos buscan lograr el objetivo de modelar geométricamente una escena 3D, no solo definiendo la forma de los objetos, sino también cómo se manipulan y se visualizan desde diferentes puntos de vista, ofreciendo una comprensión completa del pipeline de gráficos.

Descripción de la práctica: (En esta sección se deben describir todos los pasos ejecutados durante la sesión práctica realizada en el laboratorio.)

Explicación de las partes principales del Código

Las variables globales de movimiento, movX, movY, y movZ controlan la posición de la cámara. Rot, controla la rotación de la cámara alrededor del eje Y.

```
const GLint WIDTH = 800, HEIGHT = 600;
float movX=0.0f;
float movY=0.0f;
float movZ=-5.0f;
float rot = 0.0f;
```

Creación de los vértices del cubo, 36 vértices (6 caras × 6 vértices por cara). Cada vértice tiene 3 componentes de posición (x, y, z) y 3 componentes de color (r, g, b).

```
// use with Perspective Projection
float vertices[] = {
    -0.5f, -0.5f, 0.5f, 1.0f, 0.0f,0.0f,//Front
    0.5f, -0.5f, 0.5f,  1.0f, 0.0f,0.0f,
    0.5f,  0.5f, 0.5f,  1.0f, 0.0f,0.0f,
    0.5f,  0.5f, 0.5f,  1.0f, 0.0f,0.0f,
    -0.5f,  0.5f, 0.5f, 1.0f, 0.0f,0.0f,
    -0.5f, -0.5f, 0.5f, 1.0f, 0.0f,0.0f,
```

VBO (Vertex Buffer Object): Almacena los datos de los vértices en la GPU.

```
//2.- Copiamos nuestros arreglo de vertices en un buffer de vertices para que OpenGL lo use
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3.Copiamos nuestro arreglo de indices en  un elemento del buffer para que OpenGL lo use
/*glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);*/
```

Función Inputs(): Entradas de teclado, permite controlar la cámara:

