ESCUELA POLITÉCNICA NACIONAL
FACULTAD DE INGENIERÍA DE SISTEMAS
CARRERA DE INGENIERIA EN CIENCIAS DE LA COMPUTACIÓN
COMPUTACIÓN GRAFICA

PROYECTO IIB

Integrantes:

- Nelson Casa
- Cristian Diaz
- Marcela Cabrera
- Santiago Perea
- Danny Tipán

Tema:

- Examen bimestral.

Objetivos:

- Desarrollar un producto de software multimedia interactivo en OPENGL, usando conceptos como texturas 3D, Visualización, Colors, Materials, etc.

Cyber Race Corporate

Misión:

En Cyber Racer Corporate, nuestra misión es revolucionar el mundo de los videojuegos de carreras, proporcionando experiencias inmersivas y divertidas que transporten a los jugadores a un universo de velocidad y competencia. Nos enfocamos en desarrollar juegos con gráficos retro y actuales, generando una combinación de lo clásico y lo moderno, manteniendo siempre el foco en la creatividad y la excelencia técnica. Buscamos inspirar a la comunidad gamer a superar sus límites y vivir la pasión por las carreras en cada partida.

Visión:

Convertirnos en líderes potenciales en el desarrollo de videojuegos en el ámbito de la velocidad, y automotriz, somos reconocidos por nuestra capacidad de innovar y usar lo tradicional. Aspiramos a crear un legado de juegos emblemáticos que marquen una diferencia en la industria

y en la vida de nuestros jugadores. Imaginamos un futuro donde nuestro software no solo ayude al entretenimiento, sino que también ayude a la comunicación y unión de diferentes culturas y generaciones a través de una pasión, pasión llamada motores y velocidad.

Descripción:

"Cyber Racer" es un emocionante juego de conducción en el que el jugador toma el control de un carro que debe esquivar una serie de obstáculos en una pista hasta llegar a la meta. El objetivo principal es sobrevivir el mayor tiempo posible evitando colisiones con los obstáculos que aparecen en el camino. A medida que el jugador avanza, la velocidad y la dificultad aumentan, ofreciendo un desafío creciente.

Desarrollo:

Se inicia el desarrollo incluyendo varias bibliotecas y configuraciones que son fundamentales para el desarrollo de gráficos en 3D utilizando OpenGL donde se utiliza GLAD para cargar funciones de OpenGL y GLFW para gestionar ventanas y eventos. Utiliza GLM para operaciones matemáticas y transformaciones en 3D. Además, carga y maneja shaders, cámaras y modelos 3D con bibliotecas personalizadas, y stb_image para cargar texturas desde imágenes. También incorpora bibliotecas estándar de C++ para manejo de datos y generación de números aleatorios.

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include <learnopengl/shader.h>
#include <learnopengl/camera.h>
#include <learnopengl/model.h>

#define STB_IMAGE_IMPLEMENTATION
#include <learnopengl/stb_image.h>

#include <iostream>
#include <vector>
#include <random>
```

Figura 1. Declaración de librerías.

En la siguiente figura se gestiona eventos de entrada y la cámara. También se define funciones para manejar el tamaño de la ventana, el desplazamiento del mouse, y la entrada del usuario. Además, incluye una función para verificar colisiones entre objetos 3D y obstáculos, así como una función para cargar texturas. Se utiliza una estructura para representar obstáculos y una instancia de la clase Camera para controlar la vista en 3D.

```
void framebuffer_size_callback(GLFWwindow* window, int width, int height);
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset);
void processInput(GLFWwindow* window);
bool cubesStopped = false;
bool controlesVisible = true;
bool gameOverVisible = false;
bool winnerVisible = false;

bool CheckCollision(glm::vec3 objectPos, glm::vec3 objectScale, glm::vec3 obstaclePos, glm::vec3 obstacleScale) {
    bool collisionX = objectPos.x + objectScale.x >= obstaclePos.x &&
        obstaclePos.x + obstacleScale.x >= objectPos.x;
    bool collisionY = objectPos.y + objectScale.y >= obstaclePos.y &&
        obstaclePos.y + obstacleScale.y >= objectPos.y;
    bool collisionZ = objectPos.z + objectScale.z >= obstaclePos.z &&
        obstaclePos.z + obstacleScale.z >= objectPos.z;
    return collisionX && collisionY && collisionZ;
}

unsigned int loadTexture(const char* path);

