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 01

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: 22-02-2026

CALIFICACIÓN: _____

Ejercicio 1: Cambio de color de fondo aleatorio cada 2 segundos

En este ejercicio se modificó el ciclo principal del programa para que el color de fondo cambiara automáticamente cada 2 segundos utilizando valores aleatorios en el rango RGB.

Primero se agregó la inicialización del generador de números aleatorios con:

```
// Para que al ejecutar varias veces NO salga la misma secuencia de colores
srand((unsigned int)time(NULL));
```

Esto es importante porque sin esa línea el programa genera siempre la misma secuencia de números al ejecutarse, lo que provocaría que el orden de colores fuera el mismo cada vez. Al usar time(NULL) como semilla, la secuencia cambia en cada ejecución.

Después, dentro del while principal, se utilizó glfwGetTime() para obtener el tiempo transcurrido desde que inició el programa. Se almacenó el momento en el que ocurrió el último cambio de color y se comparó con el tiempo actual. Cuando la diferencia es mayor o igual a 2 segundos, se generan nuevos valores aleatorios para R, G y B normalizados entre 0 y 1:

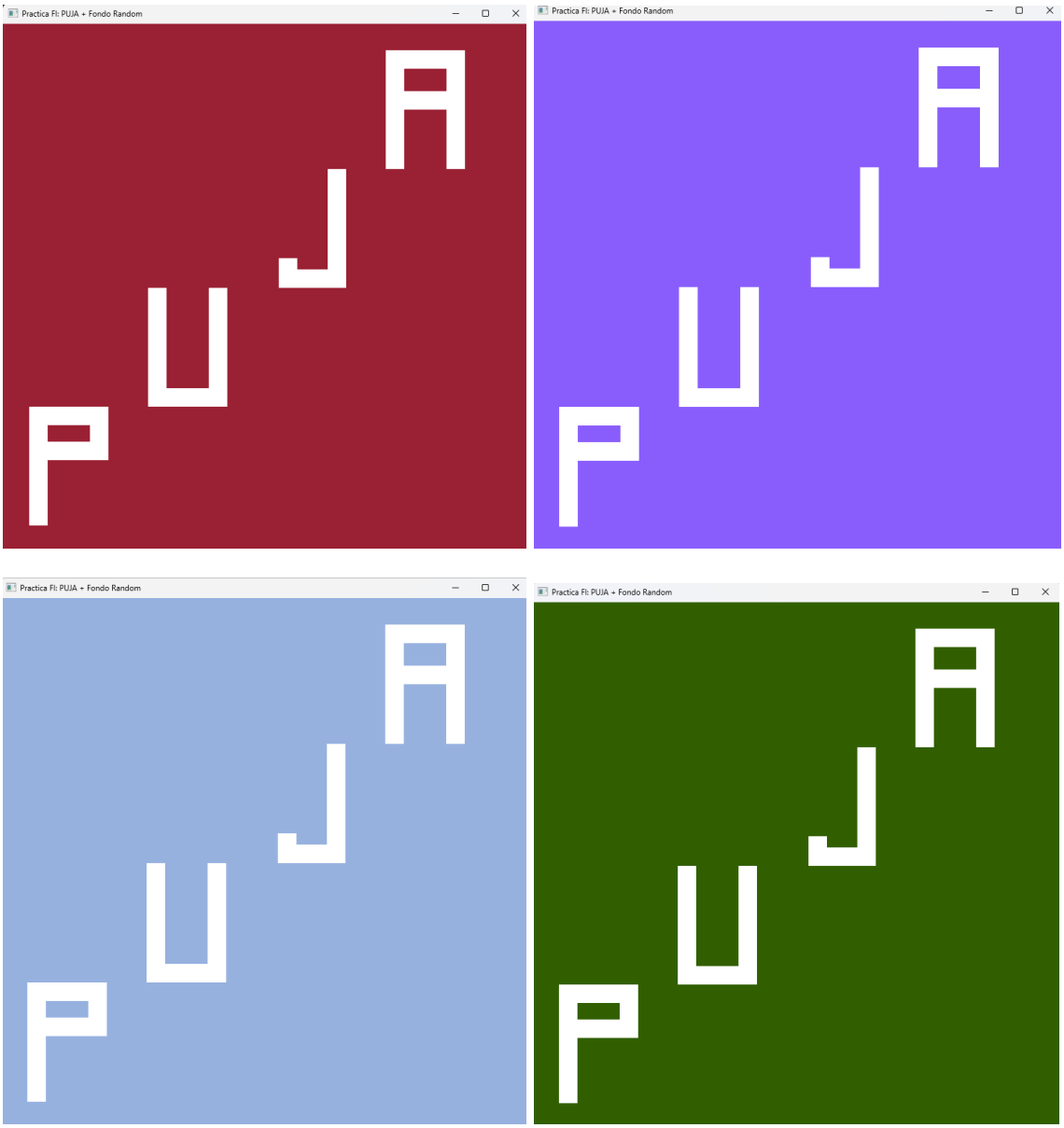
```
// 1) Fondo random RGB cada 2 segundos
double tiempo = glfwGetTime();
if (tiempo - lastChange >= 2.0)
{
    lastChange = tiempo;

    bgR = (float)rand() / (float)RAND_MAX;
    bgG = (float)rand() / (float)RAND_MAX;
    bgB = (float)rand() / (float)RAND_MAX;
}
```

