Project Plan

Team Members:

Bădescu Alexia

Gefenitor Rudolf

Iwata Junya

Kanstrén Valtteri

Must Deniel

Contents

1. Scope of Work

1.1 Overview of the Game

Our game will be a simple Turn-Based Strategy Game, loosely taking inspiration from 4 main games: Civilization, Total War: Warhammer, Risk, Age of Empires. At its core, it is a simple resource gathering and battling strategy game, with map conquering elements. Its map style is inspired by Civilization, and the art follows a simple pixel style top-down style.

1.2 Core Gameplay Features and Functionalities

1.2.1 Resource Gathering

The main way to gather resources is to construct buildings on different tiles which give passive resource income after every turn. For now, we will restrict players to construct one building per tile

1.2.2 Colonization

Players start with one army unit (which could later be a faction chief or head) that can move one tile per turn. Any neutral tile can be colonized by placing an army unit on it and paying a resource cost, which gradually increases with each captured tile to encourage strategic expansion. For enemy territory, players must first neutralize occupied tiles by stationing an army on them for a turn. Once neutralized, players can advance further into enemy territory. Movement into an enemy tile requires that it borders a neutral tile, or one already occupied by the attacking player.

1.2.3 Base-Building

A player's base is their starting location, with surrounding territory serving both as a defence and a source of resources.

1.2.4 Construction

All buildings that the player can place are for resource gathering purposes. The buildings themselves cost resources to construct, and possibly units/resources to upkeep.

1.2.5 Units/ Unit Management

Units will be grouped into armies, which move as single entities on the map. Each army can consist of various unit types, each bringing unique benefits. Additional units can be purchased if the army is in the players colonized territory. These armies will be able to traverse the map and conquer tiles. However, the number of units also dictates the number of resources the player will pay to upkeep them.

1.2.6 Combat

When an army steps on a tile with an enemy army on it, it initiates combat, where the number of units acts as the main factor in determining the outcome of the battle, with some other factors also considered.

1.2.7 Victory Condition

The only victory condition is conquering the enemy player's base tile (through the combat mechanism).

1.3 Technical Aspects

1.3.1 2D Level Loader

To load the level and manage the UI / Graphics for the game, we will be using the SFML library.

1.3.2 UI

The UI will rely on hexagon or square tiles as regions on the map, and most interaction will be available in the form of buttons in menus, or simple click and select actions on the map UI.

1.3.3 Same Screen Switch Mode

For the first implementation, players will take turns within the same game instance. After each turn, control will switch to the next player.

1.4 Additional Features

1.4.1 AI

Introduces a basic AI for solo play, designed with standard victory conditions to provide a straightforward challenge for players.

1.4.2 Networked Multiplayer

Supports online multiplayer with a lobby system, where players can join games using a unique code. Players who disconnect can reconnect seamlessly to continue the match.

1.4.3 Factions with Unique Abilities

Players can choose from various factions, each offering unique buffs and debuffs that impact gameplay strategies and unit strengths.

1.4.4 Custom Level Editor

Includes an in-game level editor allowing players to create, save, and load custom maps using a selection of predefined tiles. These custom maps can be saved for future matches, expanding replay ability and player creativity.

2. High-Level Structure: Modules and Classes

2.1 Core Game Module

- **Classes:** GameState, Player, PlayerAI, Building, Army
- **Description:** This is our core module. It encapsulates the game's primary classes and functionality. The GameState class manages the current status of every aspect of the game, including players, armies, buildings, and the terrain grid, acting as the primary container for game data. This module oversees each turn's operations, invoking the nextTurn() methods for Building, Army, and potentially Player. The Player class holds player-specific details like resources and turn order, and PlayerAI builds on this to control non-human players, adding a difficulty level. Building manages individual buildings on the grid, while Army handles player-controlled forces, each with their own units (stored most likely as a list of structs or maybe even classes) and upkeep costs. We designed this module, especially the GameState class, with the objective of providing seamless interaction between UI and GameState. We want the UI to get all the information it needs from the GameState alone, without having to interact with other classes. This will make its behaviour more predictable and easier to debug. This also means that the interfaces that we design for Building, Army and Player are facing only towards GameState. We therefore avoid having to create generalized all-purpose interfaces for these classes, which carry the risk of making classes unnecessary bloated and abstract.

