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: CARBAJAL REYES IRVIN JAIR

Nº de Cuenta: 422042084

GRUPO DE LABORATORIO: 11

GRUPO DE TEORÍA: 04

SEMESTRE 2025-2

FECHA DE ENTREGA LÍMITE: 26 DE FEBRERO DE 2025

CALIFICACIÓN: _____

Actividades

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

Para esta actividad, dentro de la función *CrearLetrasyFiguras* iremos creando primero el arreglo que contiene los vértices de cada una de las letras iniciando con su posición en *x,y, z* seguido del color RGB, esto para cada uno de los vértices que forman cada triángulo, de este modo empezamos creando el arreglo con los vértices de la letra J.

```
116     GLfloat vertices_j[] = {
117         -0.8f, 0.4f,0.0f,0.0f,  0.0f,   1.0f,
118         -0.8f, 0.3f, 0.0f, 0.0f,    0.0f,   1.0f,
119         -0.6f, 0.3f, 0.0f, 0.0f,    0.0f,   1.0f,
120         -0.8f,0.4f, 0.0f, 0.0f, 0.0f,   1.0f,
121         -0.6f, 0.4f,0.0f,0.0f,  0.0f,   1.0f,
122         -0.6f,0.3f, 0.0f, 0.0f, 0.0f,   1.0f,
123         -0.6f,0.4f, 0.0f,0.0f,  0.0f,   1.0f,
124         -0.4f,0.4f, 0.0f,0.0f,  0.0f,   1.0f,
125         -0.4f,-0.2f, 0.0f,0.0f, 0.0f,   1.0f,
126         -0.6f,0.4f, 0.0f,0.0f,  0.0f,   1.0f,
127         -0.6f,-0.2f, 0.0f,0.0f, 0.0f,   1.0f,
```

Se generaron un total de 33 vértices para dicha letra, luego la agregamos a la estructura *MeshColorList* ocupando el lugar uno.

```
151         MeshColor* j = new MeshColor();
152         j->CreateMeshColor(vertices_j, 198);
153         meshColorList.push_back(j);
```

Esto sucede de igual forma para la letra C generando 18 vértices y ocupando el lugar dos de la estructura.

```
155     GLfloat vertices_c[] = {
156         -0.1f,0.4f, 0.0f, 1.0f, 0.5f, 0.0f,
157         0.3f,0.4f, 0.0f, 1.0f, 0.5f, 0.0f,
158         0.3f,0.2f, 0.0f,  1.0f, 0.5f, 0.0f,
159         -0.2f, 0.4f,0.0f, 1.0f, 0.5f, 0.0f,
160         -0.1f, 0.4f, 0.0f, 1.0f, 0.5f, 0.0f,
161         -0.1f, -0.4f, 0.0f, 1.0f, 0.5f, 0.0f,
162         -0.2f,0.4f, 0.0f, 1.0f, 0.5f, 0.0f,
```

```
175         MeshColor* c = new MeshColor();
176         c->CreateMeshColor(vertices_c, 108);
177         meshColorList.push_back(c);
```

Y también para la letra R generando 36 vértices y ocupando la posición tres de la estructura.

```
179     GLfloat vertices_r[] = {
180         0.5f, 0.4f,0.0f, 0.5f, 0.0f, 0.5f,
181         0.5f, 0.0f, 0.0f, 0.5f, 0.0f, 0.5f,
182         0.7f, 0.4f, 0.0f, 0.5f, 0.0f, 0.5f,
183         0.7f,0.4f, 0.0f, 0.5f, 0.0f, 0.5f,
184         0.7f, 0.0f,0.0f, 0.5f, 0.0f, 0.5f,
185         0.5f,0.0f, 0.0f,  0.5f, 0.0f, 0.5f,
186         0.7f,0.4f, 0.0f, 0.5f, 0.0f, 0.5f,
```

```
217         MeshColor* r = new MeshColor();
218         r->CreateMeshColor(vertices_r, 216);
219         meshColorList.push_back(r);
```

Luego, es necesario tener proyección ortogonal 2D dentro de la función *main*.

```
345     glm::mat4 projection = glm::ortho(-1.0f, 1.0f, -1.0f, 1.0f, 0.1f, 100.0f);
```

Para las letras, usamos el shader creado en la estructura *shaderList* en la posición uno.

