Presentación Día 2: Implementación de OpenGL - Enlazando Teoría y Práctica

Paso 1A: Implementación Clásica de OpenGL - Sin Librerías Auxiliares

Parte A

Este paso muestra cómo implementar OpenGL usando funciones básicas sin frameworks auxiliares como **GLFW** o **GLEW**, lo que te ayudará a entender el funcionamiento esencial de OpenGL antes de usar implementaciones modernas.

Parte A: Dibujar un Triángulo con OpenGL Clásico

1. Crear un Proyecto en Visual Studio Code

1. **Descargar e Instalar Visual Studio Code**:
    - Ve a la página oficial de Visual Studio Code: https://code.visualstudio.com/
    - Descarga e instala la versión más reciente.
2. **Instalar las Extensiones Necesarias**:
    - Abre Visual Studio Code y busca las extensiones necesarias para desarrollar en C++:
        - **C/C++** de Microsoft: Para soporte básico de C/C++.
        - **Code Runner** (opcional): Para ejecutar código rápidamente.
3. **Configurar un Proyecto en C++**:
    - Crea una nueva carpeta y abre Visual Studio Code.
    - Crea un nuevo archivo `main1.cpp` y configura un archivo `tasks.json` para compilar el proyecto con g++. Puedes usar MinGW en Windows para acceder al compilador GCC.

Instalar FreeGLUT (una versión moderna de GLUT)

- **Windows**:

- Utiliza **MSYS2** para simplificar la instalación:
    1. Descarga e instala MSYS2 desde https://www.msys2.org/.
    2. Abre la terminal de MSYS2 (MSYS2 MinGW).

Instala FreeGLUT ejecutando el comando:

```
pacman -S mingw-w64-x86_64-freeglut
```

3. Esto instalará FreeGLUT y configurará automáticamente las rutas necesarias para MinGW.

Linux (Ubuntu/Debian):

```
sudo apt-get update
sudo apt-get install freeglut3-dev
```

MacOS:

```
brew install freeglut
```

Abrir c_cpp_properties.json

- Presiona **Ctrl + Shift + P** (o **Cmd + Shift + P** en Mac) para abrir la paleta de comandos.
- Busca "C/C++: Edit Configurations (UI)" y selecciona esa opción.
- Alternativamente, navega a `.vscode/c_cpp_properties.json`.

Añadir el Include Path Correcto

Modifica `c_cpp_properties.json` para incluir la ruta de la librería `glut.h`:

```
{
    "configurations": [
        {
```

```
            "name": "MinGW",
            "includePath": [
                "${workspaceFolder}/**",
                "C:/msys64/mingw64/include" // Solo añade esto
            ],
            "defines": [],
            "compilerPath": "C:/Program Files/mingw64/bin/g++.exe",
            "cStandard": "c11",
            "cppStandard": "c++17",
            "intelliSenseMode": "windows-gcc-x64"
        }
    ],
    "version": 4
}
```

Verificar que la Ruta es Correcta

- Asegúrate de que `glut.h` esté en `C:/msys64/mingw64/include/GL`.
- Si está en otra carpeta, cambia la ruta en `includePath`.

Guardar y Reiniciar VS Code

- Guarda los cambios en `c_cpp_properties.json`.
- Cierra y abre VS Code nuevamente para aplicar los cambios.

Añadir el directorio MSYS2 Bin al PATH del sistema

- Pulsa `Win + S` y busca «Variables de entorno».
- Haz clic en «Editar las variables de entorno del sistema».
- En la ventana Propiedades del sistema, haz clic en «Variables de entorno».
- En «Variables del sistema», desplázate hacia abajo y busca «Ruta», selecciónala y haz clic en «Editar».
- Haga clic en «Nuevo» y añada la ruta a la carpeta `bin`, por ejemplo:

```
C:\msys64\mingw64\bin
```

        - Haga clic en Aceptar en todas las ventanas para guardar los cambios.
    - Después de añadir esto, intente ejecutar su programa de nuevo.
- **Reinicie su Terminal/VS Code**
    - Tras copiar la DLL o cambiar el PATH del sistema, reinicie su terminal o VS Code para asegurarse de que los cambios surten efecto.

2. Implementar un Triángulo con OpenGL Clásico

Incluir las Librerías Estándar:

```
#include <GL/glut.h>
```

Inicializar OpenGL y Configurar la Ventana:

```
void display() {
    glClear(GL_COLOR_BUFFER_BIT); // Limpia la pantalla
    glBegin(GL_TRIANGLES);        // Inicia el dibujo de un triángulo
        glVertex2f(-0.5f, -0.5f); // Vértice 1
        glVertex2f(0.5f, -0.5f);  // Vértice 2
        glVertex2f(0.0f, 0.5f);   // Vértice 3
    glEnd();                      // Finaliza el dibujo
    glFlush();                    // Asegura que todo se renderice
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);               // Inicializa GLUT
    glutCreateWindow("Triángulo Simple"); // Crea la ventana
    glutDisplayFunc(display);            // Establece la función de display
    glutMainLoop();                      // Bucle principal
    return 0;
}
```

Código completo:

```
#include <GL/glut.h>  // Incluye la biblioteca GLUT para gestionar ventanas y dibujar gráficos en OpenGL.

// Función de callback que se encarga de renderizar la escena.
void display() {
    // Limpia el buffer de color para preparar la pantalla para dibujar.
    glClear(GL_COLOR_BUFFER_BIT);

    // Inicia el dibujo de un triángulo.
    glBegin(GL_TRIANGLES);
        // Define el primer vértice del triángulo en la posición (-0.5, -0.5).
        glVertex2f(-0.5f, -0.5f);
        // Define el segundo vértice del triángulo en la posición (0.5, -0.5).
        glVertex2f(0.5f, -0.5f);
        // Define el tercer vértice del triángulo en la posición (0.0, 0.5).
        glVertex2f(0.0f, 0.5f);
    // Finaliza la secuencia de dibujo del triángulo.
    glEnd();

    // Asegura que todas las órdenes de dibujo se ejecuten y se envíen a la pantalla.
    glFlush();
}

// Función principal del programa.
int main(int argc, char** argv) {
    // Inicializa GLUT y procesa cualquier argumento que pueda ser necesario.
    glutInit(&argc, argv);

    // Configura el modo de display para utilizar el buffer simple de color (GLUT_SINGLE) y el espacio de color RGB.
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);

    // Establece el tamaño inicial de la ventana en 640x480 píxeles.
    glutInitWindowSize(640, 480);
```

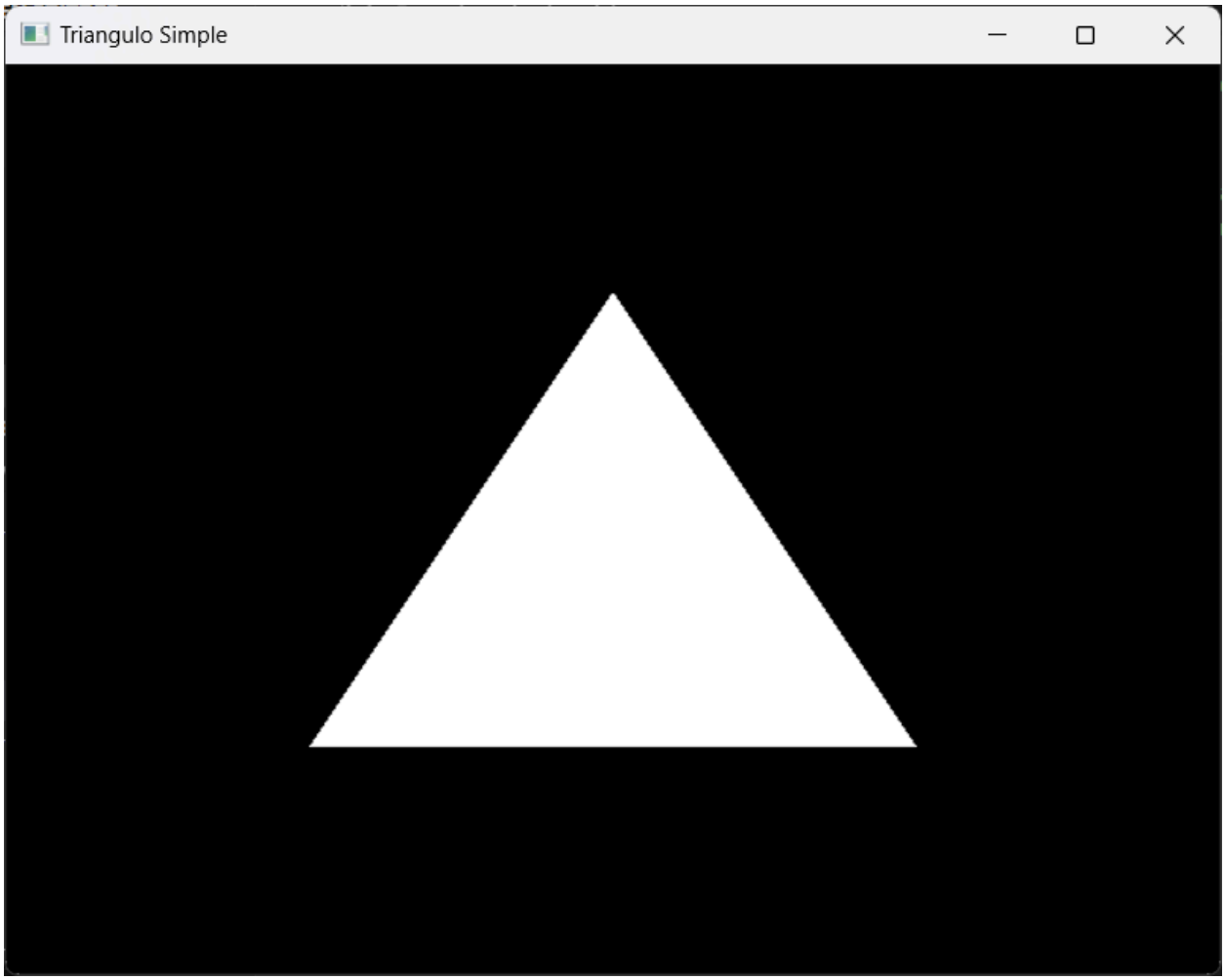
```
    // Crea una ventana con el título "Triángulo Simple".
    glutCreateWindow("Triangulo Simple");

    // Establece la función de display (renderizado), que se llama cada vez que
es necesario redibujar.
    glutDisplayFunc(display);

    // Entra en el bucle principal de GLUT. Este bucle gestiona eventos como el
redibujado y la interacción del usuario.
    glutMainLoop();

    return 0;  // Retorna 0 para indicar una ejecución exitosa.
}
```

