Documentación
Práctica OpenGL

David Bercial Blázquez
Grado Diseño y Desarrollo de Videojuegos y Entornos Virtuales
Promoción: 2021/2025

Índice:

Scene

La clase **Scene** maneja la creación, renderización y la gestión de la escena 3D en OpenGL. Incluye la configuración de la cámara, los objetos 3D (como planos, conos, cilindros), las texturas, y la skybox. Además, se encarga de compilar los shaders, manejar la entrada del usuario (teclado y ratón) y aplicar transformaciones a los objetos de la escena.

Atributos principales

- **angle**: Almacena el ángulo de rotación de los objetos, que se actualiza en cada frame para animar los objetos (por ejemplo, los conos).
- **plane, cylinder, cone:** Son instancias de objetos 3D (como planos, cilindros y conos) que se renderizan en la escena.
- **camera**: Un objeto de la clase Camera que gestiona la vista y la proyección de la cámara en la escena.
- **skybox**: Un objeto de la clase Skybox que maneja la representación del entorno de fondo (cielo).
- **terrain**: Representa el terreno de la escena, cargado desde una imagen de mapa de altura (heightmap).
- **program_id**: ID del programa de shaders principal utilizado para la renderización de los objetos 3D.
- **skybox_shader_program**: ID del programa de shaders para la skybox.
- **model_view_matrix_id, projection_matrix_id**: Identificadores de las matrices de modelo y proyección para enviar al shader.

Métodos principales

Scene(unsigned width, unsigned height): Éste es el constructor de la clase Scene, que inicializa todos los componentes necesarios para crear la escena 3D, incluyendo los objetos, las texturas y la cámara.

Parámetros:

- **width:** El ancho de la ventana de la escena.
- **height:** El alto de la ventana de la escena.

Descripción:

Configura las opciones básicas de **OpenGL**, como el **culling** y el **depth testing**. Compila y activa los shaders para los objetos de la escena y la **skybox**. Carga las texturas necesarias para los objetos (como madera, cilindros, conos) y la skybox.

void process_input(const Uint8* keystate, float delta_time): Este método se encarga de procesar la entrada del teclado, permitiendo mover la cámara a través de las teclas presionadas.

Parámetros:

- **keystate:** Un puntero al estado actual de las teclas.
- **delta_time:** El tiempo transcurrido desde el último frame, usado para ajustar el movimiento de la cámara de manera suave.

Descripción:

Se llama al método **process_keyboard** de la clase Camera, que ajusta la posición de la cámara según las teclas presionadas.

void process_mouse_motion(float x_offset, float y_offset): Este método maneja el movimiento del ratón para ajustar la orientación de la cámara.

Parámetros:

- **x_offset:** Desplazamiento en el eje X del ratón.
- **y_offset:** Desplazamiento en el eje Y del ratón.

Descripción:
Se llama al método **process_mouse_motion** de la clase Camera, que ajusta la dirección de la cámara según el movimiento del ratón.

void update(): Este método actualiza el estado de la escena en cada frame.

Descripción:
Incrementa el ángulo de rotación de los objetos para crear la animación. Aumenta la velocidad de movimiento (variable **movement_Speed**) para animaciones de objetos o cámaras.

void render(): Este método es responsable de renderizar todos los elementos de la escena, incluyendo la skybox y los objetos 3D.

Descripción:
Renderiza la skybox utilizando el shader de skybox. Renderiza el **plano, cilindro, conos, y terreno**, aplicando las transformaciones necesarias (**traslaciones y rotaciones**) para cada objeto. Las texturas correspondientes a cada objeto se cargan y se envían al shader para la correcta visualización de los materiales. Para los objetos con transparencia (como los conos), se activa el **blending** (mezcla de colores) para obtener un efecto visual de transparencia.

void resize(unsigned width, unsigned height): Este método se encarga de ajustar la ventana de la escena cuando cambia su tamaño.

Parámetros:
- **width:** El nuevo ancho de la ventana.
- **height:** El nuevo alto de la ventana.