Camera camera(glm::vec3(0.0f, 3.7f, -11.0f));

bool firstMouse = true;

float deltaTime = 0.0f;
float lastFrame = 0.0f;

bool movingForward = false;
bool cameraStopped = false;

struct Obstacle {
    glm::vec3 position;
    glm::vec3 scale;
};
```

Figura 2. Declaración de variables globales.

Luego de esto la función InitializeObstacles se encarga de crear y configurar una serie de obstáculos en la escena. Primero, limpia la lista de obstáculos. Luego, utiliza un generador de números aleatorios para determinar la posición de cada obstáculo dentro de un rango definido por LEFT_LIMIT y RIGHT_LIMIT. Se generan 10 obstáculos, cada uno con una posición específica en el eje Z (espaciado a intervalos de 20 unidades) y una escala fija de 3x3x3. Finalmente, los obstáculos se añaden a la lista obstacles.

```
void InitializeObstacles() {
    obstacles.clear();
    std::default_random_engine generator;
    std::uniform_real_distribution<float> distribution(LEFT_LIMIT, RIGHT_LIMIT);

    for (int i = 0; i < 10; ++i) {
        Obstacle obstacle;
        obstacle.position = glm::vec3(distribution(generator), 1.5f, -float(i * 20));
        obstacle.scale = glm::vec3(3.0f, 3.0f, 3.0f);
        obstacles.push_back(obstacle);
    }
}
```

Figura 3. Función inicializa los obstáculos.

En la siguiente imagen se tiene la función UpdateObstacles la cual actualiza la posición de los obstáculos en la escena. Si no se está moviendo hacia adelante o los obstáculos están detenidos, la función termina sin hacer nada. En caso contrario, recorre cada obstáculo y mueve su posición en el eje Z hacia adelante, ajustándola según el tiempo transcurrido (deltaTime). Si un obstáculo supera una cierta distancia respecto a la posición del coche (carPosition.z + 50.0f), se reposiciona al otro lado del coche, en un nuevo valor aleatorio dentro de los límites definidos en el eje X.

```cpp
void UpdateObstacles(float deltaTime, glm::vec3 carPosition) {
    if (!movingForward || cubesStopped) {
        return;
    }

    for (auto& obstacle : obstacles) {
        obstacle.position.z += deltaTime * 5.0f;

        if (obstacle.position.z > carPosition.z + 50.0f) {
            obstacle.position.z = carPosition.z - 50.0f;
            obstacle.position.x = LEFT_LIMIT + static_cast<float>(rand()) / (static_cast<float>(RAND_MAX / (RIGHT_LIMIT - LEFT_LIMIT)));
        }
    }
}
```

Figura 4. Actualización de los obstáculos.

En la figura 5 se define un arreglo de vértices para el cubo que representará los obstáculos en OpenGL. Cada vértice está compuesto por tres componentes para la posición (x, y, z), tres para las normales (x, y, z), y dos para las coordenadas de textura (s, t). El arreglo incluye los vértices para todas las caras del cubo, con las normales y coordenadas de textura adecuadas para cada cara. Esto permite a OpenGL renderizar un cubo con iluminación y texturización, utilizando los datos de posición, normales y texturas especificadas.

```cpp
float vertices[] = {
    // positions          // normals           // texture coords
    -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f,  0.0f,
     0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  1.0f,  0.0f,
     0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  1.0f,  1.0f,
     0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  1.0f,  1.0f,
    -0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f,  1.0f,
    -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f,  0.0f,

    -0.5f, -0.5f,  0.5f,  0.0f,  0.0f,  1.0f,  0.0f,  0.0f,
     0.5f, -0.5f,  0.5f,  0.0f,  0.0f,  1.0f,  1.0f,  0.0f,
     0.5f,  0.5f,  0.5f,  0.0f,  0.0f,  1.0f,  1.0f,  1.0f,
     0.5f,  0.5f,  0.5f,  0.0f,  0.0f,  1.0f,  1.0f,  1.0f,
    -0.5f,  0.5f,  0.5f,  0.0f,  0.0f,  1.0f,  0.0f,  1.0f,
    -0.5f, -0.5f,  0.5f,  0.0f,  0.0f,  1.0f,  0.0f,  0.0f,