Compilación y Ejecución:

```
g++ main1.cpp -o OpenGLTriangle1 -I"C:/msys64/mingw64/include"
-L"C:/msys64/mingw64/lib" -lfreeglut -lopengl32 -lglu32
./OpenGLTriangle1
```

- `g++ main1.cpp`: Compila el archivo `main1.cpp` en C++.
- `-o OpenGLTriangle1`: Genera un ejecutable llamado `OpenGLTriangle1`.
- `-I"C:/msys64/mingw64/include"`: Añade una ruta para buscar archivos de encabezado.
- `-L"C:/msys64/mingw64/lib"`: Añade una ruta para buscar bibliotecas.
- `-lfreeglut`: Enlaza con la biblioteca FreeGLUT para manejo de ventanas.
- `-lopengl32`: Enlaza con la biblioteca OpenGL para funciones gráficas.
- `-lglu32`: Enlaza con la biblioteca GLU para utilidades gráficas.

Resultado:



3. Ventajas y Limitaciones del Enfoque Clásico

- **Ventajas**:
    - **Simplicidad**: Fácil de entender los conceptos básicos de cómo OpenGL procesa vértices y renderiza primitivas.
    - **Sin Dependencias Externas**: No requiere configurar librerías adicionales.
- **Limitaciones**:
    - **Anticuado**: Utiliza funciones obsoletas en OpenGL moderno.

    - **Flexibilidad Reducida**: No permite el uso de shaders ni efectos visuales complejos.

4. Conexión con la Teoría

- **Pipeline Fijo**: Utiliza el pipeline fijo de OpenGL, como se explicó en la sesión teórica. No permite el uso de shaders programables.
- **Primer Contacto con Vértices**: Los comandos `glBegin()`, `glVertex()`, y `glEnd()` ayudan a comprender el procesamiento de vértices antes de introducir técnicas modernas.

Este paso inicial proporciona una base para trabajar con OpenGL sin bibliotecas externas, facilitando la comprensión de cómo OpenGL maneja vértices y geometría de manera fundamental.

Paso 1B: Implementación con Código Moderno de OpenGL - Parte B

En este paso, evolucionaremos de la implementación clásica de OpenGL a una versión moderna utilizando **Vertex Buffer Objects (VBO)** y **Vertex Array Objects (VAO)**. Esta evolución es clave para entender la eficiencia y el rendimiento que se logran con las técnicas actuales de OpenGL.

Parte A: Dibujar un Triángulo con OpenGL Moderno

1. Crear un Proyecto Moderno en Visual Studio Code

1. **Pre-requisitos**:
    - **Visual Studio Code** debe estar instalado y configurado desde el Paso 1A.
    - Necesitarás instalar bibliotecas adicionales: **GLEW** y **GLFW**.
2. **Crear un Archivo Nuevo**:
    - Crea un nuevo archivo llamado `main2.cpp` en la carpeta del proyecto.
    - Este archivo será donde desarrollaremos el código moderno para dibujar un triángulo usando VBO y VAO.

2. Instalación de GLEW y GLFW

Para trabajar con OpenGL moderno, es necesario instalar las bibliotecas **GLEW** y **GLFW**.

¿Qué es GLEW?

- **GLEW (OpenGL Extension Wrangler Library)** facilita el acceso a las funciones modernas de OpenGL, manejando extensiones que no siempre están disponibles en todas las plataformas.

¿Qué es GLFW?

- **GLFW** gestiona ventanas, contextos OpenGL y dispositivos de entrada (como teclado y mouse), siendo una alternativa moderna y flexible a GLUT.

Instalación de GLEW y GLFW en Windows

1. **Instalar MSYS2** (si no se hizo en el Paso 1A):
    - Descarga MSYS2 de https://www.msys2.org/.
    - Sigue las instrucciones de instalación.
2. **Abrir la Terminal de MSYS2**:
    - Abre la terminal **MSYS2 MinGW 64-bit**.
3. **Actualizar Paquetes**:

Ejecuta:

```
pacman -Syu
```

    - Cierra y abre nuevamente la terminal para aplicar los cambios.
4. **Instalar GLEW y GLFW**:

Ejecuta:

```
pacman -S mingw-w64-x86_64-glew
pacman -S mingw-w64-x86_64-glfw
```

3. Entendiendo VBO y VAO

En la teoría aprendimos sobre el pipeline de OpenGL y la evolución de técnicas más eficientes.

- **VBO (Vertex Buffer Object)**: Almacena los datos de los vértices en la memoria de la GPU, mejorando la eficiencia al no necesitar constantes transferencias desde la CPU.
- **VAO (Vertex Array Object)**: Almacena las configuraciones de cómo OpenGL procesa los datos de los vértices, simplificando el código y mejorando la reutilización de configuraciones.

4. Implementar el Triángulo con VBO y VAO

Incluir las Librerías Modernas:

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
```

Inicializar GLFW y Crear la Ventana:

```
if (!glfwInit()) {
    return -1;
}

GLFWwindow* window = glfwCreateWindow(800, 600, "Triángulo Moderno", NULL,
NULL);
glViewport(0, 0, 800, 600);
if (!window) {
    glfwTerminate();
    return -1;
}

glfwMakeContextCurrent(window);
```

Inicializar GLEW:

```
if (glewInit() != GLEW_OK) {
    return -1;
}
```

Crear un VBO para Almacenar los Vértices:

```
float vertices[] = {
    -0.5f, -0.5f, 0.0f,  // Vértice 1
     0.5f, -0.5f, 0.0f,  // Vértice 2
     0.0f,  0.5f, 0.0f   // Vértice 3
};
```

```
unsigned int VBO;
glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

Configurar el VAO:

```
unsigned int VAO;
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

Renderizar el Triángulo:

```
while (!glfwWindowShouldClose(window)) {
    glClear(GL_COLOR_BUFFER_BIT);

    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES, 0, 3);

    glfwSwapBuffers(window);
    glfwPollEvents();
}

glfwTerminate();
```

5. Código Completo

```
#include <GL/glew.h>     // Incluye GLEW para acceder a las funciones modernas
de OpenGL.
#include <GLFW/glfw3.h>  // Incluye GLFW para gestionar la ventana y el
```

```
contexto de OpenGL.

int main() {
    // Inicializa GLFW, una biblioteca para crear ventanas y manejar el
contexto de OpenGL.
    if (!glfwInit()) {
        // Si la inicialización falla, retorna un valor de error.
        return -1;
    }

    // Crear una ventana de 800x600 píxeles titulada "Triángulo Moderno".
    GLFWwindow* window = glfwCreateWindow(800, 600, "Triángulo Moderno", NULL,
NULL);
    if (!window) {
        // Si la ventana no se pudo crear, termina GLFW y retorna un error.
        glfwTerminate();
        return -1;
    }

    // Hace que el contexto de OpenGL de esta ventana sea el actual.
    glfwMakeContextCurrent(window);

    // Inicializa GLEW para cargar todas las funciones modernas de OpenGL.
    if (glewInit() != GLEW_OK) {
        // Si la inicialización de GLEW falla, retorna un error.
        return -1;
    }

    // Configura un callback para ajustar el viewport si la ventana cambia de
tamaño.
    glfwSetFramebufferSizeCallback(window, [](GLFWwindow* window, int width,
int height) {
        // Cada vez que la ventana cambia de tamaño, OpenGL ajusta el área de
renderizado (viewport).
        glViewport(0, 0, width, height);
    });