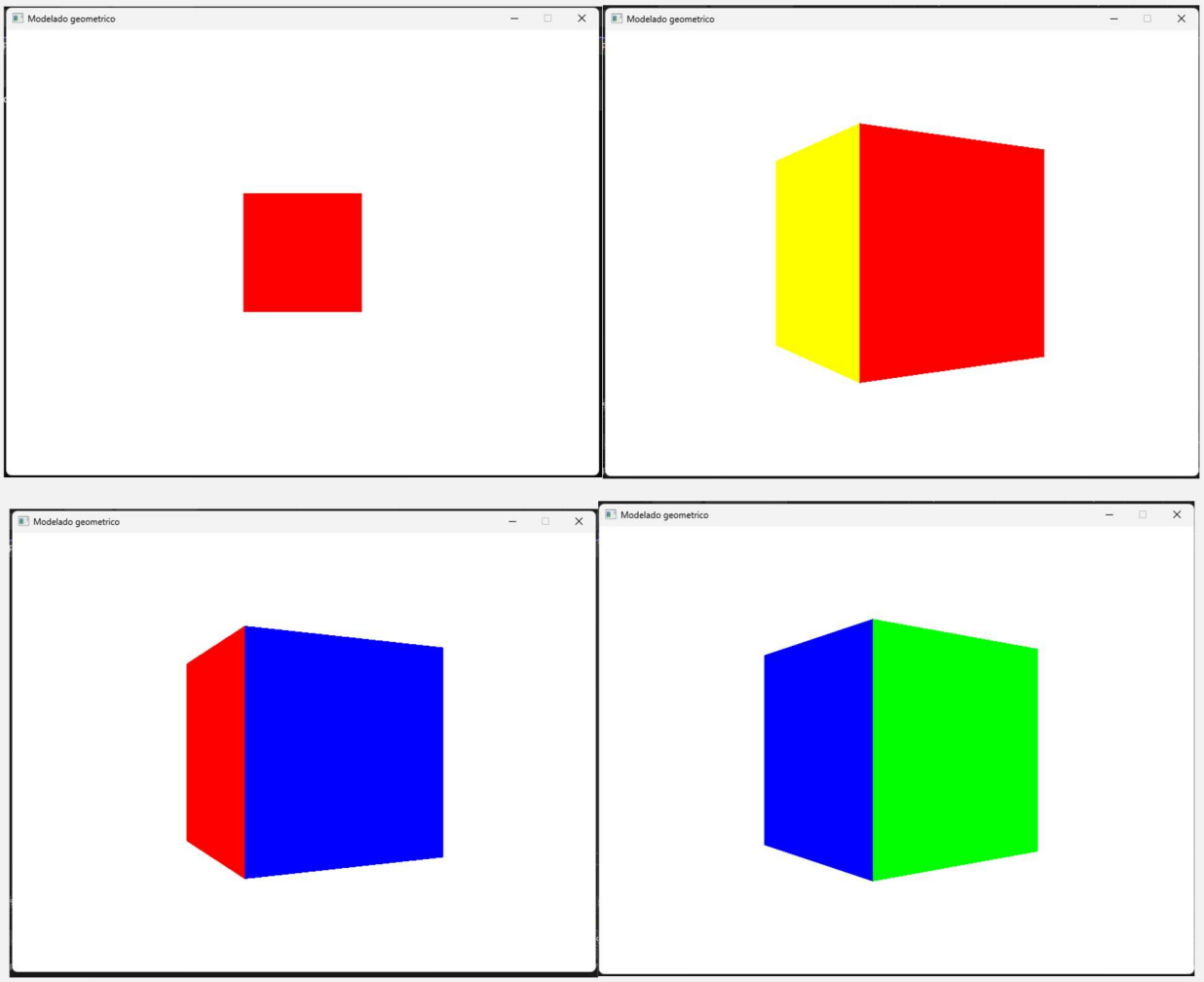
- W/S: Acercar/alejar la cámara (eje Z).
- A/D: Mover izquierda/derecha (eje X).
- PgUp/PgDown: Mover arriba/abajo (eje Y).
- Flechas izquierda/derecha: Rotar la cámara alrededor del eje Y.
- ESC: Cerrar la ventana.

```
void Inputs(GLFWwindow *window) {
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)  //GLFW_RELEASE
        glfwSetWindowShouldClose(window, true);
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        movX += 0.08f;
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        movX -= 0.08f;
    if (glfwGetKey(window, GLFW_KEY_PAGE_UP) == GLFW_PRESS)
        movY += 0.08f;
    if (glfwGetKey(window, GLFW_KEY_PAGE_DOWN) == GLFW_PRESS)
        movY -= 0.08f;
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        movZ -= 0.08f;
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        movZ += 0.08f;
    if (glfwGetKey(window, GLFW_KEY_RIGHT) == GLFW_PRESS)
        rot += 0.4f;
    if (glfwGetKey(window, GLFW_KEY_LEFT) == GLFW_PRESS)
        rot -= 0.4f;
}
```

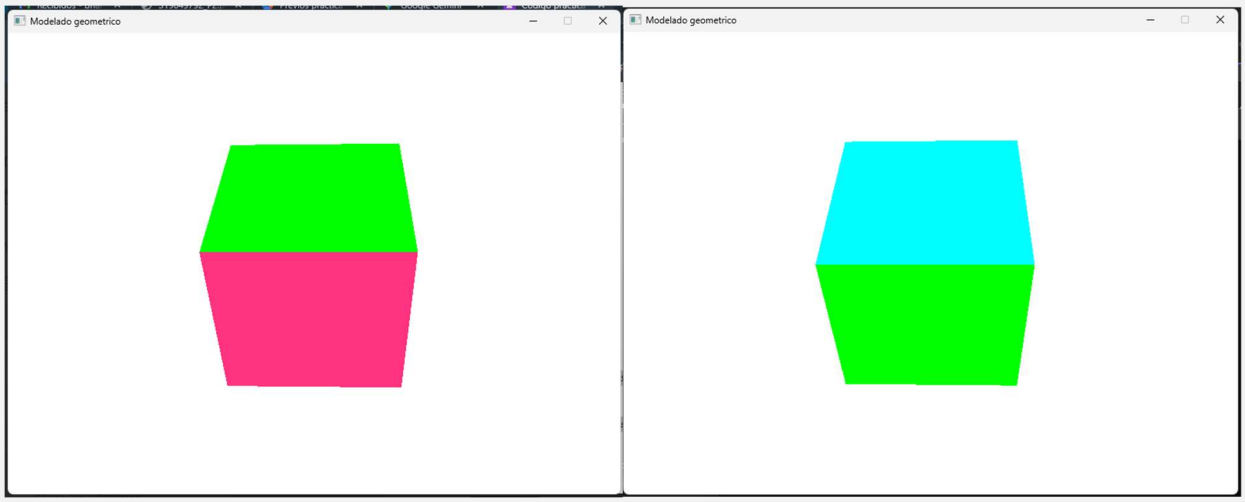
“glm::translate”, mueve la cámara según movX, movY, movZ. “glm::rotate”, rota la cámara alrededor del eje Y.