Descripción:
Calcula una nueva matriz de proyección utilizando el método glm::perspective, adaptando el campo de visión a las nuevas dimensiones de la ventana.

void textureLoader(std::string route): Este método carga una textura desde un archivo y la aplica al objeto.

Parámetros:
route: La ruta del archivo de la textura que se desea cargar.

Descripción:
Usa la librería **stb_image** para cargar la textura desde un archivo en disco. Configura la textura en **OpenGL**, ajustando los parámetros de filtrado y envoltura. Crea una nueva textura en la GPU y la asocia a la escena.

GLuint load_skybox_texture(std::vector<std::string> faces): Este método carga las texturas para la skybox.

Parámetros:
faces: Un vector que contiene las rutas de las seis imágenes que forman las caras de la skybox.

Descripción:
Carga las imágenes de la skybox utilizando **stb_image**. Crea una textura de tipo cubemap en **OpenGL**, asociando cada cara del cubo a una de las imágenes proporcionadas. Configura los parámetros de la textura para la **skybox**.

GLuint compile_shaders(): Este método compila los shaders de vértices y fragmentos para los objetos 3D en la escena.

Descripción:
Crea los objetos de shader para el vértice y el fragmento.
Carga el código fuente de los shaders desde las cadenas **vertex_shader_code** y **fragment_shader_code**. Compila los shaders y verifica que no haya errores en la compilación. Crea un programa de shaders, adjunta los shaders y los enlaza en un programa único. Retorna el identificador del programa de shaders.

GLuint compile_skybox_shaders(): Este método compila los shaders de vértices y fragmentos específicos para la skybox.

Descripción:
Similar al método **compile_shaders**, pero enfocado en los shaders de la skybox. Crea y compila los shaders de vértices y fragmentos para la skybox, y los enlaza en un programa de shaders.

Camera

La clase **Camera** proporciona la funcionalidad para controlar la cámara en una escena 3D. Permite mover la cámara mediante teclas y cambiar su orientación mediante el ratón. Además, calcula las matrices de vista necesarias para que la cámara se mueva y rote correctamente en el espacio 3D.

Atributos principales

- **position:** Almacena la posición de la cámara en el espacio 3D.
- **world_up:** Un vector que define la dirección "arriba" global, utilizado para calcular las direcciones de la cámara.
- **yaw:** El ángulo de rotación en torno al eje Y (horizontal), controlado por el movimiento del ratón.
- **pitch:** El ángulo de rotación en torno al eje X (vertical), controlado por el movimiento del ratón.
- **speed:** La velocidad de movimiento de la cámara.
- **sensitivity:** La sensibilidad al movimiento del ratón para ajustar la rotación de la cámara.
- **front:** Un vector que representa la dirección hacia la que está mirando la cámara.
- **right:** Un vector que representa la dirección "derecha" de la cámara, calculado como el producto cruzado entre front y up.
- **up:** Un vector que representa la dirección "arriba" de la cámara, recalculado a partir de right y front.

Métodos principales

Camera(glm::vec3 start_position, glm::vec3 start_up, float start_yaw, float start_pitch): Este es el constructor de la clase Camera. Inicializa la cámara con una posición, una dirección "arriba" global y los ángulos de rotación iniciales.

Parámetros:

- **start_position:** La posición inicial de la cámara en el espacio 3D.
- **start_up:** La dirección "arriba" global de la cámara.
- **start_yaw:** El ángulo de rotación inicial en torno al eje Y.
- **start_pitch:** El ángulo de rotación inicial en torno al eje X.

Descripción:

Inicializa los atributos de la cámara y luego llama al método **update_camera_vectors** para actualizar los vectores front, right y up con base en los valores iniciales de yaw y pitch.

void update_camera_vectors(): Este método calcula los vectores de dirección **front**, **right** y **up** de la cámara basándose en los ángulos de rotación **yaw** y **pitch**.