    // Definición de los vértices del triángulo en coordenadas del espacio
normalizado de OpenGL.
    float vertices[] = {
        -0.5f, -0.5f, 0.0f,  // Vértice 1: esquina inferior izquierda.
         0.5f, -0.5f, 0.0f,  // Vértice 2: esquina inferior derecha.
         0.0f,  0.5f, 0.0f   // Vértice 3: parte superior.
    };
```

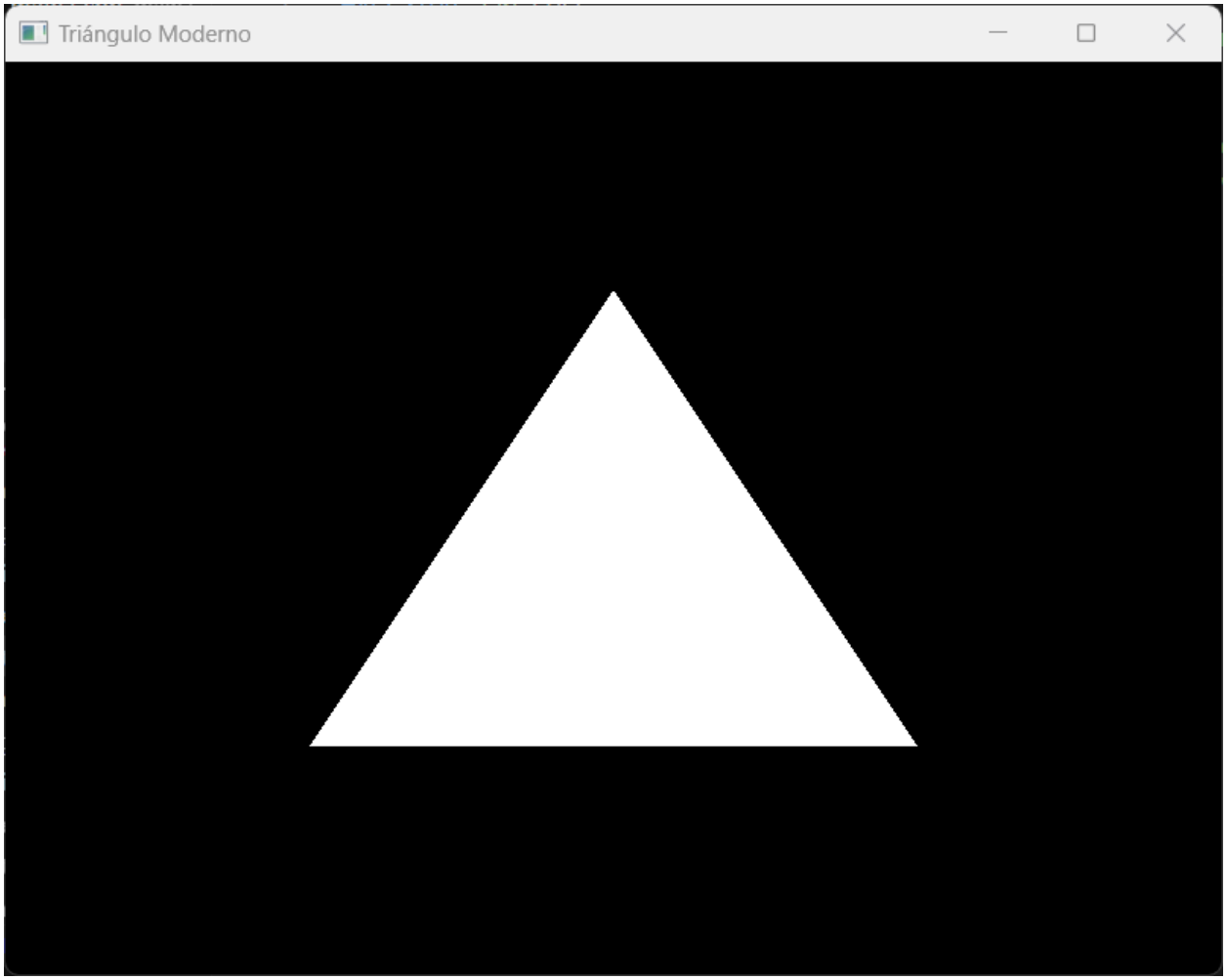
```
    // Crear un VBO (Vertex Buffer Object) para almacenar los datos de los
vértices en la GPU.
    unsigned int VBO;
    glGenBuffers(1, &VBO);  // Genera un buffer.
    glBindBuffer(GL_ARRAY_BUFFER, VBO);  // Enlaza el buffer generado como el
buffer actual de tipo GL_ARRAY_BUFFER.
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// Carga los datos de los vértices en el buffer.

    // Crear un VAO (Vertex Array Object) para almacenar la configuración de
los atributos de los vértices.
    unsigned int VAO;
    glGenVertexArrays(1, &VAO);  // Genera un VAO.
    glBindVertexArray(VAO);  // Enlaza el VAO como el VAO actual.

    // Enlazar el VBO al VAO.
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    // Define cómo interpretar los datos de los vértices.
    // Parámetros: location 0, 3 componentes por vértice (x, y, z), tipo float,
no normalizado, sin desplazamiento.
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float),
(void*)0);
    // Habilita el atributo de los vértices definido en la ubicación 0.
    glEnableVertexAttribArray(0);

    // Bucle principal de renderizado: se ejecuta hasta que se cierre la
ventana.
    while (!glfwWindowShouldClose(window)) {
        // Limpia el buffer de color para prepararse para dibujar el siguiente
cuadro.
        glClear(GL_COLOR_BUFFER_BIT);

        // Enlaza el VAO (que contiene el VBO y la configuración de los
vértices).
        glBindVertexArray(VAO);
        // Dibuja los vértices como un triángulo.
        glDrawArrays(GL_TRIANGLES, 0, 3);

        // Intercambia los buffers de la ventana para mostrar el resultado en
pantalla.
        glfwSwapBuffers(window);
        // Procesa los eventos de la ventana (como teclas y movimiento del
ratón).
        glfwPollEvents();
    }
```

```
    // Limpia y finaliza GLFW, liberando todos los recursos utilizados.
    glfwTerminate();
    return 0;
}
```

Compilación y Ejecución:

```
g++ main2.cpp -o OpenGLTriangle2 -I"C:/msys64/mingw64/include"
-L"C:/msys64/mingw64/lib" -lglfw3 -lglew32 -lopengl32 -lglu32
./OpenGLTriangle2
```

Resultado:



6. Conexión con la Teoría

- **Pipeline Programable**: En este paso no usamos shaders, pero estamos construyendo la base para un pipeline programable, que veremos en el Paso 1C.
- **Eficiencia**: Utilizando **VBO** y **VAO**, optimizamos el rendimiento al aprovechar la GPU, tal como discutimos en la teoría sobre la importancia de delegar tareas gráficas a la GPU.

Este paso moderniza la implementación que realizamos en el Paso 1A y prepara el terreno para introducir técnicas más avanzadas como los **shaders programables**. Con este enfoque, comprenderás cómo se almacenan y gestionan los vértices en la GPU, haciendo el código más eficiente y moderno.

Paso 2A: Implementación de Shaders en OpenGL Moderno - Parte A

En este paso, se introducirá la implementación de shaders en OpenGL. Los shaders permiten controlar de manera programable el pipeline gráfico, otorgando una flexibilidad y un control total sobre cómo se procesan los vértices y se renderizan los fragmentos. Nos enfocaremos en escribir shaders básicos que apliquen color a un triángulo y en entender cómo integrarlos al código moderno de OpenGL.

Parte A: Dibujar un Triángulo con Shaders Simples

1. Crear un Proyecto con Shaders

- **Requisitos Previos**: Tener instalado Visual Studio Code, GLEW, y GLFW (pasos completados en los apartados 1A y 1B).
- **Archivos Necesarios**:
    - Crearemos tres archivos: `main3.cpp` para el código principal, `vertex_shader.glsl` y `fragment_shader.glsl` para los shaders.

2. Creación de los Shaders

Vertex Shader (`vertex_shader.glsl`******)**: Este shader procesa los vértices, especificando la posición de cada uno.

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

void main() {
    gl_Position = vec4(aPos, 1.0);
}
```

- **Explicación**: Este es un shader de vértices básico. La función `main()` se ejecuta para cada vértice. La entrada `aPos` representa la posición del vértice, y `gl_Position` define la posición final en la pantalla. La salida debe ser un vector de

4 componentes (`vec4`), por lo que agregamos `1.0` como la cuarta componente. Esto nos permite convertir las coordenadas del espacio de vértices a las coordenadas de la pantalla, que es lo que OpenGL utiliza para posicionar los elementos en el render.

Fragment Shader (`fragment_shader.glsl`******)**: Este shader controla el color del fragmento que se va a renderizar.

```
#version 330 core
out vec4 FragColor;

void main() {
    FragColor = vec4(1.0, 0.5, 0.2, 1.0); // Color naranja
}
```

- **Explicación**: Este shader de fragmentos asigna un color a cada fragmento generado por el pipeline gráfico. `FragColor` define el color de salida. En este caso, se establece un color naranja con componentes RGBA (rojo, verde, azul, alfa). Cada vez que un fragmento pasa por este shader, será coloreado según lo que especifiquemos aquí.

3. Cargar y Compilar los Shaders en el Código Principal (`main3.cpp`)

Incluir las librerías necesarias:

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <iostream>
#include <fstream>
#include <sstream>
```

- **Explicación**: Importamos las librerías necesarias para utilizar OpenGL, GLFW y GLEW. `GLFW` se utiliza para crear ventanas y manejar entradas del usuario, mientras que `GLEW` nos permite acceder a funciones modernas de OpenGL que no están disponibles por defecto.

`iostream` se utiliza para imprimir mensajes de error, y `fstream` y `sstream` son necesarios para leer el contenido de los archivos de shaders.

Agregar funciones para compilar shaders:

```
std::string loadShaderSource(const char* filepath) {
    std::ifstream file(filepath);
    std::stringstream buffer;
    buffer << file.rdbuf();
    return buffer.str();
}
```

- **Explicación**: Esta función se encarga de leer un archivo de shader y devolver su contenido como una cadena (`std::string`). `ifstream` se usa para abrir el archivo, y `stringstream` para almacenar todo el contenido del archivo. Esto nos permitirá pasar el código del shader a OpenGL para que pueda compilarlo.

```
unsigned int compileShader(unsigned int type, const std::string& source) {
    unsigned int shader = glCreateShader(type); // Crear un objeto shader
    const char* src = source.c_str(); // Convertir el código del shader a
formato C
    glShaderSource(shader, 1, &src, nullptr); // Pasar el código fuente del
shader a OpenGL
    glCompileShader(shader); // Compilar el shader

    int success;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &success); // Comprobar si la
compilación fue exitosa
    if (!success) {
        char infoLog[512];
        glGetShaderInfoLog(shader, 512, nullptr, infoLog); // Obtener el
mensaje de error si falló
        std::cerr << "Error al compilar el shader:\n" << infoLog << std::endl;
    }
    return shader; // Devolver el ID del shader compilado
}
```

- **Explicación**: Esta función crea y compila un shader de OpenGL. Primero, `glCreateShader()` se usa para crear un objeto shader. Luego, `glShaderSource()` pasa

el código fuente al objeto shader, y `glCompileShader()` lo compila. Si la compilación falla, `glGetShaderInfoLog()` nos proporciona información detallada sobre el error, que imprimimos con `std::cerr`. Esta función es genérica y se puede utilizar tanto para vertex shaders como para fragment shaders.

Crear un programa de shaders:

```
unsigned int createShaderProgram(const char* vertexPath, const char*
fragmentPath) {
    std::string vertexCode = loadShaderSource(vertexPath); // Cargar el código
del vertex shader
    std::string fragmentCode = loadShaderSource(fragmentPath); // Cargar el
código del fragment shader

    unsigned int vertexShader = compileShader(GL_VERTEX_SHADER, vertexCode); //
Compilar el vertex shader
    unsigned int fragmentShader = compileShader(GL_FRAGMENT_SHADER,
fragmentCode); // Compilar el fragment shader

    unsigned int shaderProgram = glCreateProgram(); // Crear el programa de
shaders
    glAttachShader(shaderProgram, vertexShader); // Adjuntar el vertex shader
al programa
    glAttachShader(shaderProgram, fragmentShader); // Adjuntar el fragment
shader al programa
    glLinkProgram(shaderProgram); // Enlazar el programa de shaders

    int success;
    glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success); // Comprobar si el
enlace fue exitoso
    if (!success) {
        char infoLog[512];
        glGetProgramInfoLog(shaderProgram, 512, nullptr, infoLog); // Obtener
el mensaje de error si falló
        std::cerr << "Error al enlazar el programa de shaders:\n" << infoLog <<
std::endl;
    }

    glDeleteShader(vertexShader); // Eliminar el vertex shader ya que está
enlazado al programa
    glDeleteShader(fragmentShader); // Eliminar el fragment shader ya que está
```

```
enlazado al programa
    return shaderProgram; // Devolver el ID del programa de shaders
}
```

- **Explicación**: Esta función se encarga de crear un programa de shaders. Los shaders individuales (vertex y fragment) se compilan primero usando `compileShader()`. Luego, estos shaders se adjuntan al programa con `glAttachShader()` y se enlazan con `glLinkProgram()`. Finalmente, eliminamos los shaders individuales porque ya no los necesitamos por separado, ya que están integrados en el programa.

4. Integrar los Shaders en el Renderizado del Triángulo

Dentro de `main3.cpp`, luego de crear el VAO y el VBO (como en el Paso 1B):

- Cargar el programa de shaders:

```
unsigned int shaderProgram = createShaderProgram("vertex_shader.glsl",
"fragment_shader.glsl");
```

- **Explicación**: Creamos el programa de shaders utilizando los archivos `vertex_shader.glsl` y `fragment_shader.glsl` que hemos escrito anteriormente. Este programa de shaders será utilizado para procesar y colorear los vértices del triángulo.
- Utilizar el programa de shaders en el bucle principal:

```
while (!glfwWindowShouldClose(window)) {
    glClear(GL_COLOR_BUFFER_BIT); // Limpiar el buffer de color antes de cada
frame

    glUseProgram(shaderProgram); // Usar el programa de shaders
    glBindVertexArray(VAO); // Enlazar el VAO (Vertex Array Object) que
contiene los vértices
    glDrawArrays(GL_TRIANGLES, 0, 3); // Dibujar los vértices como un triángulo

    glfwSwapBuffers(window); // Intercambiar los buffers para mostrar el frame
actual
    glfwPollEvents(); // Comprobar si hubo eventos (teclado, ratón, etc.)
```

```
}
```

- **Explicación**: En cada iteración del bucle principal, primero limpiamos la pantalla con `glClear()`. Luego, activamos nuestro programa de shaders con `glUseProgram()`. Después, enlazamos el VAO, que contiene la información sobre cómo se dibujan nuestros vértices. `glDrawArrays()` se usa para dibujar el triángulo utilizando los vértices del VAO. Finalmente, `glfwSwapBuffers()` muestra el contenido del frame, y `glfwPollEvents()` se encarga de procesar cualquier interacción del usuario.

5. Código Completo

Vertex Shader (`vertex_shader.glsl`******):**

```
#version 330 core
layout(location = 0) in vec3 aPos;  // Especifica que la entrada (input) de este shader es un vector de 3 componentes llamado aPos, que se encuentra en la ubicación 0 del VAO.

void main() {
    // La variable `gl_Position` es una variable reservada de OpenGL que define la posición final del vértice.
    // Aquí convertimos el `aPos` de un vec3 (x, y, z) a un vec4 añadiendo el valor `1.0` como la componente w.
    gl_Position = vec4(aPos, 1.0);
}
```

- **Explicación**: Este shader de vértices transforma cada vértice a coordenadas de pantalla usando `gl_Position`. Los vértices se pasan como un `vec3` (tres componentes), y luego se convierten a un `vec4` para que OpenGL pueda procesarlos correctamente.

Fragment Shader (`fragment_shader.glsl`******):**

```
#version 330 core
out vec4 FragColor;  // Especifica que la salida (output) de este shader es un vector de 4 componentes llamado FragColor.

void main() {
    // Asigna el color a cada fragmento. Aquí se define el color naranja (RGB: 1.0, 0.5, 0.2) con una opacidad (alfa) de 1.0.
    FragColor = vec4(1.0, 0.5, 0.2, 1.0);
}
```

- **Explicación**: Este shader de fragmentos asigna un color a cada fragmento para renderizar el triángulo de color naranja. `FragColor` es la salida de este shader y controla el color de cada píxel dentro del triángulo.

Código Principal (`main3.cpp`):

```
#include <GL/glew.h>     // Biblioteca para manejar extensiones de OpenGL.
#include <GLFW/glfw3.h>  // Biblioteca para crear ventanas y manejar eventos del sistema.
#include <iostream>      // Biblioteca estándar para entrada y salida.
#include <fstream>       // Biblioteca para manejar archivos.
#include <sstream>       // Biblioteca para manejar flujos de cadenas.

// Función que carga el contenido de un archivo y lo devuelve como un string.
// Esto es útil para cargar el código fuente de los shaders.
std::string loadShaderSource(const char* filepath) {
    std::ifstream file(filepath);
    std::stringstream buffer;
    buffer << file.rdbuf(); // Lee el contenido del archivo y lo almacena en el buffer.
    return buffer.str();    // Convierte el contenido a string y lo devuelve.
}

// Función para compilar un shader a partir de su código fuente.
// Parámetros:
// - type: Tipo de shader (GL_VERTEX_SHADER o GL_FRAGMENT_SHADER).
// - source: Código fuente del shader en forma de string.
unsigned int compileShader(unsigned int type, const std::string& source) {
    // Crear el objeto shader en OpenGL del tipo especificado.
```

```
    unsigned int shader = glCreateShader(type);
    const char* src = source.c_str();  // Convierte el código fuente a un
puntero de tipo char.
    glShaderSource(shader, 1, &src, nullptr);  // Carga el código fuente en el
objeto shader.
    glCompileShader(shader);  // Compila el shader.

    // Verificación de errores de compilación.
    int success;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
    if (!success) {
        // Si hubo un error en la compilación, se obtiene el mensaje de error y
se imprime.
        char infoLog[512];
        glGetShaderInfoLog(shader, 512, nullptr, infoLog);
        std::cerr << "Error al compilar el shader:\n" << infoLog << std::endl;
    }
    return shader;  // Devuelve el identificador del shader compilado.
}

// Función para crear un programa de shader que combina un shader de vértices y
uno de fragmentos.
// Parámetros:
// - vertexPath: Ruta al archivo del shader de vértices.
// - fragmentPath: Ruta al archivo del shader de fragmentos.
unsigned int createShaderProgram(const char* vertexPath, const char*
fragmentPath) {
    // Cargar los códigos fuente de los shaders desde archivos.
    std::string vertexCode = loadShaderSource(vertexPath);
    std::string fragmentCode = loadShaderSource(fragmentPath);

    // Compilar los shaders.
    unsigned int vertexShader = compileShader(GL_VERTEX_SHADER, vertexCode);
    unsigned int fragmentShader = compileShader(GL_FRAGMENT_SHADER,
fragmentCode);

    // Crear el programa de shaders y enlazar los shaders compilados.
    unsigned int shaderProgram = glCreateProgram();
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);
    glLinkProgram(shaderProgram);  // Enlaza el programa completo de shaders.

    // Verificación de errores en el enlace del programa.
    int success;
    glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
```

```
    if (!success) {
        // Si hubo un error al enlazar el programa, se obtiene el mensaje de
error y se imprime.
        char infoLog[512];
        glGetProgramInfoLog(shaderProgram, 512, nullptr, infoLog);
        std::cerr << "Error al enlazar el programa de shaders:\n" << infoLog <<
std::endl;
    }

    // Eliminar los shaders individuales porque ya han sido enlazados al
programa y no se necesitan más.
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);
    return shaderProgram;  // Devuelve el identificador del programa de
shaders.
}

int main() {
    // Inicializar GLFW (una biblioteca para crear ventanas y contextos de
OpenGL).
    if (!glfwInit()) {
        std::cerr << "No se pudo inicializar GLFW" << std::endl;
        return -1;
    }

    // Crear una ventana de 800x600 píxeles titulada "Triángulo con Shaders".
    GLFWwindow* window = glfwCreateWindow(800, 600, "Triángulo con Shaders",
NULL, NULL);
    if (!window) {
        std::cerr << "No se pudo crear la ventana GLFW" << std::endl;
        glfwTerminate();
        return -1;
    }

    // Hacer que el contexto de la ventana creada sea el contexto actual de
OpenGL.
    glfwMakeContextCurrent(window);

    // Inicializar GLEW (necesario para manejar extensiones modernas de
OpenGL).
    if (glewInit() != GLEW_OK) {
        std::cerr << "No se pudo inicializar GLEW" << std::endl;
        return -1;
    }
```

```
    // Definir las coordenadas de los vértices del triángulo.
    float vertices[] = {
        -0.5f, -0.5f, 0.0f,  // Vértice 1: esquina inferior izquierda.
         0.5f, -0.5f, 0.0f,  // Vértice 2: esquina inferior derecha.
         0.0f,  0.5f, 0.0f   // Vértice 3: parte superior.
    };

    // Crear un VBO (Vertex Buffer Object) para almacenar los datos de los
vértices en la GPU.
    unsigned int VBO;
    glGenBuffers(1, &VBO);  // Genera un buffer.
    glBindBuffer(GL_ARRAY_BUFFER, VBO);  // Enlaza el buffer generado como el
buffer actual de tipo GL_ARRAY_BUFFER.
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// Carga los datos de los vértices en el buffer.

