



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 N° 02**

**NOMBRE COMPLETO:** Casillo Martinez Diego Leonardo

**N.º de Cuenta:** 319041538

**GRUPO DE LABORATORIO:** 11

**GRUPO DE TEORÍA:** 06

**SEMESTRE 2024-2**

**FECHA DE ENTREGA LÍMITE:** 28/08/2024

**CALIFICACIÓN:** \_\_\_\_\_

## REPORTE DE PRÁCTICA:

### 1.- Desarrollo

**Ejercicio 1:** Dibujar las iniciales de nuestros nombres, cada letra de un color diferente.

Para este ejercicio se plantea aprovechar los vectores de las iniciales de nuestro nombre que se hicieron en la práctica anterior. Sin embargo, en esta práctica se planteó que cada inicial tenga un color diferente.

Lo que se hizo para esta práctica fue en primera instancia agregar colores RGB para cada letra, dentro del arreglo `vertices_Letras`, esto se hizo de la siguiente forma:

```
GLfloat vertices_Letras[] = {  
    //X      Y      Z      R  
    // Triángulo 1  
    -0.9f,  0.4f, 0.0f,  0.8f, 0.6f, 1.0f,  
    -0.8f,  0.4f, 0.0f,  0.8f, 0.6f, 1.0f,  
    -0.9f,  0.35f, 0.0f,  0.8f, 0.6f, 1.0f,  
  
    // Triángulo 2  
    -0.8f,  0.4f, 0.0f,  0.8f, 0.6f, 1.0f,  
    -0.8f,  0.35f, 0.0f,  0.8f, 0.6f, 1.0f,  
    -0.9f,  0.35f, 0.0f,  0.8f, 0.6f, 1.0f,  
  
    // Triángulo 12  
    -0.9f, -0.35f, 0.0f,  0.8f, 0.6f, 1.0f,  
    -0.8f, -0.35f, 0.0f,  0.8f, 0.6f, 1.0f,  
    -0.9f, -0.4f, 0.0f,  0.8f, 0.6f, 1.0f,  
  
    // Triángulo 11  
    -0.8f, -0.35f, 0.0f,  0.8f, 0.6f, 1.0f,  
    -0.8f, -0.4f, 0.0f,  0.8f, 0.6f, 1.0f,  
    -0.9f, -0.4f, 0.0f,  0.8f, 0.6f, 1.0f,  
  
    // Palo izquierdo de la d  
    // Triángulo 14  
    -0.9f,  0.35f, 0.0f,  0.8f, 0.6f, 1.0f,  
    -0.85f, 0.35f, 0.0f,  0.8f, 0.6f, 1.0f,  
    -0.85f, -0.35f, 0.0f,  0.8f, 0.6f, 1.0f,  
  
    // todo eso fue para la D ;-;  
    -0.5f,  0.37f, 0.0f,  1.0f, 0.75f, 0.8f,  
    -0.5f, -0.4f, 0.0f,  1.0f, 0.75f, 0.8f,  
    -0.45f, 0.37f, 0.0f,  1.0f, 0.75f, 0.8f,  
  
    -0.5f, -0.4f, 0.0f,  1.0f, 0.75f, 0.8f,  
    -0.45f, 0.37f, 0.0f,  1.0f, 0.75f, 0.8f,  
    -0.45f, -0.4f, 0.0f,  1.0f, 0.75f, 0.8f,  
  
    -0.5f, -0.4f, 0.0f,  1.0f, 0.75f, 0.8f,  
    -0.5f, -0.35f, 0.0f,  1.0f, 0.75f, 0.8f,  
    -0.25f, -0.4f, 0.0f,  1.0f, 0.75f, 0.8f,  
  
    -0.5f, -0.35f, 0.0f,  1.0f, 0.75f, 0.8f,  
    -0.25f, -0.4f, 0.0f,  1.0f, 0.75f, 0.8f,  
    -0.25f, -0.35f, 0.0f,  1.0f, 0.75f, 0.8f,  
  
    //eso fue la L
```

Nota: Se debe considerar que para este programa se crearon aproximadamente 600 vértices, por lo cual solo se muestran algunos en la captura para evitar saturar el reporte con código repetitivo.