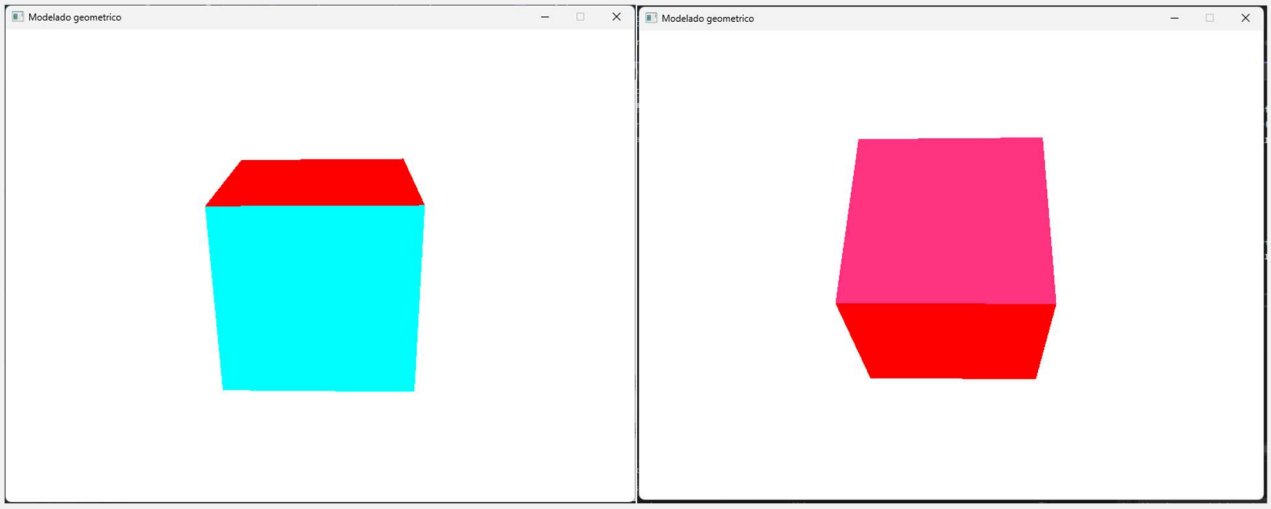
```
view = glm::translate(view, glm::vec3(movX,movY, movZ));
view = glm::rotate(view, glm::radians(rot), glm::vec3(1.0f, 1.0f, 1.0f));
```

La ejecución del programa muestra las caras laterales del cubo, y podemos mover la cámara a través del eje X, para visualizar las distintas caras del cubo.