2.2 Map and Terrain Module

- **Classes:** MapValues (nested or as part of GameState)
- **Description:** This module manages the 2D grid representing the game map, where each cell stores terrain type and may contain a pointer to a Building. This module could include functionality for terrain interaction, pathfinding, and dynamic terrain effects on gameplay, e.g., movement restrictions or resource bonuses. The exact layout of the MapValues class is not fully clear yet, as we might not even need a separate class for it depending on the complexity of our game map.

2.3 UI Module

- **Classes:** UI
- **Description:** This module handles all player interactions, including displaying game data, responding to mouse events, and managing the Fog of War. The UI class tracks the current player's visibility, limiting their field of view based on their units' locations, and provides access to game menus, tooltips, and visual representations of game elements. As mentioned above, the UI class only interacts with GameState.

2.4 AI Module

- **Classes:** PlayerAI
- **Description:** The AI module manages the logic for non-human players, implementing difficulty settings, resource management, and strategic decision-making. PlayerAI uses inherited methods from Player but adds complexity according to difficulty, influencing its decisions and interaction with other game elements like buildings and armies. This module's focus lies on the algorithms that the AI will use to make decisions.

2.5 Level Loader and Creator Module

- **Classes:** None explicitly listed (yet), but interfaces with GameState and map initialization code
- **Description:** This module allows players to create, edit, and save maps. The **Level Loader** component loads saved maps and configurations into GameState, while the **Level Creator** provides tools for designing maps within a 2D grid, enabling custom gameplay scenarios and map editing. Since our MapValues class carries all the information for a grid piece, using this class here would be unnecessary. Using a 2D vector of enums should suffice to store the terrain information.

2.6 Game Snapshot Module

- **Classes:** None explicitly listed; interacts with GameState
- **Description:** This module supports saving and loading game snapshots, simplifying debugging by allowing us to examine the game’s state at any turn. Players could also use snapshots to revisit previous turns or view post-game summaries, which can be saved as snapshots and revisited. The basic idea here is that each turn will be saved in some format, which can later be translated back into a visual representation of the game state at that point.