Estos valores se asignan mediante glClearColor, que define el color con el que se limpia la ventana antes de dibujar:

```
glClearColor(bgR, bgG, bgB, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
```

El resultado es que el fondo cambia cada 2 segundos con combinaciones RGB diferentes y no repetidas entre ejecuciones.



En esta secuencia de imágenes se puede observar el comportamiento del programa correspondiente al primer ejercicio, donde el color de fondo cambia automáticamente durante la ejecución. Se muestran diferentes combinaciones de colores generadas dentro del rango RGB, evidenciando que el cambio ocurre de forma periódica y no sigue un patrón fijo. Esto confirma que el fondo se actualiza dinámicamente mientras el programa se encuentra en ejecución.

```
int main()
{
    // Para que al ejecutar varias veces NO salga la misma secuencia de colores
    srand((unsigned int)time(NULL));

    if (!glfwInit()) { printf("Falló inicializar GLFW"); return 1; }

    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);

    GLFWwindow* mainWindow = glfwCreateWindow(WIDTH, HEIGHT, "Practica FI: PUJA + Fondo Random", NULL, NULL);
    if (!mainWindow) { glfwTerminate(); return 1; }

    int BufferWidth, BufferHeight;
    glfwGetFramebufferSize(mainWindow, &BufferWidth, &BufferHeight);
    glfwMakeContextCurrent(mainWindow);

    glewExperimental = GL_TRUE;
    if (glewInit() != GLEW_OK) { glfwDestroyWindow(mainWindow); glfwTerminate(); return 1; }

    glViewport(0, 0, BufferWidth, BufferHeight);

    CrearTriangulo();
    CompileShaders();

    // Color de fondo y control de tiempo (cada 2 segundos)
    float bgR = 0.0f, bgG = 0.0f, bgB = 0.0f;
    double lastChange = -2.0; // para que cambie inmediatamente al iniciar
    while (!glfwWindowShouldClose(mainWindow))
    {
        glfwPollEvents();

        // 1) Fondo random RGB cada 2 segundos
        double tiempo = glfwGetTime();
        if (tiempo - lastChange >= 2.0)
        {
            lastChange = tiempo;

            bgR = (float)rand() / (float)RAND_MAX;
            bgG = (float)rand() / (float)RAND_MAX;
            bgB = (float)rand() / (float)RAND_MAX;
        }

        glClearColor(bgR, bgG, bgB, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);

        // 2) Dibujar PUJA con triángulos
        glUseProgram(shader);
        glBindVertexArray(VAO);

        glDrawArrays(GL_TRIANGLES, 0, gVertexCount);

        glBindVertexArray(0);
        glUseProgram(0);

        glfwSwapBuffers(mainWindow);
    }

    return 0;
}
```

