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: Pérez Uribe José Alberto

Nº de Cuenta: 318143213

GRUPO DE LABORATORIO: 02

GRUPO DE TEORÍA: 07

SEMESTRE 2026-2

FECHA DE ENTREGA LÍMITE: 01-03-2026

CALIFICACIÓN: _____

Ejercicio 1: Iniciales PUJA en diagonal con colores distintos

En este ejercicio se retomó la forma de construir letras de la Práctica 1 (armarlas con rectángulos hechos de dos triángulos), pero ahora se integró al proyecto de la Práctica 2 usando MeshColor, shaders, matrices de proyección y transformaciones para dibujar las letras en una escena tipo 3D (con perspectiva).

Primero se creó una función auxiliar llamada AddRectColor, que agrega al arreglo de vértices un rectángulo formado por dos triángulos. La diferencia en como lo implemente a la Práctica 1 es que ahora cada vértice lleva color, por eso cada vértice se guarda como:

- Posición: x, y, z
- Color: r, g, b

```
//================= retomando la idea de como realize el ejercicio de la PRACTICA 1 ==================
static void AddRectColor(std::vector<GLfloat>& v,
    float x1, float y1, float x2, float y2, float z,
    float r, float g, float b)
{
    // Triángulo 1
    v.insert(v.end(), { x1, y1, z, r, g, b });
    v.insert(v.end(), { x2, y1, z, r, g, b });
    v.insert(v.end(), { x2, y2, z, r, g, b });

    // Triángulo 2
    v.insert(v.end(), { x1, y1, z, r, g, b });
    v.insert(v.end(), { x2, y2, z, r, g, b });
    v.insert(v.end(), { x1, y2, z, r, g, b });
}
```

Con esto, cada “barra” de una letra se puede construir solo mandando coordenadas, y OpenGL lo dibuja como triángulos.

Después, en la función CrearLetrasPUJA() se definieron proporciones generales de cada letra:

- w: ancho
- h: alto
- t: grosor del trazo
- z: profundidad (en este ejercicio las letras quedan planas y luego se posicionan con la matriz model)

También se definió un color distinto por letra usando glm::vec3:

```
// Proporciones
float w = 0.30f;  // ancho de letra
float h = 0.45f;  // alto de letra
float t = 0.07f;  // grosor del trazo
float z = 0.0f;   // letras planas en XY; luego las movemos con model

// Colores por letra (PUJA)
glm::vec3 colP(1.0f, 0.2f, 0.2f);   // P rojo
glm::vec3 colU(0.2f, 1.0f, 0.2f);   // U verde
glm::vec3 colJ(0.2f, 0.2f, 1.0f);   // J azul
glm::vec3 colA(1.0f, 1.0f, 0.2f);   // A amarillo
```

Cada letra se armó con varias llamadas a AddRectColor, igual que en práctica 1, pero ahora el color se envía desde aquí.

Ejemplo “P”:

```
// ===== LETRA P =====
{
    std::vector<GLfloat> v;
    float x = 0.0f, y = 0.0f;

    AddRectColor(v, x, y, x + t, y + h, z, colP.r, colP.g, colP.b);                          // barra izquierda
    AddRectColor(v, x, y + h - t, x + w, y + h, z, colP.r, colP.g, colP.b);                  // barra superior
    AddRectColor(v, x, y + h * 0.55f, x + w * 0.85f, y + h * 0.55f + t, z, colP.r, colP.g, colP.b); // barra media
    AddRectColor(v, x + w - t, y + h * 0.55f, x + w, y + h, z, colP.r, colP.g, colP.b);      // barra derecha arriba
```

Al final de cada letra, se crea un objeto MeshColor y se mete a meshColorList. Esto permite dibujar cada letra como una malla independiente.

```
    MeshColor* P = new MeshColor();
    P->CreateMeshColor(v.data(), (unsigned int)v.size());
    meshColorList.push_back(P);
```