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

Luego iremos agregando cada una de las letras

```
366         model = glm::mat4(1.0);
367         model = glm::translate(model, glm::vec3(0.0f, 0.0f, -4.0f));
368         glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));//FALSE ES
369         glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
370         meshColorList[1]->RenderMeshColor();
```

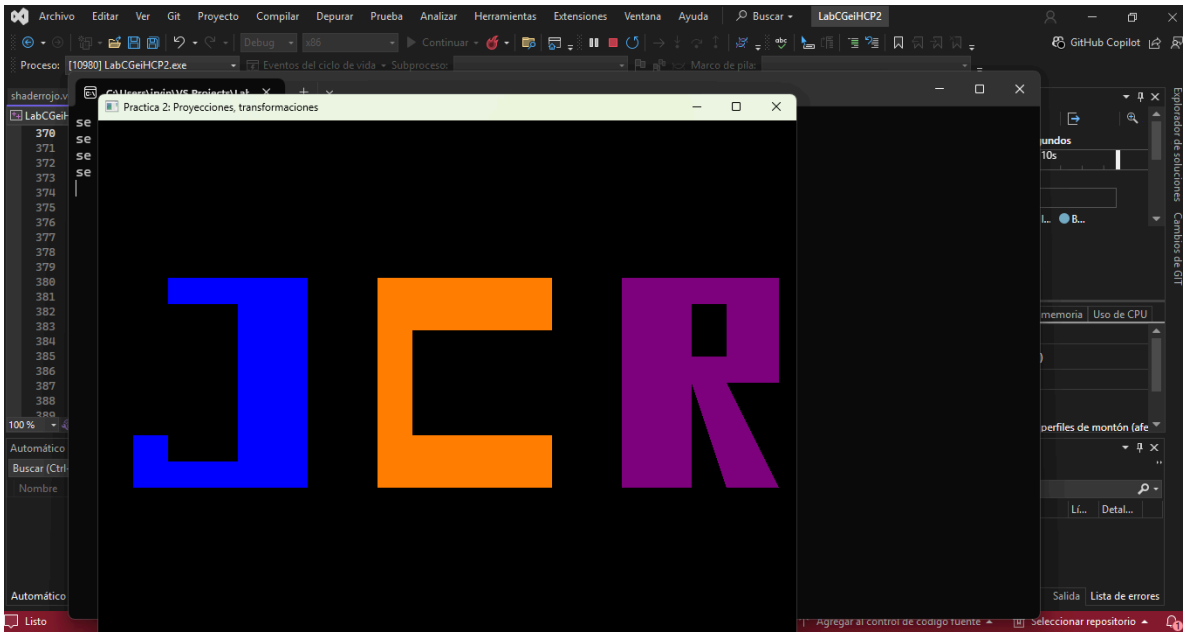
No es necesario trasladar la letra ya que sus posiciones se definieron en cada uno de sus vértices al generarlos para que se puedan visualizar las tres letras.

```
372         //Agregamos letra C
373         model = glm::mat4(1.0);
374         model = glm::translate(model, glm::vec3(0.0f, 0.0f, -4.0f));
375         glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));//FALSE ES
376         glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
377         meshColorList[2]->RenderMeshColor();
```

```
379         //Agregamos letra R
380         model = glm::mat4(1.0);
381         model = glm::translate(model, glm::vec3(0.0f, 0.0f, -4.0f));
382         glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));//FALSE ES
383         glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
384         meshColorList[3]->RenderMeshColor();
```

Notamos como el valor de la posición *meshColorList* va cambiando según la letra que se quiera agregar.

