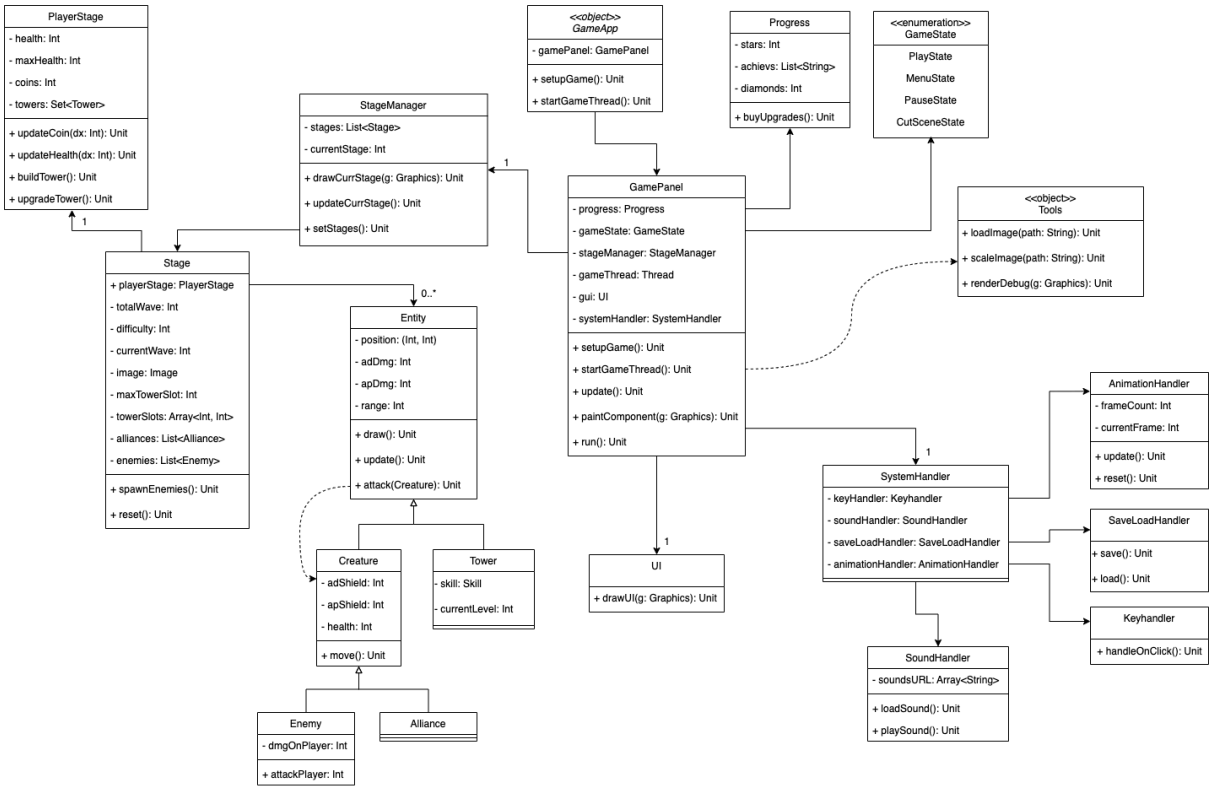
TECHNICAL PLAN

-Tower Defense Game-

Student Name: Nguyen Tran Dang Minh
Student Number: 102554946
Degree Program: BsC in Quantum Technology
Year of Studies: 2024-2027
Date: 27 January, 2025

Class Structure



class_structure.png

Use Case

1. Starting the Game

- The player opens the game and clicks "Start Campaign" in the Main Menu.
- The game loads their progress and shows the Stage Selection Screen.
- The player selects Level 1 and clicks "Start."

- The game loads the map, showing the enemy path and tower placement spots.

2. Starting the Battle

- The player clicks "Start Wave."
- Enemies appear and follow the path while towers attack them automatically.
- Earn coins by defeating enemies and spend coins for upgrading towers

3. Winning or Losing

- If all enemies are defeated, a Victory Screen appears.
- If too many enemies reach the base, a Defeat Screen appears.
- The game saves progress and unlocks the next level if the player wins.

Algorithms

1. Enemy Movement

- Enemies will follow a fixed path from the start to the end point, avoiding any need for pathfinding. The key task is to move them along this path, adjusting their speed and handling when they reach a tower or the end.