    // Crear un VAO (Vertex Array Object) para almacenar la configuración de
los atributos de los vértices.
    unsigned int VAO;
    glGenVertexArrays(1, &VAO);  // Genera un VAO.
    glBindVertexArray(VAO);  // Enlaza el VAO como el VAO actual.

    // Enlazar el VBO al VAO.
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    // Especificar cómo interpretar los datos del VBO: 3 componentes (x, y, z),
tipo float.
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float),
(void*)0);
    glEnableVertexAttribArray(0);  // Habilitar el atributo del vértice.

    // Crear el programa de shaders usando archivos de shaders para vértices y
fragmentos.
    unsigned int shaderProgram = createShaderProgram("vertex_shader.glsl",
"fragment_shader.glsl");

    // Bucle principal de renderizado: sigue ejecutándose hasta que se cierre
la ventana.
    while (!glfwWindowShouldClose(window)) {
        // Limpiar la pantalla para prepararse para el próximo cuadro.
        glClear(GL_COLOR_BUFFER_BIT);

        // Utilizar el programa de shaders.
        glUseProgram(shaderProgram);
        // Enlazar el VAO y dibujar el triángulo.
        glBindVertexArray(VAO);
```

```
        glDrawArrays(GL_TRIANGLES, 0, 3);  // Dibujar los vértices
especificados como un triángulo.

        // Intercambiar los buffers de la ventana para mostrar el resultado en
pantalla.
        glfwSwapBuffers(window);
        // Procesar los eventos de la ventana (como teclas y movimiento del
ratón).
        glfwPollEvents();
    }

    // Terminar GLFW y limpiar todos los recursos utilizados.
    glfwTerminate();
    return 0;
}
```

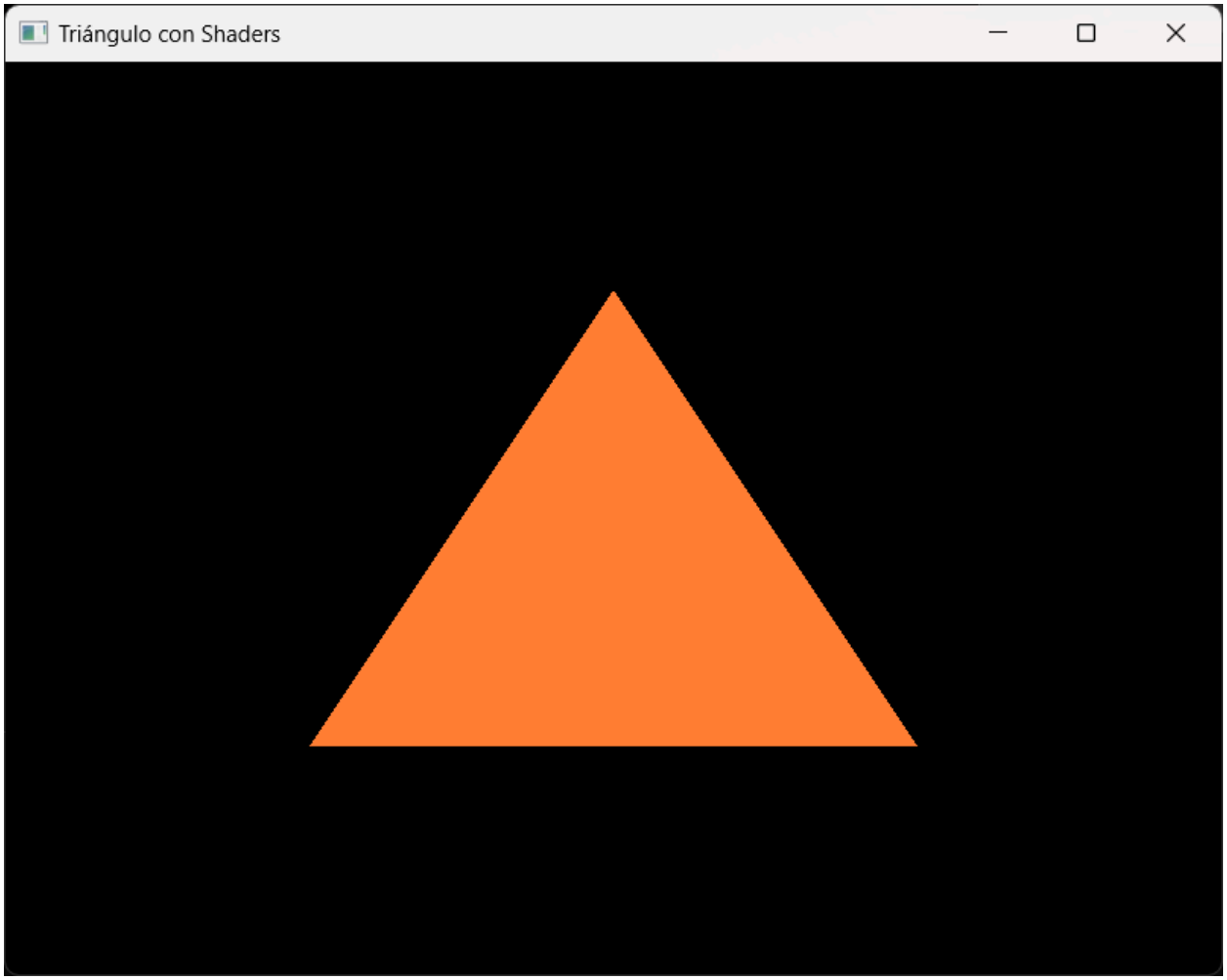
- **Explicación**: El programa principal inicializa GLFW y GLEW, crea el VAO y VBO para almacenar los datos de los vértices del triángulo, y luego usa el programa de shaders para renderizar el triángulo en la pantalla. Cada parte del código está diseñada para dar una introducción gradual a la programación gráfica moderna. Desde inicializar las bibliotecas, configurar los buffers y VAOs, hasta crear y usar shaders, cada paso se enfoca en enseñar cómo construir un pipeline gráfico en OpenGL moderno.

6. Compilación y Ejecución

Compila el proyecto incluyendo las bibliotecas de GLFW y GLEW:

```
g++ main3.cpp -o OpenGLTriangle3 -I"C:/msys64/mingw64/include"
-L"C:/msys64/mingw64/lib" -lglfw3 -lglew32 -lopengl32 -lglu32
./OpenGLTriangle3
```

Resultado:



7. Explicación y Conexión con la Teoría

- **Shaders y el Pipeline Programable**: Los shaders nos permiten pasar de un pipeline fijo a uno completamente programable, proporcionando un control completo sobre la apariencia y el comportamiento del gráfico.
- **Vertex Shader**: Establece cómo se transforman los vértices y cómo se posicionan en la pantalla.
- **Fragment Shader**: Controla cómo se colorean los fragmentos, permitiendo efectos visuales avanzados como gradientes, texturas, y sombras.

Este paso introduce la programación de shaders, que es una parte fundamental del desarrollo moderno de OpenGL, y permite experimentar con efectos visuales más complejos y personalizables. Con estos conocimientos, estarás listo para aplicar transformaciones más avanzadas y texturas en los siguientes pasos.

Paso 2B: Implementación de Iluminación Difusa en OpenGL

En este paso, vamos a mejorar el shader que implementamos en el Paso 2A para agregar **iluminación difusa**. La iluminación difusa da la sensación de una luz que incide sobre una superficie de manera directa, lo cual hace que los objetos se vean más tridimensionales y realistas.

Instalación de la Biblioteca GLM

Para poder realizar este proyecto, necesitas instalar la biblioteca GLM (OpenGL Mathematics). GLM es una biblioteca de matemáticas para gráficos que proporciona funcionalidad útil para trabajar con vectores, matrices y otros elementos matemáticos comunes en gráficos 3D, como transformaciones.

Puedes instalar GLM de varias maneras, dependiendo de tu sistema operativo y el gestor de paquetes que uses. Aquí te dejo algunos ejemplos:

Windows con MSYS2: Puedes instalar GLM usando el gestor de paquetes de MSYS2 con el siguiente comando:

```
pacman -S mingw-w64-x86_64-glm
```

Linux: Si estás en Linux y utilizas APT como gestor de paquetes, puedes instalar GLM con el siguiente comando:

```
sudo apt-get install libglm-dev
```

- **Instalación Manual**: También puedes descargar GLM directamente desde su repositorio en GitHub. Luego de descargarlo, deberás descomprimir la carpeta de GLM dentro de tu proyecto y asegurarte de que las rutas de inclusión sean correctas.

GLM es una biblioteca de solo encabezados, lo cual significa que no necesitas vincular ningún archivo adicional al compilar el proyecto, solo asegurarte de que la carpeta de encabezados esté incluida.

1. Actualización del Vertex Shader (`vertex_shader.glsl`)

En el vertex shader necesitamos definir las **normales de los vértices**, que son vectores perpendiculares a la superficie que nos ayudan a determinar cómo la luz afecta a la superficie. Como nuestro triángulo está en el plano XY, vamos a definir una normal constante.

```
#version 330 core
layout(location = 0) in vec3 aPos;
layout(location = 1) in vec3 aNormal;

out vec3 FragPos;   // Posición del fragmento en el espacio del mundo.
out vec3 Normal;    // Normal del vértice.

void main() {
    FragPos = aPos; // Pasar la posición del vértice al fragment shader.
    Normal = aNormal; // Pasar la normal al fragment shader.
    gl_Position = vec4(aPos, 1.0);
}
```

- **Explicación**:
    - Hemos añadido un atributo de entrada llamado `aNormal`, que representa la normal de cada vértice.
    - `FragPos` y `Normal` se pasarán al fragment shader para realizar los cálculos de iluminación.

2. Actualización del Fragment Shader (`fragment_shader.glsl`)

En el fragment shader agregaremos los cálculos para la iluminación difusa. Necesitamos definir la dirección de la luz y calcular cómo incide sobre la superficie del triángulo.

