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 3

NOMBRE COMPLETO: Araiza Valdés Diego Antonio

Nº de Cuenta: 423032833

GRUPO DE LABORATORIO: 02

GRUPO DE TEORÍA: 06

SEMESTRE: 2026-1

FECHA DE ENTREGA LÍMITE: 07/09/2025

CALIFICACIÓN: _____

REPORTE DE PRÁCTICA:

1. Ejecución de los ejercicios que se dejaron, comentar cada uno y capturas de pantalla de bloques de código generados y de ejecución del programa.
   1.1. Generar una pirámide rubik (pyraminx) de 9 pirámides por cara. Cada cara de la pyraminx que se vea de un color diferente y que se vean las separaciones entre instancias (las líneas oscuras son las que permiten diferenciar cada pirámide pequeña)

```
moveSpeed = startMoveSpeed*0.1f;
turnSpeed = startTurnSpeed*1.2f;
if (keys[GLFW_KEY_SPACE]) // subir en Y mundial
{
    position += worldUp * velocity;
}

if (keys[GLFW_KEY_LEFT_CONTROL] || keys[GLFW_KEY_RIGHT_CONTROL]) // bajar en Y mundial
{
    position -= worldUp * velocity;
}
```

Primero se agregaron las teclas spaces y ctrl en la función void Camera::keyControl para poder subir y bajar la cámara sin importar la orientación de la misma, es decir que se desplazan sobre el eje Y+ y Y- del mundo. Además se modificó la velocidad de movimiento y la de la cámara para facilitar el desplazamiento y orientación por el mundo.

```
#version 330
layout (location =0) in vec3 pos;
layout (location =1) in vec3 color;
out vec4 vColor;
uniform mat4 model;
uniform mat4 projection;
uniform mat4 view;

void main()
{
    //gl_Position=projection*model*vec4(pos.x,pos.y,pos.z,1.0f);
    gl_Position = projection * view * model * vec4(pos, 1.0);
    vColor=vec4(color,1.0f);
}
```

Posteriormente se cambió el shadercolor.vert para que el gl_Position fuera igual al del shader.vert.

```cpp
// Pirámide triangular regular2
void CrearPiramideTriangular2()
{
    unsigned int indices_piramide_triangular2[] = {
            0,1,2,
            0,2,3,
            0,3,1,
            1,3,2

    };
    GLfloat vertices_piramide_triangular2[] = {
        0.0f,  1.0f, 0.0f,  //0
        0.0f, 0.0f, 0.707107f,  //1
        -0.612372f, 0.0f, -0.353553f,   //2
        0.612372f, 0.0f, -0.353553f //3
    };
    Mesh* obj1 = new Mesh();
    obj1->CreateMesh(vertices_piramide_triangular2, indices_piramide_triangular2, 12, 12);
    meshList.push_back(obj1);

}
```

Posteriormente, guiándonos de la usada en clase se declaró la pirámide triangular, intentado que fuese una pirámide triangular equilátera con una altura de 1.

```cpp
// Pirámide triangular regular3
void CrearPiramideTriangular3()
{
    GLfloat vertices_piramide_triangular3[] = {
        // X            Y           Z          R     G     B
        // Cara 1 (rojo)
        - 0.081650f,  1.033333f,  0.047140f,   1.0f, 0.0f, 0.0f,
        - 0.081650f,  0.033333f,  0.754247f,   1.0f, 0.0f, 0.0f,
        - 0.694022f,  0.033333f, -0.306413f,   1.0f, 0.0f, 0.0f,

        // Cara 2 (verde)
          0.000000f,  1.033333f, -0.094281f,   0.0f, 1.0f, 0.0f,
        - 0.612372f,  0.033333f, -0.447834f,   0.0f, 1.0f, 0.0f,
          0.612372f,  0.033333f, -0.447834f,   0.0f, 1.0f, 0.0f,

        // Cara 3 (azul)
          0.081650f,  1.033333f,  0.047140f,   0.0f, 0.0f, 1.0f,
          0.694022f,  0.033333f, -0.306413f,   0.0f, 0.0f, 1.0f,
          0.081650f,  0.033333f,  0.754247f,   0.0f, 0.0f, 1.0f,

        // Cara 4 (amarillo)
          0.000000f, -0.100000f,  0.707107f,   1.0f, 1.0f, 0.0f,
          0.612372f, -0.100000f, -0.353553f,   1.0f, 1.0f, 0.0f,
        - 0.612372f, -0.100000f, -0.353553f,   1.0f, 1.0f, 0.0f,

    };
    MeshColor* obj1 = new MeshColor();
    obj1->CreateMeshColor(vertices_piramide_triangular3, 72);
    meshColorList.push_back(obj1);

}
```

