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: Carandia Lorenzo Brenda Fernanda

Nº de Cuenta: 319018961

GRUPO DE LABORATORIO: 03

GRUPO DE TEORÍA: 05

SEMESTRE 2025-2

FECHA DE ENTREGA LÍMITE: 22 de febrero de 2024

CALIFICACIÓN: _____

PRÁCTICA 02: PROYECCIONES Y PUERTOS DE VISTA.

TRANSFORMACIONES GEOMÉTRICAS

ACTIVIDADES REALIZADAS

1.- Dibujar las iniciales de sus nombres, cada letra de un color diferente

Bloque de código

- **FUNCIÓN PARA CREAR LAS LETRAS**

```
34  void CrearLetrasyFiguras()
35  {
36      GLfloat vertices_letra_f[] = { // Letra F
37
38          //x     y       z       R       G       B  Verde palido 218, 247, 166
39          -0.4f, 0.7f, 0.0f,  0.855f, 0.969f, 0.651f,
40          -0.4f, 0.6f, 0.0f,  0.855f, 0.969f, 0.651f,
41          -0.5f, 0.7f, 0.0f,  0.855f, 0.969f, 0.651f,
42
43          -0.5f, 0.7f, 0.0f,  0.855f, 0.969f, 0.651f,
44          -0.4f, 0.6f, 0.0f,  0.855f, 0.969f, 0.651f,
45          -0.6f, 0.6f, 0.0f,  0.855f, 0.969f, 0.651f,
46
47          -0.5f, 0.7f, 0.0f,  0.855f, 0.969f, 0.651f,
48          -0.6f, 0.6f, 0.0f,  0.855f, 0.969f, 0.651f,
49          -0.7f, 0.7f, 0.0f,  0.855f, 0.969f, 0.651f,
50
51          -0.7f, 0.7f, 0.0f,  0.855f, 0.969f, 0.651f,
52          -0.6f, 0.6f, 0.0f,  0.855f, 0.969f, 0.651f,
53          -0.7f, 0.5f, 0.0f,  0.855f, 0.969f, 0.651f,
54
55          -0.6f, 0.6f, 0.0f,  0.855f, 0.969f, 0.651f,
56          -0.7f, 0.5f, 0.0f,  0.855f, 0.969f, 0.651f,
57          -0.6f, 0.5f, 0.0f,  0.855f, 0.969f, 0.651f,
58
59          -0.6f, 0.5f, 0.0f,  0.855f, 0.969f, 0.651f,
60          -0.5f, 0.5f, 0.0f,  0.855f, 0.969f, 0.651f,
61          -0.5f, 0.4f, 0.0f,  0.855f, 0.969f, 0.651f,
62
63          -0.6f, 0.5f, 0.0f,  0.855f, 0.969f, 0.651f,
64          -0.5f, 0.4f, 0.0f,  0.855f, 0.969f, 0.651f,
65          -0.6f, 0.4f, 0.0f,  0.855f, 0.969f, 0.651f,
66
67          -0.7f, 0.5f, 0.0f,  0.855f, 0.969f, 0.651f,
68          -0.6f, 0.4f, 0.0f,  0.855f, 0.969f, 0.651f,
69          -0.6f, 0.5f, 0.0f,  0.855f, 0.969f, 0.651f,
70
71          -0.7f, 0.5f, 0.0f,  0.855f, 0.969f, 0.651f,
72          -0.6f, 0.4f, 0.0f,  0.855f, 0.969f, 0.651f,
73          -0.7f, 0.2f, 0.0f,  0.855f, 0.969f, 0.651f,
74
75          -0.7f, 0.2f, 0.0f,  0.855f, 0.969f, 0.651f,
76          -0.6f, 0.4f, 0.0f,  0.855f, 0.969f, 0.651f,
77          -0.6f, 0.2f, 0.0f,  0.855f, 0.969f, 0.651f,
78      };
79
80      MeshColor* letra_f= new MeshColor();
81      letra_f->CreateMeshColor(vertices_letra_f, 180); // Ponemos los vertices para dibujarlos
82      meshColorList.push_back(letra_f);
```

```
GLfloat vertices_letra_c[] = { // Letra C

    //x     y       z       R    G    B     Naranja 255, 195, 0
    0.7f, 0.54f, 0.0f,  1.0f, 0.765f, 0.0f,
    0.6f, 0.54f, 0.0f,  1.0f, 0.765f, 0.0f,
    0.7f, 0.6f, 0.0f,   1.0f, 0.765f, 0.0f,

    0.7f, 0.6f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.6f, 0.54f, 0.0f,  1.0f, 0.765f, 0.0f,
    0.6f, 0.6f, 0.0f,   1.0f, 0.765f, 0.0f,