En esta práctica se usaron dos sets de shaders (como el código que vimos en clase), pero para este ejercicio se utilizó el shader que permite color desde los vértices:

```
//================== Shaders ==================
void CreateShaders()
{
    Shader* shader1 = new Shader(); // índices
    shader1->CreateFromFiles(vShader, fShader);
    shaderList.push_back(*shader1);

    Shader* shader2 = new Shader(); // color por vértice (MeshColor) -> letras
    shader2->CreateFromFiles(vShaderColor, fShaderColor);
    shaderList.push_back(*shader2);
}
```

En el main se configuró una matriz de proyección en perspectiva para trabajar con una escena donde la cámara “mira” hacia el eje Z negativo. Por eso las letras se mandan a z = -5.0f para que entren en el campo de visión.

```
glm::mat4 projection = glm::perspective(
    glm::radians(60.0f),
    (float)mainWindow.getBufferWidth() / (float)mainWindow.getBufferHeight(),
    0.1f, 100.0f
);
```

Luego se definió un arreglo pos[4] para colocar las letras en el orden P-U-J-A y en diagonal de abajo hacia arriba, manteniendo las posiciones similares a como las desarrolle en la Práctica 1.

```
// (diagonal de abajo hacia arriba)
glm::vec3 pos[4] = {
    glm::vec3(-1.20f, -1.20f, -5.0f), // P
    glm::vec3(-0.40f, -0.40f, -5.0f), // U
    glm::vec3(0.40f,  0.40f, -5.0f), // J
    glm::vec3(1.20f,  1.20f, -5.0f)  // A
};
```

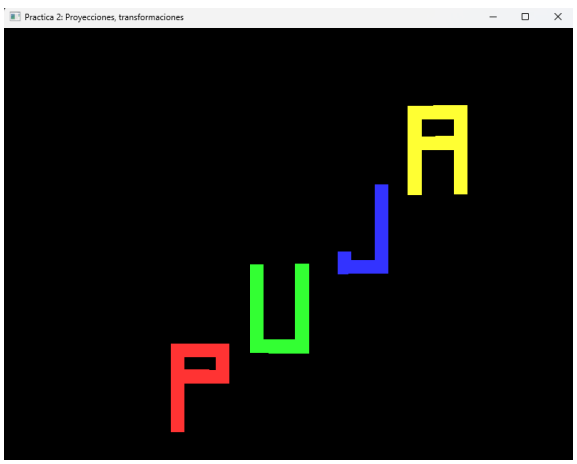
Dentro del while, se limpia la pantalla y se activa el shader de letras (shaderList[1]). Se obtienen los uniforms model y projection para mandar matrices al shader.

```
shaderList[1].useShader();
uniformModel = shaderList[1].getModelLocation();
uniformProjection = shaderList[1].getProjectLocation();

glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
```

Finalmente, se recorren las 4 letras con un for y para cada una se construye su matriz model con:

1. translate (posición de la diagonal)
2. rotate (rotación lenta)
3. scale (tamaño general)

```
for (int i = 0; i < 4; i++)
{
    model = glm::mat4(1.0f);
    model = glm::translate(model, pos[i]);
    model = glm::rotate(model, angulo, glm::vec3(0.0f, 1.0f, 0.0f));
    model = glm::scale(model, glm::vec3(scaleL, scaleL, 1.0f));
    glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));
    meshColorList[i]->RenderMeshColor();
}
```



En estas imágenes se puede ver cómo se muestra el resultado del programa mientras está corriendo. Las letras PUJA aparecen acomodadas en diagonal y cada una tiene un color diferente, lo que hace que se distingan claramente entre sí.