Descripción:

Calcula el vector **front** como la dirección en la que la cámara está mirando usando las fórmulas trigonométricas basadas en **yaw** y **pitch**. Luego, calcula los vectores **right** y **u**p mediante el producto cruzado de los vectores correspondientes.

void process_keyboard(const Uint8* keystate, float delta_time): Este método maneja la entrada del teclado para mover la cámara.

Parámetros:

keystate: Un puntero al estado actual de las teclas (usualmente proporcionado por SDL).
delta_time: El tiempo transcurrido desde el último frame, utilizado para ajustar el movimiento de la cámara de manera suave.

Descripción:

Dependiendo de las teclas presionadas (W, A, S, D), mueve la cámara en la dirección **fron**t (adelante/atrás) o en la dirección **right** (izquierda/derecha). Ajusta la velocidad de movimiento de la cámara utilizando el valor **speed** y **delta_time**.

void process_mouse_motion(float x_offset, float y_offset): Este método maneja la entrada del ratón para rotar la cámara.

Parámetros:

- **x_offset**: Desplazamiento en el eje X del ratón.
- **y_offset:** Desplazamiento en el eje Y del ratón.

Descripción:

Modifica los ángulos de rotación yaw y pitch según los desplazamientos del ratón, ajustados por la sensibilidad (sensitivity). Asegura que el ángulo pitch no sobrepase los límites de 89.0º y -89.0º para evitar que la cámara se voltee. Llama al método **update_camera_vectors** para actualizar los vectores de dirección de la cámara con los nuevos ángulos.

glm::mat4 get_view_matrix() const: Este método devuelve la matriz de vista de la cámara, que es utilizada para transformar las coordenadas de los objetos en el espacio de la cámara.

Descripción:

Usa la función **glm::lookAt** para calcular una matriz de vista que define la posición de la cámara, el punto hacia el cual está mirando (position + front), y la dirección "arriba" (up). Esta matriz se utiliza para transformar los objetos en la escena según la posición y orientación de la cámara.

void set_speed(float new_speed): Este método establece una nueva velocidad para el movimiento de la cámara.

Parámetros:

- **new_speed:** El valor de la nueva velocidad de la cámara.

Descripción:
Actualiza la velocidad de la cámara, lo que afectará cómo de rápido se mueve en respuesta a las teclas de entrada.

void set_sensitivity(float new_sensitivity): Este método establece una nueva sensibilidad para el movimiento del ratón.

Parámetros:
- **new_sensitivity:** El valor de la nueva sensibilidad del ratón.

Descripción:
Actualiza la sensibilidad del ratón, lo que afectará la cantidad de rotación de la cámara por cada movimiento del ratón.

Heightmap

La clase Heightmap se encarga de cargar un mapa de altura (heightmap), generar una malla 3D basada en dicho mapa y renderizarla utilizando OpenGL. El mapa de altura es una imagen en escala de grises donde los valores de píxel determinan la altura de los vértices de la malla generada.

Atributos principales

- **vao_id:** ID del objeto de arreglo de vértices (VAO) de OpenGL, que almacena los datos de la malla para el renderizado.
- **vbo_id:** ID del objeto de buffer de vértices (VBO), que almacena los datos de los vértices de la malla.
- **ebo_id:** ID del objeto de buffer de elementos (EBO), que almacena los índices de los triángulos de la malla.
- **vertices:** Un vector que contiene los datos de los vértices de la malla (posición, normal y coordenadas UV).
- **normals:** Un vector que contiene los datos de las normales de la malla, que se pueden calcular posteriormente para la iluminación.
- **uvs:** Un vector que contiene las coordenadas UV de la malla, utilizadas para mapear texturas.
- **rows:** El número de filas del mapa de altura (correspondiente a la dimensión en Z de la malla).
- **cols:** El número de columnas del mapa de altura (correspondiente a la dimensión en X de la malla).

Métodos principales

Heightmap(const std::string& heightmap_path, float width, float depth, float max_height): Este es el constructor de la clase Heightmap, que carga el mapa de altura desde un archivo y genera la malla correspondiente.

