Project Documentation

Overview

Our project is a Clash Royale-ish type of strategy game. The game will work something like the following.

You gain “elixir” (should probably call it something else) just by waiting. With elixir you can "activate" cards that spawn entities on the battlefield. The entities will then march toward the enemy encampment (towers, etc). If they meet enemies they will begin attacking. Troops will have a simple AI: they will attack the first thing that they see until it dies. Every entity has their strengths and weaknesses (also different kinds of attacks such as splash damage, air attacks etc). The player can also use spells that have different effects of the map/troops/towers.

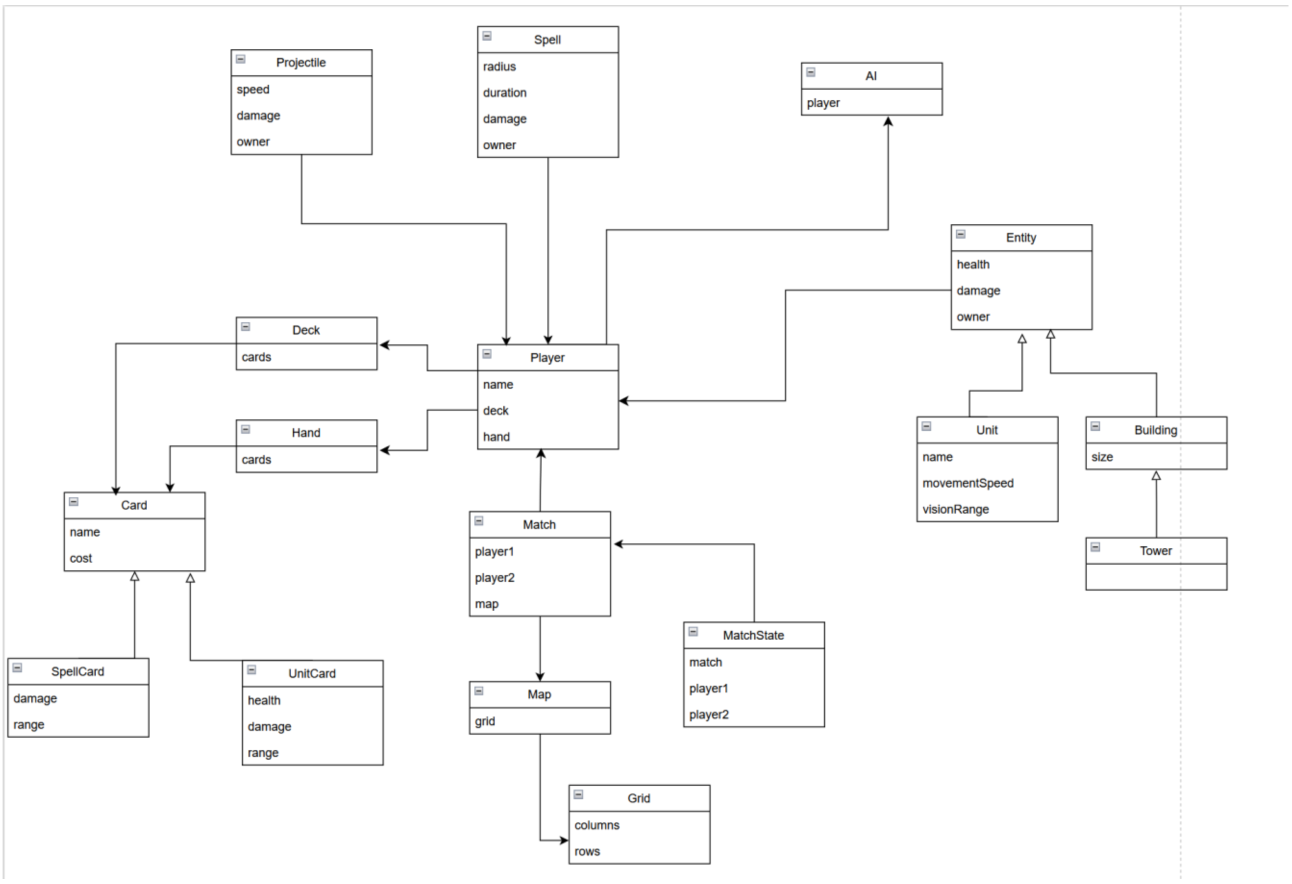
Both players have one main “king” tower. The player who destroys their opponent's king tower first will win unless time runs out, in which case the player who has destroyed more other towers or done more damage wins. Players can choose a certain amount of cards to be included in their decks before the game begins. Players can also choose which kinds of defensive towers they have.