En este bloque del main se integró la parte dinámica del ejercicio, permitiendo que el programa no solo dibuje las letras, sino que también tenga un comportamiento visual que cambia con el tiempo. Se añadió el control necesario dentro del ciclo principal para que el fondo de la ventana se actualice automáticamente mientras el programa está en ejecución. De esta forma, el renderizado no es estático, sino que muestra una variación continua en pantalla sin afectar el dibujo de las letras.

Ejercicio 2: Construcción de las letras iniciales de mi nombre con triángulos

En este ejercicio se construyeron las letras P, U, J y A utilizando exclusivamente triángulos, ya que OpenGL trabaja principalmente con primitivas triangulares.

Para mantener el mismo formato que venimos trabajando en las clases, se utilizó un solo VAO y un solo VBO. En lugar de declarar directamente todos los vértices manualmente, se implementó una función auxiliar llamada AddRect, la cual genera un rectángulo utilizando dos triángulos (6 vértices). De esta manera, cada parte de la letra se construyó a partir de rectángulos.

Cada letra fue formada por varios rectángulos que representan sus trazos verticales y horizontales. Por ejemplo:

- La letra **P** se construyó con una barra vertical izquierda y tres barras horizontales/verticales superiores.
- La letra **U** se construyó con dos barras verticales y una barra inferior.
- La letra **J** se construyó con una barra vertical derecha y un gancho inferior.
- La letra **A** se construyó con dos barras verticales y dos horizontales.

Las letras fueron colocadas en forma diagonal utilizando un desplazamiento progresivo en las coordenadas X y Y. Se respetó el rango de coordenadas normalizadas de OpenGL (-1 a 1), asegurando que todas las letras estuvieran dentro del área visible.

Todos los vértices se almacenaron en un único arreglo y posteriormente se enviaron a la GPU mediante:

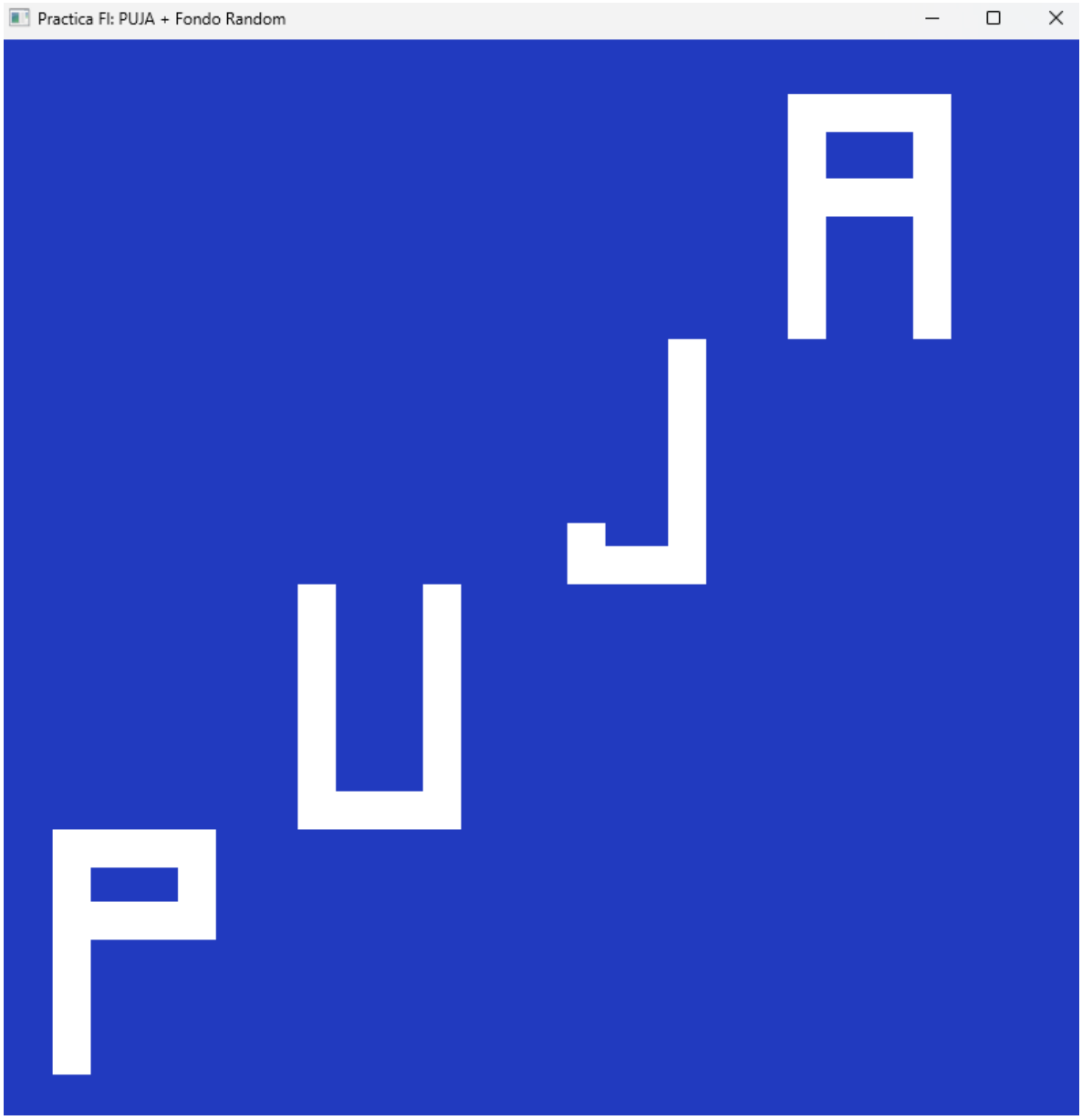
```
glBufferData(GL_ARRAY_BUFFER, idx * sizeof(GLfloat), vertices, GL_STATIC_DRAW);
```

Finalmente, se dibujaron utilizando:

```
glDrawArrays(GL_TRIANGLES, 0, gVertexCount);
```

Todas las letras se muestran con el mismo color porque el Fragment Shader define un único color fijo para todos los fragmentos:

```
color = vec4(1.0f,1.0f,1.0f,1.0f);
```



En la imagen se observa el resultado final del segundo ejercicio, donde las letras PUJA aparecen formadas por triángulos y acomodadas en forma diagonal dentro del área visible de la ventana. Se puede apreciar que todas mantienen el mismo color, mientras que el fondo cambia dinámicamente.

