Technical Design Document

csd2400f24-csd2401f24-uxg2400f24-daa2402f24

Fall 2024

- **Group Members**-

Table of Contents

[1. Overview 4](#_Toc196258494)

[1.1 Project Brief 4](#_Toc196258495)

[1.2 Engine Architecture Overview 4](#_Toc196258496)

[1.3 Managers 4](#_Toc196258497)

[1.4 Entity Component System (ECS) 5](#_Toc196258498)

[1.5 Utilities 6](#_Toc196258499)

[2. ECS Implementation 8](#_Toc196258500)

[2.1 Overview 8](#_Toc196258501)

[2.2 Entities 8](#_Toc196258502)

[2.2.1 Code Implementation 9](#_Toc196258503)

[2.2.2 Creating Entity Explanation and Example 9](#_Toc196258504)

[2.2.3 Creating Entity Explanation and Example 10](#_Toc196258505)

[2.3 Systems 13](#_Toc196258506)

[2.3.1 Creating System Explanation and Example 13](#_Toc196258507)

[2.3.2 Deletion System Explanation and Example 14](#_Toc196258508)

[2.3.3 How Systems Interact with Components and Entities 14](#_Toc196258509)

[2.4 Components 15](#_Toc196258510)

[2.4.1 Component Main Storage 15](#_Toc196258511)

[2.4.2 Component Identification 16](#_Toc196258512)

[2.4.2 Creating Component Explanation and Example 16](#_Toc196258513)

[2.4.4 Deletion Component Explanation and Example 20](#_Toc196258514)

[2.4.5 How Components Interact with Systems and Entities 21](#_Toc196258515)

[3. Coding Methods 23](#_Toc196258516)

[3.1 Naming Conventions 23](#_Toc196258517)

[3.2 File Naming Convention 23](#_Toc196258518)

[3.3 File Location 24](#_Toc196258519)

[3.4 Code Documentation 24](#_Toc196258520)

[3.5 Source Version Control 24](#_Toc196258521)

[4. Debugging Implementation 24](#_Toc196258522)

[4.1 Overview 24](#_Toc196258523)

[5. Tools 25](#_Toc196258524)

[5.1 Editing Tools 25](#_Toc196258525)

[5.2 External Libraries 25](#_Toc196258526)

[5.3 Compiler 25](#_Toc196258527)

[5.4 Other Third-Party Tools 25](#_Toc196258528)

[6. Scripting Language 26](#_Toc196258529)

[6.1 C++ Language 26](#_Toc196258530)

# 1. Overview

## 1.1 Project Brief

The GAM300 game engine is a modular, component-based engine designed for efficient game development. It employs an Entity Component System (ECS) architecture to promote separation of concerns, data-oriented design, and enhanced performance through cache-friendly operations. The engine follows modern game engine design principles, emphasizing:

Key features of the engine include:

* A complete Entity Component System implementation
* Singleton managers for various engine subsystems
* Comprehensive math and utility libraries
* Precise timing mechanisms for stable game loop execution
* Efficient memory management with component pools
* Logging capabilities for debugging and diagnostics

## 1.2 Engine Architecture Overview

The GAM300 engine is built around several core subsystems:

1. **Manager System**: Singleton managers that handle specific engine subsystems
2. **Entity Component System (ECS)**: Core architecture for game object representation and logic
3. **Math Utilities**: Vector and mathematical operations supporting game logic
4. **Timing Utilities**: Clock and timing mechanisms for frame rate management and game loop timing

The engine uses a main game loop controlled by the GameManager. This loop maintains a consistent update rate, processes input, updates all active systems, and renders the game state.

## 1.3 Managers

The engine utilizes several singleton managers to handle different aspects of the game's functionality:

**GameManager (GM)**

* Central controller for the engine
* Manages the main game loop
* Controls frame timing and rate
* Handles game state (start/stop)

**LogManager (LM)**

* Handles debug and error logging
* Writes timestamped messages to log files
* Supports formatted log output

**ECSManager (EM)**

* High-level interface to the ECS
* Manages entity creation and destruction
* Coordinates component attachment and removal
* Registers and updates systems

**ComponentManager (CM)**

* Manages component storage and retrieval
* Maintains type-specific component arrays
* Handles efficient component memory allocation

**SystemManager (SM)**

* Manages all systems in the ECS
* Handles system registration and prioritization
* Updates systems each frame
* Notifies systems about entity changes

Each manager follows a standard pattern with startUp() and shutDown() methods, and they are accessed through global macros (GM, LM, EM, CM, SM) for convenience.