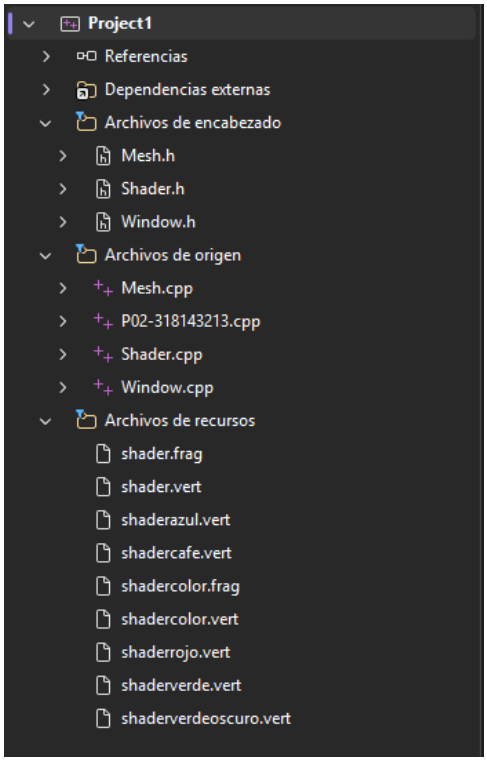
En algunas capturas las letras se ven completas y fáciles de reconocer, y en otras se ven más delgadas, casi como si estuvieran “de lado”. Esto pasa porque la vista va cambiando ligeramente y entonces no siempre se ven exactamente desde el frente.

Aun así, se mantiene el orden de las letras, su posición en diagonal y sus colores. Esto demuestra que el programa está dibujando correctamente cada letra dentro de la escena y que el resultado se mantiene estable durante la ejecución.

Ejercicio 2: Casa con cubos y pirámides usando shaders por color

En este ejercicio se recreó el dibujo de la casa de la clase, pero en lugar de usar triángulos y cuadrados directamente, se armó todo usando primitivas 3D: usando cubos y piramides. Además, el color ya no se asigna “a mano” en el arreglo de vértices, sino que se controla con shaders diferentes por color (rojo, verde, azul, café y verde oscuro).

Para cumplir la parte de shaders por color, se agregaron archivos .vert nuevos (uno por color que se usaran en la recreación del ejercicio de clase) y un fragment shader común. En el proyecto se pueden ver dentro de “Archivos de recursos”.



Primero se definieron las dos figuras base:

- CreaPiramide() crea una pirámide con un arreglo de vértices y otro de índices.
- CrearCubo() crea un cubo con sus 8 vértices y un arreglo de índices para dibujar sus caras.

```cpp
// ---------- Crear pirámide y cubo ----------
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* obj = new Mesh();
    obj->CreateMesh(vertices, indices, 12, 12);
    meshList.push_back(obj);
}
```

```cpp
void CrearCubo()
{
    unsigned int idx[] = {
        // 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 v[] = {
        // 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(v, idx, 24, 36);
    meshList.push_back(cubo);
}
```

Con esto se guardan en meshList:

- meshList[0] = pirámide
- meshList[1] = cubo

En CreateShaders() se cargaron diferentes vertex shaders, uno por color, todos usando el mismo fragment shader (shadercolor.frag). Esto se hizo para que cada parte de la casa se dibuje con un shader específico según el color que debe llevar.

```
// ---------- Shaders por color  ----------
void CreateShaders()
{
    // 0 rojo
    Shader* rojo = new Shader();
    rojo->CreateFromFiles("shaders/shaderrojo.vert", "shaders/shadercolor.frag");
    shaderList.push_back(*rojo);

    // 1 verde
    Shader* verde = new Shader();
    verde->CreateFromFiles("shaders/shaderverde.vert", "shaders/shadercolor.frag");
    shaderList.push_back(*verde);

    // 2 azul
    Shader* azul = new Shader();
    azul->CreateFromFiles("shaders/shaderazul.vert", "shaders/shadercolor.frag");
    shaderList.push_back(*azul);

    // 3 cafe
    Shader* cafe = new Shader();
    cafe->CreateFromFiles("shaders/shadercafe.vert", "shaders/shadercolor.frag");
    shaderList.push_back(*cafe);

    // 4 verde oscuro
    Shader* verdeOsc = new Shader();
    verdeOsc->CreateFromFiles("shaders/shaderverdeoscuro.vert", "shaders/shadercolor.frag");
    shaderList.push_back(*verdeOsc);
}
```