La ejecución queda de la siguiente manera:



2. Generar el dibujo de la casa de la clase, pero en lugar de instanciar triangulos y cuadrados será instanciando piramides 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

Para esta actividad, se crearon cinco archivos .vert para los shaders, donde únicamente se modificó el color en cada uno de estos.

```
#version 330
layout (location =0) in vec3 pos;
out vec4 vColor;
uniform mat4 model;
uniform mat4 projection;
void main()
{
    gl_Position=projection*model*vec4(pos,1.0f);
    vColor=vec4(1.0f,0.0f,0.0f,1.0f);
    //vColor=vec4(clamp(pos,0.0f,1.0f),1.0f);
}
```

Luego, cada uno de estos archivos se declararon como shaders en el archivo *main*.

```
22      //Vertex Shader
23      static const char* rShader = "shaders/shaderrojo.vert";
24      static const char* aShader = "shaders/shader.vert";
25      static const char* vShader = "shaders/shaderverde.vert";
26      static const char* voShader = "shaders/shaderverobs.vert";
27      static const char* cShader = "shaders/shadercafe.vert";
28      static const char* fShader = "shaders/shader.frag";
29      static const char* vShaderColor = "shaders/shadercolor.vert";
30      static const char* fShaderColor = "shaders/shadercolor.frag";
```

Luego, en la función *CrearShaders* se crean cada uno de estos y se almacenan dentro de la estructura *shaderList*.

```
305 void CreateShaders()
306 {
307
308     Shader *shader1 = new Shader(); //shader azul para usar índices: objetos: cubo y  pirámide
309     shader1->CreateFromFiles(aShader, fShader);
310     shaderList.push_back(*shader1);
311
312     Shader *shader2 = new Shader();//shader para usar color como parte del VAO: letras
313     shader2->CreateFromFiles(vShaderColor, fShaderColor);
314     shaderList.push_back(*shader2);
315
316     Shader* shader3 = new Shader(); //shader rojo para usar índices: objetos: cubo y  pirámide
317     shader3->CreateFromFiles(rShader, fShader);
318     shaderList.push_back(*shader3);
```

Dentro de la función *main* usamos la proyección perspectiva 3D

```
346     glm::mat4 projection = glm::perspective(glm::radians(60.0f) ,mainWindow.getBufferWidth() / mainWindow.getBu
```

Ahora, antes de colocar un cubo o pirámide de shader distinto es importante usar el índice correspondiente al shader del color que se quiere usar.

```
464             shaderList[2].useShader();
465             uniformModel = shaderList[2].getModelLocation();
466             uniformProjection = shaderList[2].getProjectLocation();
467             angulo += 0.01;
```

En este caso, al usar el índice dos en *shaderList* hacemos referencia al shader rojo.

Luego, colocamos el cubo o la pirámide tomando en cuenta el índice de la estructura *MeshList*.

```
470             model = glm::mat4(1.0);
471             model = glm::translate(model, glm::vec3(0.0f, -0.3f, -3.5f));
472             //model = glm::rotate(model, glm::radians(angulo), glm::vec3(0.0f, 1.0f, 0.0f));
473             glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));//FALSE ES PA
474             glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
475             meshList[1]->RenderMesh();
```

Donde *translate* nos permitirá desplazar la figura en cualquier eje, *rotate* (que se encuentra comentado) nos permite rotar la figura, y el índice uno en *meshList* hace referencia al cubo.