```
#version 330 core
in vec3 FragPos;   // Posición del fragmento.
in vec3 Normal;    // Normal del fragmento.

out vec4 FragColor;

uniform vec3 lightPos;   // Posición de la fuente de luz.
uniform vec3 lightColor; // Color de la luz.
uniform vec3 objectColor; // Color del objeto.

void main() {
    // Calcular la dirección de la luz.
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - FragPos);

    // Calcular la iluminación difusa usando el producto punto entre la normal
y la dirección de la luz.
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * lightColor;

    // Calcular el color final del fragmento.
    vec3 result = diffuse * objectColor;
    FragColor = vec4(result, 1.0);
}
```

- **Explicación**:
    - `lightPos`: Es la posición de la fuente de luz.
    - `lightColor`: Es el color de la luz. Esto nos permite cambiar la intensidad y el tono de la luz.
    - `objectColor`: Es el color del objeto, que en este caso es el triángulo.
    - Calculamos la dirección de la luz (`lightDir`) y luego el **producto punto** entre `norm` y `lightDir`. Esto nos da la cantidad de luz que incide en la superficie.
    - Finalmente, combinamos el resultado de la luz difusa con el color del objeto para determinar el color final del fragmento.

3. Actualización del Código Principal (`main3.cpp`)

Ahora necesitamos actualizar nuestro código de C++ para pasar la información adicional de la luz y las normales de los vértices a los shaders.

3.1 Definir las Normales y Actualizar el VAO

Actualizaremos los datos de los vértices para incluir las normales.

```
float vertices[] = {
    // posiciones         // normales
    -0.5f, -0.5f, 0.0f,    0.0f, 0.0f, 1.0f,
     0.5f, -0.5f, 0.0f,    0.0f, 0.0f, 1.0f,
     0.0f,  0.5f, 0.0f,    0.0f, 0.0f, 1.0f
};

unsigned int VBO, VAO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);

glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

// Atributo de posición
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

// Atributo de normal
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 *
sizeof(float)));
glEnableVertexAttribArray(1);
```

- **Explicación**:
    - Hemos actualizado el arreglo `vertices` para incluir las normales de cada vértice.
    - Cada vértice ahora tiene 6 valores: 3 para la posición y 3 para la normal.
    - Definimos dos atributos: uno para la posición (`location = 0`) y otro para la normal (`location = 1`).

3.2 Pasar la Información de la Luz al Shader

Necesitamos definir la posición de la luz y su color, y pasar esta información al shader.

```
// Crear el programa de shaders
unsigned int shaderProgram = createShaderProgram("vertex_shader.glsl",
"fragment_shader.glsl");

// Definir la posición y el color de la luz
glm::vec3 lightPos(1.2f, 1.0f, 2.0f);
glm::vec3 lightColor(1.0f, 1.0f, 1.0f);
glm::vec3 objectColor(1.0f, 0.5f, 0.2f);

while (!glfwWindowShouldClose(window)) {
    // Limpiar la pantalla
    glClear(GL_COLOR_BUFFER_BIT);

    // Usar el programa de shaders
    glUseProgram(shaderProgram);

    // Pasar la posición y el color de la luz al fragment shader
    int lightPosLoc = glGetUniformLocation(shaderProgram, "lightPos");
    glUniform3fv(lightPosLoc, 1, &lightPos[0]);

    int lightColorLoc = glGetUniformLocation(shaderProgram, "lightColor");
    glUniform3fv(lightColorLoc, 1, &lightColor[0]);

    int objectColorLoc = glGetUniformLocation(shaderProgram, "objectColor");
    glUniform3fv(objectColorLoc, 1, &objectColor[0]);

    // Dibujar el triángulo
    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES, 0, 3);

    // Intercambiar buffers
    glfwSwapBuffers(window);
    glfwPollEvents();
}

glfwTerminate();
return 0;
```

- **Explicación**:
    - Definimos la posición de la luz (`lightPos`) y su color (`lightColor`).
    - Utilizamos `glGetUniformLocation` para obtener la ubicación de cada variable uniforme en el shader y `glUniform3fv` para pasar los valores correspondientes.
    - Esto permite que el shader tenga toda la información necesaria para calcular la iluminación difusa.

3.5 Código Completo (`main4.cpp`)

```
#include <GL/glew.h>     // Biblioteca para manejar extensiones de OpenGL.
#include <GLFW/glfw3.h>  // Biblioteca para crear ventanas y manejar eventos
del sistema.
#include <iostream>      // Biblioteca estándar para entrada y salida.
#include <fstream>       // Biblioteca para manejar archivos.
#include <sstream>       // Biblioteca para manejar flujos de cadenas.
#include <glm/glm.hpp>   // Biblioteca para vectores y matrices matemáticas.

// Función que carga el contenido de un archivo y lo devuelve como un string.
std::string loadShaderSource(const char* filepath) {
    std::ifstream file(filepath);
    std::stringstream buffer;
    buffer << file.rdbuf(); // Lee el contenido del archivo y lo almacena en el
buffer.
    return buffer.str();    // Convierte el contenido a string y lo devuelve.
}

// Función para compilar un shader a partir de su código fuente.
unsigned int compileShader(unsigned int type, const std::string& source) {
    unsigned int shader = glCreateShader(type);
    const char* src = source.c_str();  // Convierte el código fuente a un
puntero de tipo char.
    glShaderSource(shader, 1, &src, nullptr);  // Carga el código fuente en el
objeto shader.
    glCompileShader(shader);  // Compila el shader.

    int success;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
    if (!success) {
        char infoLog[512];
        glGetShaderInfoLog(shader, 512, nullptr, infoLog);
```

```
        std::cerr << "Error al compilar el shader:\n" << infoLog << std::endl;
    }
    return shader;  // Devuelve el identificador del shader compilado.
}

// Función para crear un programa de shader que combina un shader de vértices y
uno de fragmentos.
unsigned int createShaderProgram(const char* vertexPath, const char*
fragmentPath) {
    std::string vertexCode = loadShaderSource(vertexPath);
    std::string fragmentCode = loadShaderSource(fragmentPath);

    unsigned int vertexShader = compileShader(GL_VERTEX_SHADER, vertexCode);
    unsigned int fragmentShader = compileShader(GL_FRAGMENT_SHADER,
fragmentCode);

    unsigned int shaderProgram = glCreateProgram();
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);
    glLinkProgram(shaderProgram);  // Enlaza el programa completo de shaders.

    int success;
    glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
    if (!success) {
        char infoLog[512];
        glGetProgramInfoLog(shaderProgram, 512, nullptr, infoLog);
        std::cerr << "Error al enlazar el programa de shaders:\n" << infoLog <<
std::endl;
    }

    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);
    return shaderProgram;  // Devuelve el identificador del programa de
shaders.
}

int main() {
    // Inicializar GLFW
    if (!glfwInit()) {
        std::cerr << "No se pudo inicializar GLFW" << std::endl;
        return -1;
    }

    // Crear una ventana de 800x600 píxeles
    GLFWwindow* window = glfwCreateWindow(800, 600, "Triángulo con Iluminación
```

```
Difusa", NULL, NULL);
    if (!window) {
        std::cerr << "No se pudo crear la ventana GLFW" << std::endl;
        glfwTerminate();
        return -1;
    }

    glfwMakeContextCurrent(window);

    // Inicializar GLEW
    if (glewInit() != GLEW_OK) {
        std::cerr << "No se pudo inicializar GLEW" << std::endl;
        return -1;
    }

    // Definir las coordenadas de los vértices del triángulo, incluyendo las
normales
    float vertices[] = {
        // posiciones         // normales
        -0.5f, -0.5f, 0.0f,    0.0f, 0.0f, 1.0f,
         0.5f, -0.5f, 0.0f,    0.0f, 0.0f, 1.0f,
         0.0f,  0.5f, 0.0f,    0.0f, 0.0f, 1.0f
    };

    unsigned int VBO, VAO;
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);

    glBindVertexArray(VAO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    // Atributo de posición
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float),
(void*)0);
    glEnableVertexAttribArray(0);

    // Atributo de normal
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float),
(void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(1);

    // Crear el programa de shaders
    unsigned int shaderProgram = createShaderProgram("vertex_shader.glsl",
```

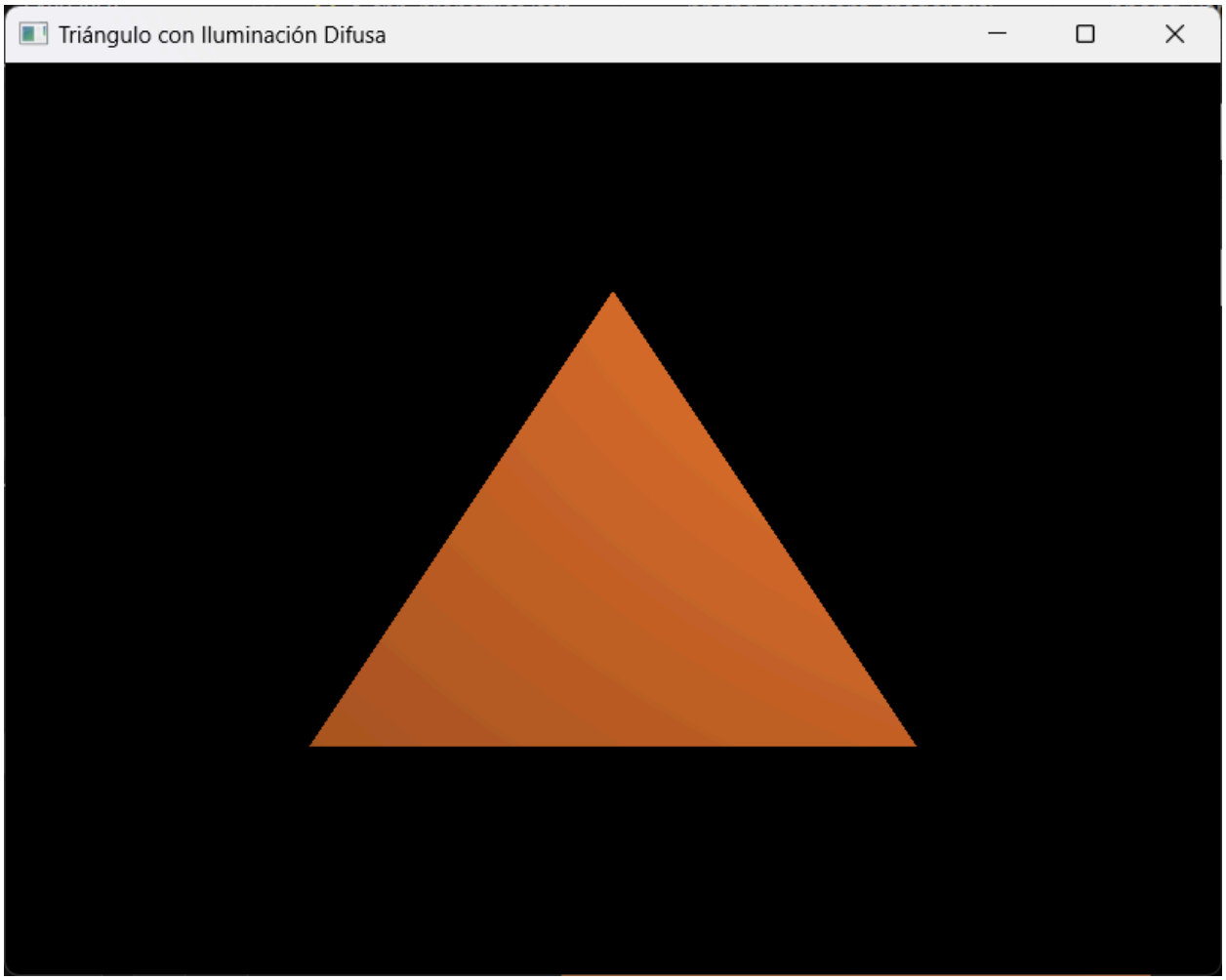
```
"fragment_shader.glsl");