Para no repetir código, se creó DrawMeshColor() que recibe:

- shaderIndex: qué shader usar (color)
- uniformModel y uniformProjection: para mandar matrices
- model y projection: matrices actuales
- mesh: qué figura dibujar (cubo o pirámide)

```
static void DrawMeshColor(int shaderIndex, GLuint uniformModel, GLuint uniformProjection,
    const glm::mat4& model, const glm::mat4& projection, Mesh* mesh)
{
    shaderList[shaderIndex].useShader();
    glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));
    glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
    mesh->RenderMesh();
}
```

```
int main()
{
    mainWindow = Window(800, 800);
    mainWindow.Initialise();

    glEnable(GL_DEPTH_TEST);

    CreaPiramide(); // meshList[0]
    CrearCubo();    // meshList[1]
    CreateShaders();

    glm::mat4 projection = glm::perspective(
        glm::radians(60.0f),
        (float)mainWindow.getBufferWidth() / (float)mainWindow.getBufferHeight(),
        0.1f, 100.0f
    );
```

En esta parte del main se realiza la configuración inicial necesaria para poder trabajar en la escena 3D. Primero se crea e inicializa la ventana donde se va a dibujar todo, lo que permite que OpenGL tenga un contexto activo para renderizar. Después se activa el GL_DEPTH_TEST, que es fundamental en 3D porque permite que los objetos se dibujen respetando la profundidad, es decir, que los que están más cerca tapen correctamente a los que están más lejos.

Posteriormente se crean las figuras base (la pirámide y el cubo) que se utilizarán para construir la casa, y se cargan los shaders correspondientes a cada color. Finalmente, se define la matriz de proyección en perspectiva, que es la que hace que la escena tenga efecto de profundidad y se vea como un espacio tridimensional. En conjunto, este bloque deja preparada la escena antes de comenzar a dibujar los objetos dentro del ciclo principal.

```
while (!mainWindow.getShouldClose())
{
    glfwPollEvents();

    glClearColor(1.0f, 1.0f, 1.0f, 1.0f); // fondo blanco
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Rotación lenta para apreciar los cubos y piramides
    angulo += 0.0001f;
    glm::mat4 scene(1.0f);
    scene = glm::translate(scene, glm::vec3(0.0f, 0.0f, -8.0f));
    scene = glm::rotate(scene, angulo, glm::vec3(0.0f, 1.0f, 0.0f));
```

En estas líneas se ejecuta el ciclo principal del programa, que es el que mantiene la ventana activa mientras no se cierre. Primero se actualizan los eventos del teclado y mouse con glfwPollEvents(), y luego se limpia la pantalla con un fondo blanco, además de reiniciar tanto el buffer de color como el de profundidad para evitar que se mezclen los dibujos del cuadro anterior. Después se incrementa ligeramente la variable angulo, lo que permite que la escena rote lentamente.