    0.7f, 0.6f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.6f, 0.6f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.6f, 0.7f, 0.0f,   1.0f, 0.765f, 0.0f,

    0.6f, 0.7f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.6f, 0.6f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.4f, 0.6f, 0.0f,   1.0f, 0.765f, 0.0f,

    0.6f, 0.7f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.4f, 0.6f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.4f, 0.7f, 0.0f,   1.0f, 0.765f, 0.0f,

    0.4f, 0.7f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.4f, 0.6f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.3f, 0.6f, 0.0f,   1.0f, 0.765f, 0.0f,

    0.3f, 0.6f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.4f, 0.6f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.3f, 0.3f, 0.0f,   1.0f, 0.765f, 0.0f,

    0.4f, 0.6f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.3f, 0.3f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.4f, 0.3f, 0.0f,   1.0f, 0.765f, 0.0f,

    0.3f, 0.3f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.4f, 0.3f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.4f, 0.2f, 0.0f,   1.0f, 0.765f, 0.0f,

    0.4f, 0.3f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.4f, 0.2f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.6f, 0.2f, 0.0f,   1.0f, 0.765f, 0.0f,

    0.4f, 0.3f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.6f, 0.2f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.6f, 0.3f, 0.0f,   1.0f, 0.765f, 0.0f,

    0.6f, 0.3f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.6f, 0.2f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.7f, 0.3f, 0.0f,   1.0f, 0.765f, 0.0f,

    0.6f, 0.36f, 0.0f,  1.0f, 0.765f, 0.0f,
    0.6f, 0.3f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.7f, 0.3f, 0.0f,   1.0f, 0.765f, 0.0f,

    0.7f, 0.3f, 0.0f,   1.0f, 0.765f, 0.0f,
    0.6f, 0.36f, 0.0f,  1.0f, 0.765f, 0.0f,
    0.7f, 0.36f, 0.0f,  1.0f, 0.765f, 0.0f,
};

MeshColor* letra_c = new MeshColor();
letra_c->CreateMeshColor(vertices_letra_c, 252); // Ponemos los vertices para dibujarlo
meshColorList.push_back(letra_c);
```

```
148     GLfloat vertices_letra_l[] = { // Letra     L
149         //x     y       z       R    G    B     255, 87, 51
150         -0.1f, -0.2f, 0.0f, 1.0f, 0.341f, 0.2f,
151         0.0f, -0.2f, 0.0f,  1.0f, 0.341f, 0.2f,
152         -0.1f, -0.7f, 0.0f, 1.0f, 0.341f, 0.2f,
153
154         0.0f, -0.2f, 0.0f,  1.0f, 0.341f, 0.2f,
155         -0.1f, -0.7f, 0.0f, 1.0f, 0.341f, 0.2f,
156         0.0f, -0.7f, 0.0f,  1.0f, 0.341f, 0.2f,
157
158         0.0f, -0.6f, 0.0f,  1.0f, 0.341f, 0.2f,
159         0.0f, -0.7f, 0.0f,  1.0f, 0.341f, 0.2f,
160         0.2f, -0.7f, 0.0f,  1.0f, 0.341f, 0.2f,
161
162         0.2f, -0.7f, 0.0f,  1.0f, 0.341f, 0.2f,
163         0.0f, -0.6f, 0.0f,  1.0f, 0.341f, 0.2f,
164         0.2f, -0.6f, 0.0f,  1.0f, 0.341f, 0.2f,
165     };
166
167     MeshColor* letra_l = new MeshColor();
168     letra_l->CreateMeshColor(vertices_letra_l, 72);// Ponemos los vertices para dibujarlos
169     meshColorList.push_back(letra_l);
170 };
```