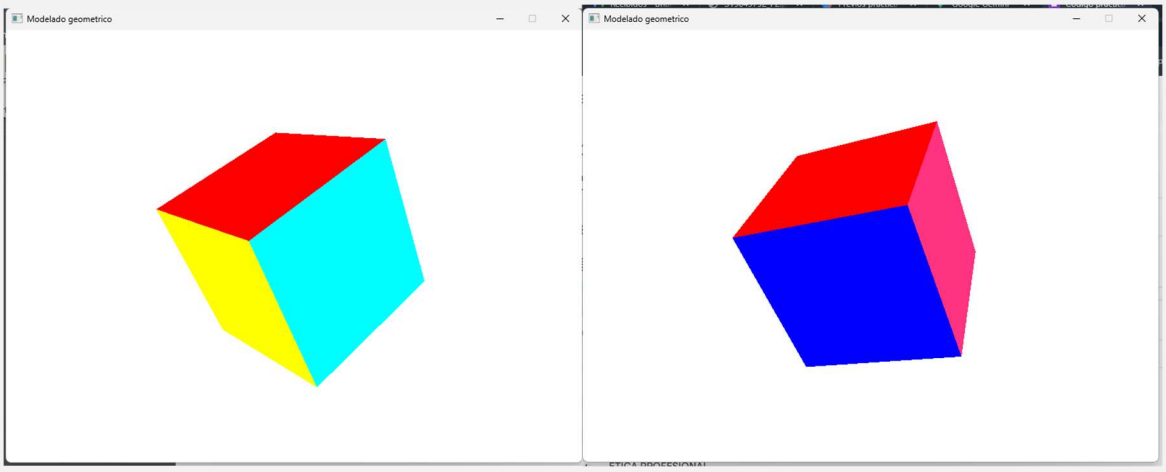


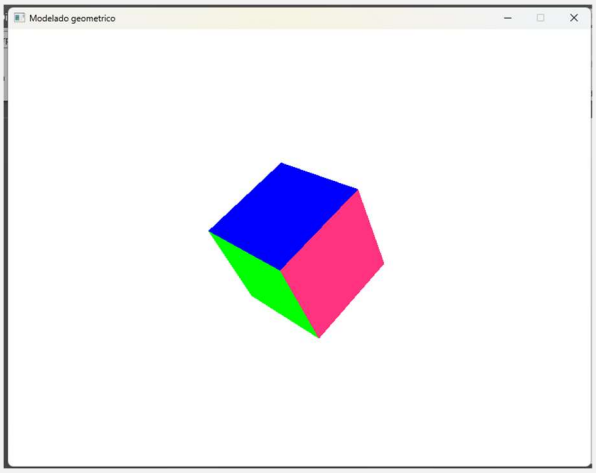
Posteriormente implementamos la rotación de la cámara a través del eje Y, para poder visualizar la cara superior e inferior del cubo.





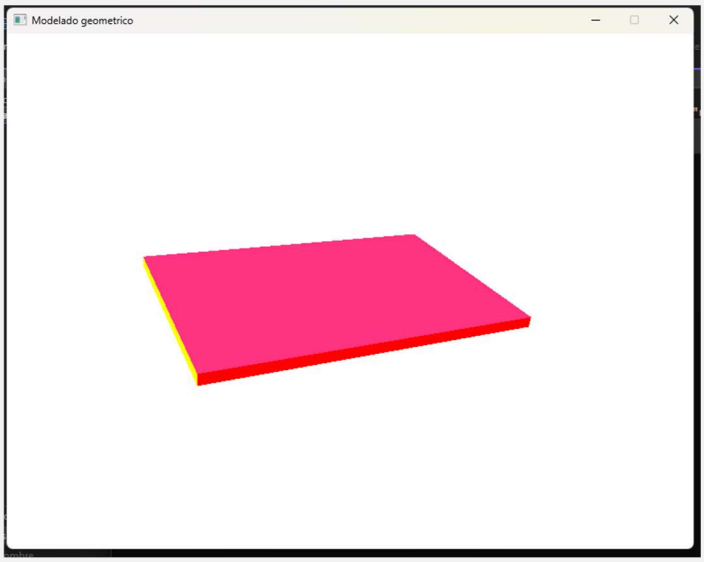
Además, podemos rotar la cámara a través del eje Y, también podemos girar la cámara en todos los ejes al mismo tiempo.





Una vez que teníamos la proyección en perspectiva y que pudimos aplicar las operaciones de transformación. La siguiente actividad era modelar una mesa completa, desde sus 4 patas, su tablero y un objeto encima de la mesa.