*Posteriormente, a partir de la pirámide creada anterior mente y de sus normales ((-0.816497, 0.333333, 0.471404), (0.000000, 0.333333, -0.942809), (0.816497, 0.333333, 0.471404), (0.000000, -1.000000, 0.000000), se creó otra cuya separación entre caras fuera de 0.1 de las uniones hacia cada cara, de modo que el NuevoVertice=Vertice+0.1*Normal y se pintó cada cara de un color distinto.*

```
CrearCubo();//índice 0 en MeshList
CrearPiramideTriangular();//índice 1 en MeshList
CrearCilindro(25, 1.0f);//índice 2 en MeshList
CrearCono(25, 2.0f);//índice 3 en MeshList
CrearPiramideCuadrangular();//índice 4 en MeshList
CrearPiramideTriangular2();//índice 5 en MeshList
CrearPiramideTriangular3();//índice 0 en MeshColorList
CreateShaders();
```

Por consiguiente, se agregaron las funciones al main para poder hacer llamadas a las mismas.

```
glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
```

Después se cambio el color del fondo a color blanco.

```
#define GLM_ENABLE_EXPERIMENTAL
#include<gtc\quaternion.hpp>
#include<gtx\quaternion.hpp>
glm::quat rotx = glm::angleAxis(glm::radians(0.0f), glm::vec3(1, 0, 0));
glm::quat roty = glm::angleAxis(glm::radians(0.0f), glm::vec3(0, 1, 0));
glm::quat rotz = glm::angleAxis(glm::radians(00.0f), glm::vec3(0, 0, 1));
glm::quat rot = rotx * roty * rotz;
glm::mat4 localRot = glm::toMat4(rot);
```

Posteriormente, se declararon las librerías de quaterion y se activo el GLM experimental para poder trabajar con él. Además, se declararon e inicializaron las variables rotx, roty, rotz y rot, que representan rotaciones en torno a los ejes X, Y y Z.Y a partir de estas rotaciones se construyó una matriz (localRot), la cual utilizaremos para aplicar las transformaciones de rotación a los objetos

```
// -------------------- Grupo: rotación alrededor del pivote --------------------
glm::vec3 pivot = glm::vec3(0.0f, 0.0f, -4.0f);
glm::mat4 parentModel = glm::mat4(1.0f);

// Construir parentModel para rotar alrededor de 'pivot':
// parentModel = T(pivot) * Rz * Ry * Rx * T(-pivot)
parentModel = glm::translate(parentModel, pivot);
parentModel = glm::rotate(parentModel, glm::radians(mainWindow.getrotaz()), glm::vec3(0.0f, 0.0f, 1.0f));
parentModel = glm::rotate(parentModel, glm::radians(mainWindow.getrotay()), glm::vec3(0.0f, 1.0f, 0.0f));
parentModel = glm::rotate(parentModel, glm::radians(mainWindow.getrotax()), glm::vec3(1.0f, 0.0f, 0.0f));
parentModel = glm::translate(parentModel, -pivot);
```

