Reporte de Práctica de Laboratorio de Computación Gráfica e Interacción Humano-Computadora

Información de la práctica

Nombre del alumno:	Martínez Pérez Brian Erik		
Número de Cuenta:	319049792	Práctica número:	6
Título de la práctica:	Texturizado		
Fecha de entrega:	13 de octubre de 2025		

Introducción: (descripción inicial sobre la práctica)

El texturizado es una técnica fundamental en gráficos por computadora que permite agregar realismo y detalle a los modelos 3D mediante la aplicación de imágenes o patrones sobre su superficie. En la práctica nosotros aprenderemos a cargar, configurar y aplicar texturas en modelos geométricos, aplicando conceptos clave como el mapeo de coordenadas UV, el uso de samplers en los shaders y la configuración de parámetros de textura para controlar aspectos como el filtrado y la repetición.

Aplicar texturizado es útil para cualquier aplicación gráfica moderna, ya que permite simular materiales, superficies y entornos complejos sin aumentar la complejidad geométrica del modelo. En esta práctica, se utilizan herramientas y bibliotecas como stb_image.h para la carga de imágenes, y se implementarán flujos de trabajo estándar en OpenGL para la creación y gestión de texturas mediante VBOs, VAOs y EBOs. También se abordarán técnicas adicionales como la generación de mipmaps para garantizar una visualización correcta.

Resumen Teórico: (introducción sobre los fundamentos teóricos en los que se basa la práctica y cuáles son los objetivos teóricos que persigue.)

Texturas: Las texturas funcionan como "papel tapiz" que se proyecta sobre la geometría 3D mediante un sistema de coordenadas de textura (coordenadas UV). Este sistema permite mapear puntos específicos de una imagen 2D a vértices particulares de la malla 3D.

Fases del texturizado

- Carga de imágenes: Conversión de formatos de imagen (PNG, JPG, BMP) a datos manipulables
- Creación de texturas en GPU: Almacenamiento eficiente en memoria de gráficos
- Configuración de parámetros: Definición de comportamiento de muestreo y filtrado
- Mapeo en shaders: Aplicación mediante programas de vertex y fragment shaders

Coordenadas UV: Las coordenadas de textura (U,V) establecen la correspondencia entre la textura 2D y la geometría 3D, permitiendo deformaciones y ajustes precisos.

Objetivos

- Entender el proceso completo desde la carga de imágenes hasta su renderizado
- Entender la teoría detrás de los mipmaps y su importancia para el rendimiento
- implementar la comunicación entre C++ y shaders.

Reporte de Práctica de Laboratorio de Computación Gráfica e Interacción Humano-Computadora

Descripción de la práctica: (En esta sección se deben describir todos los pasos ejecutados durante la sesión práctica realizada en el laboratorio.)

Lo primero que realizamos fue el cargado de los archivos necesario para aplicar texturizado a un modelo 3d, agregamos los archivos “lamp.frag” y “lamp.vs” en la sección de archivos de origen que se utilizaran como shaders. Además de agregar el código principal “Texturizado.cpp” en la sección de Archivos de origen.

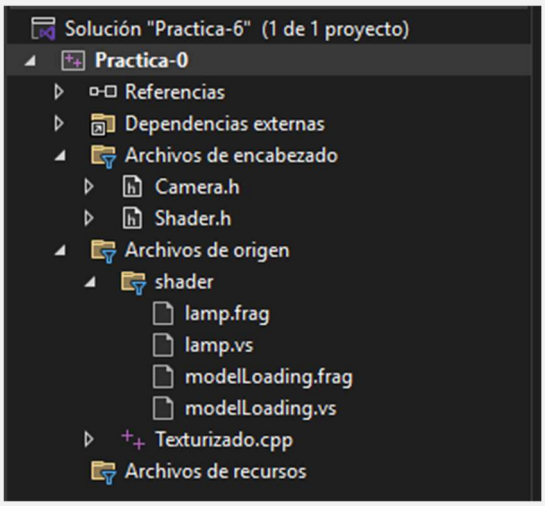


Imagen 1.1 – organización de los archivos utilizamos en la solución del proyecto.