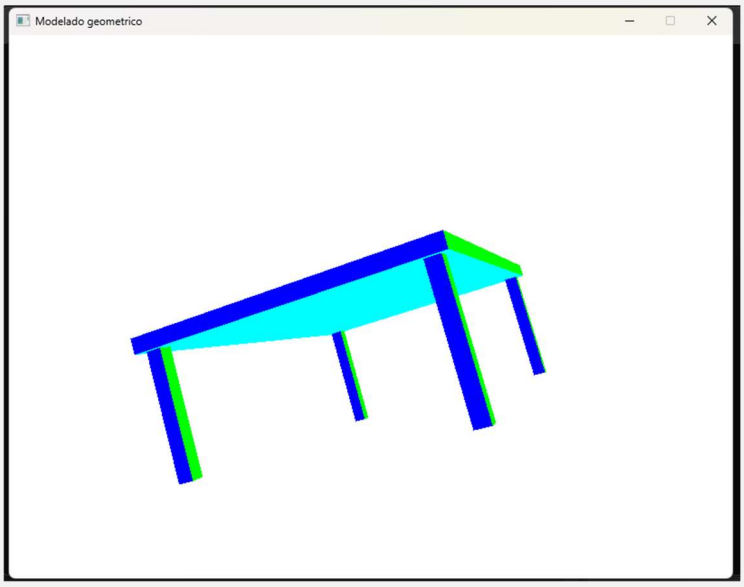
El primer paso su crear el tablero de la mesa, donde bebíamos de tener en cuenta que a partir del cubo, teníamos que modificarlo y hacer el cubo más largo en el eje X que del eje Z, y lo más pequeña posible del eje Y, el objeto que generamos lo mostramos a continuación.



El Código que genera la mesa es el siguiente, para crear la superficie de una mesa a partir de un cubo unitario. Primero se inicializa la matriz de modelo como una matriz identidad. Luego se aplica un escalamiento para darle forma de tabla con dimensiones 3.0 unidades de ancho, 0.1 unidades de grosor y 2.0 unidades de profundidad. Posteriormente se traslada la superficie 0.5 unidades hacia arriba en el eje Y para elevarla del suelo. Finalmente, se envían estas transformaciones al shader y se dibuja el objeto utilizando los 36 vértices del cubo. El orden de las operaciones es crítico: primero el escalamiento y luego la traslación, para evitar distorsiones en la posición final del objeto.

```
//tablero de la mesa
model = glm::mat4(1.0f);
model = glm::scale(model, glm::vec3(3.0f, 0.1f, 2.0f)); // se escal
model = glm::translate(model, glm::vec3(0.0f, 0.5f, 0.0f));// trasl
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);
```

Después de que teníamos la superficie de la mesa, ahora teníamos que generar las 4 patas, para ello debíamos tener en cuenta que las patas se generaban a partir del cubo también. Las patas tenían que ser de tamaño pequeñas en los ejes X y Z, pero muy altas en el eje Y.



La función pataDraw se utiliza para crear y dibujar las patas de una mesa mediante transformaciones geométricas. Recibe como parámetros una matriz de modelo, un vector de escala, un vector de traslación y la ubicación del uniforme en el shader.

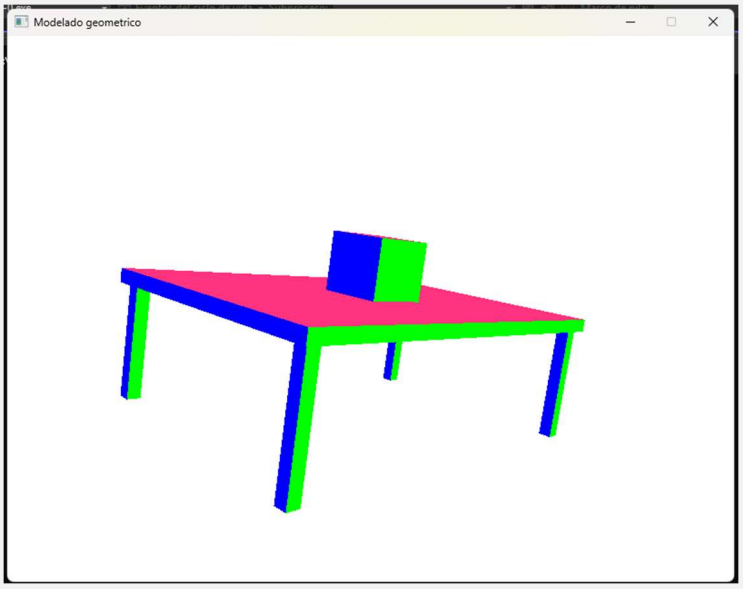
Inicializa la matriz de modelo como identidad, aplica un escalamiento para definir las dimensiones de la pata (0.1 unidades en ancho, 0.9 unidades en altura y 0.1 unidades en profundidad), y luego aplica una traslación para posicionar la pata en una de las esquinas de la mesa. Finalmente, envía la matriz transformada al shader y dibuja la pata utilizando los 36 vértices del cubo base.