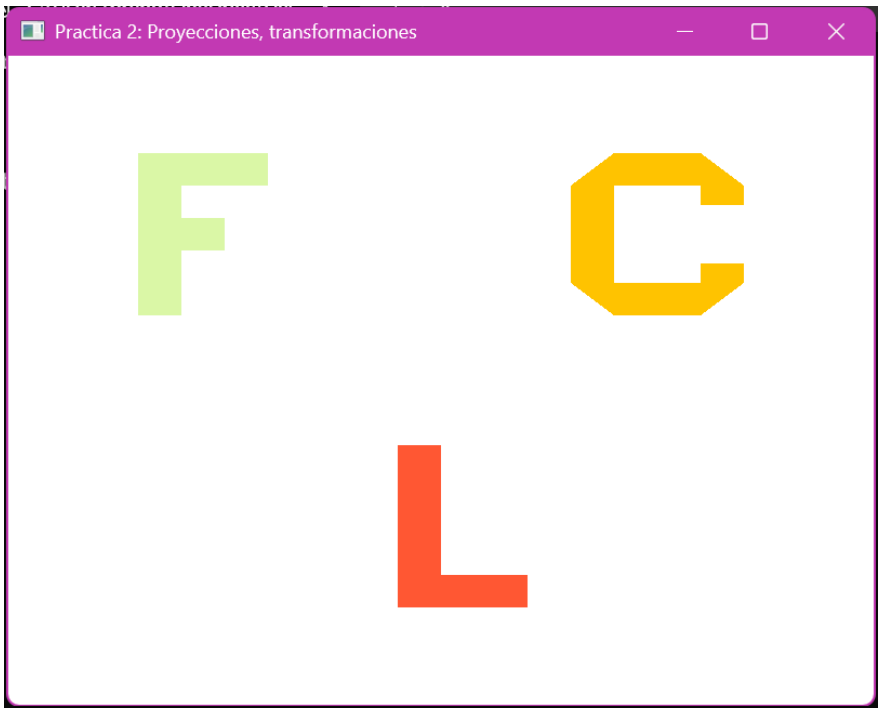
En la función `CrearLetrasyFiguras()` tenemos los vértices para crear los triángulos correspondientes y que formen las letras F, C y L, solo que en este caso, también se agregaron los vértices de los colores para cada letra, es por cada vértice de un triángulo, se tiene el vértice del color RGB (normalizado), por ejemplo para la letra F, se eligió el color verde pálido y se necesitaron 180 vértices, para la letra C el color naranja con 252 vértices y para la letra L el color rojo con 72 vértices. Y estos se pasan a `meshColorList` para posteriormente dibujarlos conforme a su color y posición en la lista

- **MAIN**

Mandamos a llamar a la función CrearLetrasyFiguras`()` para posteriormente declarar una proyección ortogonal para apreciar a las letras en 2D, ponemos el fondo de color blanco, y dentro del while, se usa el segundo set de shaders `ShaderList` con índice en 1, se manda a renderizar a las letra con su respectivo color conforme a la posición en que fueron declaradas F en la posición 0, C en 1 y L en 2, esto con `meshColorList[i] -> RenderMeshColor()`.

```cpp
int main()
{
    mainWindow = Window(800, 600);
    mainWindow.Initialise();
    CrearLetrasyFiguras();
    CreateShaders();
    GLuint uniformProjection = 0;
    GLuint uniformModel = 0;
    //Projection: Matriz de Dimensión 4x4 para indicar si vemos en 2D( orthogonal) o en 3D) perspectiva
    glm::mat4 projection = glm::ortho(-1.0f, 1.0f, -1.0f, 1.0f, 0.1f, 100.0f);
    //glm::mat4 projection = glm::perspective(glm::radians(60.0f)   ,mainWindow.getBufferWidth() / mainWindow.getBufferHeight(), 0.1f, 100.0f);

    //Model: Matriz de Dimensión 4x4 en la cual se almacena la multiplicación de las transformaciones geométricas.
    glm::mat4 model(1.0); //fuera del while se usa para inicializar la matriz con una identidad

    //Loop mientras no se cierra la ventana
    while (!mainWindow.getShouldClose())
    {
        //Recibir eventos del usuario
        glfwPollEvents();
        //Limpiar la ventana
        glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); //Se agrega limpiar el buffer de profundidad


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

        //Inicializar matriz de dimensión 4x4 que servirá como matriz de modelo para almacenar las transformaciones geométricas
        model = glm::mat4(1.0);
        model = glm::translate(model, glm::vec3(0.0f, 0.0f, -4.0f));
        //
        glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));//FALSE ES PARA QUE NO SEA TRANSPUESTA y se envían al shader como variables de tipo uniform
        glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
        meshColorList[0]->RenderMeshColor(); //Renderiza/Dibuja a la letra F
        meshColorList[1]->RenderMeshColor();//Renderiza/Dibuja a la letra C
        meshColorList[2]->RenderMeshColor();//Renderiza/Dibuja a la letra L

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

- **Ejecución del programa (Cambio de color en las letras)**



Se puede observar que cada letra tiene un color distinto.

2. **Generar el dibujo de la casa de la clase, pero en lugar de instanciar triángulos y cuadrados será instanciando pirámides y cubos, para esto se requiere crear shaders diferentes de los colores: rojo, verde, azul, café y verde oscuro en lugar de usar el shader con el color clamp**

Bloque de código - CREACIÓN DE SHADER . FRAG

ROJO