Parámetros:

- **heightmap_path:** La ruta del archivo del mapa de altura (debe ser una imagen en escala de grises).
- **width:** El ancho de la malla generada.
- **depth:** La profundidad de la malla generada.
- **max_height:** La altura máxima de la malla, que se asigna según los valores de intensidad del mapa de altura.

Descripción:

Llama al método **load_heightmap** para cargar los datos del mapa de altura desde un archivo. Llama al método **generate_mesh** para generar la malla de acuerdo con el tamaño especificado (width y depth). Crea los buffers de OpenGL necesarios (VAO, VBO, EBO) para almacenar los datos de la malla.

~Heightmap(): Destructor de la clase Heightmap. Elimina los buffers de OpenGL cuando el objeto Heightmap se destruye.

Descripción:

Llama a las funciones de OpenGL para borrar los objetos de buffers y arreglos de vértices (VAO, VBO, EBO), liberando los recursos utilizados por la malla.

void load_heightmap(const std::string& path, float max_height): Este método carga un mapa de altura desde una imagen en escala de grises y calcula las posiciones de los vértices de la malla.

Parámetros:

- **path:** La ruta del archivo de imagen que contiene el mapa de altura.
- **max_height:** La altura máxima a la que se deben escalar los valores de intensidad del mapa de altura.

Descripción:

Usa la librería **stb_image** para cargar la imagen del mapa de altura. Recorre cada píxel de la imagen, asignando su valor de intensidad a la altura (y) de los vértices de la malla. Calcula las posiciones de los vértices y asigna valores iniciales para las normales (que se calcularán más tarde) y las coordenadas UV.

void generate_mesh(float width, float depth): Este método genera la malla 3D utilizando los vértices calculados a partir del mapa de altura.

Parámetros:

- **width:** El ancho de la malla.

- **depth:** La profundidad de la malla.

Descripción:
Escala las posiciones de los vértices de acuerdo con el tamaño especificado (width y depth). Crea los índices para los triángulos de la malla. Cada celda del mapa de altura se divide en dos triángulos, por lo que se generan dos índices por celda. La malla resultante se usa para el renderizado en OpenGL.

void render(): Este método se encarga de renderizar la malla generada en OpenGL.

Descripción:
Llama a **glBindVertexArray** para vincular el VAO de la malla. Usa **glDrawElements** para dibujar los triángulos de la malla usando los índices almacenados en el EBO. Desvincula el VAO al finalizar el renderizado para evitar que otros objetos lo modifiquen.

Skybox

La clase Skybox se encarga de cargar las texturas para una skybox (el entorno de fondo de la escena) y renderizarla en OpenGL. La skybox es un cubo con texturas aplicadas a sus caras, que da la apariencia de un entorno infinito cuando se ve desde dentro de una escena 3D.

Atributos principales

- **faces:** Un vector de cadenas que contiene las rutas de las imágenes para las seis caras del cubo (skybox).
- **vao_id:** ID del objeto de arreglo de vértices (VAO) de OpenGL, que contiene los datos de los vértices de la skybox.
- **vbo_id:** ID del objeto de buffer de vértices (VBO), que contiene los datos de los vértices de la skybox.
- **texture_id:** ID de la textura cargada para la skybox, que se aplica al cubo.

Métodos principales

Skybox(const std::vector<std::string>& faces): Este es el constructor de la clase Skybox. Inicializa la skybox cargando las texturas y configurando los buffers de OpenGL para renderizarla.

Parámetros:
- **faces:** Un vector que contiene las rutas de las imágenes para las seis caras de la skybox.

Descripción:

Llama al método **load_textures** para cargar las texturas para las seis caras de la skybox. Llama al método **setup_buffers** para configurar los buffers de OpenGL necesarios para el renderizado.

~Skybox(): Destructor de la clase Skybox. Elimina los recursos de OpenGL cuando el objeto Skybox se destruye.

Descripción:
Elimina los buffers y la textura de la skybox de OpenGL, liberando los recursos utilizados.

void load_textures(): Este método carga las texturas de las seis caras de la skybox y las asocia a un cubo (cubemap) en OpenGL.