Al implementar esto, simplemente lo llamamos en el bucle de la ventana que se encuentra en nuestro main. Es importante tener en cuenta que, para este ejercicio, utilizamos el segundo set de shaders con índice 1 en `ShaderList`, dado que `ShaderList[0]` se emplea para objetos que no tienen información de color en los vértices. En estos casos, los colores se aplican uniformemente en todo el objeto a través de un archivo `fShader`. Por otro lado, `ShaderList[1]`, que se maneja con `shaderList[1].useShader`, se usa para objetos donde los colores están definidos en los vértices del objeto.

A continuación, se muestra la implementación de nuestro código:

```
// ESTO ES PARA DIBUJAR LAS LETRAS PARA EL EJERCICIO 2 SE DEBE COMENTAR
//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
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
glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
meshColorList[0] -> RenderMeshColor(); //----- ojo que aqui se imprime mi nombre
```

Gracias a esto obtenemos la siguiente ejecución:








**Ejercicio 2:** generar el dibujo de la casa de la clase, pero en lugar de instanciar triángulos y cuadrados será instanciando pirámides y cubos.

Para este ejercicio, se siguió una secuencia de pasos para agregar colores a nuestras pirámides y cubos. Es importante tener en cuenta que el programa utiliza una proyección ortogonal. Además, se deben considerar los siguientes puntos:

1. El fondo de la ventana se cambió a color blanco.
2. Para evitar que las letras del ejercicio anterior interfieran, se comentaron en este punto. Si deseamos volver a visualizar solo las letras, debemos comentar la sección del código relacionada con las pirámides y cubos, y descomentar la parte correspondiente a las letras.

Con lo mencionado, podemos comenzar a detallar la secuencia de pasos que se realizaron:

1.- Para este ejercicio, lo primero que se hizo fue crear nuevos archivos .frag, los cuales se agregan a una lista de shaders en el código para cada color específico. Por lo tanto, realizamos 5 archivos .frag nuevos, cada uno correspondiente a un color específico: verde claro, verde, café, rojo y azul. Estos archivos se encuentran en nuestra carpeta de shaders y tienen la siguiente apariencia:

 verdeClaro	27/08/2024 10:43 p. m.	Archivo FRAG	1 KB
 azul	27/08/2024 09:24 p. m.	Archivo FRAG	1 KB
 cafe	27/08/2024 09:24 p. m.	Archivo FRAG	1 KB
 verde	27/08/2024 09:24 p. m.	Archivo FRAG	1 KB
 rojo	27/08/2024 09:24 p. m.	Archivo FRAG	1 KB

El contenido de esto si bien es repetido tiene la misma estructura:

```
#version 330
in vec4 vColor;
out vec4 color;
void main()
{
    color= vec4(0.0, 1.0, 0.0,1.0);
}
```

La única diferencia es que se modifica la estructura de color RGB para cada color en particular.

2.- Una vez que estos archivos están en la carpeta de shaders, se declaran y se agregan en la función createShaders de la siguiente manera:

Para la declaración:

```
static const char* vShader = "shaders/shader.vert";
static const char* fShader = "shaders/shader.frag";
static const char* vShaderColor = "shaders/shadercolor.vert";
static const char* fShaderColor = "shaders/shadercolor.frag";
//shaders nuevos se crearían acá
static const char* fShaderRojo = "shaders/rojo.frag";
static const char* fShaderVerde = "shaders/verde.frag";
static const char* fShaderAzul = "shaders/azul.frag";
static const char* fShaderCafe = "shaders/cafe.frag";
static const char* fShaderVerdeClaro = "shaders/verdeClaro.frag";
```