    // Definir la posición y el color de la luz
    glm::vec3 lightPos(1.2f, 1.0f, 2.0f);
    glm::vec3 lightColor(1.0f, 1.0f, 1.0f);
    glm::vec3 objectColor(1.0f, 0.5f, 0.2f);

    while (!glfwWindowShouldClose(window)) {
        // Limpiar la pantalla
        glClear(GL_COLOR_BUFFER_BIT);

        // Usar el programa de shaders
        glUseProgram(shaderProgram);

        // Pasar la posición y el color de la luz al fragment shader
        int lightPosLoc = glGetUniformLocation(shaderProgram, "lightPos");
        glUniform3fv(lightPosLoc, 1, &lightPos[0]);

        int lightColorLoc = glGetUniformLocation(shaderProgram, "lightColor");
        glUniform3fv(lightColorLoc, 1, &lightColor[0]);

        int objectColorLoc = glGetUniformLocation(shaderProgram,
"objectColor");
        glUniform3fv(objectColorLoc, 1, &objectColor[0]);

        // Dibujar el triángulo
        glBindVertexArray(VAO);
        glDrawArrays(GL_TRIANGLES, 0, 3);

        // Intercambiar buffers
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    // Limpiar y terminar
    glfwTerminate();
    return 0;
}
```

4. Compilación y Ejecución

Compila el proyecto incluyendo las bibliotecas de GLFW, GLEW, y GLM:

```
g++ main4.cpp -o OpenGLTriangleWithLighting -I"C:/msys64/mingw64/include"
-I"path/to/glm" -L"C:/msys64/mingw64/lib" -lglfw3 -lglew32 -lopengl32 -lglu32
./OpenGLTriangleWithLighting
```

Resultado:



Explicación y Conexión con la Teoría

- **Modelo de Iluminación de Phong**: La iluminación difusa se basa en el modelo de iluminación de Phong, que simula cómo la luz se refleja de una superficie rugosa en todas las direcciones.

- **Reflejo Dependiente del Ángulo**: La intensidad de la luz reflejada depende del ángulo entre la dirección de la luz y la normal de la superficie. A mayor perpendicularidad, mayor iluminación.
- **Shaders Personalizados**: Usamos **vertex shader** y **fragment shader** para definir cómo la luz afecta a cada vértice y fragmento, logrando así un control detallado sobre el sombreado.
- **Normales de Vértices**: Las **normales** son fundamentales para los cálculos de iluminación, ya que indican la dirección en la que está orientada cada superficie.
- **Iluminación Completa**: La iluminación difusa es solo una parte del modelo de Phong. En futuros pasos, se puede agregar iluminación especular para simular reflejos brillantes y obtener un mayor realismo.

Conclusión

En este paso, hemos implementado un modelo básico de **iluminación difusa** en nuestro triángulo, lo cual hace que se vea más tridimensional y realista. Esto se logró calculando cómo la luz incide sobre cada fragmento usando las **normales** de los vértices y la dirección de la luz. Esto es el comienzo de la aplicación de efectos de iluminación más complejos, que mejoran la percepción de profundidad y realismo en una escena 3D.

Si deseas seguir con la implementación de **iluminación especular** o aplicar múltiples fuentes de luz, estaré encantado de guiarte en los siguientes pasos.

Paso 3A: Implementación de Phong Shading en OpenGL

En este paso, vamos a llevar la iluminación un paso más allá y agregar **Phong Shading**. Este tipo de iluminación nos permite obtener un aspecto mucho más realista al incorporar componentes de luz ambiental, difusa y especular.

1. Actualización del Código Principal (`main5.cpp`)

Vamos a actualizar nuestro código principal de C++ para implementar Phong Shading. Esto implica pasar información adicional al shader, como la posición de la cámara y definir un segundo triángulo para mejorar la percepción de la iluminación.

```
#include <GL/glew.h>     // Biblioteca para manejar extensiones de OpenGL.
#include <GLFW/glfw3.h>  // Biblioteca para crear ventanas y manejar eventos
del sistema.
#include <iostream>      // Biblioteca estándar para entrada y salida.
#include <fstream>       // Biblioteca para manejar archivos.
#include <sstream>       // Biblioteca para manejar flujos de cadenas.
#include <glm/glm.hpp>   // Biblioteca para vectores y matrices matemáticas.
#include <glm/gtc/matrix_transform.hpp> // Incluye funciones para transformar
matrices
#include <glm/gtc/type_ptr.hpp> // Incluye funciones para convertir matrices a
punteros

// Función que carga el contenido de un archivo y lo devuelve como un string.
std::string loadShaderSource(const char* filepath) {
    std::ifstream file(filepath);
    std::stringstream buffer;
    buffer << file.rdbuf(); // Lee el contenido del archivo y lo almacena en el
buffer.
    return buffer.str();    // Convierte el contenido a string y lo devuelve.
}

// Función para compilar un shader a partir de su código fuente.
unsigned int compileShader(unsigned int type, const std::string& source) {
    unsigned int shader = glCreateShader(type);
    const char* src = source.c_str();  // Convierte el código fuente a un
puntero de tipo char.
    glShaderSource(shader, 1, &src, nullptr);  // Carga el código fuente en el
objeto shader.
```

```
    glCompileShader(shader);  // Compila el shader.

    int success;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
    if (!success) {
        char infoLog[512];
        glGetShaderInfoLog(shader, 512, nullptr, infoLog);
        std::cerr << "Error al compilar el shader: " << infoLog << std::endl;
    }
    return shader;  // Devuelve el identificador del shader compilado.
}

// Función para crear un programa de shader que combina un shader de vértices y
uno de fragmentos.
unsigned int createShaderProgram(const char* vertexPath, const char*
fragmentPath) {
    std::string vertexCode = loadShaderSource(vertexPath);
    std::string fragmentCode = loadShaderSource(fragmentPath);

    unsigned int vertexShader = compileShader(GL_VERTEX_SHADER, vertexCode);
    unsigned int fragmentShader = compileShader(GL_FRAGMENT_SHADER,
fragmentCode);

    unsigned int shaderProgram = glCreateProgram();
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);
    glLinkProgram(shaderProgram);  // Enlaza el programa completo de shaders.

    int success;
    glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
    if (!success) {
        char infoLog[512];
        glGetProgramInfoLog(shaderProgram, 512, nullptr, infoLog);
        std::cerr << "Error al enlazar el programa de shaders: " << infoLog <<
std::endl;
    }

    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);
    return shaderProgram;  // Devuelve el identificador del programa de
shaders.
}

int main() {
    // Inicializar GLFW
```

```
    if (!glfwInit()) {
        std::cerr << "No se pudo inicializar GLFW" << std::endl;
        return -1;
    }

    // Crear una ventana de 800x600 píxeles
    GLFWwindow* window = glfwCreateWindow(800, 600, "Triángulo con Phong
Shading", NULL, NULL);
    if (!window) {
        std::cerr << "No se pudo crear la ventana GLFW" << std::endl;
        glfwTerminate();
        return -1;
    }

    glfwMakeContextCurrent(window);

    // Inicializar GLEW
    if (glewInit() != GLEW_OK) {
        std::cerr << "No se pudo inicializar GLEW" << std::endl;
        return -1;
    }

    glEnable(GL_DEPTH_TEST); // Habilitar el buffer de profundidad desde el
inicio para Phong Shading

    // Definir las coordenadas de los vértices de los triángulos, incluyendo
las normales
    float vertices[] = {
        // Primer triángulo
        // posiciones         // normales
        -0.5f, -0.5f, 0.0f,    0.0f, 0.0f, 1.0f,
         0.5f, -0.5f, 0.0f,    0.0f, 0.0f, 1.0f,
         0.0f,  0.5f, 0.0f,    0.0f, 0.0f, 1.0f,

        // Segundo triángulo (cercano al primero para generar un efecto de
sombra o reflejo)
        // posiciones         // normales
        -0.5f, -0.5f, -0.5f,    0.0f, 0.0f, 1.0f,
         0.5f, -0.5f, -0.5f,    0.0f, 0.0f, 1.0f,
         0.0f,  0.5f, -0.5f,    0.0f, 0.0f, 1.0f
    };

    unsigned int VBO, VAO;
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
```

```
    glBindVertexArray(VAO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    // Atributo de posición
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float),
(void*)0);
    glEnableVertexAttribArray(0);