Descripción:
Crea una textura de tipo cubemap en OpenGL y carga las imágenes correspondientes a las caras del cubo usando **stb_image**. Configura las propiedades de la textura (como el filtro de minificación y magnificación, y el envolvimiento de las coordenadas de textura).

void setup_buffers(): Este método configura los buffers de OpenGL para almacenar los datos de los vértices de la skybox.

Descripción:
Define los vértices de las seis caras del cubo que representa la skybox. Cada cara del cubo se define con dos triángulos, lo que da un total de 36 vértices (6 caras x 2 triángulos por cara x 3 vértices por triángulo). Crea el VAO, el VBO y los buffers necesarios para almacenar y organizar estos vértices. Configura la matriz de atributos para que OpenGL pueda usar estos vértices para el renderizado.

void set_texture(GLuint texture_id): Este método establece el ID de la textura de la skybox.

Parámetros:
- **texture_id:** El ID de la textura que se debe usar para la skybox.

Descripción:
Asigna la textura de la skybox utilizando el ID proporcionado, que fue generado al cargar las imágenes de las caras del cubo.

GLuint get_texture_id(): Este método devuelve el ID de la textura de la skybox.

Descripción:
Retorna el ID de la textura de la skybox, lo que permite utilizarlo en el proceso de renderizado.

void render(): Este método renderiza la skybox en la escena.

Descripción:

Cambia la función de profundidad de OpenGL a **GL_LEQUAL** para asegurar que la skybox se dibuje detrás de todos los objetos de la escena (y no se recorte). Enlaza el VAO de la skybox y la textura cargada.
Desactiva el culling para que no se recorten las caras del cubo de la skybox, ya que no tiene sentido que se recorten cuando estamos dentro de ella. Utiliza **glDrawArrays** para dibujar los triángulos que componen la skybox. Restaura la función de profundidad a **GL_LESS** después de renderizar la skybox para no interferir con otros objetos de la escena.

Texture

La clase Texture se encarga de cargar y gestionar las texturas en OpenGL, permitiendo asociarlas a objetos en una escena 3D. Utiliza la librería **stb_image** para cargar las imágenes y las configura correctamente en OpenGL.

Atributos principales

- **texture_id:** ID de la textura en OpenGL, generado por la función glGenTextures para identificar la textura.
- **width:** El ancho de la textura cargada.
- **height:** La altura de la textura cargada.
- **channels:** El número de canales de la imagen cargada (por ejemplo, 3 para RGB o 4 para RGBA).

Métodos principales

- **Texture(const std::string& file_path):** Este es el constructor de la clase Texture, que carga una textura desde un archivo de imagen y la configura en OpenGL.

Parámetros:

- **file_path:** La ruta del archivo de imagen que contiene la textura a cargar.

Descripción:

Crea y vincula una textura en OpenGL utilizando **glGenTextures** y **glBindTexture**. Configura los parámetros de la textura, como el tipo de envolvimiento (wrap) y el filtro de minificación y magnificación.
Utiliza la librería **stb_image** para cargar la imagen del archivo especificado. El parámetro **stbi_set_flip_vertically_on_load(true)** asegura que la imagen se voltee verticalmente si es necesario. Si la imagen se carga correctamente, la textura se configura en OpenGL con glTexImage2D, y se genera un mipmap utilizando **glGenerateMipmap**. Si la carga falla, se muestra un mensaje de error. La memoria de la imagen cargada se libera al final con **stbi_image_free(data)**.

~Texture(): Destructor de la clase Texture, que elimina la textura de OpenGL cuando el objeto Texture es destruido.

Descripción:
Llama a glDeleteTextures para eliminar la textura de OpenGL y liberar los recursos asociados.

void bind(unsigned int unit) const: Este método se encarga de vincular la textura a una unidad de textura específica en OpenGL.

Parámetros:
- **unit:** El número de la unidad de textura a la que se debe vincular la textura (por ejemplo, **GL_TEXTURE0** para la primera unidad de textura).