    -0.5f,  0.5f,  0.5f, -1.0f,  0.0f,  0.0f,  1.0f,  0.0f,
    -0.5f,  0.5f, -0.5f, -1.0f,  0.0f,  0.0f,  1.0f,  1.0f,
    -0.5f, -0.5f, -0.5f, -1.0f,  0.0f,  0.0f,  0.0f,  1.0f,
    -0.5f, -0.5f, -0.5f, -1.0f,  0.0f,  0.0f,  0.0f,  1.0f,
    -0.5f, -0.5f,  0.5f, -1.0f,  0.0f,  0.0f,  0.0f,  0.0f,
    -0.5f,  0.5f,  0.5f, -1.0f,  0.0f,  0.0f,  1.0f,  0.0f,

     0.5f,  0.5f,  0.5f,  1.0f,  0.0f,  0.0f,  1.0f,  0.0f,
     0.5f,  0.5f, -0.5f,  1.0f,  0.0f,  0.0f,  1.0f,  1.0f,
     0.5f, -0.5f, -0.5f,  1.0f,  0.0f,  0.0f,  0.0f,  1.0f,
     0.5f, -0.5f, -0.5f,  1.0f,  0.0f,  0.0f,  0.0f,  1.0f,
     0.5f, -0.5f,  0.5f,  1.0f,  0.0f,  0.0f,  0.0f,  0.0f,
     0.5f,  0.5f,  0.5f,  1.0f,  0.0f,  0.0f,  1.0f,  0.0f,
```

Figura 5. Declaración de coordenadas de posición, color y textura.

Hecho esto se configura el cubo en OpenGL mediante la creación de un Vertex Array Object (VAO) y un Vertex Buffer Object (VBO). Primero, se generan y vinculan estos objetos para manejar los datos del cubo. Luego, se cargan los datos de los vértices en el VBO y se configuran los atributos del VAO: posición, normales y coordenadas de textura, cada uno con el tipo de dato y tamaño apropiado. Además, se carga una textura desde un archivo y se asocia con una variable en el shader para su uso durante el renderizado. Finalmente, se activa el shader y se configuran parámetros de iluminación, como el color y la posición de la luz, para iluminar el cubo correctamente en la escena.

```
unsigned int VBO, cubeVAO; //declaracion de VAO y cubeVAO
glGenVertexArrays(1, &cubeVAO);
glGenBuffers(1, &VBO);

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

glBindVertexArray(cubeVAO);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);

glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6 * sizeof(float)));
glEnableVertexAttribArray(2);

unsigned int diffuseMap = loadTexture("textures/container2.png");

ourShader.use();
ourShader.setVec3("lightColor", 1.0f, 1.0f, 1.0f);
ourShader.setVec3("lightPos", lightPos);
```

Figura 6.. configuración de vértices y texturas y enlaza buffers correspondientes.

En la figura 7 se gestiona el movimiento de la cámara en la escena de OpenGL para que el vehículo vaya aumentado la velocidad según pasa el tiempo. Para ello si se cumplen las condiciones de que la cámara se está moviendo hacia adelante (movingForward) y no está detenida (!cameraStopped), se incrementa la velocidad de movimiento de la cámara y se procesa el movimiento hacia adelante según el tiempo transcurrido (deltaTime). Si la posición de la cámara en el eje Z es menor o igual a -7075.0f y su posición en el eje X es menor o igual a 10000.0f, se detiene la cámara estableciendo cameraStopped a true y la velocidad de movimiento a 0.0f. Además, se establece winnerVisible a true para indicar que se ha alcanzado la meta o un estado de victoria en el juego.

```
if (movingForward && !cameraStopped)
{
    camera.MovementSpeed += 0.5f * deltaTime;
    camera.ProcessKeyboard(FORWARD, deltaTime);

    if (camera.Position.z <= -7075.0f && camera.Position.x <= 10000.0f) {
        cameraStopped = true;
        camera.MovementSpeed = 0.0f;
        winnerVisible = true;
    }
}
```

Figura 7. Movimiento que aumenta el movimiento de la cámara.

La figura 8 realiza la detección de los elementos que se encuentran como trampa, para que el usuario realice movimientos de forma que tenga un alto grado de dificultad, entonces en el código se realiza la ubicación del vehículo y se pone un tamaño. Luego el bucle define la constante auto y obstáculo tomando en cuenta la condición.

