Software Design Document

for

Touhou Fan Game

Team: Syntax Soil

Project: Touhou Fan Game

Team Members:
Ross Kugler
Linh (Jason) Nguyen
Huy (Harry) Ky
Ben Bordon
Toufic Majdalani
Dylan Gyori

Table of Contents

Document Revision History

Revision Number	**Revision Date**	**Description**	**Rationale**
1.0	3/23/2025	Milestone 2 Deliverable	Completing UML diagram for architecture, Sequence Diagrams, and Subsystem descriptions.

List of Figures

1. Introduction

The *Touhou Project* (or simply *Touhou*) is a bullet hell shoot-'em-up series developed by Team Shanghai Alice (formerly ZUN), renowned for its intense gameplay, intricate bullet patterns, and precise movement mechanics. Players navigate through waves of enemy projectiles while utilizing unique abilities to defeat formidable bosses. The series is highly praised for its challenging mechanics, iconic soundtrack, and vibrant cast of characters, making it a defining influence in the bullet hell genre.

Inspired by *Touhou Youyoumu ~ Perfect Cherry Blossom*, the seventh game in the series, our project's goal is to recreate its experience while exploring the addition of original mechanics and features. *Touhou 7* is arcade-like. It has a score counter, and the player’s goal is to maximize the score for a level. While creating this game, we have a strong focus on following good design patterns and architecture. Having good architecture allows us to easily swap out assets, stats, data, etc., so that in the future, we can possibly create a much more unique game that still functions similar.

1.1 Architectural Design Goals

1.1.1 Performance

Bullet hell games require real-time responsiveness to handle numerous projectiles, enemies, and collision checks while maintaining a stable frame rate. Our architecture prioritizes efficient processing and resource management to minimize latency, prevent frame drops, and sustain smooth gameplay.

Our approach follows two key performance tactics: Control Resource Demand and Manage Resources.

Control Resource Demand

- **Reduce Overhead:** The Entity-Component-System (ECS) pattern ensures that only active entities are processed, reducing unnecessary computations.
- **Limit Event Response:** Bullet and enemy spawn rates are regulated to balance computational load.
- **Prioritize Events:** Rendering and physics calculations focus on onscreen entities, optimizing performance.

Manage Resources

- **Introduce Concurrency:** Independent collision detection, movement updates, and rendering improve execution efficiency.
- **Increase Resource Efficiency:** Object pooling reuses Bullet entities, as opposed to creating new objects every time, minimizing memory allocation overhead.

- **Schedule Resources:** Bullet patterns and AI behavior updates follow fixed time steps, reducing processing jitter.
- **Level Manager:** Our individual levels are structured as JSON files, serving as a centralized data source for level configurations. These files will encapsulate essential gameplay elements, including enemy wave compositions, spawn timings, and boss encounter details. By externalizing level data, we enable seamless adjustments, easier scalability, and potential modding support, ensuring a flexible and maintainable game design.

Concrete Scenario (Performance Figure Representation)

- **Source:** Player inputs and enemy bullet patterns.
- **Stimulus:** A player action or enemy attack triggers an event (e.g., firing bullets).
- **Artifact (Environment):** The game engine updates the simulation, handling movement, collision detection, and rendering.
- **Response:** The system processes the update within a strict time budget.
- **Response Measure:** The game maintains an average frame rate of 60 FPS and input-to-render latency below 16ms to ensure smooth gameplay.

1.1.2 Testability

A modular and testable codebase is essential for efficient debugging and long-term maintainability. The ECS pattern enhances testability by keeping components separate, making it easier to isolate and verify behaviors.

Our approach follows two key testability tactics: Control and Observe System State and Limit Complexity.

Control and Observe System State

- **Record/Playback Testing:** We intend to implement the Command Pattern into our game eventually. If we have the ability to undo commands, we can do playback, which would be extremely useful for testing.
- **State Observation & Debugging Tools:** Hitbox visualization, AI behavior logging, and Bullet trajectory tracing allow real-time monitoring. We have a DebugRenderingSystem that can render these elements when turned on.

Limit Complexity

- **Reduce Coupling with ECS:** Separating concerns (e.g., movement, AI, and rendering) reduces dependencies, making testing easier.
- **Minimize Nondeterminism:** Bullet patterns and AI behavior follow predictable rules defined in JSONs, ensuring consistent testing.

Concrete Scenario (Testability Figure Representation)

- **Source:** Play-testers and debugging tools.
- **Stimulus:** Completion of a game system (e.g., movement, collision, AI) or integration of a new feature.
- **Artifact (Environment**): Game subsystems (e.g., movement system, bullet spawning system) tested in a playable environment.
- **Response:** The system displays debugging output (e.g., bullet trajectories, AI state changes).
- **Response Measure:** Play-testers encounter 0 bugs within 1 hour of gameplay.

2. Software Architecture

The following are three key use cases in our game, illustrated with sequence diagrams.