Descripción:
Activa la unidad de textura especificada con **glActiveTexture(GL_TEXTURE0 + unit)** y luego vincula la textura a esa unidad usando **glBindTexture(GL_TEXTURE_2D, texture_id)**.

void unbind() const: Este método desvincula la textura actual de OpenGL.

Descripción:
Desvincula la textura de OpenGL usando glBindTexture(**GL_TEXTURE_2D**, 0), lo que deja la unidad de textura sin textura vinculada.

Cono

La clase **Cone** se encarga de generar un cono en 3D utilizando OpenGL, incluyendo la creación de los vértices, colores, coordenadas UV y los índices necesarios para representar el cono en la escena. La clase maneja la creación de los buffers de OpenGL y el renderizado del cono.

Atributos principales

- **coordinates:** Un vector que contiene las coordenadas de los vértices del cono (base y ápice).
- **colors:** Un vector que contiene los colores asociados a cada vértice.
- **uvs:** Un vector que contiene las coordenadas UV para la asignación de texturas.
- **indices:** Un vector que contiene los índices que definen los triángulos de la malla del cono.
- **vao_id:** ID del objeto de arreglo de vértices (VAO) de OpenGL, que almacena los datos de la malla.
- **vbo_ids:** Un arreglo de IDs de objetos de buffer de vértices (VBO) que almacenan las coordenadas, colores, UVs y los índices de la malla.

Métodos principales

Cone(int radial_segments, float radius, float height): Este es el constructor de la clase Cone, que genera los vértices, colores, coordenadas UV e índices necesarios para renderizar el cono en 3D.

Parámetros:

- **radial_segments:** El número de segmentos radiales que se usarán para dividir la base del cono (cuantos más segmentos, más suave será el cono).
- **radius:** El radio de la base del cono.
- **height:** La altura del cono.
-

Descripción:

- ***Generación de la base****:* Se calculan las coordenadas de los vértices de la base del cono utilizando coordenadas polares (con el ángulo angle), y se asignan valores de color y coordenadas UV.
- ***Vértice central de la base:*** Se agrega un vértice en el centro de la base del cono, con un color rojo y una coordenada UV centrada.
- ***Vértice del ápice:*** Se agrega el vértice del ápice del cono (en la parte superior).
- ***Generación de índices:*** Se crean los índices para los triángulos que forman la base del cono y las caras laterales. La base se divide en triángulos conectando el centro de la base con los vértices de la base, y las caras laterales se crean conectando el vértice del ápice con los vértices de la base.
- ***Configuración de los buffers:*** Se crean los buffers de OpenGL (VBOs y VAO) y se envían los datos de los vértices, colores, UVs e índices para ser utilizados en el renderizado.

~Cone(): Destructor de la clase Cone, que elimina los buffers de OpenGL cuando el objeto Cone es destruido.

Descripción:

Llama a **glDeleteVertexArrays** y **glDeleteBuffers** para liberar los recursos utilizados por los buffers de OpenGL asociados al cono.

void render(): Este método se encarga de renderizar el cono en la escena.

Descripción:

Configura el modo de polígono para que el cono sea rellenado (**GL_FILL**).
Habilita el culling para asegurar que solo se dibujen las caras visibles del cono.
Enlaza el VAO del cono y utiliza **glDrawElements** para dibujar los triángulos que componen la malla del cono utilizando los índices almacenados en el EBO (Element Buffer Object).
Después de renderizar, desvincula el VAO para evitar que se modifiquen otros objetos.

Cylinder

La clase **Cylinder** se encarga de generar un cilindro 3D utilizando OpenGL, incluyendo la creación de los vértices, colores, coordenadas UV y los índices necesarios para representar el cilindro en la escena. Además, maneja la creación de los buffers de OpenGL y el renderizado del cilindro.