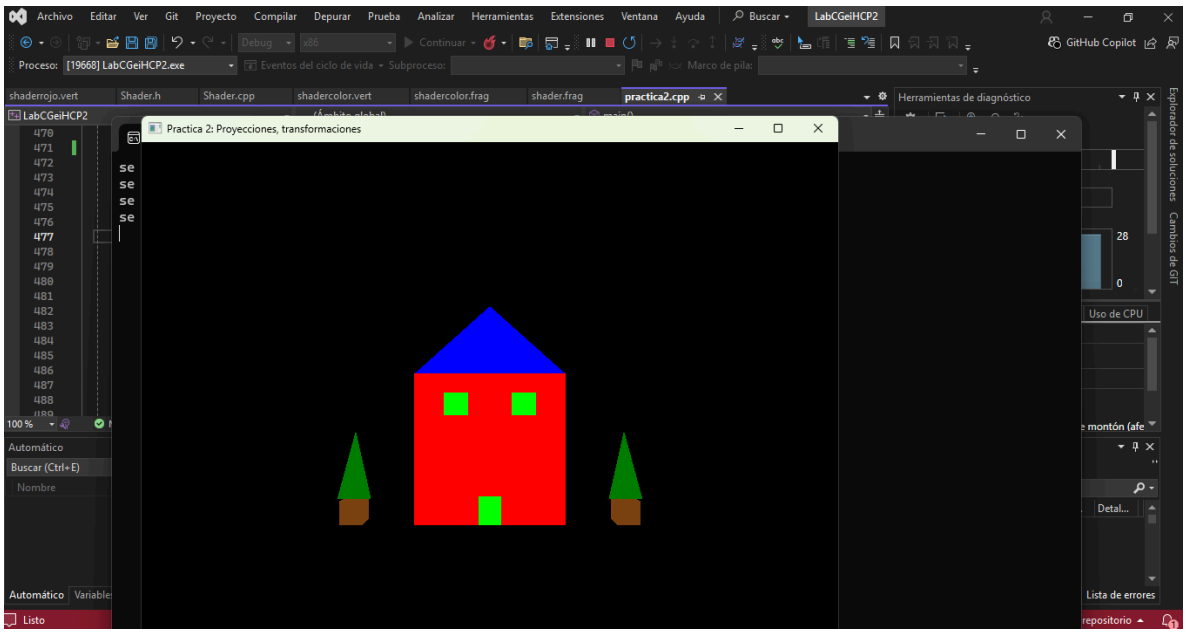
Para el techo que es una pirámide azul, primero usamos el shader azul con el índice cero.

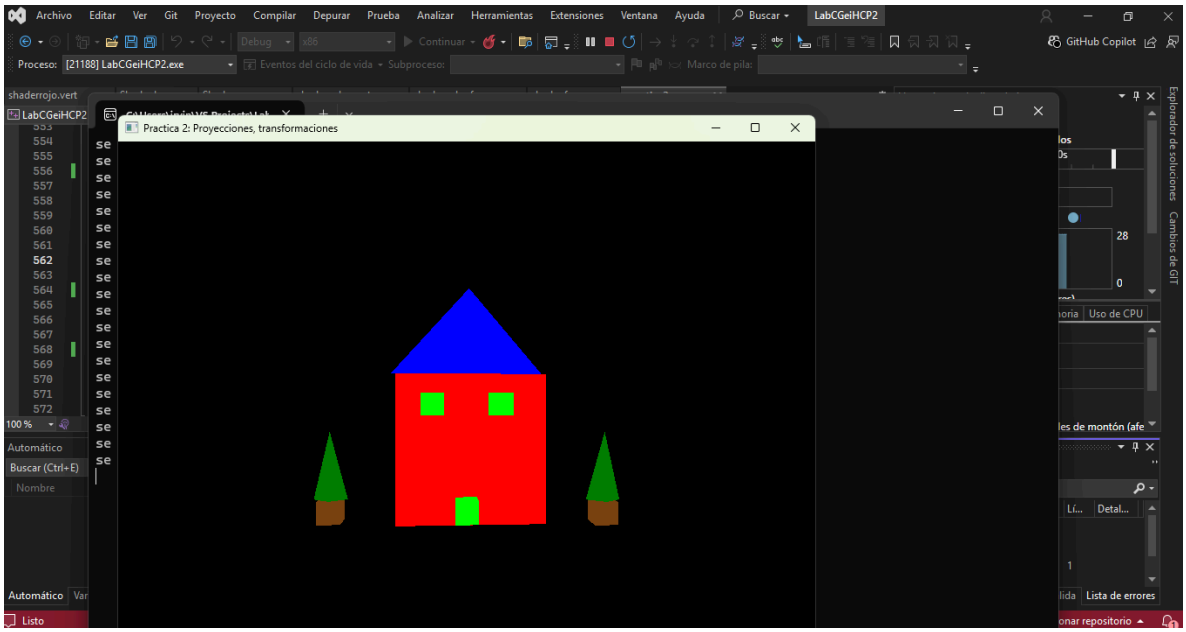
```
479             shaderList[0].useShader();
480             uniformModel = shaderList[0].getModelLocation();
481             uniformProjection = shaderList[0].getProjectLocation();
482             angulo += 0.01;
```