1. AI Spawns Enemies

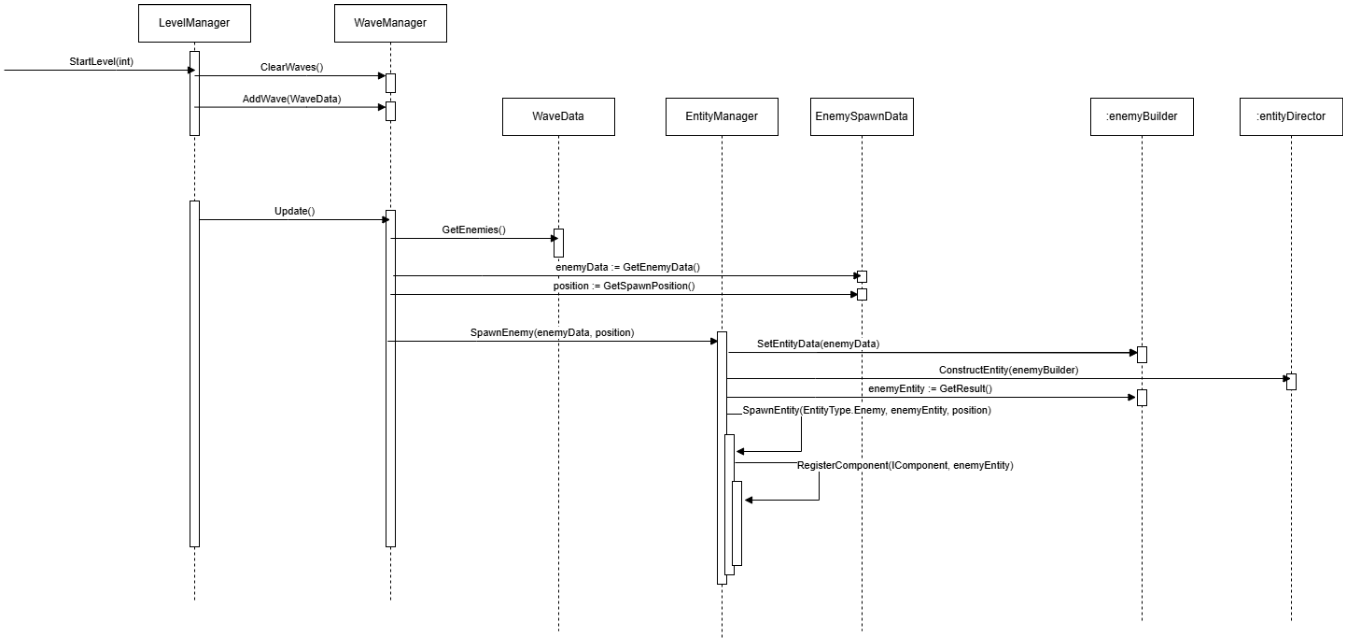

LevelManager
WaveManager
StartLevel(int)
ClearWaves()
AddWave(WaveData)
WaveData
EntityManager
EnemySpawnData
:enemyBuilder
:entityDirector
Update()
GetEnemies()
enemyData := GetEnemyData()
position := GetSpawnPosition()
SpawnEnemy(enemyData, position)
SetEntityData(enemyData)
ConstructEntity(enemyBuilder)
enemyEntity := GetResult()
SpawnEntity(EntityType.Enemy, enemyEntity, position)
RegisterComponent(IComponent, enemyEntity)

Fig 2.1 – Sequence Diagram: AI Spawns Enemies

2. User Controls Player Character to Move

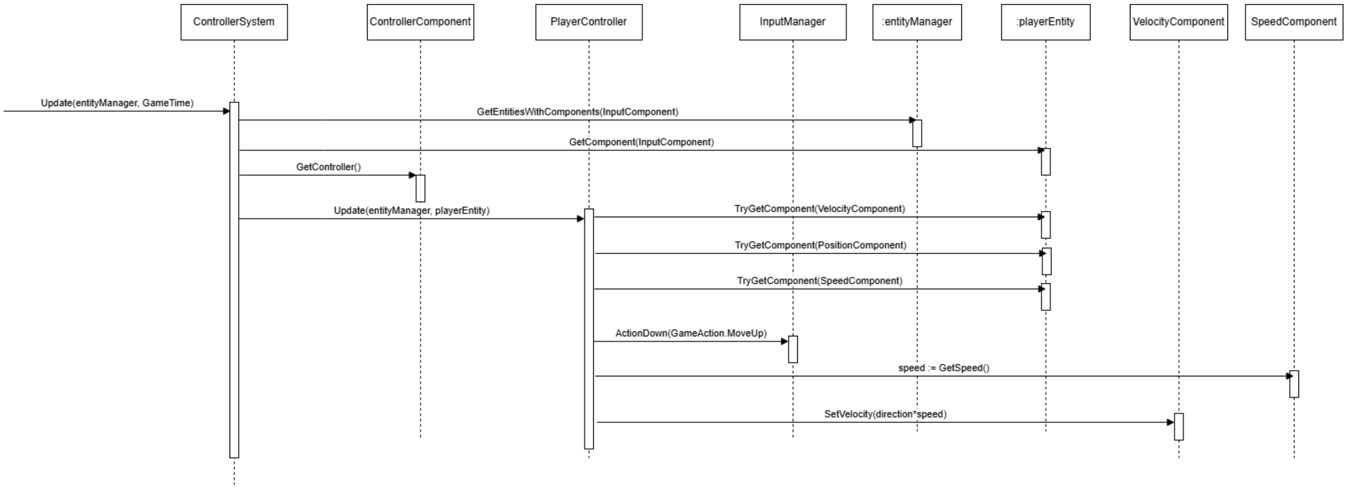

ControllerSystem
ControllerComponent
PlayerController
InputManager
:entityManager
:playerEntity
VelocityComponent
SpeedComponent
Update(entityManager, GameTime)
GetEntitiesWithComponents(InputComponent)
GetComponent(InputComponent)
GetController()
Update(entityManager, playerEntity)
TryGetComponent(VelocityComponent)
TryGetComponent(PositionComponent)
TryGetComponent(SpeedComponent)
ActionDown(GameAction.MoveUp)
speed := GetSpeed()
SetVelocity(direction*speed)