```
#version 330

out vec4 color;
void main()
{
    color = vec4(1.0f, 0.0f, 0.0f, 1.0f); // Rojo
}
```

AZUL

```
#version 330

out vec4 color;
void main()
{
    color = vec4(0.0f, 0.0f, 1.0f, 1.0f); // Azul
}
```

VERDE

```
#version 330
out vec4 color;
void main()
{
    color = vec4(0.0f, 1.0f, 0.0f, 1.0f); // Verde
}
```

CAFÉ

```
#version 330

out vec4 color;
void main()
{
    color = vec4(0.478f,    0.255f, 0.067f, 1.0f); // cafe
}
```

VERDE OSCURO

```
#version 330

out vec4 color;
void main()
{
    color = vec4(0.0f, 0.5f, 0.0f, 1.0f); // Verde oscuro
}
```

SHADER.VERT

```
#version 330
layout (location =0) in vec3 pos;

out vec4 vColor;
uniform vec4 color;
uniform mat4 model;
uniform mat4 projection;
void main()
{
    gl_Position=projection*model*vec4(pos,1.0f);
    vColor=vec4(color);
```

Se crean los `shader.frag` que contendrán los colores para las pirámides y cubos que constituyan la casa. En donde `color` será la salida del color que está definido. Y en el `shader.vert` se cambia el parámetro de `vec4`

- **RUTA DE ARCHIVOS**

```
23  // Definición de las rutas de los archivos de shaders utilizados
24  static const char* vShader = "shaders/shader.vert";
25  static const char* ShaderRojo = "shaders/shader_rojo.frag";
26  static const char* ShaderVerde = "shaders/shader_verde.frag";
27  static const char* ShaderAzul = "shaders/shader_azul.frag";
28  static const char* ShaderCafe = "shaders/shader_cafe.frag";
29  static const char* ShaderVerdeOscuro = "shaders/shader_verde_oscuro.frag";
```

Este código almacena las rutas de los archivos de shaders, es decir, en la carpeta en la que se encuentran, los cuales serán utilizados en la función `CreateShaders()` para cargarlos en OpenGL.

- **CREACIÓN DE PIRÁMIDES Y CUBOS**

```
37  void CreaPiramide()
38  {
39      unsigned int indices[] = {
40          0,1,2,
41          1,3,2,
42          3,0,2,
43          1,0,3
44
45      };
46      GLfloat vertices[] = {
47          -0.5f, -0.5f,0.0f,  //0
48          0.5f,-0.5f,0.0f,    //1
49          0.0f,0.5f, -0.25f,  //2
50          0.0f,-0.5f,-0.5f,   //3
51
52      };
53      Mesh* obj1 = new Mesh();
54      obj1->CreateMesh(vertices, indices, 12, 12);
55      meshList.push_back(obj1);
56  }
```

En la función `CreaPiramide()` se define un conjunto de índices que conectan los vértices en triángulos, formando así las caras de la pirámide. A continuación, se declara un arreglo con las

coordenadas de los cuatro vértices necesarios para representar la base y la punta de la pirámide. Luego, se crea un objeto Mesh, que se inicializa con los datos de la pirámide mediante `CreateMesh()`, *y se agrega a* `meshList` *para su posterior renderización.*