La distribución diagonal permite distinguir claramente cada letra y confirma que las coordenadas fueron ajustadas correctamente dentro del rango normalizado de OpenGL.

```
//====================== GEOMETRIA (PUJA) ======================

static void AddRect(GLfloat* data, int& idx,
    float x1, float y1, float x2, float y2, float z = 0.0f)
{
    // Triangulo 1
    data[idx++] = x1; data[idx++] = y1; data[idx++] = z;
    data[idx++] = x2; data[idx++] = y1; data[idx++] = z;
    data[idx++] = x2; data[idx++] = y2; data[idx++] = z;

    // Triangulo 2
    data[idx++] = x1; data[idx++] = y1; data[idx++] = z;
    data[idx++] = x2; data[idx++] = y2; data[idx++] = z;
    data[idx++] = x1; data[idx++] = y2; data[idx++] = z;
}

void CrearTriangulo() // (mismo nombre para apegarse al estilo)
{
    // Rango NDC: x,y en [-1,1]
    float startX = -0.90f;
    float startY = -0.90f;
    float step = 0.45f;  // separacion diagonal
    float w = 0.30f;  // ancho de cada letra
    float h = 0.45f;  // alto de cada letra
    float t = 0.07f;  // grosor de trazo

    // Max aproximado: 14 rectangulos * 6 vertices * 3 floats = 252 floats
    // Por seguridad, se da más espacio
    GLfloat vertices[600];
    int idx = 0;
// ---- LETRA P ----
{
    float x = startX + step * 0;
    float y = startY + step * 0;

    // barra izquierda
    AddRect(vertices, idx, x, y, x + t, y + h);

    // barra superior
    AddRect(vertices, idx, x, y + h - t, x + w, y + h);

    // barra media
    AddRect(vertices, idx, x, y + h * 0.55f, x + w * 0.85f, y + h * 0.55f + t);

    // barra derecha (solo arriba, para formar el "hueco" de la P)
    AddRect(vertices, idx, x + w - t, y + h * 0.55f, x + w, y + h);
}

// ---- LETRA U ----
{
    float x = startX + step * 1;
    float y = startY + step * 1;

    // barra izquierda
    AddRect(vertices, idx, x, y, x + t, y + h);

    // barra derecha
    AddRect(vertices, idx, x + w - t, y, x + w, y + h);

    // barra inferior
    AddRect(vertices, idx, x, y, x + w, y + t);
}
```

```
// ---- LETRA J ----
{
    float x = startX + step * 2;
    float y = startY + step * 2;

    // barra vertical derecha (casi toda la altura)
    AddRect(vertices, idx, x + w - t, y + t, x + w, y + h);

    // base inferior (horizontal hacia la izquierda)
    AddRect(vertices, idx, x + w * 0.15f, y, x + w, y + t);

    // pequeño gancho vertical izquierdo
    AddRect(vertices, idx, x + w * 0.15f, y, x + w * 0.15f + t, y + h * 0.25f);
}
// ---- LETRA A ----
{
    float x = startX + step * 3;
    float y = startY + step * 3;

    // barra izquierda
    AddRect(vertices, idx, x, y, x + t, y + h);

    // barra derecha
    AddRect(vertices, idx, x + w - t, y, x + w, y + h);

    // barra superior
    AddRect(vertices, idx, x, y + h - t, x + w, y + h);

    // barra media
    AddRect(vertices, idx, x, y + h * 0.5f, x + w, y + h * 0.5f + t);
}

// Guardamos cantidad de vertices
gVertexCount = idx / 3; // 3 floats por vertice
// VAO/VBO
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);

glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);

glBufferData(GL_ARRAY_BUFFER, idx * sizeof(GLfloat), vertices, GL_STATIC_DRAW);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(0);

glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

En este bloque se organizó la geometría necesaria para dibujar las letras dentro de un solo arreglo de vértices. Se definieron posiciones iniciales y parámetros como ancho, alto y separación diagonal para mantener proporción y orden en la distribución de las letras. A partir de esos valores se calcularon las coordenadas específicas de cada trazo.

También se determinó la cantidad total de vértices generados y posteriormente se configuró el VAO y el VBO para enviar toda la información a la GPU. De esta manera, la geometría quedó lista para ser renderizada en una sola llamada de dibujo dentro del ciclo principal.

2.- Problemas encontrados y cómo se resolvieron

Uno de los primeros problemas que tuve fue que el fondo no cambiaba de manera realmente aleatoria. Al principio sí cambiaba cada 2 segundos, pero al cerrar y volver a ejecutar el programa los colores salían en el mismo orden. Después de revisar, me di cuenta de que no estaba inicializando la semilla del generador de números aleatorios. Lo resolví agregando srand(time(NULL)); al inicio del main, y con eso cada ejecución empezó a mostrar una secuencia diferente.