Fig 2.2 – Sequence Diagram: User Controls the Player Character to Move

3. Player Character is Hit by Enemy Bullets

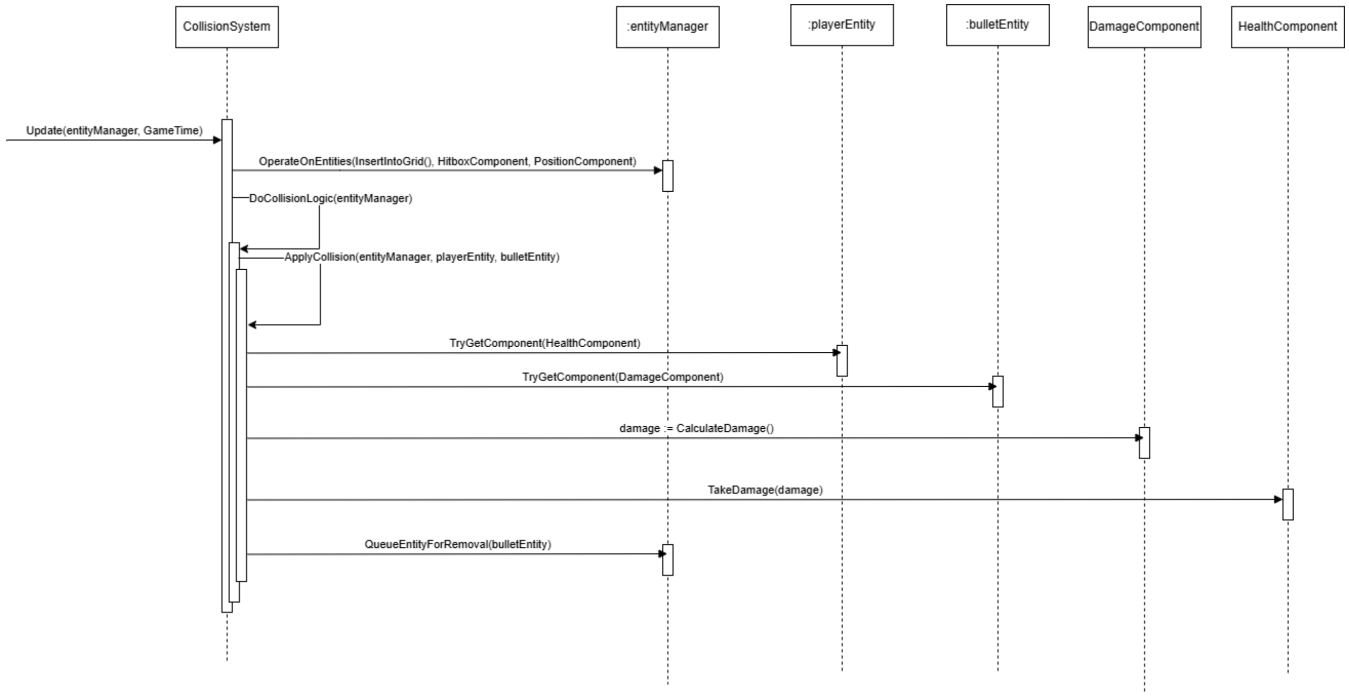

CollisionSystem
:entityManager
:playerEntity
:bulletEntity
DamageComponent
HealthComponent
Update(entityManager, GameTime)
OperateOnEntities(InsertIntoGrid(), HitboxComponent, PositionComponent)
DoCollisionLogic(entityManager)
ApplyCollision(entityManager, playerEntity, bulletEntity)
TryGetComponent(HealthComponent)
TryGetComponent(DamageComponent)
damage := CalculateDamage()
TakeDamage(damage)
QueueEntityForRemoval(bulletEntity)

Fig 2.3 – Sequence Diagram: Player Character is Hit by Enemy Bullets

2.1 Overview

Game Architecture: Multi-layered, Entity-Component System

Description: Our software architecture is an open multi-layered architecture combined with ECS. We separate our system into multiple layers: Presentation, Logic, Data-Access, and Data. Our system also uses ECS, where Bullets, Enemies, Collectibles, etc. are Entities that have a list of Components that hold necessary data, and there are systems that operate on the entities with certain components.

Rationale: We chose multi-layered architecture as it is easily compatible with our ECS design. It also allows us to easily separate the subsystems dealing with JSONs, JSON loading, actual game logic, and display. This promotes encapsulation/cohesion and reduces coupling. We also made our architecture open (higher layers can directly access several layers below) because for a game, one of our main focuses is performance. Our scenes, which are on the presentation layer, can skip the logic layer and go to the data-access layer for loading data required for a scene.

Breakdown: The system can be broken into 4 layers.

1. Presentation
    - Involves Scenes and UI classes. We construct scenes which all have Draw functions to display everything to the screen. The UI classes are utilized in scenes as reusable elements.
2. Logic
    - All the logic for managing, updating, and the game is in this layer. This layer is where the ECS happens, where Systems operate on Entities with Components.
3. DataAccess
    - This layer is where we have DataLoader classes for reading and constructing an object from a JSON.
4. Data
    - This layer is where just JSON files are.

Subsystems: The overall game subsystems we have are listed below.