Y usamos el índice cero en *MeshList* para implementar la pirámide, además de hacer uso de *scale* para modificar su tamaño.

```
484             model = glm::mat4(1.0);
485             model = glm::translate(model, glm::vec3(0.0f, 0.45f, -3.0f));
486             model = glm::scale(model, glm::vec3(1.0f, 0.5f, 1.0f));
487             //model = glm::rotate(model, glm::radians(angulo), glm::vec3(0.0f, 1.0f, 0.0f));
488             glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));//FALSE ES P
489             glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
490             meshList[0]->RenderMesh();
```

Con esta misma estructura colocamos el resto de los cubos y pirámides, quedando la ejecución de la siguiente manera:





Problemas presentados

Durante la práctica se presentó un problema cuando empezamos a compilar el programa con los cubos y las pirámides, ya que mostraba el error que no encontraba el archivo *Debug/LabCGeiHCP2.exe*, por lo que no mostraba la ejecución. La solución que se presentó fue eliminar la carpeta *Debug* y volver a compilar el programa. Esto daba solución al momento pero después de realizar algunos cambios al código se presentaba nuevamente y se elimina la carpeta para volver a compilar sin errores. Al final se lograron ejecutar las actividades, pero el error sigue surgiendo en ocasiones.

Al generar los vértices para las letras no se tenía muy en claro cómo almacenar en el arreglo los vértices y sus colores, ya que se pensaba colocar todos los vértices de las tres letras dentro del mismo arreglo, lo que provocó errores en la ejecución, sin embargo, después se hicieron arreglos independientes para cada letra y cada una se añadió a la estructura correspondiente.

Durante la elaboración hubo algunas dificultades en la implementación de los archivos *.vert* de los *shaders* y poder implementarlos en los cubos y pirámides, pero leyendo las líneas de código correspondientes se logró entender cómo es que se hacía uso de los archivos y como se accedían a las estructuras de datos tanto para guardar los *shader* como para utilizarlos en las figuras.

Conclusiones

Durante el desarrollo de cada una de las actividades de esta práctica, la complejidad fue adecuada para los conocimientos que se buscaban adquirir, como lo fue el manejo de VAO y VBO, shader entre otros elementos que nos permiten visualizar figuras tanto en 2D como en 3D. La explicación dada en el laboratorio fue adecuada, sin embargo al ser muchos conceptos puede que no se adquieran todos al cien por ciento, por lo que el manual de prácticas fue de mucha ayuda para recordar lo visto en clase y resolver los ejercicios propuestos.

Es importante mencionar que las actividades sirvieron para entender las proyecciones y transformaciones tanto en 2D como en 3D, revisando conceptos como rotación, traslación y escala, para manipular las figuras a lo que nosotros buscamos proyectar.

Bibliografía

- Tutorial 3 : Matrices. (n.d.). http://www.opengl-tutorial.org/es/beginners-tutorials/tutorial-3-matrices/