En esta sección se define los vértices del plano cuadrado, donde cargaremos las distintas imágenes que utilizaremos como texturas de nuestro objeto. Definimos la Posición (x, y, z), color (r, g, b), todos blancos en este caso y Coordenadas de textura (u, v).

```
// Set up vertex data (and buffer(s)) and attribute pointers
GLfloat vertices[] =
{
    // Positions           // Colors            // Texture Coords
    -0.5f, -0.5f, 0.0f,    1.0f, 1.0f,1.0f,     0.0f,0.0f,
    0.5f, -0.5f, 0.0f,     1.0f, 1.0f,1.0f,     1.0f,0.0f,
    0.5f,  0.5f, 0.0f,     1.0f, 1.0f,1.0f,     1.0f,1.0f,
    -0.5f,  0.5f, 0.0f,    1.0f, 1.0f,1.0f,     0.0f,1.0f,


};
```

Imagen 1.2 – Código para la definición de vértices.

El siguiente fragmento de código para reutilizar vértices, cuando múltiples triángulos comparten los mismos vértices. En lugar de repetir vértices, solo almacenamos sus índices.

```
GLuint indices[] =
{  // Note that we start from 0!
    0,1,3,
    1,2,3

};
```

Imagen 1.3 – código para la definición de los índices.

```
// Load textures
GLuint texture1;
glGenTextures(1, &texture1);
glBindTexture(GL_TEXTURE_2D,texture1);
int textureWidth, textureHeight,nrChannels;
stbi_set_flip_vertically_on_load(true);
unsigned char *image;
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST_MIPMAP_NEAREST);
// Diffuse map
image = stbi_load("images/checker_Tex.png", &textureWidth, &textureHeight, &nrChannels,0);
glBindTexture(GL_TEXTURE_2D, texture1);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, textureWidth, textureHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, image);
glGenerateMipmap(GL_TEXTURE_2D);
```

Imagen 1.4 - Carga y Configuración de Textura

```
//// Create camera transformations
glm::mat4 view;
view = camera.GetViewMatrix();
glm::mat4 projection = glm::perspective(camera.GetZoom(), (GLfloat)SCREEN_WIDTH / (GLfloat)SCREEN_HEIGHT, 0.1f, 100.0f);
glm::mat4 model(1);
// Get location objects for the matrices on the lamp shader (these could be different on a different shader)
// Get the uniform locations
GLint modelLoc = glGetUniformLocation(lampShader.Program, "model");
GLint viewLoc = glGetUniformLocation(lampShader.Program, "view");
GLint projLoc = glGetUniformLocation(lampShader.Program, "projection");
```

Imagen 1.5 - Sistema de Cámara y Transformaciones

El archivo “lamp.frag” contiene el código Fragment Shader en GLSL que maneja el texturizado.

```
#version 330 core
out vec4 outColor;

in vec3 Color;
in vec2 TexCoord;

uniform sampler2D ourTexture;

void main()
{
    outColor = vec4(Color,1.0)*texture(ourTexture, TexCoord);
}
```

Imagen 1.6 – código del archivo lamp.frag

El archivo “lamp.vs” tiene el código que toma vértices en espacio local y los transforma a su posición final en pantalla, mientras prepara los datos para el texturizado en el fragment shader.

```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 inColor;
layout (location = 2) in vec2 inTexCoord;

out vec3 Color;
out vec2 TexCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * view *  model * vec4(position, 1.0f);
    Color=inColor;
    TexCoord=inTexCoord;
}
```

Imagen 1.7 – código del archivo lamp.vs

La línea de codigo `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)`, Evita la baja calidad al realizar zoom en las texturas de los modelos 3d.

Al ejecutar el código obtenemos, podemos observar como se le aplican la textura de la imagen que le proporcionamos al código, para poder aplicárselo al plano cuadrado que generamos.