Approach:

- Predefined Path : The path is made up of waypoints that enemies follow in order. Each enemy moves from one waypoint to the next until they reach the end.
- Simple Movement: Enemies move at a fixed speed along the path. We just check the current position and update it towards the next waypoint until the enemy reaches it.

2. Tower Targeting & Attacking

- Towers need to target and attack enemies within range, choosing the best enemy to attack based on proximity or health.

Approach:

- Spatial Querying: Towers detect enemies in range using a spatial grid. The game area is divided into grid cells, each tracking enemies within it. Towers query nearby cells to efficiently find targets (e.g., closest enemy) without iterating all enemies.
- Priority Queue: Once the nearby enemies are found, the tower uses a priority queue to choose the target, based on distance or health.

3. Wave Management & Enemy Spawning

- The game will spawn enemies in waves, with each wave increasing in difficulty by adding more enemies or increasing their strength.

Approach:

- Timer-based Spawning: Enemies spawn at regular intervals within each wave.

- Scaling Difficulty: Each wave increases the number of enemies and their strength. The difficulty can be scaled based on the wave number, like so:
enemy_count = base_count + (wave_index × 2)

4. Game State Management

- The game needs to manage the state of the towers, enemies, and other entities (like projectiles) efficiently.

Approach:

- Queue: Actions like enemy movement, tower attacks, and other game events are handled through a queue, ensuring that they are processed in the correct order.

Data Structures

1. Game Map Representation

- Array/Grid: The game map can be represented as a grid, where each tile corresponds to a cell in the array. This is simple and efficient for a predefined path and helps with managing towers and enemies.

2. Enemy and Tower Management

- List for active enemies: Stores all the enemies currently on the map, making it easy to manage their movements and interactions.
- Priority Queue for Target Selection (Towers): Ensures that towers select the best enemy to attack.

3. Player Actions and Events

- Queue: Actions like placing towers, upgrading, or enemies moving are stored in a queue to ensure smooth game progression and synchronized events

Files

1. Progress Files

- Stores the player’s campaign progress, including completed stages, stars earned, and possibly unlocked towers or achievements.

File Format:

- The progress file will be a binary .dat file for efficient storage and faster load times.
It will store serialized player data, including levels completed, stars earned per level, and any unlocked content (e.g., towers).
The data will be saved as a sequence of binary records, where each record corresponds to a game state (e.g., level, stars, towers unlocked).

2. Configuration Files

Stores settings such as sound volume, screen resolution, and control preferences.

File Format:

- The configuration file will be a text file in the .txt format.
- It will store value-key in a predetermined way, making it easy to parse and edit.

3. Game Data Files

File Format:

- Game data could be stored in a .json file for easy readability and modification.

Example:

```
{
  "waves": [
    { "wave_number": 1, "enemies": ["grunt", "archer"], “number”: [100, 50] },
    { "wave_number": 2, "enemies": ["orc", "archer"] , “number”: [200, 100] },
  ],
}
```

4. Graphics and Assets Files

Contains images, sounds, and animations used in the game. These files will be loaded dynamically at runtime for efficient memory management and faster loading times.

File Format:

- Image Files: PNG for 2D sprites and background images.
- Sound Files: WAV for background music and sound effects.

Schedule

Weeks 1–2 (Planning & Setup, ~10h)

- Define project scope, key features, and architecture.
- Set up the development environment and initial repository.
- Outline the main components and dependencies.

Weeks 3–4 (Core Implementation, ~15h)

- Implement foundational classes and data structures.
- Create stub methods for unimplemented features.
- Start working on basic program logic.

Weeks 5–6 (Functionality Expansion, ~20h)

- Implement key features and interactions between components.
- Develop preliminary testing for core functions.

- Handle input/output processing and data validation.

Weeks 7–8 (Testing & Refinement, ~15h)

- Conduct system testing through user interaction and automated tests.
- Refactor code for efficiency and maintainability.
- Identify and fix bugs in core functionality.

Weeks 9–10 (Finalization & Documentation, ~10h)

- Improve UI and optimize performance.
- Write documentation and finalize the project.
- Conduct final testing and validation.

Testing Plan

System Testing

- Use a mix of UI-based and direct function testing.
- Test core functionalities, error handling, and edge cases.
- Implement test classes to validate system operations.

Unit Testing

- Focus on key logic-heavy methods and core components.
- Test data handling, calculations, and program flow.
- Write unit tests to verify expected input/output behavior.

References

- Scala API Documentation: https://www.scala-lang.org/api/3.6.3/
- Sprites, Animation: itch.io & jimmyyyao88/assets