```
58      //Vértices de un cubo
59      void CrearCubo()
60      {
61          unsigned int cubo_indices[] = {
62              // front
63              0, 1, 2,
64              2, 3, 0,
65              // right
66              1, 5, 6,
67              6, 2, 1,
68              // back
69              7, 6, 5,
70              5, 4, 7,
71              // left
72              4, 0, 3,
73              3, 7, 4,
74              // bottom
75              4, 5, 1,
76              1, 0, 4,
77              // top
78              3, 2, 6,
79              6, 7, 3
80          };
81
82          GLfloat cubo_vertices[] = {
83              // front
84              -0.5f, -0.5f,  0.5f,
85              0.5f, -0.5f,  0.5f,
86              0.5f,  0.5f,  0.5f,
87              -0.5f,  0.5f,  0.5f,
88              // back
89              -0.5f, -0.5f, -0.5f,
90              0.5f, -0.5f, -0.5f,
91              0.5f,  0.5f, -0.5f,
92              -0.5f,  0.5f, -0.5f
93          };
94          Mesh* cubo = new Mesh();
95          cubo->CreateMesh(cubo_vertices, cubo_indices, 24, 36);
96          meshList.push_back(cubo);
97      }
```

En la función CrearCubo() primero, se declara un arreglo de índices que agrupa los vértices en triángulos para formar cada una de las seis caras del cubo. Luego, se define un arreglo con las coordenadas de los ocho vértices del cubo en el espacio 3D. Posteriormente, se crea un objeto de tipo Mesh, al cual se le asignan los vértices e índices mediante la función `CreateMesh()`, *y finalmente, se almacena en la lista* `meshList` *para su posterior uso en la renderización.*

- **CREACION DE SHADERS**

```
100  void CreateShaders() {
101
102      // Crea un shader azul
103      Shader* shaderAzul = new Shader();  // Reserva memoria para un nuevo shader
104      shaderAzul->CreateFromFiles(vShader, ShaderAzul);  // Carga los archivos de shaders (vertex y fragment azul)
105      shaderList.push_back(*shaderAzul);  // Agrega el shader a la lista de shaders [0]
106
107      // Crea un shader rojo
108      Shader* shaderRojo = new Shader();
109      shaderRojo->CreateFromFiles(vShader, ShaderRojo);
110      shaderList.push_back(*shaderRojo); // Agrega el shader a la lista de shaders [1]
111
112      // Crea un shader verde
113      Shader* shaderVerde = new Shader();
114      shaderVerde->CreateFromFiles(vShader, ShaderVerde);
115      shaderList.push_back(*shaderVerde); // Agrega el shader a la lista de shaders [2]
116
117      // Crea un shader café
118      Shader* shaderCafe = new Shader();
119      shaderCafe->CreateFromFiles(vShader, ShaderCafe);
120      shaderList.push_back(*shaderCafe);// Agrega el shader a la lista de shaders [3]
121
122      // Crea un shader verde oscuro
123      Shader* shaderVerdeOscuro = new Shader();
124      shaderVerdeOscuro->CreateFromFiles(vShader, ShaderVerdeOscuro);
125      shaderList.push_back(*shaderVerdeOscuro);// Agrega el shader a la lista de shaders [4]
126
127  }
```

Esta función `CreateShaders()` crea shaders de diferentes colores (azul, rojo, verde, café y verde oscuro) y los almacena en shaderList. azul en cero, rojo en uno, verde en dos, café en tres y verde oscuro en 4. Cada shader se crea con el mismo vertex shader (`vShader`), pero con un fragment shader diferente, que define su color.

- **MAIN**

```
int main()
{
    mainWindow = Window(800, 800);
    mainWindow.Initialise();
    CreateShaders();
    CreaPiramide(); //indice 0 en MeshList
    CrearCubo();//indice 1 en MeshList
    GLuint uniformProjection = 0;
    GLuint uniformModel = 0;

    //Projection: Matriz de Dimensión 4x4 para indicar si vemos en 2D( orthogonal) o en 3D) perspectiva
    //glm::mat4 projection = glm::ortho(-1.0f, 1.0f, -1.0f, 1.0f, 0.1f, 100.0f);
    glm::mat4 projection = glm::perspective(glm::radians(50.0f), mainWindow.getBufferWidth() / mainWindow.getBufferHeight(), 0.1f, 100.0f);