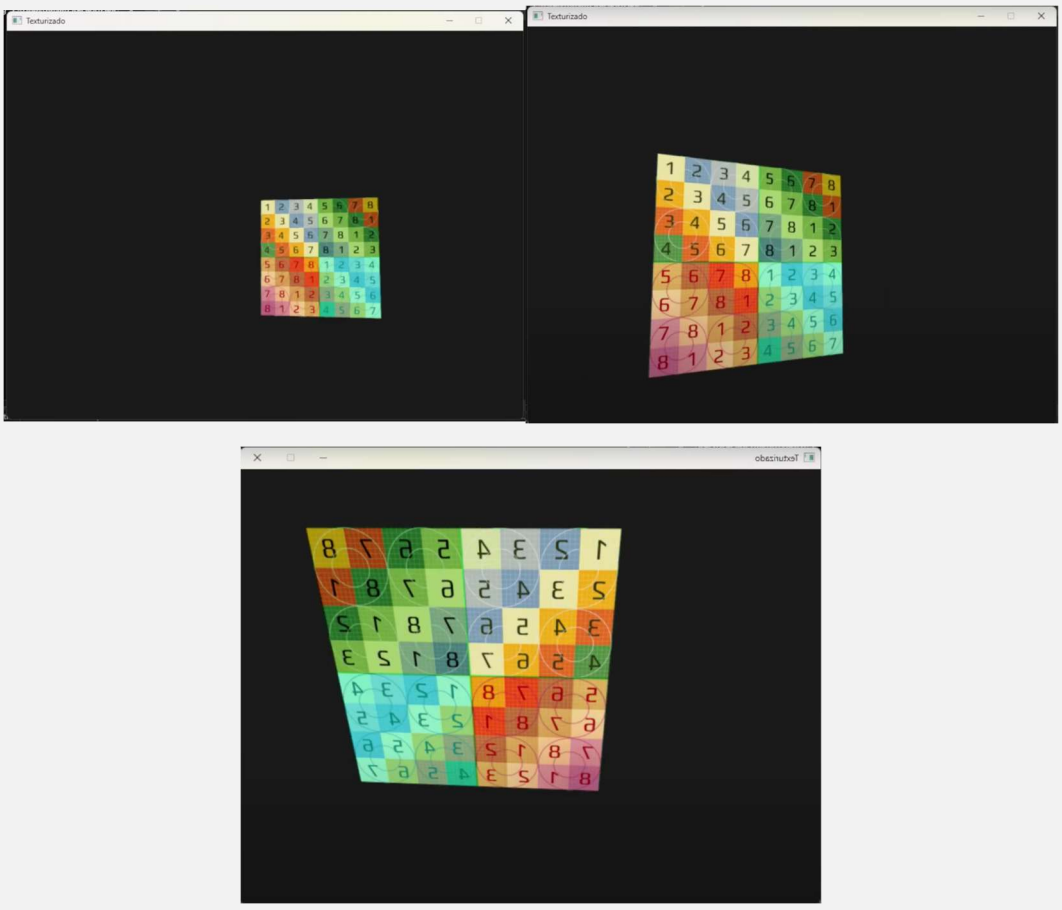


Imagen 2.1 – ejecución del texturizado con la imagen “checker_Tex.png”

Cambiamos los índices de la textura por (3,2,1) y (3,2,1)

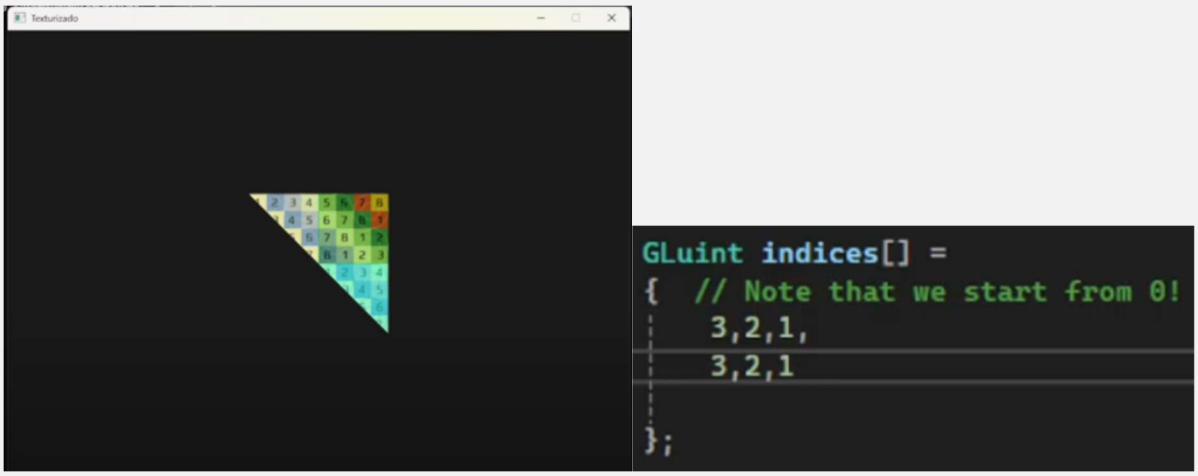


Imagen 2.2 – ejecución con distintos valores de índices de textura

Ahora cambiamos los índices de la textura por (5,2,1) y (4,2,1)

