High-Level Design Document for Physics Engine 3D Game:

Table of Contents

1. Introduction

This document provides a high-level design for a 3D engine implemented in C++. It covers the major components, their relationships, and interactions within the system. The engine demonstrates various computer graphics concepts, including rendering, lighting, shadow mapping, and physics.

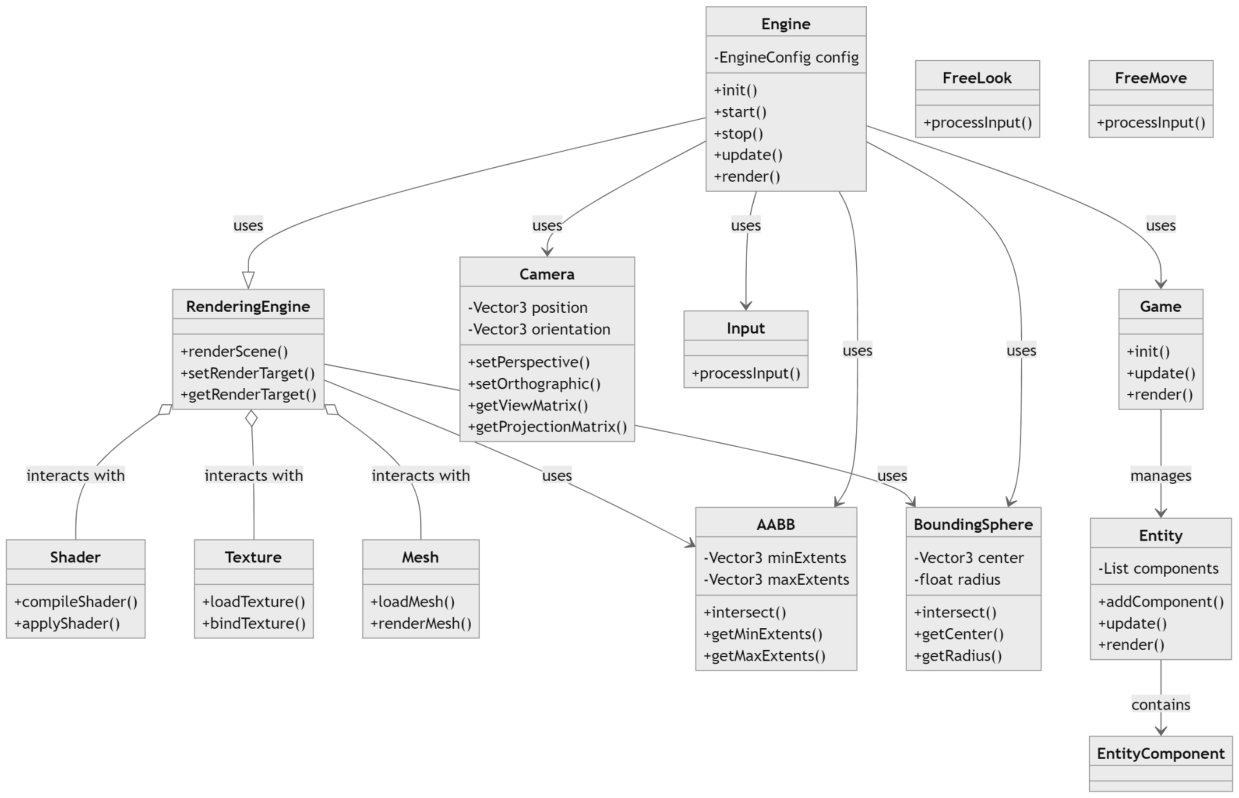
2. System Architecture

Overview

The 3D engine is composed of several subsystems that interact to provide rendering, physics, input handling, and game logic. The main subsystems include:

- **Core Engine**
- **Rendering Engine**
- **Physics Engine**
- **Input Handling**
- **Game Logic**

HLD Diagram



3. Major Components

3.1 Core Engine

- **3DEngine.h/cpp**: Manages the initialization, execution, and termination of the engine.
- **CoreEngine.h/cpp**: Contains the main game loop and engine core functions.