Se llaman cuatro instancias de esta función para crear las cuatro patas de la mesa, posicionándolas en las esquinas de la superficie mediante traslaciones específicas en X y Z, con ligeras variaciones en Y para ajustar su altura relativa. Las patas se ubican simétricamente en las coordenadas (14, -0.4, 9), (14, -0.5, -9), (-14, -0.5, 9) y (-14, -0.5, -9), completando así la estructura básica de la mesa.

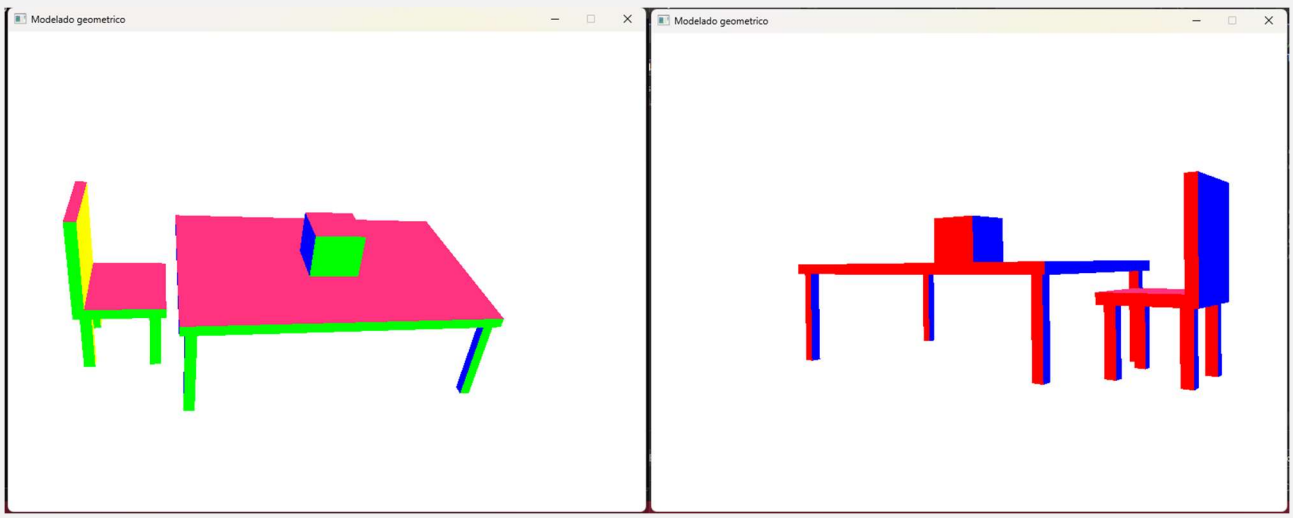
```
pataDraw(model, glm::vec3(0.1f, 0.9f, 0.1f), glm::vec3(14.0f, -0.4f, 9.0f), modelLoc);
pataDraw(model, glm::vec3(0.1f, 0.9f, 0.1f), glm::vec3(14.0f, -0.5f, -9.0f), modelLoc);
pataDraw(model, glm::vec3(0.1f, 0.9f, 0.1f), glm::vec3(-14.0f, -0.5f, 9.0f), modelLoc);
pataDraw(model, glm::vec3(0.1f, 0.9f, 0.1f), glm::vec3(-14.0f, -0.5f, -9.0f), modelLoc);
```

```
void pataDraw(glm::mat4 modelo, glm::vec3 escala, glm::vec3 traslado, GLint uniformModel) {
    modelo = glm::mat4(1);
    modelo = glm::scale(modelo, escala); // tamaño de la pata
    modelo = glm::translate(modelo, traslado);// colocamos la pata en una esquina
    glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(modelo));
    glDrawArrays(GL_TRIANGLES, 0, 36);
}
```

Además, agregamos un objeto encima de la mesa, el cual era un cubo, para ello ocupamos el mismo bloque de código de la superficie de la mesa, solo que lo escalamos para que fuera más pequeño y además agregamos traslación paras simular que esta encima de la mesa.