    //Model: Matriz de Dimensión 4x4 en la cual se almacena la multiplicación de las transformaciones geométricas.
    glm::mat4 model(1.0); //fuera del while se usa para inicializar la matriz con una identidad
```

Se declara la función `CreaPiramide()` y `CrearCubo()` para poder dibujarlos posteriormente, el primero con índice 0 en MeshList y el segundo con 0 en esta misma. Además, se declara una matriz en perspectiva para poder ver los objetos en 3D

- **CICLO WHILE**

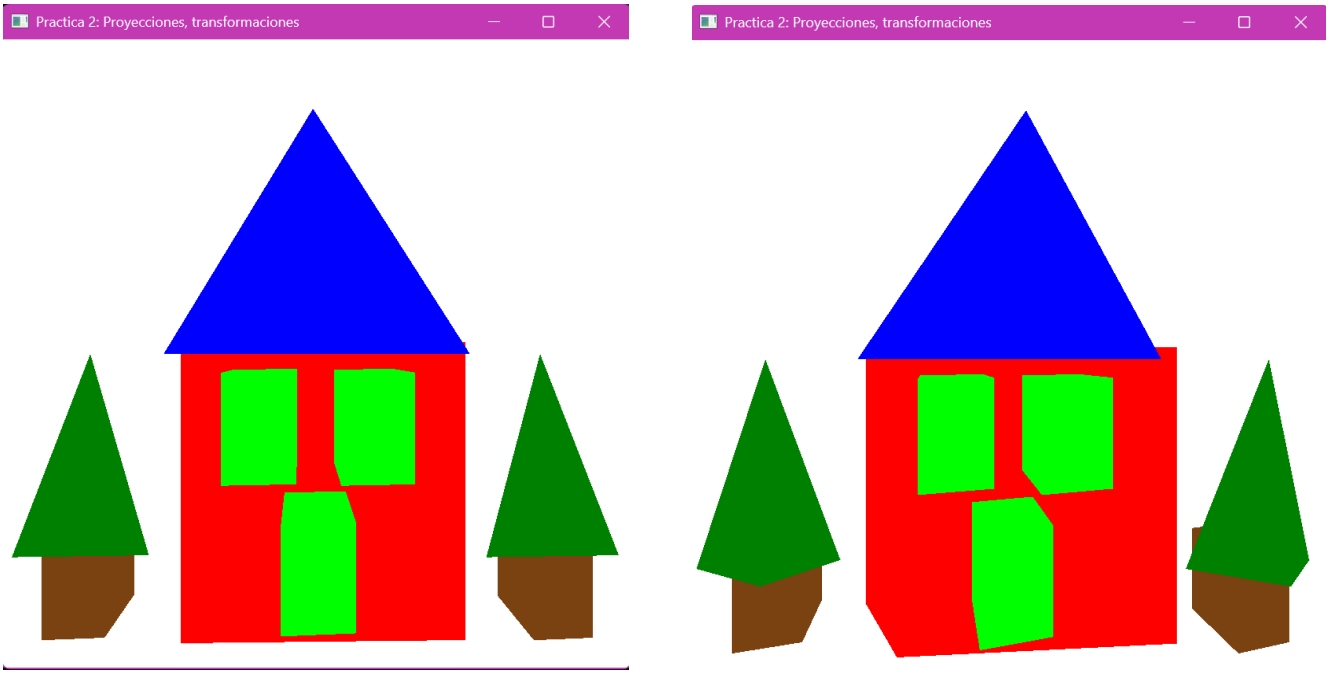
```
while (!mainWindow.getShouldClose())
{
    //Recibir eventos del usuario
    glfwPollEvents();
    //  //Limpiar la ventana
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);//Fondo Blanco
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);//Se agrega limpiar el buffer de profundidad


    angulo += 0.001;
    glm::mat4 model(1.0);

    // Piramide azul,
    shaderList[0].useShader(); // Usa shader azul
    uniformModel = shaderList[0].getModelLocation();
    uniformProjection = shaderList[0].getProjectLocation();

    model = glm::mat4(1.0);
    model = glm::translate(model, glm::vec3(0.0f, 0.4f, -2.2f));//Al centro y arriba
    model = glm::scale(model, glm::vec3(1.0f, 0.8f, 0.0f));
    model = glm::rotate(model, glm::radians(angulo), glm::vec3(0.0f, 0.1f, 0.0f));

    glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));//FALSE ES PARA QUE NO SEA TRANSPUESTA
    glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
    meshList[0]->RenderMesh();