```
glm::vec3 carPosition = camera.Position + glm::vec3(0.0f, -3.0f, -12.0f);
glm::vec3 carScale = glm::vec3(1.0f, 1.0f, 1.0f);
for (const auto& obstacle : obstacles) {
    if (CheckCollision(carPosition, carScale, obstacle.position, obstacle.scale)) {
        cameraStopped = true;
        camera.MovementSpeed = 0.0f;
        cubesStopped = true;
        gameOverVisible = true;
        break;
    }
}
```

Figura 8. Detección de colisión.

En la siguiente imagen se puede observar cómo se configura y dibuja varios modelos en una escena de OpenGL. Primero, se activa el shader con ourShader.use() y se configuran las matrices de proyección y vista. Luego, se definen y aplican matrices de modelo para diferentes objetos: el coche (objetoModel), la pista (pistaModel), la luna (moonModel) y la línea de llegada (llegadaModel). Cada objeto se posiciona, rota y escala según sea necesario, y luego se dibuja en la escena usando su respectivo método Draw.

```
ourShader.use();
glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom), (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 1000.0f);
glm::mat4 view = camera.GetViewMatrix();
ourShader.setMat4("projection", projection);
ourShader.setMat4("view", view);

glm::mat4 model = glm::mat4(1.0f);
model = glm::translate(model, carPosition);
ourShader.setMat4("model", model);
objetoModel.Draw(ourShader);

model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(-2.0f, 0.0f, -3475.0f));
model = glm::rotate(model, glm::radians(90.0f), glm::vec3(0.0f, 1.0f, 0.0f));
model = glm::scale(model, glm::vec3(0.1f, 0.1f, 0.1f));
ourShader.setMat4("model", model);
pistaModel.Draw(ourShader);

model = glm::mat4(1.0f);

glm::vec3 moonPosition = glm::vec3(camera.Position.x, 10.0f, camera.Position.z - 200.0f);
model = glm::translate(model, moonPosition);
model = glm::scale(model, glm::vec3(5.0f, 5.0f, 5.0f));
ourShader.setMat4("model", model);
moonModel.Draw(ourShader);

model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(-2.0f, 0.0f, -7075.0f));
model = glm::rotate(model, glm::radians(-90.0f), glm::vec3(0.0f, 1.0f, 0.0f));
model = glm::scale(model, glm::vec3(0.1f, 0.1f, 0.1f));
ourShader.setMat4("model", model);
llegadaModel.Draw(ourShader);
```

Figura 9. Renderización de los modelos implementados.

En la figura 10 se renderizan tres modelos (controlesModel, gameoverModel, y winnerModel) según su visibilidad. Para cada modelo, se configura una matriz de transformación que incluye traslación y escala para posicionar y ajustar el tamaño del modelo. Esta matriz se envía al shader y el modelo se dibuja en la escena con las transformaciones aplicadas.

```
if (controlesVisible) {
    model = glm::mat4(1.0f);
    model = glm::translate(model, glm::vec3(-2.0f, -1.0f, -55.0f));
    model = glm::scale(model, glm::vec3(0.1f, 0.1f, 0.1f));
    ourShader.setMat4("model", model);
    controlesModel.Draw(ourShader);
}

if (gameOverVisible) {
    model = glm::mat4(1.0f);
    glm::vec3 gameOverPosition = glm::vec3(camera.Position.x, -5.0f, camera.Position.z - 35.0f); // Calcula la posición Z
    model = glm::translate(model, gameOverPosition);
    model = glm::scale(model, glm::vec3(0.1f, 0.1f, 0.1f));
    ourShader.setMat4("model", model);
    gameoverModel.Draw(ourShader);
}

if (winnerVisible) {
    model = glm::mat4(1.0f);
    glm::vec3 winnerPosition = glm::vec3(camera.Position.x, -10.0f, camera.Position.z - 100.0f); // Calcula la posición Z
    model = glm::translate(model, winnerPosition);
    model = glm::scale(model, glm::vec3(0.1f, 0.1f, 0.1f));
    ourShader.setMat4("model", model);
    winnerModel.Draw(ourShader);
}
```

Figura 10. Visibilidad de texto y controles.

A continuación se puede observar cómo se configura y se utiliza un shader para renderizar obstáculos en la escena. Se activa la textura y se la vincula al shader. Luego, para cada obstáculo en la lista obstacles, se establece una matriz de transformación (modelObstacle) para posicionar y escalar el obstáculo en la escena. Esta matriz se envía al shader, y se dibujan los obstáculos usando glDrawArrays. Finalmente, se desactiva el VertexArray para limpiar el estado.