Al integrarlo en la función createShaders:

```

void CreateShaders()

    Shader *shader1 = new Shader(); //shader para usar índices: objetos: cubo y pirámide
    shader1->CreateFromFiles(vShader, fShader);
    shaderList.push_back(*shader1);

    Shader *shader2 = new Shader(); //shader para usar color como parte del VAO: letras
    shader2->CreateFromFiles(vShaderColor, fShaderColor);
    shaderList.push_back(*shader2);
    // Shader para color rojo
    Shader* shaderRojo = new Shader();
    shaderRojo->CreateFromFiles(vShader, fShaderRojo);
    shaderList.push_back(*shaderRojo);

    // Shader para color verde
    Shader* shaderVerde = new Shader();
    shaderVerde->CreateFromFiles(vShader, fShaderVerde);
    shaderList.push_back(*shaderVerde);

    // Shader para color azul
    Shader* shaderAzul = new Shader();
    shaderAzul->CreateFromFiles(vShader, fShaderAzul);
    shaderList.push_back(*shaderAzul);

```

3.- Una vez realizados los pasos anteriores, podemos comenzar a instanciar nuestras pirámides y cubos y asignarles el color deseado de la siguiente manera:

```

// Usar el shader rojo
// notaciones para el shader list
// 2 es color rojo, 3 es verde , 4 es azul , 5 cafe , 6 verde clarito
// este es el cubo que forma la casa
shaderList[2].useShader();
uniformModel = shaderList[0].getModelLocation();
uniformProjection = shaderList[0].getProjectLocation();
// Aplicar transformaciones al cubo
model = glm::mat4(1.0f);
// escalamos el cubo a la mital pls
model = glm::scale(model, glm::vec3(0.8f, 0.9f, 0.9f)); // recortamos un poco el eje x
model = glm::translate(model, glm::vec3(0.0f, -0.6f, -4.0f));
glUniformMatrix4fv(uniformModel, 1, GL_FALSE, glm::value_ptr(model));
glUniformMatrix4fv(uniformProjection, 1, GL_FALSE, glm::value_ptr(projection));
meshList[1]->RenderMesh(); // recordemos que 1 s para el cubo y cero es para la piramide

```

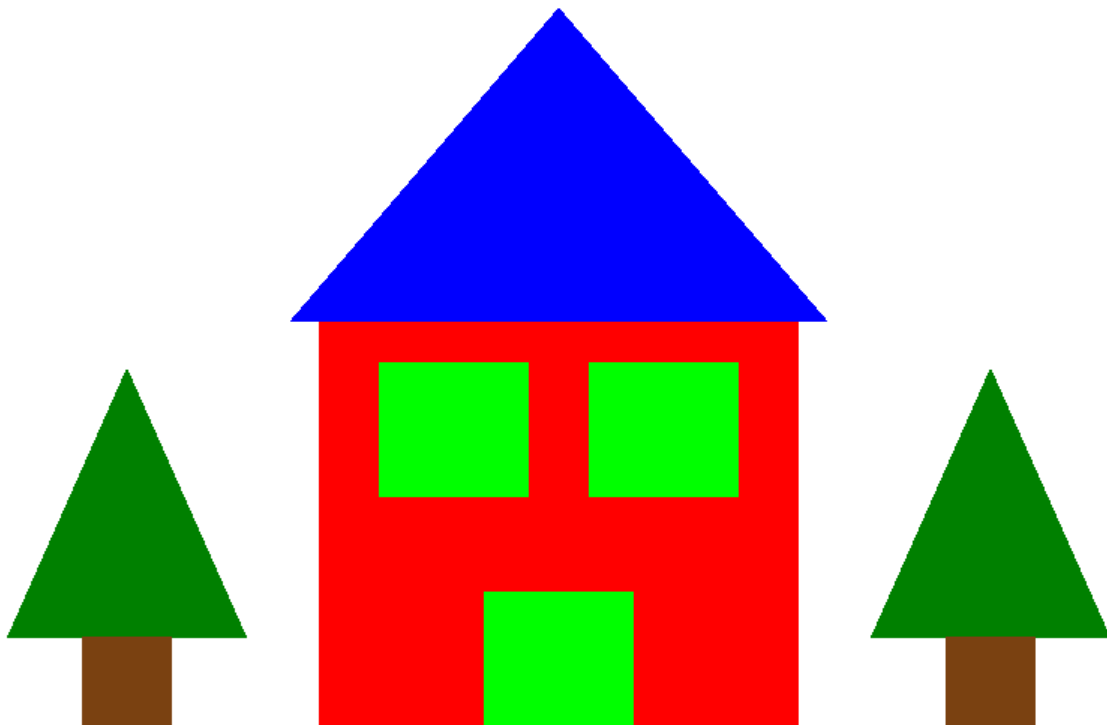
Nota: Esta captura solo imprime el cuadro rojo de la casa. La impresión de las demás partes no se muestra como captura ya que tienen la misma estructura y lo único que cambia es el índice del arreglo.