    // Cubo rojo
    shaderList[1].useShader(); // Usa shader rojo
    uniformModel = shaderList[1].getModelLocation();
    uniformProjection = shaderList[1].getProjectLocation();

    model = glm::mat4(1.0);
    model = glm::translate(model, glm::vec3(0.0f, -0.46f, -2.5f));//Al centro y abajo
    model = glm::scale(model, glm::vec3(0.95f, 1.0f, 0.5f));
    model = glm::rotate(model, glm::radians(angulo), glm::vec3(0.0f, 0.2f, 0.0f));

    glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));//FALSE ES PARA QUE NO SEA TRANSPUESTA
    glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
    meshList[1]->RenderMesh();

    // Cubo verde izquierda
    shaderList[2].useShader(); // Usa shader verde
    uniformModel = shaderList[2].getModelLocation();
    uniformProjection = shaderList[2].getProjectLocation();

    model = glm::mat4(1.0);
    model = glm::translate(model, glm::vec3(-0.08f, -0.1f, -1.0f));//ventana izquierda y en la parte superior del cubo rojo
    model = glm::scale(model, glm::vec3(0.10f, 0.15f, 0.2f)); // se hace más pequeña para que quepa en el cubo rojo
    model = glm::rotate(model, glm::radians(angulo), glm::vec3(0.0f, 0.2f, 0.0f));
    glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));//FALSE ES PARA QUE NO SEA TRANSPUESTA y se envían
    glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
    meshList[1]->RenderMesh();

    // Cubo verde derecha
    shaderList[2].useShader(); // Usa shader verde
    uniformModel = shaderList[2].getModelLocation();
    uniformProjection = shaderList[2].getProjectLocation();

    model = glm::mat4(1.0);
    model = glm::translate(model, glm::vec3(0.08f, -0.1f, -1.0f)); // ventana derecha y en la parte superior del cubo rojo
    model = glm::scale(model, glm::vec3(0.10f, 0.15f, 0.2f));// se hace más pequeña para que quepa en el cubo rojo
    model = glm::rotate(model, glm::radians(angulo), glm::vec3(0.0f, 0.2f, 0.0f));
    glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));//FALSE ES PARA QUE NO SEA TRANSPUESTA y se envían
    glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
    meshList[1]->RenderMesh();

    // Cubo verde abajo
    shaderList[2].useShader(); // Usa shader verde
    uniformModel = shaderList[2].getModelLocation();
    uniformProjection = shaderList[2].getProjectLocation();

    model = glm::mat4(1.0);
    model = glm::translate(model, glm::vec3(0.0f, -0.3f, -1.0f));//puerta al centro y abajo del cubo rojo
    model = glm::scale(model, glm::vec3(0.10f, 0.15f, 0.2f));// se hace más pequeña para que quepa en el cubo rojo
    model = glm::rotate(model, glm::radians(angulo), glm::vec3(0.0f, 0.2f, 0.0f));
    glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));//FALSE ES PARA QUE NO SEA TRANSPUESTA y se envían
    glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
    meshList[1]->RenderMesh();
```

```
// Cubo cafe derecha
shaderList[3].useShader(); // Usa shader cafe
uniformModel = shaderList[3].getModelLocation();
uniformProjection = shaderList[3].getProjectLocation();

model = glm::mat4(1.0);
model = glm::translate(model, glm::vec3(0.4f, -0.39f, -1.2f));//Tronco derecho del pino, se mueve fuera del cubo rojo
model = glm::scale(model, glm::vec3(0.10f, 0.15f, 0.2f));// se hace más pequeña para dentro del espacio de ventana
model = glm::rotate(model, glm::radians(angulo), glm::vec3(0.0f, 0.2f, 0.0f));

glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));//FALSE ES PARA QUE NO SEA TRANSPUESTA y se envían a
glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
meshList[1]->RenderMesh();

// Cubo cafe izquierda
shaderList[3].useShader(); // Usa shader cafe
uniformModel = shaderList[3].getModelLocation();
uniformProjection = shaderList[3].getProjectLocation()

model = glm::mat4(1.0);
model = glm::translate(model, glm::vec3(-0.4f, -0.39f, -1.2f));//Tronco izquierdo del pino, se mueve fuera del cubo rojo
model = glm::scale(model, glm::vec3(0.10f, 0.15f, 0.2f));// se hace más pequeña para dentro del espacio de ventana
model = glm::rotate(model, glm::radians(angulo), glm::vec3(0.0f, 0.2f, 0.0f));

glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));//FALSE ES PARA QUE NO SEA TRANSPUESTA y se envían a
glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
meshList[1]->RenderMesh();