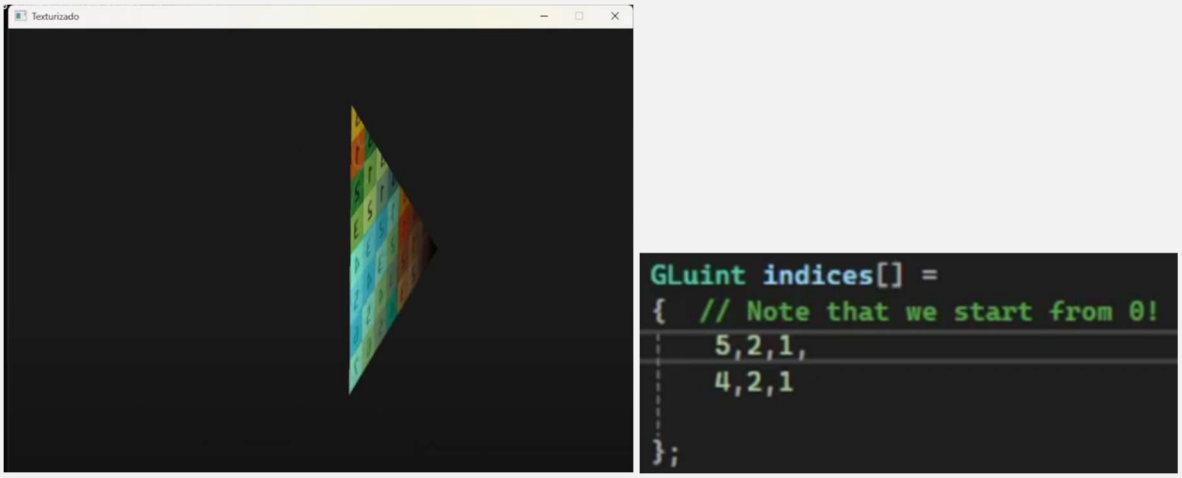


Imagen 2.3 – ejecución con distintos valores de índices de textura

Posteriormente cambiamos la imagen de la textura que colocamos en el modelo 3d, `image = stbi_load("images/perrito.png", &textureWidth, &textureHeight, &nrChannels,0);` cambiamos la imagen por otra de tipo png.