```
    ourShader.use();
    ourShader.setInt("texture_diffuse1", 0);

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, diffuseMap);

    glBindVertexArray(cubeVAO);
    for (const auto& obstacle : obstacles) {
        glm::mat4 modelObstacle = glm::mat4(1.0f);
        modelObstacle = glm::translate(modelObstacle, obstacle.position);
        modelObstacle = glm::scale(modelObstacle, obstacle.scale);
        ourShader.setMat4("model", modelObstacle);
        glDrawArrays(GL_TRIANGLES, 0, 36);
    }
    glBindVertexArray(0);
```

Figura 11. Configuración de shader y textura para el renderizado de obstáculos.

En la imagen numero 12 se carga una textura en OpenGL generando un identificador, cargando la imagen con stb_image, y configurando su formato según los componentes de color. Luego, se define la textura, se generan mipmaps, y se configuran parámetros de envolvimiento y filtrado. Finalmente, se libera la memoria de la imagen y se maneja cualquier error en la carga.

```
unsigned int textureID;
glGenTextures(1, &textureID);

int width, height, nrComponents;
unsigned char* data = stbi_load(path, &width, &height, &nrComponents, 0);
if (data)
{
    GLenum format;
    if (nrComponents == 1)
        format = GL_RED;
    else if (nrComponents == 3)
        format = GL_RGB;
    else if (nrComponents == 4)
        format = GL_RGBA;

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, textureID);
    glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format, GL_UNSIGNED_BYTE, data);
    glGenerateMipmap(GL_TEXTURE_2D);

    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_LINEAR);

    stbi_image_free(data);
}
else
{
    std::cout << "Texture failed to load at path: " << path << std::endl;
    stbi_image_free(data);
}

return textureID;
```

Figura 12. Carga de una imagen y establece parámetros para el renderizado.

Pruebas:

La figura 13 presenta la pantalla inicial del video juego, además muestra algunos parámetros que fueron implementados, esto para hacer una interfaz intuitiva y que muestre al usuario el uso del mismo, para comodidad y eficacia del software hacia el cliente. Por ejemplo, tenemos los controles que se usaran para el movimiento del auto, como la letra A, que se usara para darle movimiento hacia la izquierda, D que dará el movimiento a la izquierda, W para empezar la carrera y finalmente R para volver a jugar.

Figura 13. Pantalla de menú.

Figura 14. Pantalla de inicio del juego.

Enlace de GitHub:

https://github.com/NelsonCasa14/2024A_GR1CC_GR4.git

Conclusiones y recomendaciones:

Conclusiones:

- Se desarrolló un videojuego, con la ayuda de modelos implementados, shaders, texturas 3D, movimientos de cámara y esencialmente usando diferentes librerías de OPENGL.
- "Cyber Racer" combina elementos de simulación, detección de colisiones y generación dinámica de contenido para ofrecer una experiencia de carrera inmersiva y desafiante. Las técnicas de optimización y los efectos visuales avanzados garantizan que el juego no solo sea atractivo visualmente, sino también eficiente en términos de rendimiento.
- El juego está diseñado para ser altamente interactivo, con controles de usuario intuitivos y respuestas inmediatas a las acciones del jugador. Esto es crucial para mantener el interés y la satisfacción del jugador.

Recomendaciones:

- Es recomendable tener un código ordenado para identificar cada función que se realiza para generar el video juego.
- Para garantizar un entorno de desarrollo estable y libre de problemas, es crucial que todas las librerías necesarias estén bien instaladas y gestionadas.

Bibliografía:

Solórzano Alcívar, N., Moscoso Poveda, S., & Elizalde Ríos, E. (2019). *Scielo*. Obtenido de http://scielo.senescyt.gob.ec/scielo.php?script=sci_arttext&pid=S2588-09342019000200125

Zabaleta Cruz, F. A. (15 de Febrero de 2021). *Niixer*. Obtenido de https://niixer.com/index.php/2021/02/15/video-videojuegos-y-computacion-grafica/

Zahumenszky, C. (19 de Agosto de 2015). *Gizmodo*. Obtenido de https://es.gizmodo.com/asi-funcionaban-los-graficos-de-tus-juegos-favoritos-en-1725014252