Ya por último decidí agregar una silla en la parte de la derecha de la mesa.



Estas fueron las instrucciones de código que utilizamos para generar las 4 patas de la silla, el asiento y el respaldo.

```
//asiento
model = glm::mat4(1.0f);
model = glm::scale(model, glm::vec3(0.8f, 0.1f, 0.8f)); // se escala en x y z para hacer
model = glm::translate(model, glm::vec3(2.5f, -2.0f, 0.0f));// trasladamos en Y (Se subio
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);

//respaldo de la silla
model = glm::mat4(1.0f);
model = glm::scale(model, glm::vec3(0.1f, 0.9f, 0.8f)); // se escala en x y z para hacer
model = glm::translate(model, glm::vec3(23.5f, 0.3f, 0.0f));// trasladamos en Y (Se subio
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);

//patas de la silla
pataDraw(model, glm::vec3(0.1f, 0.65f, 0.1f), glm::vec3(23.0f, -0.8f, 3.0f), modelLoc);
pataDraw(model, glm::vec3(0.1f, 0.65f, 0.1f), glm::vec3(23.0f, -0.8f, -3.0f), modelLoc);
pataDraw(model, glm::vec3(0.1f, 0.65f, 0.1f), glm::vec3(17.0f, -0.8f, 3.0f), modelLoc);
pataDraw(model, glm::vec3(0.1f, 0.65f, 0.1f), glm::vec3(17.0f, -0.8f, -3.0f), modelLoc);
```

Resultados: (explicar lo que se logró o aquello que no lograron realizar o comprender)

Se logró visualizar objetos sólidos en 3D utilizando primitivas gráficas, específicamente triángulos, para construir un cubo. El código utiliza un arreglo de vértices que define las posiciones y los colores de las caras del cubo.

La práctica implementa con éxito transformaciones afines como la traslación, rotación, y escalamiento para manipular objetos. Se utilizó la librería GLM para crear y aplicar matrices de modelo (model) a las formas. Por ejemplo, un cubo se escaló para formar la parte superior de una mesa y se trasladó a su posición, y otros cubos más pequeños se escalaron y trasladaron para crear las patas.

Se implementó un control de cámara que permite la interacción del usuario a través del teclado. Las teclas (W, A, S, D, PAGE UP, PAGE DOWN) controlan el movimiento de la cámara, mientras que las flechas y las teclas J/K controlan su rotación. Esto se logró manipulando la matriz de vista (view), proporcionando una experiencia interactiva para explorar la escena.

Conclusiones:

Logre el objetivo de integrar los principios de las matemáticas y la geometría computacional con la programación práctica en OpenGL para crear y manipular una escena 3D interactiva. Las actividades principales incluyeron el uso de la librería GLM para aplicar transformaciones afines como traslación, rotación y escalamiento, esenciales para el posicionamiento de objetos en el espacio virtual. Se logró modelar una escena compuesta por una mesa y un objeto encima.

La práctica demostró una gestión de los datos de los vértices con VBOs y VAOs y utilizando shaders para la visualización. Además, se implementó un sistema de control de cámara a través del teclado, lo que permitió al usuario interactuar y explorar la escena desde diferentes ángulos.

En conjunto, la práctica me ayudo a entender la capacidad de aplicar los fundamentos teóricos para modelar geométricamente una escena 3D, no solo definiendo la forma de los objetos, sino también su manipulación y visualización, cumpliendo con los objetivos establecidos.

Bibliografía consultada:

Khronos Group. (2023). *OpenGL Reference Pages*. https://www.opengl.org/documentation/

Joey de Vries. (2023). *LearnOpenGL: Getting started*. https://learnopengl.com/Getting-started

GLFW. (2023). *GLFW Documentation*. https://www.glfw.org/docs/latest/

GLM. (2023). *OpenGL Mathematics (GLM)*. https://github.com/g-truc/glm