2.7 Class Diagram

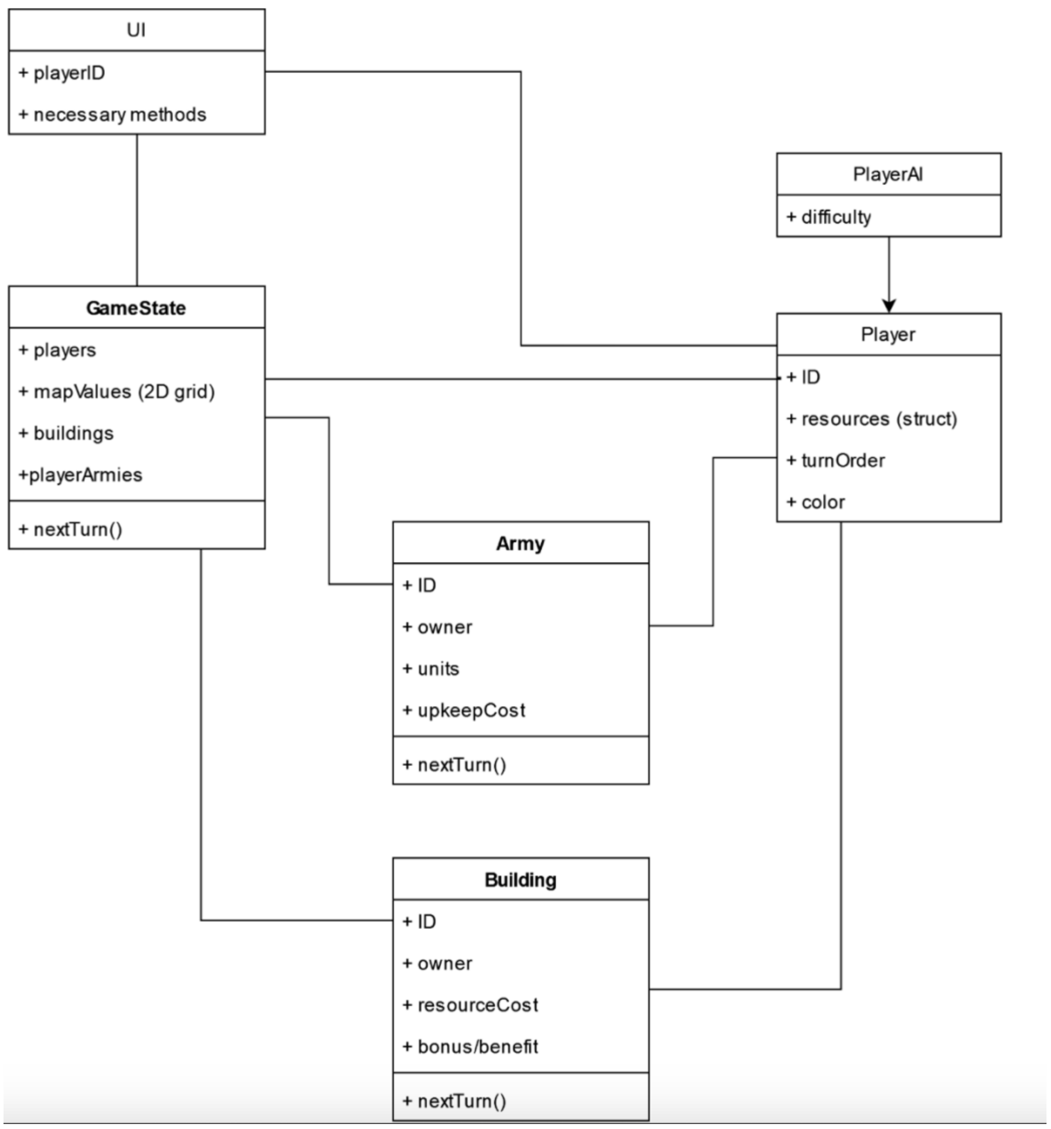

Figure 1 / Class Diagram

3. Use of External Libraries

SFML

The only library we will need for our game is SFML. SFML will provide the user input, graphics, and audio functionalities we need, and is cross-platform so it can be used on Windows, MacOS, and Linux. SFML also provides some extra functionality if we want to add more to our game, including a networking API that could be used for network multiplayer. There are many useful tutorials on the SFML tutorials page, including guides for installation and using different modules. The documentation can be found on the SFML website.

4. Plans for the Sprints

Each sprint involves two weekly team meetings:

- Tuesday 16:00 (online)
- Thursday 16:00 (offline or online)

The progress will be tracked by having team members report on completed tasks and upcoming goals during each meeting. These meetings also provide an opportunity to discuss and resolve technical issues that may hinder progress.

Sprint 1: 18.10 - 1.11

The objective of this sprint is to lay the groundwork for development. This includes establishing team agreements on working practices and defining the general game architecture. The team will be ready to start implementation by the end of this sprint.

Sprint 2: 1.11 - 15.11

The team is to begin the implementation. The initial playable version is expected to be complete by the end of this sprint, featuring all the core mechanics.

Sprint 3: 15.11 - 29.11

All planned features of the game are expected to be generally complete by the end of this sprint. The version produced in this sprint must be devoid of any major defects.

Sprint 4: 29.11 - 13.12

This sprint is allocated to finalizing the project. The tasks in this sprint include testing, fixing minor defects, as well as some final modifications. The team aims to complete the game by 6.12, allowing a buffer period of 7 days to accommodate any additional tasks that might arise.