Finalmente, se crea una matriz llamada scene que mueve toda la escena hacia atrás en el eje Z (para que pueda verse con la cámara en perspectiva) y luego aplica una rotación en el eje Y, haciendo que los cubos y pirámides se puedan apreciar desde distintos ángulos mientras el programa está en ejecución.

```
// ---------------- CASA ----------------
// Cuerpo (cubo rojo)
{
    glm::mat4 model = scene;
    model = glm::translate(model, glm::vec3(0.0f, -0.2f, 0.0f));
    model = glm::scale(model, glm::vec3(3.0f, 2.8f, 1.2f));
    DrawMeshColor(0, uniformModel, uniformProjection, model, projection, meshList[1]);
}

// Techo (pirámide azul)
{
    glm::mat4 model = scene;
    model = glm::translate(model, glm::vec3(0.0f, 2.0f, 0.65f));
    model = glm::scale(model, glm::vec3(3.4f, 1.6f, 2.51f));
    DrawMeshColor(2, uniformModel, uniformProjection, model, projection, meshList[0]);
}

// Ventana izquierda (cubo verde)
{
    glm::mat4 model = scene;
    model = glm::translate(model, glm::vec3(-0.9f, 0.5f, 0.65f));
    model = glm::scale(model, glm::vec3(0.8f, 0.8f, 0.1f));
    DrawMeshColor(1, uniformModel, uniformProjection, model, projection, meshList[1]);
}

// Ventana derecha (cubo verde)
{
    glm::mat4 model = scene;
    model = glm::translate(model, glm::vec3(0.9f, 0.5f, 0.65f));
    model = glm::scale(model, glm::vec3(0.8f, 0.8f, 0.1f));
    DrawMeshColor(1, uniformModel, uniformProjection, model, projection, meshList[1]);
}

// Puerta (cubo verde)
{
    glm::mat4 model = scene;
    model = glm::translate(model, glm::vec3(0.0f, -1.1f, 0.65f));
    model = glm::scale(model, glm::vec3(0.9f, 1.2f, 0.1f));
    DrawMeshColor(1, uniformModel, uniformProjection, model, projection, meshList[1]);
}
```

En este bloque se está construyendo la casa pieza por pieza usando la matriz scene como base. Para cada parte (cuerpo, techo, ventanas y puerta), primero se copia scene en una nueva matriz llamada model, y sobre ella se aplican transformaciones específicas. Con glm::translate se posiciona cada objeto en el lugar correcto dentro de la escena, y con glm::scale se ajusta su tamaño para que tenga las proporciones adecuadas.

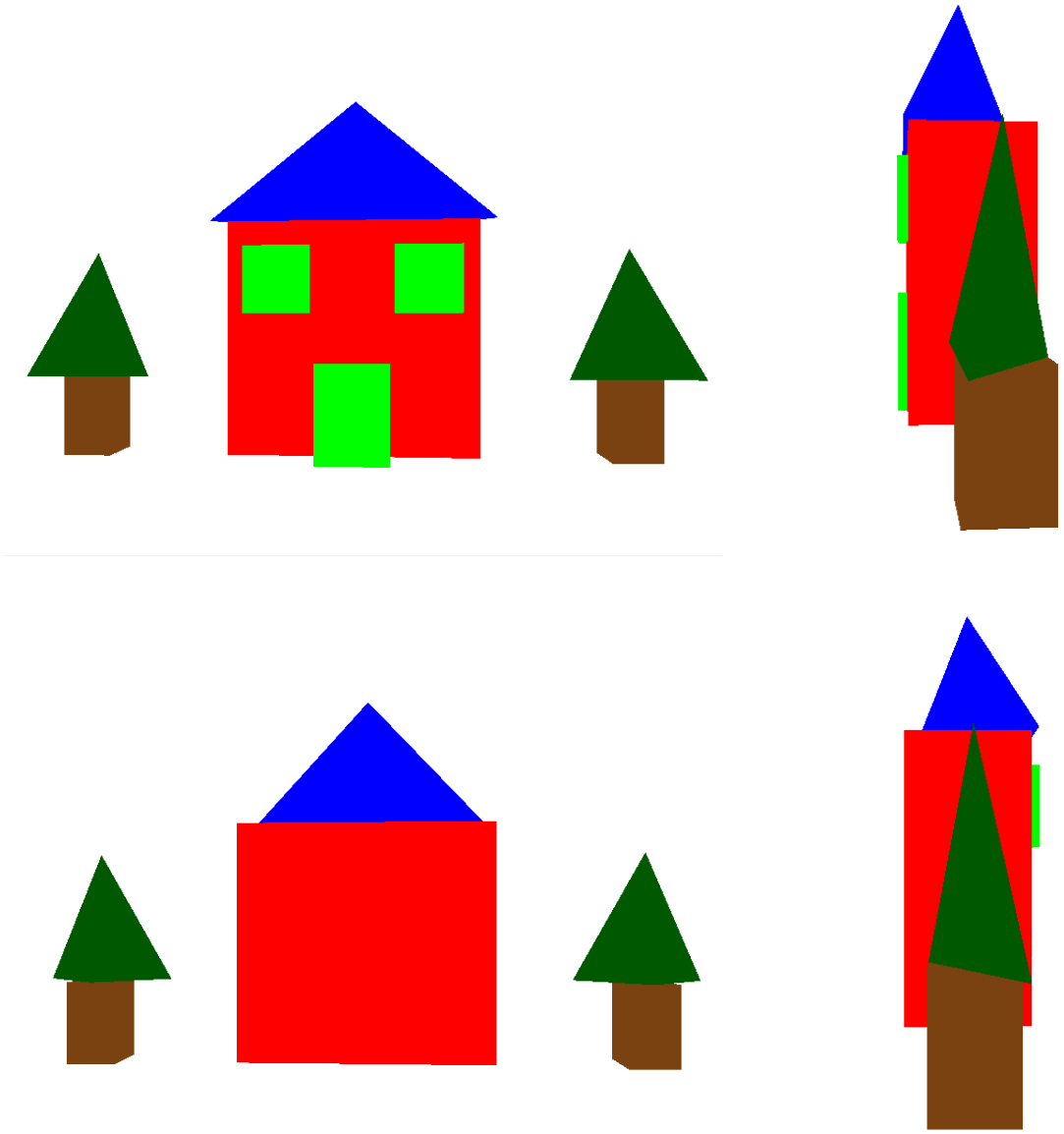
El cuerpo de la casa se dibuja como un cubo rojo grande, el techo como una pirámide azul colocada encima, y las ventanas y la puerta como cubos verdes más delgados colocados al frente. Finalmente, en cada caso se llama a