*Después, para iniciar a dibujar, nos apoyamos de las funciones creadas previamente para la rotación y se construyó una matriz que representa la rotación global del grupo alrededor del punto (0,0,-4). Usando la técnica T(pivot) * R * T(-pivot), forma estándar para rotar alrededor de un punto arbitrario (no el origen).*

```
// -------------------- Pirámide base --------------------
shaderList[0].useShader();
glm::mat4 model = glm::mat4(1.0f);

// conserva la posicion local original del objeto principal
model = glm::translate(model, glm::vec3(0.0f, 0.0f, -4.0f));
model = parentModel * model; // aplicar la rotación/traslación del grupo
glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));
glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
glUniformMatrix4fv(uniformView, 1, GL_FALSE, glm::value_ptr(camera.calculateViewMatrix()));
meshList[5]->RenderMesh();
```

*Después, creamos la pirámide grande con el shader 0 y la trasladamos al punto (0, 0, -4). Además, haciendo uso de model = parentModel * model; aplica la transformación de grupo sobre la transformación local del objeto, es decir, hace que la malla principal rote alrededor del pivote junto al resto. Pasamos sus matrices al shader y finalmente la dibujamos.*

```
// -------------------- Pirámides pequeñas  --------------------
shaderList[1].useShader();
glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
glUniformMatrix4fv(uniformView, 1, GL_FALSE, glm::value_ptr(camera.calculateViewMatrix()));

auto renderChild = [&](glm::vec3 pos, glm::vec3 scaleVec, bool hasLocalRot = false, glm::mat4 localRot = glm::mat4(1.0f)) {
    glm::mat4 model = glm::mat4(1.0f);
    model = glm::translate(model, pos);
    model = glm::scale(model, scaleVec);
    if (hasLocalRot) {
        // si localRot no es identidad, aplicarla
        model = model * localRot;
    }
    glm::mat4 finalModel = parentModel * model; // aplica rotación/traslación del grupo
    glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(finalModel));
    meshColorList[0]->RenderMeshColor();
    };
```

Después, cambiamos de shader para dibujar las pirámides pequeñas con su respectivo color ya declarado y mandamos al shader la matriz de proyección y de vista ya que estas no cambian sin importar la figura. Posteriormente declaramos la función renderChild, que nos permitirá realizar la inicialización y dibujo de los triángulos de manera más rápida y sencilla. Recibe 4 parametros:

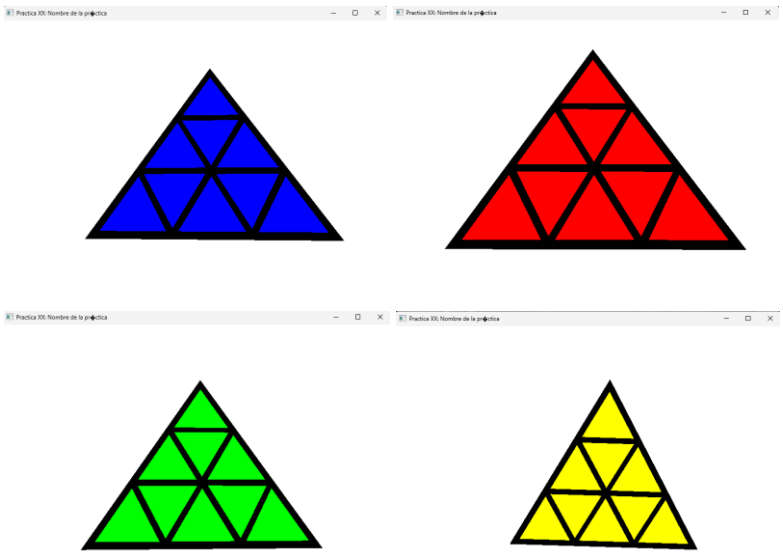
1. *La posición a la que se trasladará la pirámide*
2. *El escalamiento de la pirámide*
3. *Un bool cuyo valor por defecto es false que permite saber si se realizarán rotaciones locales sobre este objeto. Al tener un valor por defecto es opcional incluirla en la función.*
4. *La matriz cuyo valor por defecto es la matriz identidad que permite realizar las rotaciones en caso de que el bool sea verdadero. Al tener un valor por defecto es opcional incluirla en la función. La matriz cuyo valor por defecto*