Otro problema fue que las letras no se veían completas dentro de la ventana. Algunas partes quedaban fuera del rango visible, especialmente cuando ajusté la separación diagonal. Me di cuenta de que estaba usando valores muy grandes para el desplazamiento, así que tuve que reducir el step y ajustar las coordenadas hasta que todas las letras quedaran dentro del rango (-1,1).

También tuve un detalle con la letra J, ya que al inicio no se distinguía bien su forma y parecía una L invertida. Lo solucioné modificando los rectángulos que forman el gancho inferior hasta que visualmente se entendiera mejor como una J.

En una ocasión el programa dejó de dibujar completamente después de hacer cambios en los vértices. Revisando el código noté que el conteo de vértices (gVertexCount) no coincidía con los datos enviados al VBO. Corrigiendo ese valor volvió a renderizar correctamente.

En general, los problemas no fueron tanto de sintaxis sino más bien de lógica y de acomodar correctamente las coordenadas dentro del espacio normalizado de OpenGL.

3.Conclusión

a) Los ejercicios del reporte: Complejidad y explicación

Los ejercicios tuvieron una complejidad intermedia. Aunque el código base ya estaba estructurado, fue necesario entender bien cómo funcionan las coordenadas en el rango normalizado de OpenGL y cómo construir figuras a partir de triángulos. El ejercicio del fondo aleatorio fue más sencillo en cuanto a lógica, pero ayudó a comprender mejor cómo funciona el ciclo principal del programa y el uso del tiempo dentro del renderizado.

El ejercicio de las letras fue un poco más detallado, ya que implicó organizar correctamente los vértices, calcular proporciones y asegurarse de que todas las figuras se mantuvieran dentro del área visible. En general, no fue complicado a nivel matemático, pero sí requirió atención en los detalles.

b) Comentarios generales

En general, la práctica fue clara en cuanto a lo que se pedía realizar. Sin embargo, la parte de acomodar coordenadas dentro del espacio (-1 a 1) puede resultar confusa al inicio, especialmente cuando se construyen figuras más complejas a partir de varias partes.

Sería útil que en futuras prácticas se mostrara un ejemplo más detallado de cómo planear las coordenadas antes de comenzar a construir las figuras, o explicar con más calma cómo visualizar mentalmente el espacio normalizado. También ayudaría ir un poco más lento en la parte donde se combinan varios rectángulos o triángulos para formar letras u otras figuras.

c) Conclusión

Esta práctica ayudó a reforzar varios conceptos importantes de OpenGL que antes solo se veían de forma más básica. No solo se trató de dibujar figuras, sino de entender cómo se construyen realmente a partir de triángulos y cómo se organizan los datos para enviarlos a la GPU.

También permitió comprender mejor la estructura de un programa gráfico completo: inicialización, creación de buffers, shaders y ciclo de renderizado. Al final, ver que el fondo cambia dinámicamente mientras las letras permanecen estables ayuda a notar cómo cada parte del código tiene una función específica dentro del proceso de renderizado.

En general, fue una práctica útil para ganar más confianza modificando el código base y entendiendo cómo pequeños cambios en valores y lógica pueden alterar completamente el resultado visual en pantalla.

Bibliografía

Stevewhims. (n.d.). Función glVertexPointer (Gl.h) - Win32 apps. Microsoft Learn. https://learn.microsoft.com/es-es/windows/win32/opengl/glvertexpointer

Guía Definitiva de VBOs, VAOs y EBOs en OpenGL. (n.d.). https://www.q2bstudio.com/nuestro-blog/21151/guia-definitiva-de-vbos-vaos-yebos-en-opengl?scriptscookies=1

Rogrp. (2020, June 3). OpenGL – básico (C++). https://regor.home.blog/2020/05/21/opengl-basico-c/

Shader Basics – Vertex Shader | GPU Shader Tutorial. (n.d.). https://shadertutorial.dev/basics/vertex-shader/#2-what-is-a-vertex-shader

Fragment Shader - OpenGL Wiki. (n.d.). https://wikis.khronos.org/opengl/Fragment_Shader

GlDrawArrays - OpenGL 4 - docs.gl. (n.d.). https://docs.gl/gl4/glDrawArrays