1. Entity Logic
2. Entity Rendering
3. Scene Management
4. Data Loading
5. Audio
6. Input Handling

Architecture UML diagram:

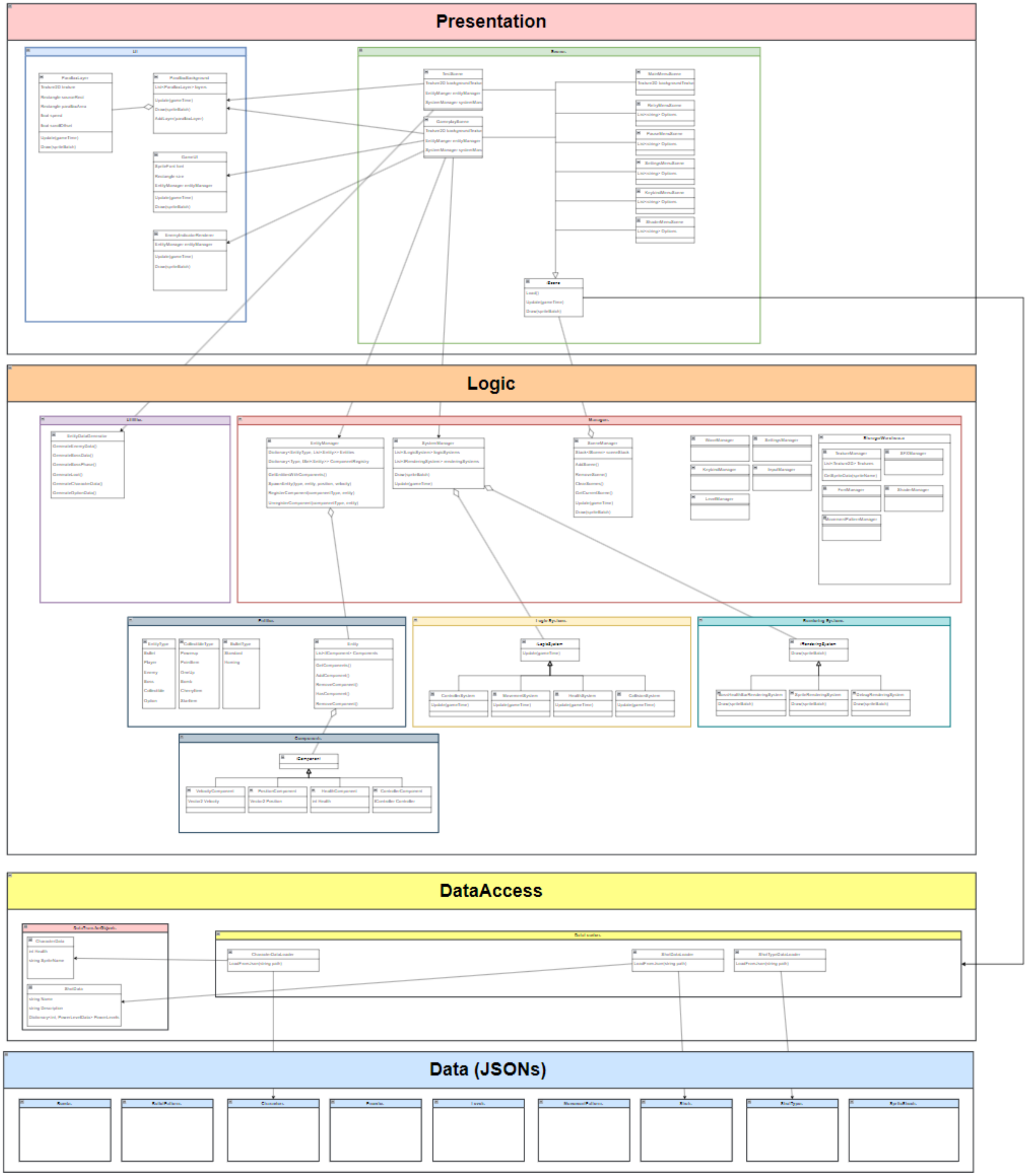


Fig 2.1.1 – Architecture UML Diagram

2.2 Subsystem Decomposition

Below we describe the different subsystems of our game in more detail.

2.2.1 - Entity Logic Subsystem

How each entity's component data is updated. In Game1's Update() function, SystemManager goes through all the LogicSystems and calls their Update() function. Each individual system has its own responsibilities. For example, the MovementSystem only handles updating each entity's position component data based on their velocity component.

2.2.2 - Entity Rendering Subsystem

Gets the Entities on the screen. In Game1's Draw() function, SystemManager's Draw() function is called. The system manager goes through all the rendering systems and calls their individual draw function. Each rendering system has its own responsibility. For example, the SpriteRenderingSystem finds all entities with a SpriteComponent, and draws it based on the data in the sprite component.

2.2.3 - Scene Management Subsystem

Handled by the SceneManager and IScenes. The SceneManager keeps a stack of scenes. The top of the stack is the scene that is currently active, and only that scene is Updated(). We can control which scenes are drawn as well (if the top scene is a pause scene, draw the scene below it too).

2.2.4 - Data Loading Subsystem

Handled by DataLoader classes that read from JSONs and deserialize the data into objects. Scenes can utilize the DataLoader classes to load up the necessary EntityData to spawn the player character, for example.