De esta manera la función trasladará, escalará y si es necesario rotará la pirámide, después aplica la transformación de grupo sobre la transformación local del objeto y manda la matriz del modelo al shader para finalmente dibujarla.

```
// --- Triangulos rectos (Azul) ---
renderChild(glm::vec3(0.0f, 0.02662f, -3.556f), glm::vec3(0.26666f));
renderChild(glm::vec3(0.1925f, 0.02662f, -3.889f), glm::vec3(0.26666f));
renderChild(glm::vec3(0.385f, 0.02662f, -4.222f), glm::vec3(0.26666f));
renderChild(glm::vec3(0.0f, 0.340325f, -3.778f), glm::vec3(0.26666f));
renderChild(glm::vec3(0.1925f, 0.340325f, -4.111f), glm::vec3(0.26666f));
renderChild(glm::vec3(0.0f, 0.654f, -4.0f), glm::vec3(0.26666f));

// --- Triangulos rectos (Rojo) ---
renderChild(glm::vec3(-0.1925f, 0.02662f, -3.889f), glm::vec3(0.26666f));
renderChild(glm::vec3(-0.385f, 0.02662f, -4.222f), glm::vec3(0.26666f));
renderChild(glm::vec3(-0.1925f, 0.340325f, -4.111f), glm::vec3(0.26666f));

// --- Triangulos rectos (Verde) ---
renderChild(glm::vec3(0.0f, 0.02662f, -4.222f), glm::vec3(0.26666f));

// -------------------- Triangulos invertidos --------------------

// Azul (invertido)
rotx = glm::angleAxis(glm::radians(210.0f), glm::vec3(1, 0, 0));
roty = glm::angleAxis(glm::radians(50.1f), glm::vec3(0, 1, 0));
rotz = glm::angleAxis(glm::radians(-58.7f), glm::vec3(0, 0, 1));
rot = rotx * roty * rotz;
localRot = glm::toMat4(rot);

renderChild(glm::vec3(0.0288f, 0.2478f, -3.76f), glm::vec3(0.26666f), true, localRot);
renderChild(glm::vec3(0.22f, 0.2478f, -4.092f), glm::vec3(0.26666f), true, localRot);
renderChild(glm::vec3(0.031f, 0.56f, -3.98f), glm::vec3(0.26666f), true, localRot);

// Rojo (invertido)
rotx = glm::angleAxis(glm::radians(210.0f), glm::vec3(1, 0, 0));
roty = glm::angleAxis(glm::radians(-50.1f), glm::vec3(0, 1, 0));
rotz = glm::angleAxis(glm::radians(58.7f), glm::vec3(0, 0, 1));
rot = rotx * roty * rotz;
localRot = glm::toMat4(rot);

renderChild(glm::vec3(-0.0288f, 0.2478f, -3.76f), glm::vec3(0.26666f), true, localRot);
renderChild(glm::vec3(-0.22f, 0.2478f, -4.092f), glm::vec3(0.26666f), true, localRot);
renderChild(glm::vec3(-0.031f, 0.56f, -3.98f), glm::vec3(0.26666f), true, localRot);

// Verde (invertido)
rotx = glm::angleAxis(glm::radians(10.0f), glm::vec3(1, 0, 0));
roty = glm::angleAxis(glm::radians(-17.0f), glm::vec3(0, 1, 0));
rotz = glm::angleAxis(glm::radians(58.8f), glm::vec3(0, 0, 1));
rot = rotx * roty * rotz;
localRot = glm::toMat4(rot);

renderChild(glm::vec3(0.245f, 0.159f, -4.18f), glm::vec3(0.26666f), true, localRot);
renderChild(glm::vec3(-0.14f, 0.159f, -4.18f), glm::vec3(0.26666f), true, localRot);
renderChild(glm::vec3(0.0535f, 0.475f, -4.066f), glm::vec3(0.26666f), true, localRot);

// Amarillo (invertido con rotacion 180° en Y)
rotx = glm::angleAxis(glm::radians(0.0f), glm::vec3(1, 0, 0));
roty = glm::angleAxis(glm::radians(180.0f), glm::vec3(0, 1, 0));
rotz = glm::angleAxis(glm::radians(0.0f), glm::vec3(0, 0, 1));
rot = rotx * roty * rotz;
localRot = glm::toMat4(rot);

renderChild(glm::vec3(0.0f, 0.0266f, -3.776f), glm::vec3(0.26666f), true, localRot);
renderChild(glm::vec3(-0.1925f, 0.0266f, -4.1085f), glm::vec3(0.26666f), true, localRot);
renderChild(glm::vec3(0.1925f, 0.0266f, -4.1085f), glm::vec3(0.26666f), true, localRot);

// -------------------- Finalizar --------------------
glUseProgram(0);
mainWindow.swapBuffers();
```