// Piramide verde oscuro derecha
shaderList[4].useShader(); // Usa shader verde oscuro
uniformModel = shaderList[4].getModelLocation();
uniformProjection = shaderList[4].getProjectLocation();

model = glm::mat4(1.0);
model = glm::translate(model, glm::vec3(0.35f, -0.15f, -1.0f));// pino derecho, se aleja del cubo rojo y cafe
model = glm::scale(model, glm::vec3(0.2f, 0.3f, 0.2f)); // se hace más pequeña para dentro del espacio de ventana
model = glm::rotate(model, glm::radians(angulo), glm::vec3(0.0f, 0.2f, 0.0f));

glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));//FALSE ES PARA QUE NO SEA TRANSPUESTA y se envían a
glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
meshList[0]->RenderMesh();

// Piramide verde oscuro izquierdo
shaderList[4].useShader(); // Usa shader verde oscuro
uniformModel = shaderList[4].getModelLocation();
uniformProjection = shaderList[4].getProjectLocation();

model = glm::mat4(1.0);
model = glm::translate(model, glm::vec3(-0.35f, -0.15f, -1.0f));// pino izquierdo, se aleja del cubo rojo y cafe
model = glm::scale(model, glm::vec3(0.2f, 0.3f, 0.2f));// se hace más pequeña para dentro del espacio de ventana
model = glm::rotate(model, glm::radians(angulo), glm::vec3(0.0f, 0.2f, 0.0f));

glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));//FALSE ES PARA QUE NO SEA TRANSPUESTA y se
glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
meshList[0]->RenderMesh();

glUseProgram(0);
mainWindow.swapBuffers();

eturn 0;
```

*En esta parte se dibuja una casa con una pirámide azul como techo, un cubo rojo como base y varios cubos verdes representando ventanas y una puerta. Además, incluye troncos de árboles con cubos cafés y copas de árboles con pirámides verde oscuro. Para lograrlo, el **vertex shader** maneja las transformaciones geométricas mediante matrices de modelo y proyección, mientras que el **fragment shader** asigna colores específicos (rojo, verde, azul, café y verde oscuro) a cada objeto sin depender del shader con color clamp. Además, se aplican transformaciones de traslación, escalado y rotación para posicionar los elementos de manera correcta y ver los*

diferentes lados de las figuras, la cual se actualiza en un bucle de renderizado con una cámara en perspectiva.

- **Ejecución del programa (Cambio de color en las letras)**



Se pude observar una casa en 3D, de manera rotada se aprecia mejor

3. Problemas presentados.

En este caso para el ejercicio no tuve problemas, sin embargo, para el ejercicio dos, si me surgieron algunos problemas. Sobre todo, al momento de linkear, ya que en una primera instancia me salía error, pero se debía a en el shader.vert en vec4 le quería pasar un cuarto parámetro y ya los tenía en *color*, después tuvo un problema con los shader frag pero era debido a que en la variable de salida estaba poniendo *vColor* y no color como corresponde.

4. Conclusión

Sin duda estos ejercicios, fueron un reto, ya que de trabajar en figuras en 2D pasamos a figuras en 3D, fue un caso considerar al eje z, ya que como no estamos acostumbrados a trabajar en 3D costo trabajo adaptarse, me costó entender en un principio como era que funcionaban los shaders pero conforme practicaba en el código, lo pude comprender. Considero que los ejercicios fueron de una complejidad aceptable para seguir conociendo un poco sobre Open GL.

5. Bibliografía

- *LearnOpenGL - Shaders*. (s. f.). https://learnopengl.com/Getting-started/Shaders
- *Función RGB - Soporte técnico de Microsoft*. (s. f.). https://support.microsoft.com/es-es/topic/funci%C3%B3n-rgb-aa04db19-fb8a-4f58-9ad6-71a1f5a43e94
- *Las etapas de renderizado*. (s. f.). [Diapositivas]. Universidad de Huelva. https://www.uhu.es/francisco.moreno/gii_rv/docs/Tema_3.pdf