2.2.5 - Audio Subsystem

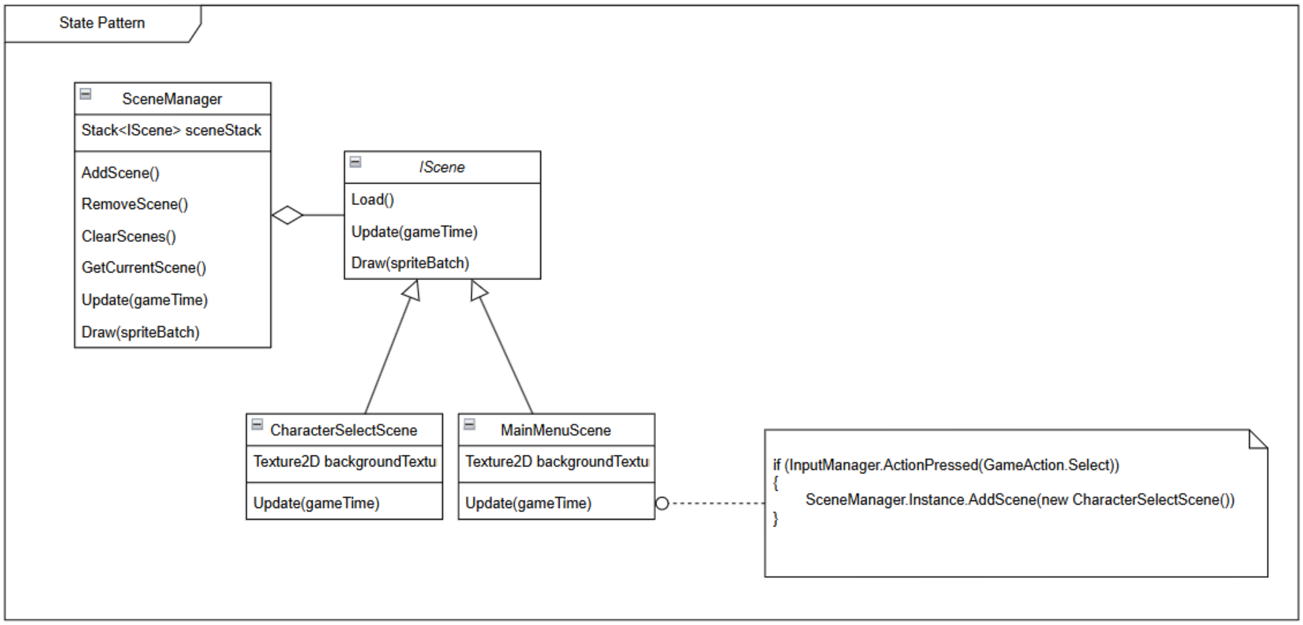
Handled by a BGMManager and SFXManager. ILogicSystems can make calls to the SFXManager in their update functions to play specific sound effects. Scenes can make calls to BGMManager to play a song (like the main menu has a theme that should be playing, the GameplayScene should play a song related to the leveldata).

2.2.6 - Input Handling Subsystem

How the user can press keys to navigate menus and play the game. An InputManager keeps track of what keys are pressed. Keys are also mapped to GameActions (as opposed to literal key presses) so that keys can be rebound easily.

2.2.7 - Design Patterns

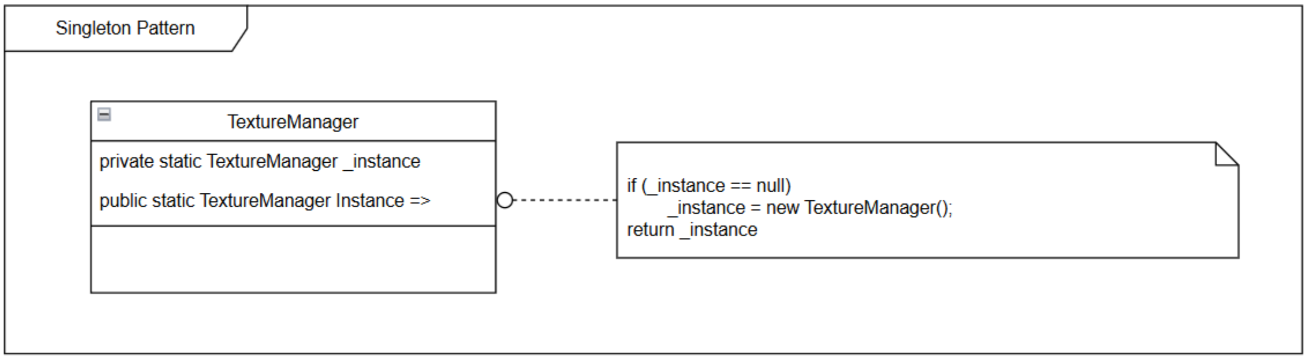
1. State – Scenes act as States, StateManager acts as the Context in the pattern. Scenes know about other scenes, and they determine the logic to switch to another scene. StateManager just Updates and Draws the current scene.



```
if (InputManager.ActionPressed(GameAction.Select))
{
        SceneManager.Instance.AddScene(new CharacterSelectScene())
}
```

Figure 2.2.7.1 – State Pattern Diagram

2. Singleton – TextureManager is a Singleton. We have many other managers that are singletons (FontManager, InputManager, etc.) but this is one example. There is only one instance of TextureManager at a time, and that instance is the one place to go for textures.


Singleton Pattern
TextureManager
private static TextureManager _instance
public static TextureManager Instance =>

```
if (_instance == null)
        _instance = new TextureManager();
return _instance
```

Figure 2.2.7.2 – Singleton Pattern Diagram

3. Flyweight – TextureManager also acts somewhat as a Flyweight, as it stores game textures in a centralized location. There is only one instance of an individual texture (the spritesheets in our case), other classes make a reference to that instance. Our SpriteComponent class acts as a ConcreteFlyweight since it references a texture but adds on an extrinsic state (the current animation, the source rect of where the texture is on the sprite sheet, etc.)