DrawMeshColor, que se encarga de usar el shader correspondiente al color indicado y renderizar el cubo o la pirámide en esa posición. Así, combinando traslación, escala y el shader correcto, se arma visualmente toda la estructura de la casa.

```
        // ---------------- ÁRBOL IZQUIERDO ----------------
        // Tronco (cubo café)
        {
            glm::mat4 model = scene;
            model = glm::translate(model, glm::vec3(-3.4f, -1.2f, 0.0f));
            model = glm::scale(model, glm::vec3(0.6f, 1.0f, 0.6f));
            DrawMeshColor(3, uniformModel, uniformProjection, model, projection, meshList[1]);
        }

        // Copa (pirámide verde oscuro)
        {
            glm::mat4 model = scene;
            model = glm::translate(model, glm::vec3(-3.4f, 0.1f, 0.3f));
            model = glm::scale(model, glm::vec3(1.6f, 1.6f, 1.2f));
            DrawMeshColor(4, uniformModel, uniformProjection, model, projection, meshList[0]);
        }

        // ---------------- ÁRBOL DERECHO ----------------
        // Tronco (cubo café)
        {
            glm::mat4 model = scene;
            model = glm::translate(model, glm::vec3(3.4f, -1.2f, 0.0f));
            model = glm::scale(model, glm::vec3(0.6f, 1.0f, 0.6f));
            DrawMeshColor(3, uniformModel, uniformProjection, model, projection, meshList[1]);
        }

        // Copa (pirámide verde oscuro)
        {
            glm::mat4 model = scene;
            model = glm::translate(model, glm::vec3(3.4f, 0.1f, 0.3f));
            model = glm::scale(model, glm::vec3(1.6f, 1.6f, 1.2f));
            DrawMeshColor(4, uniformModel, uniformProjection, model, projection, meshList[0]);
        }

        glUseProgram(0);
        mainWindow.swapBuffers();
    }

    return 0;
}
```

En este bloque se construyen los dos árboles que acompañan a la casa. Cada árbol se arma utilizando dos figuras: un cubo para el tronco y una pirámide para la copa. Igual que en la casa, primero se toma la matriz scene como base y después se aplican transformaciones de traslación y escala para colocar cada pieza en su posición correspondiente y darle el tamaño adecuado.