## 1.4 Entity Component System (ECS)

**Core Concepts**

The Entity Component System (ECS) is an architectural pattern that separates:

* **Entities**: Game objects represented as unique IDs
* **Components**: Data containers attached to entities
* **Systems**: Logic processors that operate on entities with specific components

This separation provides several advantages:

* Composition over inheritance
* Better code organization and reusability
* Cache-friendly data access patterns
* Easier parallelization

**Implementation Details**

In our engine, the ECS is implemented as follows:

**Entities**:

* Lightweight representations of game objects
* Each entity has a unique EntityID
* Contain a component mask (bitset) indicating attached components

**Components**:

* Data containers attached to entities
* Store state, but generally not behavior
* Managed in type-specific component pools for efficient memory access

**Systems**:

* Process entities with specific component combinations
* Contain game logic that operates on components
* Are prioritized to ensure correct update order

The relationships between these elements are managed by the ECSManager, which provides a high-level interface for creating entities, attaching components, and registering systems.

## 1.5 Utilities

The engine provides several utility classes to support game development:

**Vector Mathematics**

* Vector2D: 2D vector operations
* Vector3D: 3D vector operations
* Support for common vector operations (addition, normalization, dot product, etc.)

**Math Utilities**

* Common mathematical constants (PI, etc.)
* Angle conversion functions
* Interpolation functions
* Random number generation

**Timing Utilities**

* Clock class for high-precision timing
* Supports delta time calculation for frame-independent movement
* Handles microsecond precision

# 2. ECS Implementation

## 2.1 Overview

The Entity Component System (ECS) implementation follows a data-oriented approach, separating entities (game objects), components (data), and systems (logic). This separation provides several advantages:

Composition over inheritance: Game objects are constructed by combining components

Data locality: Similar components are stored together for cache-friendly access

Parallel processing: Systems can operate independently on different component types

Decoupled architecture: Changes to one part of the system minimally impact others

The ECS is managed through three main classes:

* ECSManager: High-level interface for the entire system
* ComponentManager: Handles component storage and retrieval
* SystemManager: Manages system registration and updates

These managers coordinate to ensure proper entity lifecycle management, component attachment, and system execution.

## 2.2 Entities

In the current LOF ECS implementation, entities are represented as lightweight objects with unique integer identifiers (EntityIDs) and optional names. These entities do not store any data directly but act as references to various components attached to them. The lightweight design allows for the efficient creation, destruction, and management of large numbers of entities.

Each entity in the ECS architecture also has an associated name, which provides a human-readable identifier for debugging, scene management, or gameplay logic. The name can be assigned when the entity is created and updated dynamically during the game. This feature is especially useful when loading entities from scene files or identifying specific game objects during runtime.

Additionally, each entity maintains a component mask (or bitmask) that indicates which components are associated with it. This mask enables systems to quickly determine whether an entity has the necessary components required for processing. The EntityID and its component mask are used to associate components with entities via component storage.

When an entity is destroyed, its ID and associated name are released, and any components linked to that entity are removed from their respective component stores. This ensures that unused entities and components do not occupy unnecessary memory or processing resources.

### 2.2.1 Code Implementation

Entities in the engine are implemented in the Entity class (Entity.h/Entity.cpp):

A screen shot of a computer program

AI-generated content may be incorrect.

Key aspects of the Entity implementation:

1. **EntityID**: A 32-bit unsigned integer providing a unique identifier
2. **ComponentMask**: A bitset where each bit represents a component type
3. **Minimal Storage**: Entities don't store component data directly

The ComponentMask is crucial for quick determination of which components an entity has, allowing systems to efficiently filter entities they should process.

### 2.2.2 Creating Entity Explanation and Example

Entities are created through the ECSManager using the createEntity method. This ensures proper ID assignment and registration with all necessary systems.

Creation Process:

1. ECSManager assigns a unique ID
2. Entity is added to the internal entity list
3. SystemManager is notified of the new entity
4. Systems check if the entity matches their requirements

Example:

A screenshot of a computer program

AI-generated content may be incorrect.

The createEntity method returns a reference to the newly created entity, which can be used immediately or stored by ID for later reference.

### 2.2.3 Creating Entity Explanation and Example

Entities are deleted through the ECSManager using the destroyEntity method. This ensures proper cleanup of all associated components and system references.