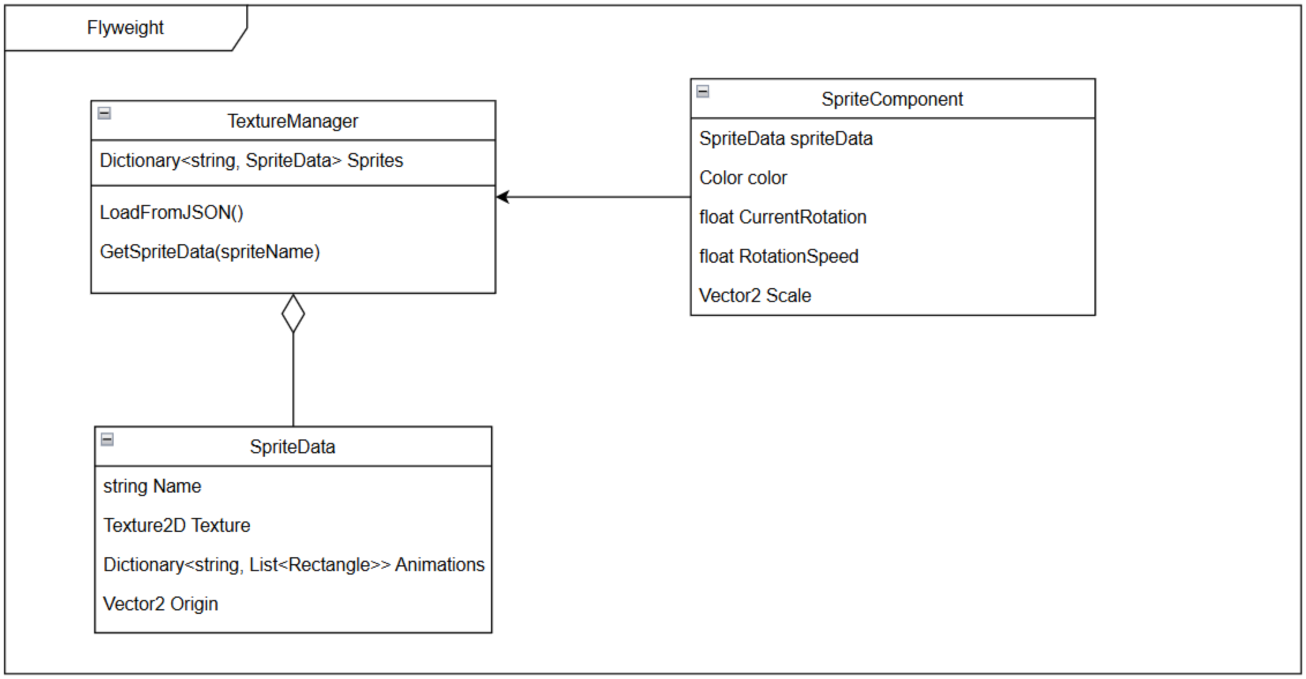


Figure 2.2.7.3 – Flyweight Pattern Diagram

4. Builder – we have multiple concrete EntityBuilder classes that construct an Entity by adding components to it based on data it is given. EntityDirector tells how to construct the Entity with a certain Builder.