3.2 Rendering Engine

- **RenderingEngine.h/cpp**: Manages the rendering process and pipeline.
- **Shader.h/cpp**: Handles shader compilation and application.
- **Texture.h/cpp**: Manages texture loading and binding.
- **Mesh.h/cpp**: Handles 3D model loading and management.
- **Camera.h/cpp**: Manages camera functionality.

3.3 Physics Engine

- **AABB.h/cpp**: Handles Axis-Aligned Bounding Box (AABB) collision detection.
- **BoundingSphere.h/cpp**: Manages bounding sphere collision detection.

3.4 Input Handling

- **Input.h/cpp**: Processes user input from keyboard and mouse.
- **FreeLook.h/cpp**: Implements free look camera control.
- **FreeMove.h/cpp**: Implements free move camera control.

3.5 Game Logic

- **Game.h/cpp**: Contains game-specific logic and functions.
- **Entity.h/cpp**: Manages entities within the game.
- **EntityComponent.h**: Handles the entity-component system.

4. Class and Object Descriptions

Core Engine

- **Engine**: Primary class managing the engine's lifecycle.
  - Methods: `init()`, `start()`, `stop()`, `update()`, `render()`

Rendering Engine

- **RenderingEngine**: Manages rendering tasks.
  - Methods: `renderScene()`, `setRenderTarget()`, `getRenderTarget()`
- **Shader**: Compiles and applies shaders.
  - Methods: `compileShader()`, `applyShader()`
- **Texture**: Loads and binds textures.
  - Methods: `loadTexture()`, `bindTexture()`
- **Mesh**: Loads and renders 3D models.
  - Methods: `loadMesh()`, `renderMesh()`
- **Camera**: Manages camera properties and matrices.
  - Methods: `setPerspective()`, `setOrthographic()`, `getViewMatrix()`, `getProjectionMatrix()`

Physics Engine

- **AABB**: Handles AABB collision detection.
  - Methods: `intersect(const AABB& other)`, `getMinExtents()`, `getMaxExtents()`
- **BoundingSphere**: Manages bounding sphere collision detection.
  - Methods: `intersect(const BoundingSphere& other)`, `getCenter()`, `getRadius()`

Input Handling

- **Input**: Processes user input.
  - Methods: `processInput()`
- **FreeLook**: Implements free look control.
  - Methods: `processInput()`
- **FreeMove**: Implements free move control.
  - Methods: `processInput()`

Game Logic

- **Game**: Contains game-specific initialization and logic.
  - Methods: `init()`, `update()`, `render()`
- **Entity**: Represents game entities.
  - Methods: `addComponent(EntityComponent* component)`, `update()`, `render()`
- **EntityComponent**: Represents components of entities.

5. Data Flow

Initialization

1. The engine initializes by setting up all required subsystems (rendering, physics, input).
2. Shaders and textures are loaded.
3. Game entities and components are initialized.

Game Loop

1. **Input Processing**: Captures user input (keyboard, mouse).
2. **Update**: Updates game logic, physics, and entity states.
3. **Render**: Renders the current frame using the rendering engine.

Termination

1. Shuts down all subsystems.
2. Cleans up resources (textures, shaders, meshes).

6. Interactions and Interfaces

Core Engine and Rendering Engine

- **CoreEngine** calls `render()` on the **RenderingEngine** to draw the scene.

Core Engine and Physics Engine

- **CoreEngine** updates the physics state by calling methods on **AABB** and **BoundingSphere**.

Input Handling

- **Input** captures and processes input, passing control to **FreeLook** and **FreeMove** for camera control.

Game Logic

- **Game** manages entities and updates their states, invoking methods on **Entity** and **EntityComponent**.

7. Use Cases

Basic Rendering

1. Initialize the engine.
2. Load shaders and textures.
3. Create and configure the camera.
4. Load 3D models.
5. Enter the game loop: process input, update game state, render the scene.

Collision Detection

1. Create game entities with AABB or bounding spheres.
2. Update entity positions.
3. Check for collisions using `intersect` methods.
4. Respond to collisions (e.g., stop movement, apply damage).

Camera Control

1. Initialize the camera.
2. Process user input for free look or free move.
3. Update the camera position and orientation based on input.
4. Use the updated camera matrices for rendering.