**Deletion Process:**

1. ECSManager finds the entity by ID
2. SystemManager is notified to remove the entity from all systems
3. ComponentManager is notified to remove all components owned by the entity
4. The entity is removed from the ECSManager's entity list

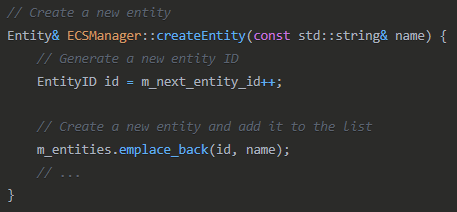
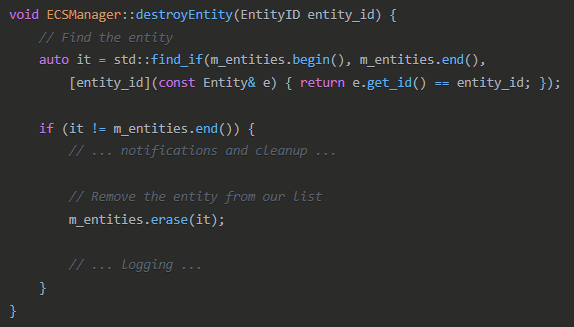
**Example:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Entity ID upon deletion:**

When you delete an entity, its ID is not recycled or reused. Here's what happens:

1. In the ECSManager class, entity IDs are assigned sequentially using an incrementing counter:
2. When an entity is deleted via destroyEntity(), the entity is removed from the vector, but nothing is done with its ID:  
   
3. The m\_next\_entity\_id counter is never decremented or reset, and there's no pool of "recycled" IDs.

**Key Points:**

1. **No ID Recycling**: Entity IDs are not pushed back or recycled when entities are deleted
2. **ID Uniqueness**: This ensures that every entity created during the lifetime of the application has a unique ID
3. **No ID Compaction**: The remaining entity IDs stay exactly as they were - there's no reordering or compaction
4. **Sequential Assignment**: New entities always get the next sequential ID from m\_next\_entity\_id

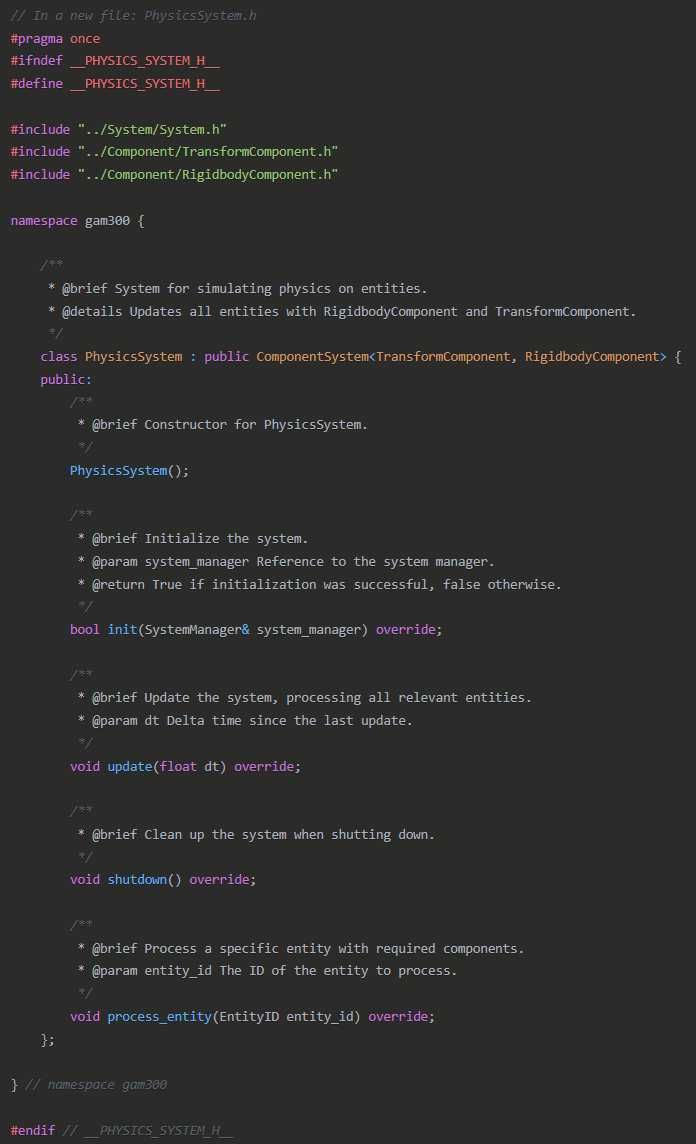
**Implications:**

* This approach is simple and prevents issues with dangling references to deleted entities
* Since EntityID is a 32-bit unsigned integer (std::uint32\_t), the system can create up to ~4 billion unique entities before overflow
* If your game creates and destroys many entities over a long period, the IDs will continue to increase
* Entity IDs should be treated as opaque identifiers, not as indices into an array

## 2.3 Systems

**Systems** define the logic that operates on the data held by components. Each system is responsible for updating entities that match a specific component signature.