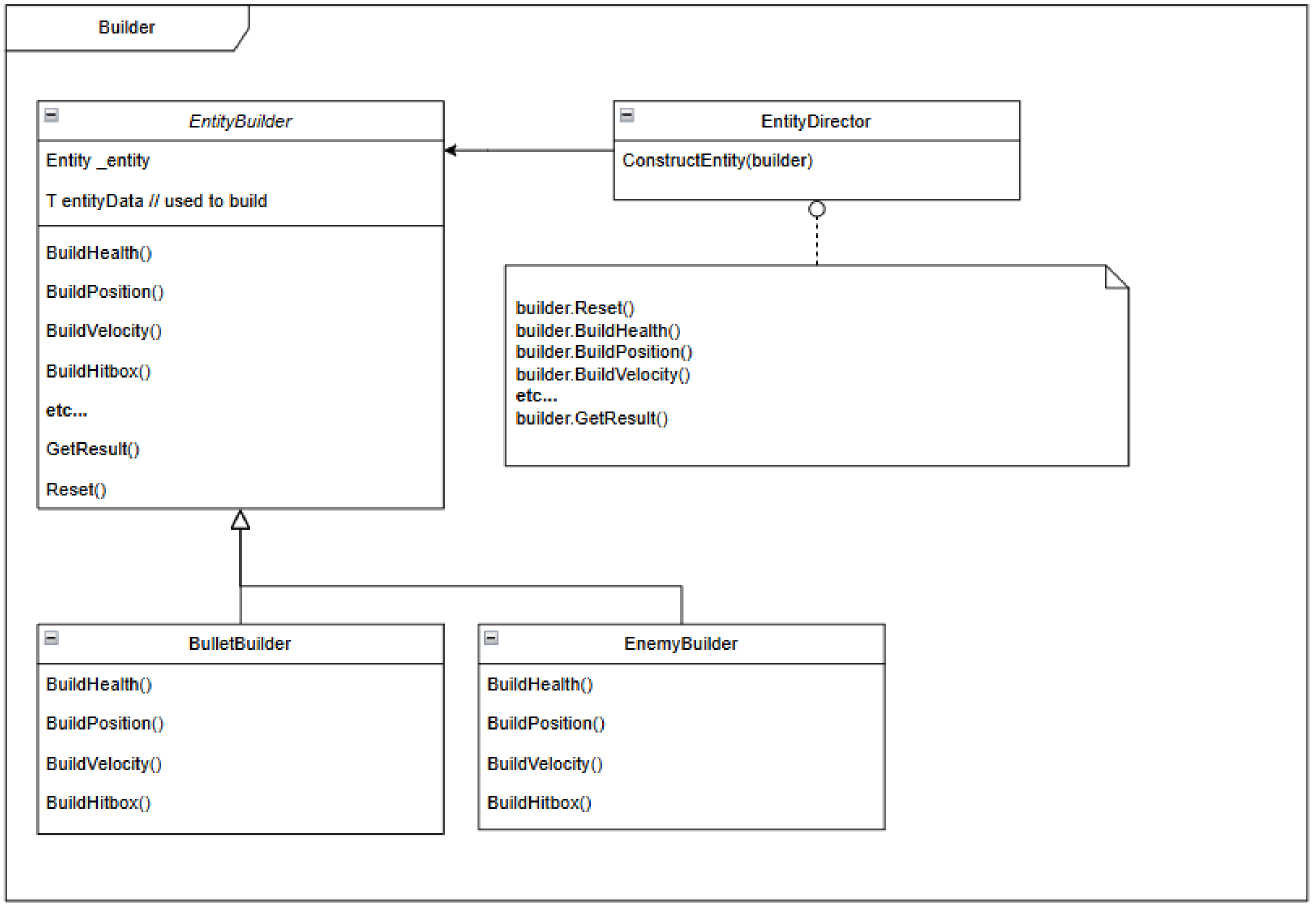


Figure 2.2.7.4 – Builder Pattern Diagram

3. Subsystem Services

3.1 – Entity Logic Subsystem

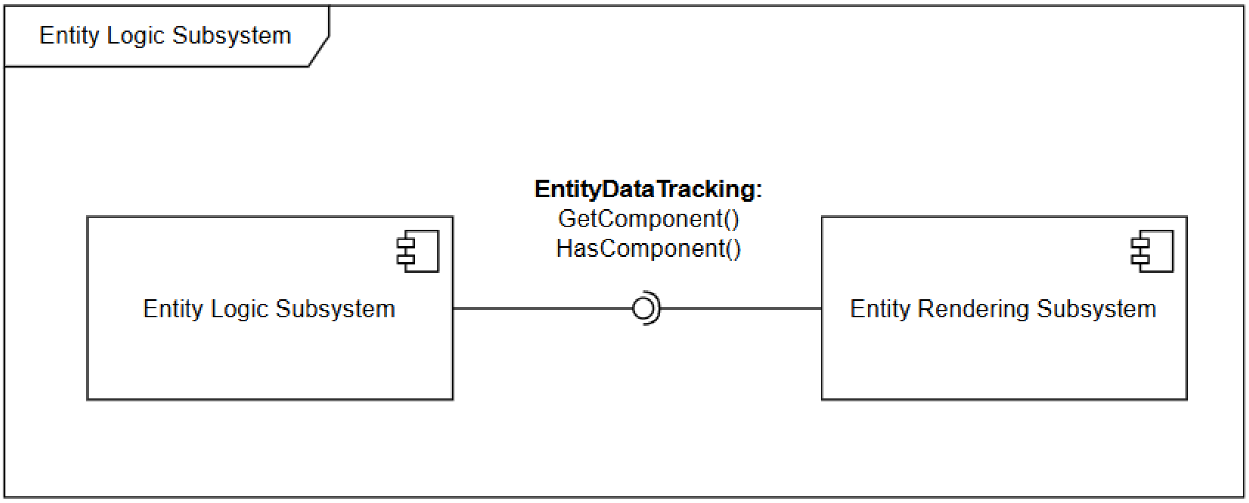


Fig 3.1.1 - Entity Logic Subsystem Diagram

Dependents:

- Entity Rendering System – Rendering depends on how the logic systems update the sprite component (is there an exposed function though? Maybe entity.GetComponent(), if you consider the components part of the logic system)

Services Provided:

Since the Entity Logic Subsystem provides a variety of services (the systems), and we are following the ECS (Entity-Component-System) pattern, these services are ultimately dependent on the type of entity involved. For example, enemy entities may utilize a different movement service than player entities, reflecting their distinct behaviors and logic. These services are done in the game Update() function.

Entity Data Tracking: The rendering system utilizes the logic system’s GetComponent() and HasComponent() to get the components necessary for rendering, like SpriteComponent

- entity.GetComponent()
- entity.HasComponent()

