Game Engine

DESIGN OVERVIEW

Game Engine

- Building a Game Engine is extremely complex.
- If your goal is to build a game, writing your own engine is often a waste of time. Use existing engines like Unity instead.
- Debug GUI should be built first for internal logging.
- Engine systems can be developed in parallel, not sequentially.

Entry Point

- The engine must have a well-defined **Entry Point** to initialize and start execution.
- The Entry Point defines how the engine starts and what APIs are exposed.
- In some engines (e.g., Unreal Engine 5), the Entry Point is defined using a macro.
- Client-side Entry Points are controlled by the user application.
- The Engine-side Entry Point transfers control to the client after setup but retains some internal control.
- Instead of macros, Entry Points can be generated at runtime.
- The contents and responsibilities of the Entry Point must be clearly defined.

Application Layer

- Manages the application lifecycle and core events.
- Handles the main **Run Loop** responsible for rendering and updating.
- Tracks time and runs game-specific code and state management.

Window Layer

The window layer manages the operating system window and user input:

- **Window:**
  - Applicable only to desktop platforms (Windows, macOS, Linux).

    - Window creation is the first step.
    - The window is the primary **render target** and main source of **events**.
    - The entire application interface operates through the window.
    - On other platforms (e.g., mobile), it is abstracted as a surface.
- **Input:**
    - All input is translated into events.
    - There are two input modes:
        1. **Messaging System** – reacts when events happen.
        2. **Polling System** – manually checks for state (e.g., asking "Is Jump pressed?").
    - Polling is useful for states like cursor position and falls outside the messaging system.
- **Events:**
    - Managed by an **Event Manager**, acting as a messaging and broadcast system.
    - Events are classified by type.
    - Multiple engine layers may subscribe to the same type.
    - Events are dispatched to all subscribed layers.
    - Based on **Component & Entity** architecture: the handler decides to consume or propagate the event.
    - Works on **Notification** rather than **Interrupt**.
    - Each layer implements its own OnEvent() function.
    - Notification systems are faster and more efficient than polling every frame.

Render

- Handles all actual rendering to the window.
- Performs a complete redraw every frame.
- One of the largest and most complex parts of the engine.

Render API Abstraction

- Rendering must be decoupled from any specific graphics library.
- Enables support for multiple backends (e.g., OpenGL, DirectX).
- **OpenGL** is simplest, most supported, and cross-platform.
- **DirectX** is not cross-platform.
- Design must allow for future expansion and backend switching.

Debugging Support

- Crucial for understanding what happens at runtime.
- Helps identify and trace problems inside the engine.
- Must support **variable debugging** through the GUI.
- **Profiling** tools must track time consumption per system.
- Only active in **Debug Mode**.

- Profiling data is visualized via ImGui or similar tools.

Scripting Language

- The engine must support script loading, parsing, and execution.
- **C#** is preferred for artists due to its managed environment and abstraction from memory management.

Memory Systems

- Memory is a limited resource in real-time engines.
- The engine must implement a custom memory system with tracking and custom allocators.
- Memory management helps prevent out-of-memory errors and performance bottlenecks.
- **Memory tracking** is active in Debug Mode.
- Must measure and minimize both **memory usage** and **allocation time**.

Entity Component System (ECS)

- Engine uses ECS to define entities as sets of components.
- Components determine entity behavior.
- Examples:
  - Scripting Component: AI or manual control.
  - Physics Component: mass, velocity, etc.
- ECS defines how the engine understands and processes game data.
- ECS is loaded and used during scene runtime.

Physics

- Implemented as part of ECS via attachable components.
- Defines physical behaviors (e.g., mass, forces) per entity.

File I/O, VFS

- Not directly detailed, but implied via asset transformation and memory usage.
- Engine must support loading, parsing, and processing binary engine-friendly assets.

Build System

- Responsible for converting raw assets to engine-ready formats.
- Artists provide files like .fbx, .png, which are unsuitable for direct use.
- Asset transformation must be done **offline** to reduce runtime overhead.
- Final engine assets are parsed and stored in a **binary** format.
- The system should support **hot-reload at runtime** with caching, limited to a small number of assets at a time.
- Any asset update triggers a **rebuild** automatically.