Documentación Entrega Programación Gráfica

Descripción General

Este proyecto implementa un motor de renderizado 3D completo utilizando C++ y OpenGL. El motor es capaz de renderizar escenas complejas que incluyen terrenos procedurales, vegetación instanciada, objetos transparentes y skyboxes, todo ello con un sistema de iluminación realista.

Tecnologías Utilizadas

- **C++17**: Lenguaje principal del proyecto
- **OpenGL 3.3**: API de renderizado gráfico
- **SDL2**: Gestión de ventanas y eventos
- **GLM**: Biblioteca matemática para gráficos 3D
- **GLAD**: Cargador de extensiones OpenGL
- **SOIL2**: Carga de texturas e imágenes
- **Assimp**: Importación de modelos 3D
- **Visual Studio 2022**: Entorno de desarrollo

Estructura del Proyecto

MainProject/

├── code/ # Código fuente principal

├── project/VisualStudio2022/ # Archivos del proyecto VS

├── project/binaries/ # Binarios

shared/

└── assets/

    ├── shaders/   # Shaders GLSL

    ├── textures/   # Texturas y heightmaps

    └── models/   # Modelos 3D

documents/ #Documentación del proyecto

libraries/ # Librerias

Arquitectura del Sistema

Sistema de Ventanas y Contexto

Archivo: `Window.hpp/cpp`

La clase `Window` encapsula la gestión de la ventana SDL y el contexto OpenGL. Características principales:

- **Configuración automática del contexto OpenGL**: La clase maneja automáticamente la configuración de versión, buffers de profundidad y sincronización vertical
- **Gestión RAII**: Implementa move semantics para transferencia segura de recursos
- **Inicialización de GLAD**: Se encarga de cargar las extensiones OpenGL necesarias

```
// Ejemplo de uso
Window window("Mi Aplicación",
        Window::Position::CENTERED,
        Window::Position::CENTERED,
```

1024, 576,

{3, 3}); // OpenGL 3.3

Sistema de Shaders

Archivos: `Shader.hpp/cpp`, `VertexShader.hpp`, `FragmentShader.hpp`, `ShaderProgram.hpp`

El sistema de shaders está diseñado con una jerarquía clara:

- **Shader**: Clase base que maneja la compilación de shaders individuales
- **VertexShader/FragmentShader**: Especializaciones para cada tipo
- **ShaderProgram**: Enlaza múltiples shaders en un programa ejecutable

Flujo de trabajo típico:

1. Crear shaders individuales desde archivos
2. Compilar cada shader y verificar errores
3. Adjuntar shaders al programa
4. Enlazar el programa completo
5. Limpiar shaders individuales (ya no necesarios)

Geometrías y Mallas

Archivo base: `Mesh.hpp/cpp`

La clase `Mesh` es abstracta y define la interfaz común para todas las geometrías:

- **Método virtual puro `initialize()`**: Cada geometría debe implementar su propia generación de vértices
- **Sistema VAO/VBO**: Gestión automática de buffers OpenGL

- **Múltiples atributos**: Posiciones, normales, colores e índices
- **Renderizado instanciado**: Soporte para dibujar múltiples instancias eficientemente

Geometrías Implementadas

Plane (`Plane.hpp/cpp`):

- Genera un plano subdividido en una grilla
- Configurable en resolución (filas/columnas) y dimensiones
- Útil como base para terrenos o superficies planas

Cone (`Cone.hpp/cpp`):

- Cono paramétrico con número de segmentos configurable
- Genera tanto la superficie lateral como la base
- Calcula automáticamente las normales para iluminación

Cube (`Cube.hpp/cpp`):

- Cubo simple con dimensiones configurables
- Usado principalmente para objetos transparentes de demostración

Sistema de Grafo de Escena

Archivos: `SceneNode.hpp`, `Camera.hpp`

El grafo de escena permite organizar objetos 3D de forma jerárquica:

SceneNode:

- **Transformaciones**: Posición, rotación y escala local
- **Jerarquía padre-hijo**: Los hijos heredan las transformaciones del padre
- **Cálculo de matrices**: Convierte transformaciones locales a matrices mundiales

- **Gestión de memoria**: Utiliza smart pointers para gestión automática

Camera:

- Hereda de SceneNode, por lo que puede ser posicionada en el grafo
- **Matrices de vista y proyección**: Calcula automáticamente las matrices necesarias
- **Parámetros de cámara**: FOV, aspecto, planos cercano y lejano configurables

Terreno Procedural

Archivos: `HeightMapTerrain.hpp/cpp`

El sistema de terreno genera superficies 3D a partir de imágenes heightmap:

Proceso de generación:

1. **Carga de heightmap**: Lee una imagen RGB y convierte cada píxel a altura
2. **Generación de vértices**: Crea una grilla de vértices con alturas basadas en la imagen
3. **Cálculo de normales**: Determina las normales por diferencias finitas entre vecinos
4. **Coloreado por altura**: Aplica diferentes colores según la elevación (agua, costa, pasto, montaña, nieve)
5. **Triangulación**: Genera triángulos para formar la superficie

Características avanzadas:

- **Consulta de altura**: Permite obtener la altura en cualquier posición mundial
- **Interpolación bilineal**: Suaviza las consultas de altura entre píxeles
- **Integración con hierba**: Proporciona información de altura para colocación de vegetación