3.2 – Data Loading Subsystem

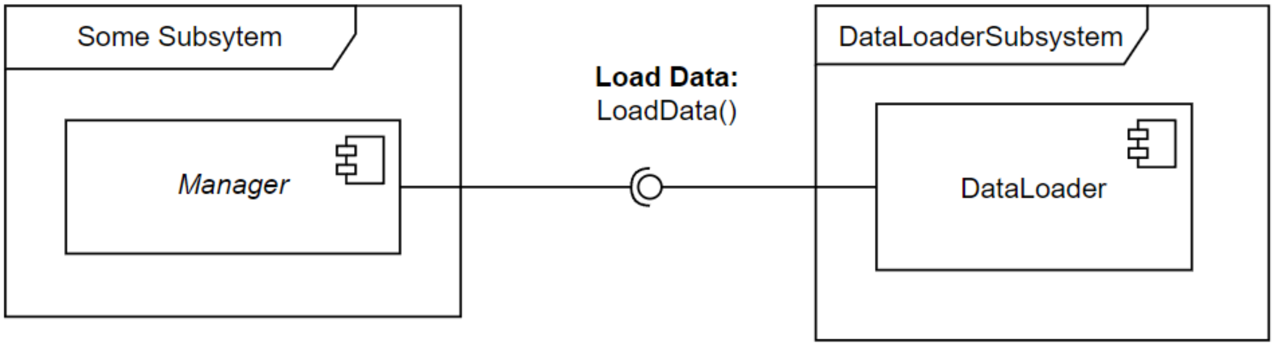


Fig 3.2.1 – UML component Diagram for Generic Data Loader

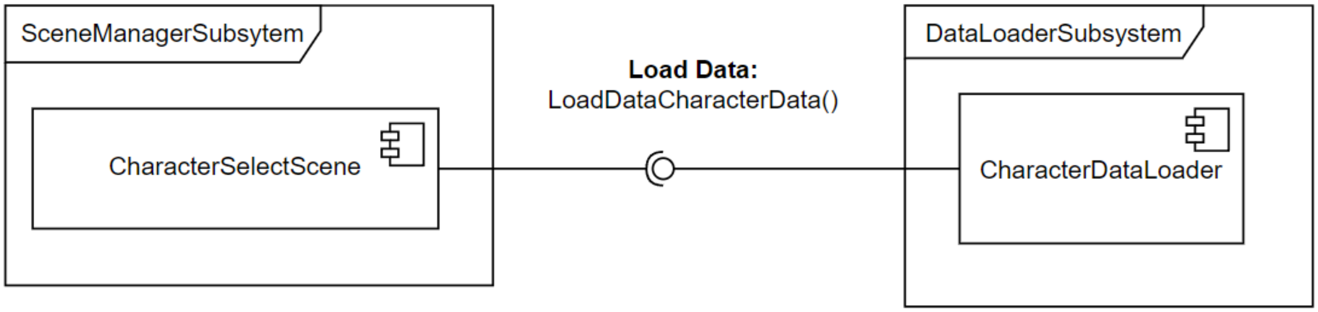


Fig 3.2.2 – UML component Diagram for Character Data Loader Subsystem

Dependents:

- Scene Management Subsystem (Individual scenes load data depending on what they need). Example: CharacterSelectScene's Load() function uses CharacterDataLoader.LoadCharacterData(). So DataLoaderSubsystem exposes LoadCharacterData() function for scenes to use.

Services Provided:

Data-Loading: There are multiple classes that provide a public function for loading and constructing data classes from JSON files.

- CharacterDataLoader.LoadCharacterData()
- LevelDataLoader.LoadLevelData()
- EnemyDataLoader.LoadEnemyData()

3.3 – Audio Subsystem

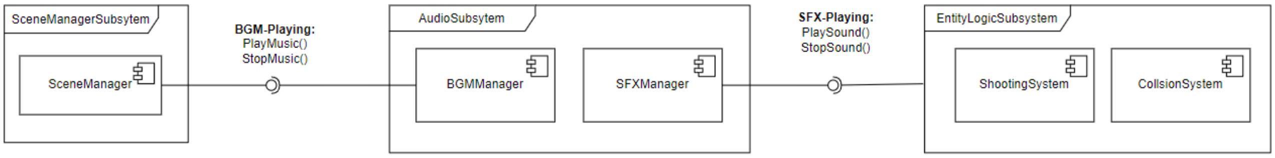


Fig 3.3.1 - Audio subsystem UML component Diagram

Dependents:

- Scene Management Subsystem: Scenes utilize the BGMManager’s PlaySong() function.
- Entity Logic Subsystem: Audio Subsytem has SFXManager that exposes a PlaySound() function to be used in the Entity Logic Subsystem. In the CollisonSystem.cs and the ShootingSystem,cs they both use the SFXmanagers PlaySound() function as seen below

Services Provided:

SFX-playing:

- SFXManager.PlaySound()
- SFXMangager.LoadSound()
- SFXManager.StopSound()
- SFXManager.SetPitch()
- SFXManager SetVolume()

BGM-playing:

- BGMManager.PlayMusic()
- BGMManager.StopMusic()

3.4 – Input Handler Subsystem

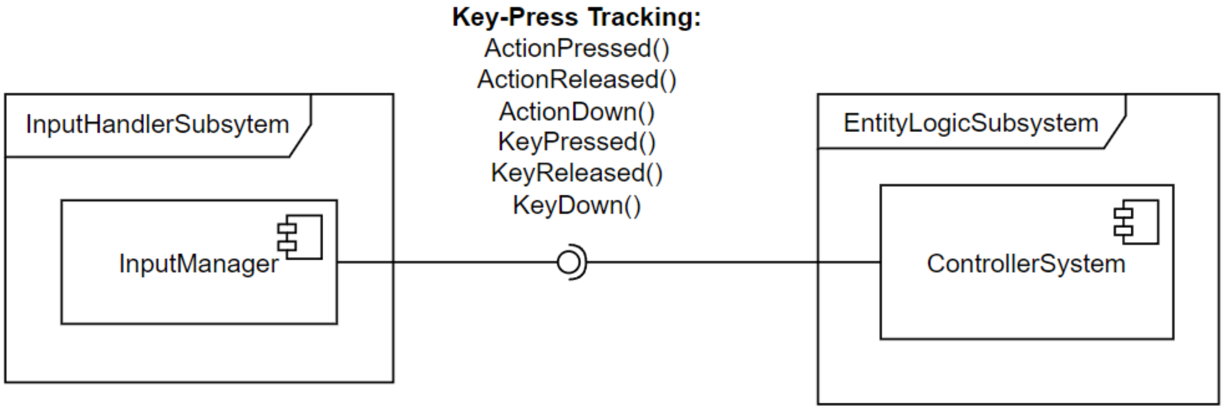


Fig 3.4.1 - Input Handler Subsystem Diagram

Dependents:

- Entity Logic Subsystem (getting inputs determines what data to change)

Services Provided:

Key-Press Tracking:

- ActionPressed()
- ActionReleased()
- ActionDown()
- KeyPressed()
- KeyReleased()
- KeyDown()