Atributos principales

- **coordinates:** Un vector que contiene las coordenadas de los vértices del cilindro (tanto para las caras laterales como para las bases).
- **colors:** Un vector que contiene los colores asociados a cada vértice del cilindro, con un ejemplo de degradado radial.
- **uvs:** Un vector que contiene las coordenadas UV para la asignación de texturas.
- **indices:** Un vector que contiene los índices que definen los triángulos de la malla del cilindro.
- **vao_id:** ID del objeto de arreglo de vértices (VAO) de OpenGL, que contiene los datos de la malla.
- **vbo_ids:** Un arreglo de IDs de objetos de buffer de vértices (VBO) que almacenan las coordenadas, colores, UVs y los índices de la malla.

Métodos principales

Cylinder(int radial_segments, int height_segments, float radius, float height): Este es el constructor de la clase Cylinder, que genera los vértices, colores, coordenadas UV e índices necesarios para renderizar el cilindro en 3D.

Parámetros:

- **radial_segments:** El número de segmentos radiales que se usarán para dividir el cilindro a lo largo de su circunferencia (cuantos más segmentos, más suave será el cilindro).
- **height_segments:** El número de segmentos que dividen el cilindro a lo largo de su altura.
- **radius:** El radio del cilindro.
- **height:** La altura del cilindro.

Descripción:

- ***Generación de vértices:*** El constructor recorre las posiciones a lo largo de la altura del cilindro y a lo largo de su circunferencia. Para cada punto, se calculan las coordenadas (x, y, z), las coordenadas UV y los colores.
- ***Bases del cilindro:*** Añade los vértices centrales de las bases inferior y superior del cilindro.
- ***Generación de índices:*** Se crean los índices para los triángulos que forman las caras laterales del cilindro, así como para las bases.
- ***Configuración de los buffers:*** Se crean los buffers de OpenGL (VBOs y VAO) y se envían los datos de los vértices, colores, UVs e índices para ser utilizados en el renderizado.

~Cylinder(): Destructor de la clase Cylinder, que elimina los buffers de OpenGL cuando el objeto Cylinder es destruido.

Descripción:
Llama a **glDeleteVertexArrays** y **glDeleteBuffers** para liberar los recursos utilizados por los buffers de OpenGL asociados al cilindro.

void render(): Este método se encarga de renderizar el cilindro en la escena.

Descripción:
Configura el modo de polígono para que el cilindro sea rellenado (GL_FILL).
Desactiva el culling para asegurar que todas las caras del cilindro sean visibles.
Enlaza el VAO del cilindro y utiliza glDrawElements para dibujar los triángulos que componen la malla del cilindro utilizando los índices almacenados en el EBO (Element Buffer Object). Después de renderizar, desvincula el VAO para evitar que se modifiquen otros objetos.

Plano

La clase **Plane** se encarga de generar un plano 3D en OpenGL, incluyendo la creación de los vértices, colores, coordenadas UV y los índices necesarios para representar el plano en la escena. Además, maneja la creación de los buffers de OpenGL y el renderizado del plano.

Atributos principales

- **coordinates:** Un vector que contiene las coordenadas de los vértices del plano.
- **colors:** Un vector que contiene los colores asociados a cada vértice del plano.
- **uvs:** Un vector que contiene las coordenadas UV para la asignación de texturas.
- **indices:** Un vector que contiene los índices que definen los triángulos de la malla del plano.
- **vao_id:** ID del objeto de arreglo de vértices (VAO) de OpenGL, que contiene los datos de la malla.
- **vbo_ids:** Un arreglo de IDs de objetos de buffer de vértices (VBO) que almacenan las coordenadas, colores, UVs y los índices de la malla.

Métodos principales

Plane(int width, int height): Este es el constructor de la clase **Plane**, que genera los vértices, colores, coordenadas UV e índices necesarios para renderizar el plano en 3D.

Parámetros:
- **width:** El número de segmentos en el eje X para dividir el plano.
- **height:** El número de segmentos en el eje Y para dividir el plano.