    // Atributo de normal
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float),
(void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(1);

    // Crear el programa de shaders
    unsigned int shaderProgram =
createShaderProgram("phong_vertex_shader.glsl", "phong_fragment_shader.glsl");

    // Definir la posición y el color de la luz y de la cámara
    glm::vec3 lightPos(1.2f, 1.0f, 2.0f);
    glm::vec3 lightColor(1.0f, 1.0f, 1.0f);
    glm::vec3 objectColor(1.0f, 0.5f, 0.2f);
    glm::vec3 viewPos(0.0f, 0.0f, 3.0f); // Posición de la cámara en el espacio
del mundo

    // Definir las matrices de transformación (modelo, vista, proyección)
    glm::mat4 view = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f,
-3.0f));
    glm::mat4 projection = glm::perspective(glm::radians(45.0f), 800.0f /
600.0f, 0.1f, 100.0f);

    while (!glfwWindowShouldClose(window)) {
        // Limpiar la pantalla
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        // Usar el programa de shaders
        glUseProgram(shaderProgram);

        // Definir las matrices de modelo para cada triángulo
        glm::mat4 model1 = glm::mat4(1.0f); // Primer triángulo sin
transformación
        glm::mat4 model2 = glm::translate(glm::mat4(1.0f), glm::vec3(0.5f,
0.0f, -0.5f)); // Mover el segundo triángulo a la derecha y hacia atrás
```

```
        // Pasar las matrices de transformación al vertex shader para el primer
triángulo
        int modelLoc = glGetUniformLocation(shaderProgram, "model");
        int viewLoc = glGetUniformLocation(shaderProgram, "view");
        int projLoc = glGetUniformLocation(shaderProgram, "projection");

        glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));
        glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projection));

        // Pasar la posición y el color de la luz al fragment shader
        int lightPosLoc = glGetUniformLocation(shaderProgram, "lightPos");
        glUniform3fv(lightPosLoc, 1, &lightPos[0]);

        int lightColorLoc = glGetUniformLocation(shaderProgram, "lightColor");
        glUniform3fv(lightColorLoc, 1, &lightColor[0]);

        int objectColorLoc = glGetUniformLocation(shaderProgram,
"objectColor");
        glUniform3fv(objectColorLoc, 1, &objectColor[0]);

        int viewPosLoc = glGetUniformLocation(shaderProgram, "viewPos");
        glUniform3fv(viewPosLoc, 1, &viewPos[0]);

        // Dibujar el primer triángulo
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model1));
        glBindVertexArray(VAO);
        glDrawArrays(GL_TRIANGLES, 0, 3);

        // Dibujar el segundo triángulo
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model2));
        glDrawArrays(GL_TRIANGLES, 3, 3);

        // Intercambiar buffers
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    // Limpiar y terminar
    glfwTerminate();
    return 0;
}
```

2. Shader de Fragmento con Phong Shading (`phong_fragment_shader.glsl`)

Actualizaremos el fragment shader para implementar el modelo completo de Phong, que incluye componentes de luz ambiental, difusa y especular.

```
#version 330 core

in vec3 FragPos;   // Posición del fragmento en el espacio del mundo.
in vec3 Normal;    // Normal del fragmento en el espacio del mundo.

out vec4 FragColor;

uniform vec3 lightPos;    // Posición de la fuente de luz.
uniform vec3 viewPos;     // Posición de la cámara.
uniform vec3 lightColor;  // Color de la luz.
uniform vec3 objectColor; // Color del objeto.

void main() {
    // Componente ambiental
    float ambientStrength = 0.1;
    vec3 ambient = ambientStrength * lightColor;

    // Componente difusa
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * lightColor;

    // Componente especular
    float specularStrength = 0.8;
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 16);
    vec3 specular = specularStrength * spec * lightColor;

    // Sumar los componentes para obtener el color final
    vec3 result = (ambient + diffuse + specular) * objectColor;
    FragColor = vec4(result, 1.0);
}
```

- **Explicación**:
    - **Componente Ambiental**: Simula la luz ambiental general que está presente en la escena.

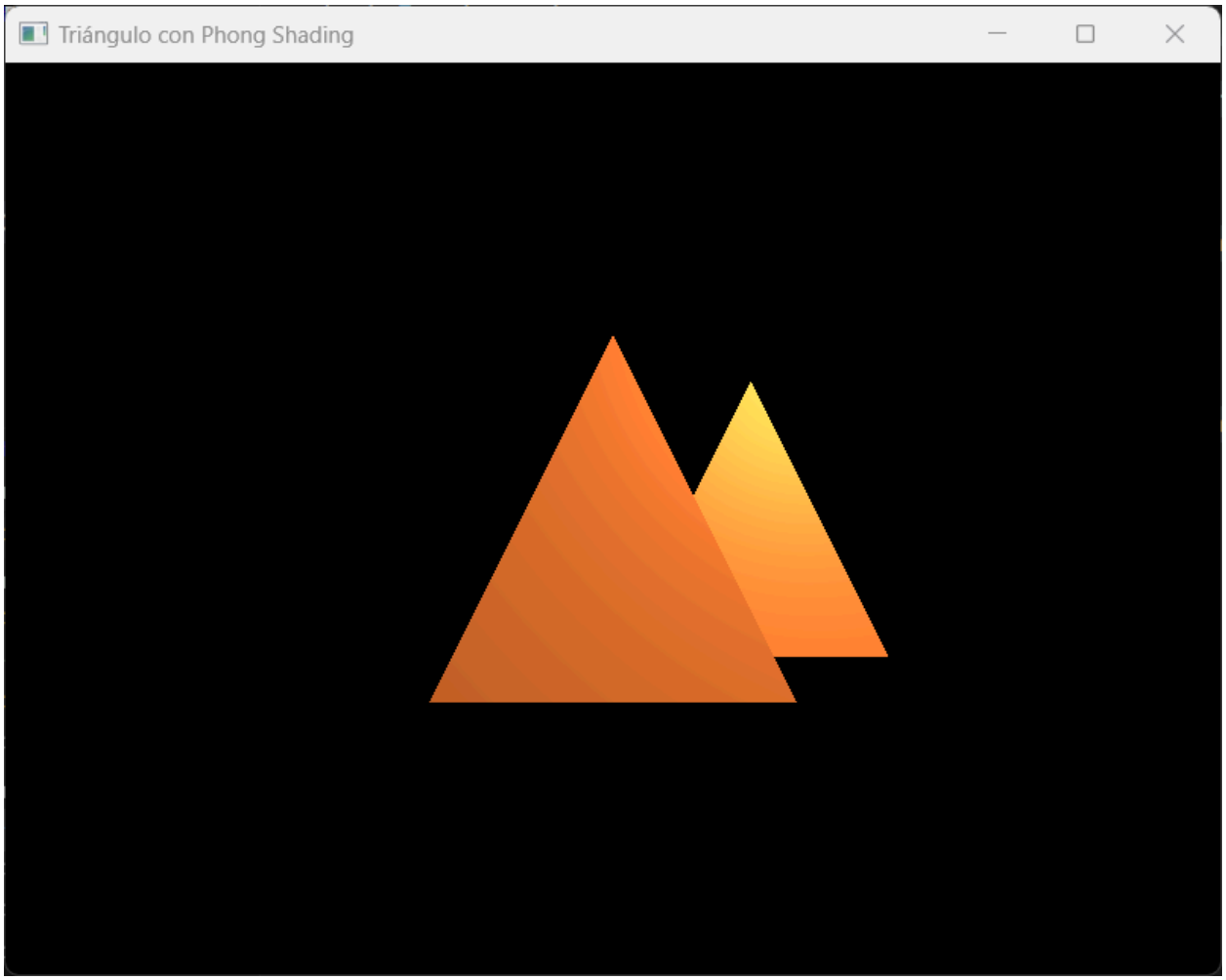
- **Componente Difusa**: Calcula la cantidad de luz reflejada desde la fuente de luz hacia el objeto.
- **Componente Especular**: Agrega los reflejos especulares para hacer que el objeto luzca brillante, simulando un material pulido.

Compilación y Ejecución

Compila el proyecto incluyendo las bibliotecas de GLFW, GLEW, y GLM:

```
g++ main5.cpp -o OpenGLTriangleWithLighting -I"C:/msys64/mingw64/include"
-I"path/to/glm" -L"C:/msys64/mingw64/lib" -lglfw3 -lglew32 -lopengl32 -lglu32
./OpenGLTriangleWithLighting
```

Resultado:



Explicación y Conexión con la Teoría

- **Modelo de Iluminación de Phong**: El Phong Shading se basa en el modelo de iluminación de Phong, que simula la forma en que la luz se refleja sobre una superficie, incorporando luz ambiental, difusa y especular.
- **Luz Ambiental, Difusa y Especular**:
    - **Luz Ambiental**: Representa la luz global que siempre está presente en la escena, independientemente de la fuente de luz.
    - **Luz Difusa**: Refleja la luz en todas las direcciones, y depende del ángulo entre la luz y la normal de la superficie.
    - **Luz Especular**: Representa el brillo visible en la superficie, similar al reflejo de una luz sobre un material brillante.
- **Shaders Personalizados**: Utilizamos **vertex shaders** y **fragment shaders** para definir cómo la luz interactúa con los vértices y los fragmentos de la geometría.
- **Normales de Vértices**: Las **normales** de los vértices se utilizan para calcular cómo la luz interactúa con la superficie en cada fragmento, lo cual es esencial para obtener un sombreado preciso y realista.

Conclusión

Con esta implementación de **Phong Shading**, hemos incorporado componentes de luz ambiental, difusa y especular, lo cual mejora la percepción de profundidad y hace que nuestros objetos luzcan mucho más realistas. La diferencia entre esta iluminación y la difusa que usamos antes es la adición del reflejo especular, que simula el brillo en la superficie del objeto.

En el siguiente paso, podemos explorar cómo implementar **mapas de texturas** y combinar texturas con la iluminación para hacer nuestras representaciones gráficas aún más complejas y visualmente atractivas.