Finalmente, hacemos uso de la función renderChild declarada anteriormente para crear y dibujar todas las pirámides pequeñas. Haciendo uso de rotx, roty, rotz, rot y localRot para crear la matriz de rotación que permitirá rotar las pirámides.

1.2. Evidencias:



2. Liste los problemas que tuvo a la hora de hacer estos ejercicios y si los resolvió explicar cómo fue, en caso de error adjuntar captura de pantalla
   R: Surgió un error de no poder escribir el archivo en un momento dado de estar realizando las pruebas y el acomodo de pirámides, como cerrar el archivo no lo resolvió reinicie la computadora y quedó resuelto.

3. Conclusión:
   1. Los ejercicios del reporte: Complejidad, Explicación.
      Respecto a los ejercicios realizados, creo que fueron bastante complicados al nivel de lo visto en clase, ya que se tuvo que investigar bastante, si bien la creación de la pirámide no fue la gran cosa, el hacer que rotaran todas juntas, la biblioteca quaternion, y el estar intentado acomodar las pirámides rotadas si fue bastante laborioso.

   2. Comentarios generales: Faltó explicar a detalle, ir más lento en alguna explicación, otros comentarios y sugerencias para mejorar desarrollo de la práctica.
      Me hubiera gustado conocer la biblioteca quaternion y la forma de hacer que todas las figuras roten sobre un mismo eje en clase, hubiera facilitado mucho la realización de la práctica.

3. Conclusión

   Personalmente creo que se cumplieron los objetivos de la práctica. Con el ejercicio aprendí a crear objetos a partir de primitivas geométricas en un espacio tridimensional y trabajar con ellos, además, aprendí a utilizar una cámara virtual. Con este ejercicio aprendí a trabajar con objetos en un espacio tridimensional, esto incluye crearlos, trasladarlos, escalarlos y rotarlos, mencionando que se aprendió a declarar objetos cuyas sus caras fueran de un color distinto. También aprendí a declarar teclas especiales con las cuales desplazarme por el espacio ya sea con relación a la cámara o del mundo mismo, así también agarré práctica para desplazarme sobre el espacio. Aprendí a rotar objetos sobre un mismo eje, como si estos fueran un mismo objeto o bloque, haciendo uso de un pivote y aprendiendo las diferencias entre las trasformaciones locales y en grupo. Finalmente, aprendí la forma de declarar una función que me permita realizar la creación y dibujo de las figuras y objetos, ahorrándome muchas línea de código.

Bibliografía en formato APA

OpenGL Tutorial. (n.d.). *Tutorial 17: Quaternions*. https://www.opengl-tutorial.org/intermediate-tutorials/tutorial-17-quaternions

MJB. (2018, febrero 21). *Quaternions*. Oregon State University. https://web.engr.oregonstate.edu/~mjb/vulkan/Handouts/quaternions.1pp.pdf

Stack Overflow. (2018, diciembre 25). *How to change pivot in OpenGL?* https://stackoverflow.com/questions/53755401/how-to-change-pivot-in-opengl