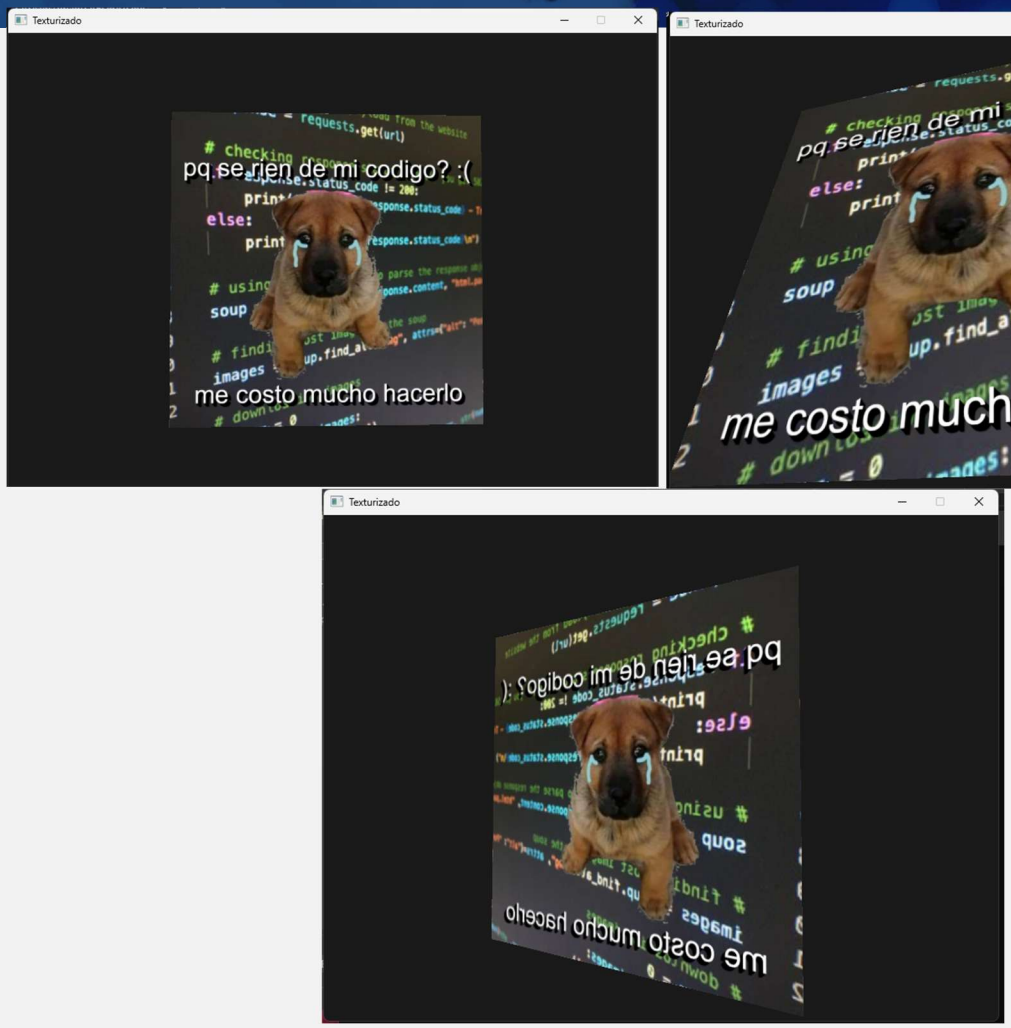
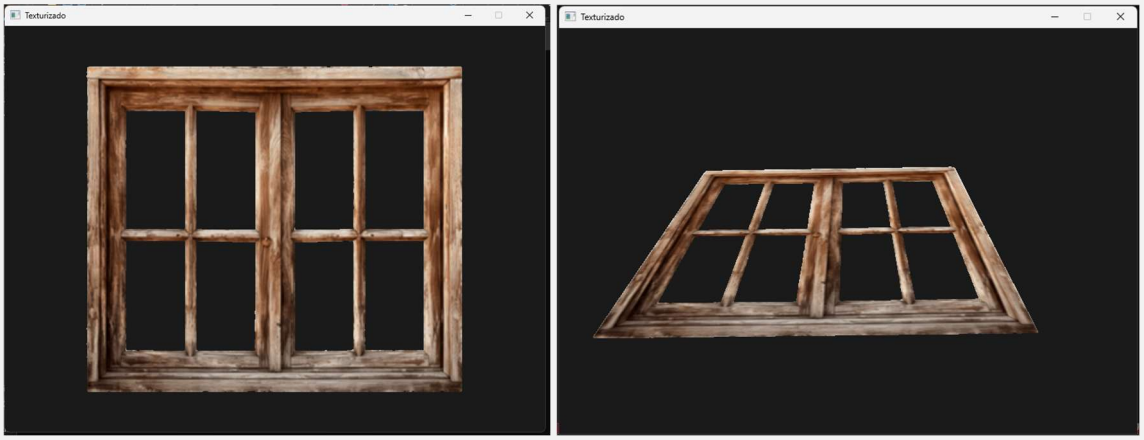


Imagen 2.4 – nueva imagen de textura aplicada

La última imagen cargada fue la de una ventana, igual colocamos el nombre de la imagen en la línea del código `image = stbi_load("images/window.png", &textureWidth, &textureHeight, &nrChannels,0);` código, el problema de esta era a la hora de ejecutar, ya que al ser de tipo png, no se mostraba transparencia en los cristales de la ventana, este problema se corrigió, agregando una condición en el archivo “lamp.frag”.

La condición establecía si el canal alpha del color resultante es menor que 0.1 (10% opacidad), descarta ese píxel lo hace completamente transparente.