En donde Para definir el color que deseamos, usamos el comando `shaderList[n].useShader()`, que activa o selecciona el shader en la posición n de la lista `shaderList`. Debemos considerar el siguiente orden de índices para los colores:

2: rojo, 3: verde, 4: azul, 5: café, 6: verde claro.

Para determinar qué objeto vamos a renderizar, utilizamos `meshList[n]->RenderMesh();`, donde `n` puede ser 0 para las pirámides y 1 para los cubos.

Una vez realizado todo esto obtenemos el siguiente resultado al ejecutar el programa:

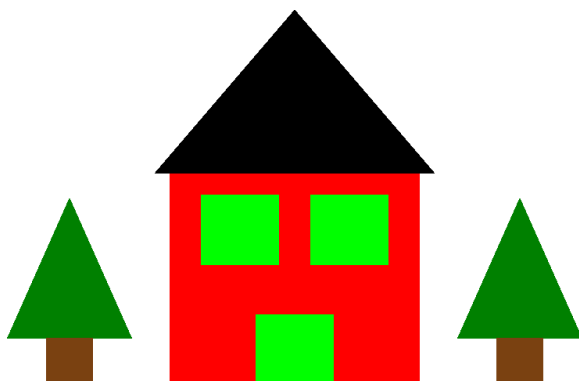


## 2.- problemas que surgieron:

Para esta práctica surgió un problema al realizar el ejercicio 2, el cual fue un problema sintactico en los archivos `.frag` ya que no estaba bien escrito el vector de índices, al tener este error el programa si bien se ejecutaba te decía que no podía cargar el contenido de un archivo, de la siguiente forma:

```
EL error al compilar el shader 35632 es:
```

Y al momento de dibujar el ejercicio se mostraba de la siguiente forma:



Si bien este no es un problema tan significativo, se tardó un tiempo en encontrar el error, fuera de este problema no surgieron problemas significativos y solo fue cuestión de ser paciente y entender la lógica de este código.

### **3.- Conclusión:**

En esta práctica se presentaron algunas dificultades al generar los archivos .frag y al trasladar los objetos (cubo y pirámide). Sin embargo, estas dificultades ayudaron a comprender mejor el funcionamiento de GLFW, así como las proyecciones, traslaciones, escalado de vértices y shaders. Aunque los ejercicios no eran extremadamente complejos, fue esencial entender cómo operan los shaders y cómo se proyectan y trasladan las figuras para que la imagen final cumpliera con las expectativas.

En general, la práctica fue entretenida y desafiante. Disfruté tanto de la explicación del programa como de la teoría. No obstante, mi única observación es que, debido al tiempo limitado de la clase y a la complejidad de la teoría abordada, algunas explicaciones se ofrecieron de manera demasiado rápida.

### **4.- Bibliografía:**

*Tutorial 3 : Matrices.* (s. f.). <https://www.opengl-tutorial.org/es/beginners-tutorials/tutorial-3-matrices/>