Para el árbol izquierdo, el cubo café se posiciona más hacia la izquierda y ligeramente abajo para simular el tronco, mientras que la pirámide verde oscuro se coloca encima para formar la copa. Luego se repite el mismo procedimiento para el árbol derecho, pero cambiando la posición en el eje X hacia el lado contrario. Finalmente, se llama a DrawMeshColor con el shader correspondiente para dibujar cada parte con su color definido. Al terminar de renderizar todos los objetos, se intercambian los buffers con swapBuffers() para mostrar en pantalla la escena completa ya actualizada.



En esta secuencia de imágenes se observa el resultado del ejercicio 2 mientras el programa está en ejecución. En la primera vista se aprecia la casa de frente, con el cuerpo rojo, el techo azul, las ventanas y puerta verdes, y los árboles a los lados. La escena mantiene la proporción y distribución similar al dibujo original de clase.

En las demás capturas se nota el efecto de la rotación aplicada a toda la escena. Al girar, se puede ver que la casa y los árboles no son figuras planas, sino que están construidos con cubos y pirámides reales. Desde ciertos ángulos se aprecia la profundidad del cuerpo de la casa, el volumen del techo y el grosor de los troncos. Esto confirma que el modelo está construido en 3D y que el Depth Test está funcionando correctamente, permitiendo que los objetos se oculten entre sí según su posición en el espacio.

2.- Problemas encontrados y cómo se resolvieron

Uno de los primeros problemas que tuve fue que no se veían las letras cuando cambié a glm::perspective. El programa sí corría, pero la pantalla salía negra. Después de revisar, me di cuenta de que estaba dejando las letras en z = 0 y con perspectiva eso casi no se aprecia o queda fuera de la vista. Lo resolví mandando las letras hacia atrás en el eje Z (por ejemplo z = -5.0f) usando glm::translate, y ya con eso se empezaron a ver.

Otro problema fue que, al inicio, las letras se veían muy raras cuando rotaban, casi como si desaparecieran. Me confundí pensando que era un error de vértices, pero realmente era por la rotación y la perspectiva, ya que las letras son planas y al girar se ven muy delgadas de lado. Para que no se sintiera como “falla”, bajé mucho la velocidad de rotación (angulo += 0.0001f) y también ajusté el tamaño con scale para que se mantuvieran visibles.

En el ejercicio de la casa tuve un problema con los shaders: el programa no dibujaba nada aunque el código parecía correcto. Después de revisar, me di cuenta de que no había agregado los archivos .vert nuevos al proyecto, solamente los había creado en la carpeta. Visual Studio no los estaba considerando realmente. Lo solucioné agregando manualmente cada archivo .vert desde “Agregar elemento existente” dentro de “Archivos de recursos”, y después de eso los shaders cargaron correctamente y la casa se mostró en pantalla.

Finalmente, para que la casa se pareciera al dibujo original, tuve que hacer varios ajustes de posición y tamaño. Al principio las ventanas y la puerta quedaban “volando” o metidas dentro del cubo rojo. Lo solucioné probando valores en translate y scale hasta que quedaran acomodadas al frente, y dejé una rotación lenta para poder revisar que realmente fueran cubos y pirámides y no solo figuras planas.

3.Conclusión

a) Los ejercicios del reporte: Complejidad y explicación

En esta práctica la complejidad fue un poco mayor que en la anterior, principalmente porque ya no solo se trataba de dibujar figuras en 2D, sino de empezar a trabajar con proyecciones y transformaciones en un entorno con profundidad. El primer ejercicio, donde se dibujaron las letras en diagonal, no fue complicado en cuanto a lógica, ya que la base venía de la práctica anterior. Sin embargo, sí cambió la forma de pensar el espacio, porque ahora se trabajó con perspectiva y con movimientos en el eje Z. Eso hizo que pequeños detalles, como la posición o la escala, influyeran mucho en cómo se veían las figuras.