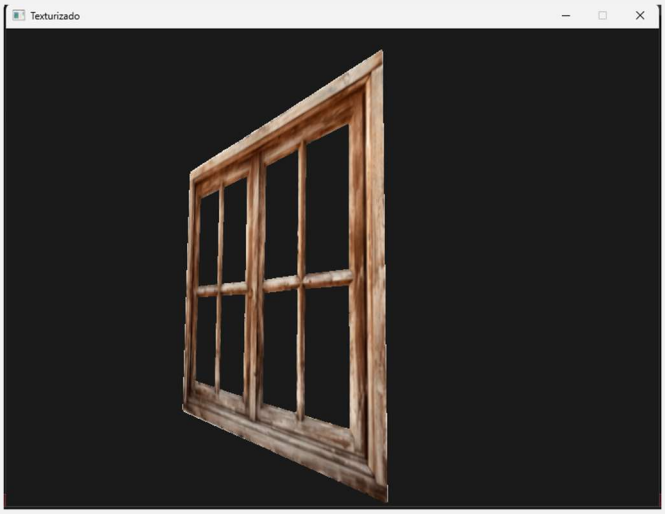


Imagen 2.5 – textura de imagen de ventana con transparencia.

Por último, se duplico el modelo 3d para comprobar que se podían colocar dos modelos. En este caso se coloco la misma ventana 2 veces, con el código.

```
model = glm::mat4(1);
model = glm::translate(model, glm::vec3(1.5f, 0.0f, 0.0f)); // Mover 1.5 unidades a la derecha
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

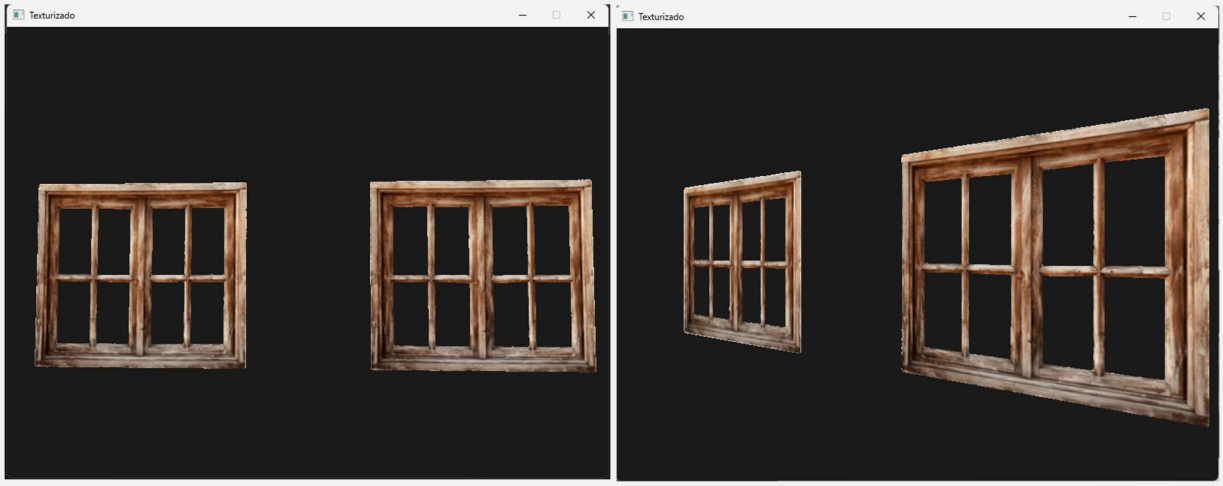


Imagen 2.6 – modelo 3d duplicado con la textura de la ventana.

Resultados: (explicar lo que se logró o aquello que no lograron realizar o comprender)

- Se logro comprender porque es importante dar texturas a un modelo 3d
- Se logro aplicar una imagen png como textura a un plano
- Se pudo aplicar distintas imágenes de archivos tipo png y jpg como texturas de un plano.
- Se soluciono el problema de transparencia en los cristales de la textura de la imagen png.
- Se comprendido como se aplican las coordenadas UV en una imagen, para aplicarlas en modelo 3d.

Conclusiones:

En esta practica logre implementar una imagen como textura de un plano, además para entender como es que se aplican las coordenadas UV, utilizamos una imagen en especial en la cual podíamos experimentar el mapeo de los puntos que deseamos que se coloquen en la textura. También logramos corregir la transparencia en el modelo de una ventana, esta transparencia produce una mayor experiencia visual, ya que produce un efecto mas realista en los modelos generados.

Bibliografía consultada:

Interactivas y Computación Gráfica, T. [@ArturoVMS]. (s/f). *Carga de Texturas en OpenGL Introducción a las UV s [[Object Object]]*. Youtube. Recuperado el 12 de octubre de 2025, de https://www.youtube.com/watch?v=21j0b3w_T8k

stb_image. (2023). stb_image.h: Public Domain Image Loader. https://github.com/nothings/stb/blob/master/stb_image.h