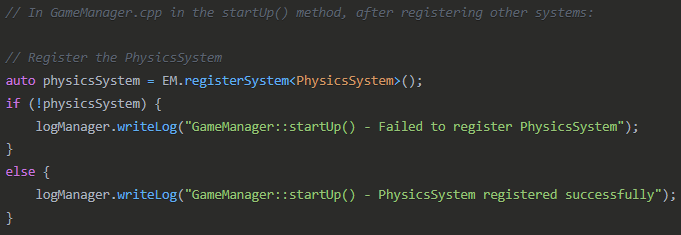
### 2.3.1 Creating System Explanation and Example

**1. Create System Header File  
**

**2. Create System Implementation File**

****

### **3. Register the System in ECSManager**



**4. Include System Header in GameManager**

****

### 2.3.2 Deletion System Explanation and Example

Systems are managed by the SystemManager and typically exist for the duration of the game. However, systems can be disabled or removed if needed.

**System Deactivation:** To temporarily disable a system without removing it:

A screen shot of a computer program

AI-generated content may be incorrect.

### 2.3.3 How Systems Interact with Components and Entities

Systems form the processing layer of the ECS architecture, operating on entities that have specific component combinations. The interaction between systems, components, and entities is managed through several mechanisms:

**Entity Filtering:**

* Each system maintains a ComponentMask indicating which component types it requires
* The SystemManager checks if an entity's component mask satisfies the system's requirements
* Only matching entities are added to a system's processing list

**Component Access:**

* Systems access components through the ComponentManager:



* Systems can also use ComponentView for more efficient iteration:

A screen shot of a computer

AI-generated content may be incorrect.

**Event Notifications:** Systems are notified of entity lifecycle events through the SystemManager:

1. entity\_created: When a new entity is created
2. entity\_destroyed: When an entity is destroyed
3. entity\_components\_changed: When components are added/removed from an entity

This notification system ensures that systems always operate on the correct set of entities and don't process entities that no longer exist or no longer have the required components.

## 2.4 Components

### 2.4.1 Component Main Storage

Components are stored in type-specific pools managed by the ComponentManager. This design ensures:

1. **Data Locality**: Components of the same type are stored contiguously in memory
2. **Efficient Iteration**: Systems can iterate through components efficiently
3. **Fast Access**: O(1) component access by entity ID

The storage is implemented through the ComponentPool template class, which maintains:

* A dense array of components (std::vector<T>)
* A mapping from entity IDs to component indices (std::unordered\_map<EntityID, size\_t>)
* A mapping from component indices to entity IDs (std::unordered\_map<size\_t, EntityID>)

When a component is removed, the last component in the array is moved to fill the gap, maintaining data contiguity.

### 2.4.2 Component Identification

Each component type is assigned a unique ID at compile time using template metaprogramming:

A computer screen shot of a code

AI-generated content may be incorrect.

This ID is used to:

1. Set bits in the entity's ComponentMask
2. Identify component arrays in the ComponentManager
3. Check if an entity has a specific component type

The system supports up to MAX\_COMPONENTS (64) different component types, as defined in ECS\_Variables.h.

### 2.4.2 Creating Component Explanation and Example

**1. Create Component Serializer Class Declaration**

**Implementation:**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**2. Add Accessor Methods to Component Class**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**3. Implement Serialization Method**

**A computer screen shot of a program

AI-generated content may be incorrect.**

**4. Implement Deserialization Method**

Component\* TransformComponentSerializer::deserialize(EntityID entityId, const std::string& jsonData) {

**A screen shot of a computer program

AI-generated content may be incorrect.**

**5. Extend extractSection Method (if needed)**

**Implementation**: Since TransformComponent uses nested objects rather than arrays, you might need to extend the extraction method:

A screen shot of a computer program

AI-generated content may be incorrect.

**6. Update SerialisationManager::startUp()**A screen shot of a computer program

AI-generated content may be incorrect.

**7. Update saveScene() to Include the New Component**

A computer screen shot of a program code

AI-generated content may be incorrect.

**8. Include Necessary Headers**

****

### 2.4.4 Deletion Component Explanation and Example

Components are removed from entities through the ECSManager:

**Component Removal Process:**

1. ECSManager updates the entity's component mask
2. ComponentManager removes the component from its storage
3. SystemManager is notified of the component change
4. Systems check if the entity still matches their requirements

**Example:**

A screen shot of a computer program

AI-generated content may be incorrect.