El segundo ejercicio fue más completo, ya que implicó construir una escena formada por varias partes: cubos y pirámides que juntos representaban una casa y árboles. Aquí la dificultad estuvo más en acomodar correctamente cada figura, ajustar tamaños y posiciones hasta que visualmente coincidieran con el dibujo original. No fue un ejercicio difícil en cuanto a fórmulas, pero sí requirió paciencia y varias pruebas hasta que todo encajara bien.

En general, los ejercicios no fueron excesivamente complicados, pero sí exigieron mayor atención a los detalles y una mejor comprensión del espacio tridimensional.

b) Comentarios generales

En general, la práctica fue clara en cuanto a lo que se pedía realizar, pero al momento de llevarlo al código surgen dudas, sobre todo cuando se empieza a trabajar con matrices de transformación y perspectiva. Al principio cuesta visualizar mentalmente qué está pasando cuando se aplica una traslación o una rotación, especialmente porque un pequeño cambio en los valores puede alterar mucho la posición final del objeto.

También considero que cuando se trabaja con varias figuras al mismo tiempo (como en la casa), sería útil mostrar más ejemplos de cómo implementar otras figuras, por ejemplo, explicando cómo decidir las coordenadas y escalas antes de empezar a escribir el código. Muchas veces el proceso es más de prueba y error que de cálculo exacto, y entender eso desde el principio ayudaría a trabajar con más seguridad.

Aun así, la práctica fue interesante porque permitió aplicar lo visto en clase de manera más visual, y eso facilita entender conceptos que en teoría pueden parecer abstractos.

c) Conclusión

En esta práctica sentí que ya empezamos a trabajar más en serio con OpenGL. En la práctica anterior todo era más plano y directo, pero aquí ya tuvimos que pensar en profundidad, perspectiva y en cómo acomodar objetos dentro de un

espacio 3D. Al principio sí fue un poco confuso entender por qué algo no se veía o por qué se veía diferente al girar la escena, pero justo eso me ayudó a comprender mejor cómo funciona el programa.

Algo que me gustó fue que no solo dibujamos figuras simples, sino que armamos algo más completo, como la casa con los árboles. Eso hizo que el ejercicio se sintiera más “real”, porque cada parte tenía que encajar con las demás. También trabajar con diferentes shaders para cada color me ayudó a entender mejor cómo se separa la parte visual del objeto de su forma.

En general, esta práctica me ayudó a dejar de ver OpenGL como solo código y empezar a verlo como un espacio donde uno acomoda objetos, los mueve, los escala y los organiza hasta formar una escena completa. Todavía hay cosas que cuestan un poco, pero ya se entiende mucho mejor qué está pasando cuando algo se dibuja en pantalla.

Bibliografía

• LearnOpenGL - Sistemas de coordenadas. (n.d.). https://learnopenglcom.translate.goog/Getting-started/CoordinateSystems?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc

• LearnOpenGL - Hello Triangle. (n.d.). https://learnopengl.com/Gettingstarted/Hello-Triangle

• LearnOpenGL - Transformations. (n.d.). https://learnopengl.com/Gettingstarted/Transformations

• LearnOpenGL - Shaders. (n.d.). https://learnopengl.com/Gettingstarted/Shaders

• GeeksforGeeks. (2025, July 23). extern Keyword in C. GeeksforGeeks. https://www.geeksforgeeks.org/c/understanding-extern-keyword-in-c/

• Khronos Group. (2023). The OpenGL® Shading Language (Version 4.60.8) (G. Leese, J. Kessenich, D. Baldwin & R. Rost). https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.60.pdf