Sistema de Vegetación

Archivos: `GrassMesh.hpp/cpp`

Implementa un sistema de instanciado masivo para renderizar miles de plantas de forma eficiente:

Carga de modelos:

- Utiliza Assimp para cargar modelos 3D complejos de hierba
- Procesa vértices, normales e índices del modelo base

Generación de instancias:

- **Distribución aleatoria**: Coloca hierba de forma natural sobre el terreno
- **Filtrado por altura**: Solo coloca hierba en elevaciones apropiadas (evita agua y montañas altas)
- **Variación procedural**: Cada instancia tiene rotación, escala y color únicos
- **Coloreado inteligente**: El color depende de la altura del terreno

Optimización de renderizado:

- **Instanced drawing**: Renderiza todas las instancias en una sola llamada a OpenGL
- **Atributos por instancia**: Posición, color, escala y rotación se pasan como atributos de instancia
- **Buffer management**: Gestión eficiente de memoria GPU

Sistema de Skybox

Archivos: `Skybox.hpp/cpp`

Implementa un skybox cúbico para fondos infinitos:

- **Cubemap texture**: Utiliza 6 texturas para las caras del cubo

- **Renderizado especial**: Se dibuja sin translación para mantenerlo infinitamente lejano
- **Integración con depth buffer**: Se asegura de que siempre aparezca detrás de otros objetos

Motor de Renderizado Principal

Archivos: `Scene.hpp/cpp`

La clase Scene orquesta todo el proceso de renderizado:

Inicialización:

- Configura múltiples programas de shaders (opaco, transparente, hierba, skybox)
- Crea el grafo de escena con terreno, cámara y objetos
- Genera la vegetación sobre el terreno
- Configura el skybox

Loop de renderizado (orden crítico):

1. **Skybox**: Se renderiza primero como fondo
2. **Objetos opacos**: Terreno y geometrías sólidas
3. **Hierba instanciada**: Miles de instancias con shader especializado
4. **Objetos transparentes**: Con blending habilitado y depth mask deshabilitado

Sistema de input:

- **Control de cámara**: WASD para movimiento, flechas para rotación
- **Control de objetos**: Teclas numéricas para velocidad de rotación del cubo
- **Funciones especiales**: Reset de cámara y toggle de animaciones

Controles del Usuario

Tecla	Función
W/A/S/D	Movimiento de cámara
Flechas	Rotación de cámara
C	Reset de rotación de cámara
R	Toggle rotación del cubo transparente
Espacio	Reset rotación del cubo a posición inicial
1-4	Velocidades predefinidas de rotación

Shaders y Efectos Visuales

Shader Principal (`vertex_shader.glsl`, `fragment_shader.glsl`)

Implementa iluminación Phong completa:

- **Componente ambiental**: Iluminación base uniforme
- **Componente difusa**: Iluminación direccional basada en normales
- **Componente especular**: Reflejos brillantes con factor de shininess
- **Transformaciones**: Convierte coordenadas de objeto a espacio de vista

Shader de Hierba (`grass_vertex_shader.glsl`)

Especializado para renderizado instanciado:

- **Transformaciones por instancia**: Aplica rotación, escala y traslación únicas
- **Matriz de rotación**: Calcula rotación Y procedural
- **Combinación de colores**: Mezcla color del vértice con color de instancia

Shader Transparente (`fragment_shader_transparent.glsl`)

Extiende el shader principal con transparencia:

- **Canal alpha**: Utiliza el cuarto componente para controlar opacidad
- **Animación de transparencia**: Soporte para efectos de "respiración"

Shader de Skybox (`skybox_vertex_shader.glsl`, `skybox_fragment_shader.glsl`)

Manejo especializado de cubemaps:

- **Coordenadas de textura**: Utiliza la posición del vértice como coordenada de cubemap
- **Profundidad infinita**: Se asegura de que el skybox siempre esté en el fondo

Configuración y Compilación

Dependencias del Proyecto

El proyecto está configurado para Visual Studio 2022 con las siguientes bibliotecas:

Debug (x64):

- SDL2-staticd.lib, SDL2maind.lib
- gladd.lib
- soil2-debug.lib
- assimp-vc143-mtd.lib
- zlibstaticd.lib

Release (x64):

- SDL2-static.lib, SDL2main.lib
- glad.lib
- soil2.lib
- assimp-vc143-mt.lib
- zlibstatic.lib

Estructura de Assets

La aplicación espera encontrar los assets en rutas relativas específicas:

- **Shaders**: `../../../shared/assets/shaders/`
- **Texturas**: `../../../shared/assets/textures/`
- **Modelos**: `../../../shared/assets/models/`

Características Destacadas

Rendimiento Optimizado

- **Frustum culling**: El skybox se renderiza de forma optimizada
- **Batch rendering**: La hierba utiliza instanced drawing para máximo rendimiento
- **Memory management**: Uso consistente de RAII y smart pointers

Escalabilidad

- **Sistema modular**: Fácil añadir nuevas geometrías heredando de Mesh
- **Grafo de escena flexible**: Permite jerarquías complejas de objetos
- **Shaders intercambiables**: Sistema de shaders permite efectos personalizados

Realismo Visual

- **Iluminación física**: Implementación completa del modelo de iluminación Phong

- **Materiales procedurales**: Colores de terreno y hierba basados en altura
- **Efectos atmosféricos**: Skybox proporciona ambiente realista
- **Transparencias avanzadas**: Orden de renderizado correcto para efectos transparentes