When a component is removed:

1. The component's memory is freed
2. The last component in the pool is moved to fill the gap (if needed)
3. The entity-to-index and index-to-entity maps are updated

This ensures efficient memory usage and maintains data contiguity within component pools.

### 2.4.5 How Components Interact with Systems and Entities

Components form the data layer of the ECS architecture, storing the state of game objects. The interaction between components, systems, and entities is managed through several mechanisms:

**Component-Entity Relationship:**

* Each component stores its owner entity ID
* Entities track which components they have via the component mask
* Components are accessed through their owner entity ID

**Component-System Relationship:**

* Systems process entities based on their component requirements
* Systems read and modify component data to implement game logic
* Components don't contain behavior; they're processed by systems

**Data Flow:**

1. Entities aggregate components
2. Components store data
3. Systems process that data to implement game behavior

This separation ensures a clean architecture where data (components) and behavior (systems) are distinct, allowing for more maintainable and flexible code.

# 3. Coding Methods

## 3.1 Naming Conventions

**Classes and Types:**

* PascalCase for class names: Entity, ComponentManager, Vector3D
* Underscores for type aliases: using EntityID = std::uint32\_t;

**Variables:**

* camelCase for member variables: m\_position, m\_owner\_id
* snake\_case for local variables: entity\_id, component\_array

**Methods and Functions:**

* camelCase for methods in classes: getPosition(), setRotation()
* snake\_case for standalone functions: get\_component\_type\_id()

**Constants:**

* UPPER\_CASE for constants: MAX\_COMPONENTS, INVALID\_ENTITY\_ID

**Global Singleton Accessors:**

* Two-letter uppercase macros: EM, CM, GM, LM, SM

## 3.2 File Naming Convention

Files in the engine follow consistent naming conventions:

**Header Files:**

* Use .h extension
* Match the class name: Entity.h, Vector3D.h, ComponentManager.h

**Implementation Files:**

* Use .cpp extension
* Match the corresponding header: Entity.cpp, Vector3D.cpp, ComponentManager.cpp

**File Naming Style:**

* PascalCase for files containing a single class: Vector3D.h
* snake\_case for utility files containing multiple functions: math\_utils.h

## 3.3 File Location

The engine organizes files into a structured directory layout:

This organization groups related files together and separates different aspects of the engine into logical directories:

Entity/: Entity-related files

Component/: Component-related files

System/: System-related files

Manager/: Manager classes that control various engine subsystems

Utility/: Utility classes and functions used throughout the engine

Assets/: Assets files

## 3.4 Code Documentation

Doxygen-style comments are used throughout the codebase, providing clear documentation for classes, functions, and methods. Complex logic is further explained using inline comments. This approach helps in both understanding and maintaining the code.

## 3.5 Source Version Control

* **GitHub**: GitHub is used for version control, ensuring all changes are tracked, reviewed, and stored securely. This allows for efficient collaboration and version tracking.
* **Telegram**: Telegram is used for team communication and issue management, where team members discuss bugs, new features, and coordinate tasks.

# 4. Debugging Implementation

## 4.1 Overview

Debugging in the project is primarily facilitated through the Log\_Manager, which records detailed logs of key events, errors, and system states to a designated log file. This includes both routine operations and exceptional events, ensuring that developers can trace the flow of execution.

When a crash occurs, the system catches exceptions and generates crash reports, including relevant system state information. The combination of regular logs and crash reports allows for easier identification of issues and more efficient debugging.

Additionally, inline assertions are used in critical sections of the code to catch potential issues during development before they escalate into larger bugs.

# 5. Tools

## 5.1 Editing Tools

The project is developed using **Visual Studio** and **VS Code** for writing, editing, and compiling the codebase.

## 5.2 External Libraries

* **GLFW**: Handles window creation and input events.
* **GLAD**: Manages the loading of OpenGL function pointers.
* **RapidJSON**: Used for fast JSON parsing and serialization/deserialization of game objects and components.
* **FMOD**: Handles the playback of audio from sound files.

## 5.3 Compiler

The project is compiled using the **ISO C++17** standard, enabling modern C++ features to improve code efficiency and readability.

## 5.4 Other Third-Party Tools

* **Github**: Utilized for source control, tracking changes and collaboration.
* **Inno Setup Compiler**: Used to create the installer script and generate the setup.exe file
* **Telegram**: Used for communication among team members, providing a platform for issue discussion and resolution.
* **Discord**: Used for team communication, offering screen sharing and a platform for issue discussion and resolution.

# 6. Scripting Language

## 6.1 C++ Language

The entire engine is built using C++, utilizing modern C++ features such as smart pointers, templates, and standard containers like `std::unordered\_map`.