Descripción:

- ***Generación de vértices:*** El constructor recorre las posiciones a lo largo de los ejes X y Y para generar los vértices del plano, asignando un valor fijo de 0 a la coordenada Y para mantener el plano en el eje XZ.
- ***Generación de colores:*** Se asignan colores interpolados a cada vértice, creando un degradado en función de su posición.
- ***Generación de coordenadas UV:*** Se asignan coordenadas UV a los vértices para su posterior mapeo de texturas.
- ***Generación de índices:*** Se crean los índices para los triángulos que forman el plano, utilizando dos triángulos por cada celda de la malla.
- ***Configuración de los buffers:*** Se crean los buffers de OpenGL (VBOs y VAO) y se envían los datos de los vértices, colores, UVs e índices para ser utilizados en el renderizado.

~Plane(): Destructor de la clase Plane, que elimina los buffers de OpenGL cuando el objeto Plane es destruido.

Descripción:

Llama a **glDeleteVertexArrays** y **glDeleteBuffers** para liberar los recursos utilizados por los buffers de OpenGL asociados al plano.

void render(): Este método se encarga de renderizar el plano en la escena.

Descripción:

Configura el modo de polígono para que el plano sea rellenado (**GL_FILL**).
Desactiva el culling para asegurar que todas las caras del plano sean visibles, ya que un plano puede ser visible desde ambos lados. Enlaza el VAO del plano y utiliza glDrawElements para dibujar los triángulos que componen la malla del plano utilizando los índices almacenados en el EBO (Element Buffer Object).
Después de renderizar, desvincula el VAO para evitar que se modifiquen otros objetos.

Cube

La clase **Cube** se encarga de crear y renderizar un cubo 3D en OpenGL. Define sus vértices, colores, y los índices para sus caras, y gestiona la configuración de los buffers de OpenGL necesarios para su renderizado.

Atributos principales

- **coordinates[]:** Un arreglo que contiene las coordenadas de los 8 vértices del cubo. Cada vértice tiene una posición en 3D (x, y, z).
- **colors[]:** Un arreglo que contiene los colores asociados a cada vértice del cubo, en formato RGB.

- **indices[]:** Un arreglo que contiene los índices que definen los triángulos de la malla del cubo, organizados por caras del cubo.
- **vao_id:** ID del objeto de arreglo de vértices (VAO) de OpenGL, que organiza y gestiona los buffers de los vértices.
- **vbo_ids[]:** Un arreglo de IDs de buffers de vértices (VBO) que almacenan las coordenadas, colores y los índices de la malla.

Métodos principales

Cube(): Este es el constructor de la clase Cube, que genera los vértices, colores y los índices necesarios para renderizar el cubo en 3D.

Descripción:

- ***Generación de VBOs:*** Se generan los VBOs (Vertex Buffer Objects) para almacenar los datos de las coordenadas, colores e índices del cubo.
- ***Generación de VAO:*** Se crea un VAO (Vertex Array Object) para organizar y gestionar los buffers de OpenGL del cubo.
- ***Enlazar los buffers:*** Los datos de las coordenadas, colores e índices se suben a los respectivos buffers de OpenGL (VBOs y EBO), y se vinculan al VAO para que puedan ser utilizados durante el renderizado.
- ***Configurar los atributos de vértices:*** Se habilitan y configuran los atributos de los vértices (coordenadas, colores y UVs) utilizando **glVertexAttribPointer** y **glEnableVertexAttribArray**.

~Cube(): Destructor de la clase Cube, que elimina los buffers de OpenGL cuando el objeto Cube es destruido.

Descripción:

Libera los recursos de OpenGL utilizados por el cubo, eliminando el VAO y los VBOs creados en el constructor.

void render(): Este método se encarga de renderizar el cubo en la escena.

Descripción:

- ***Configuración de renderizado:*** Se establece el modo de polígonos para dibujar el cubo con líneas (GL_LINE) y se desactiva el culling (para que todas las caras del cubo sean visibles).
- ***Vinculación de VAO:*** Se vincula el VAO que contiene los datos del cubo.
- ***Dibujo del cubo:*** Utiliza glDrawElements para dibujar los triángulos del cubo usando los índices almacenados en el EBO (Element Buffer Object).
- ***Desvinculación de VAO:*** Después del renderizado, desvincula el VAO para evitar que se modifiquen otros objetos.