The battlefield will be a simple 2D map. The players can see the map all at once. The map could include different obstacles etc if we see the need to. The map could also be randomly generated at the start of each game if needed to meet the requirements. The players can only place units on their own area.

The players’ cards will cycle their chosen deck (queue data structure). Meaning that they can only have a few cards “in their hand” at any point in the game.

There is also an AI the player plays against and the player can change its difficulty level.

Software structure



Picture 1, UML of the current project structure

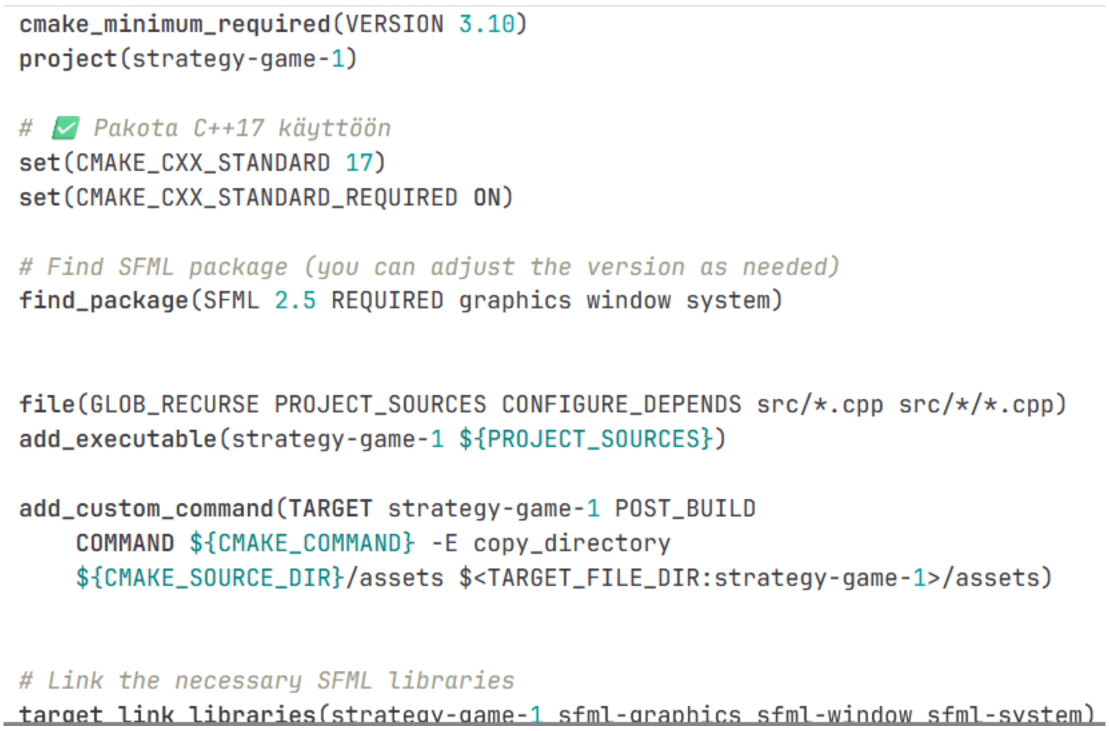
Picture 1 shows the current project class structure. The program has a main function that alternates between the different game states based on the progress of the current game or user input. These different game states are match state, menu state, pause state and end state.

The main game functionality is located in the Match, Map and Grid classes. Match contains all the units, buildings, spells and projectiles on the map and handles basic game functionalities like checking if the game is over, which player has won and adding new units on the map. Map class is responsible for creating and storing the map and also for pathfinding. Grid class is responsible for the grid structure of the map, loading the map from a text file and checking if a specific grid tile is occupied.

UnitCard and SpellCard are abstract classes inheriting from Card class that are used to create different types of units and spells. When the player plays a card the respective unit or spell instance is created based on the card. Player class holds the chosen deck of cards and hand is a rotating 4 card subcollection of the entire deck.

Both Unit and Building are subclasses of Entity and Building also has a subclass Tower. Unit class represents all the moving characters on the map, while Tower class is used for the 3 main towers of each player that the other has to destroy.

Instructions for using the program



```
cmake_minimum_required(VERSION 3.10)
project(strategy-game-1)

# ✅ Pakota C++17 käyttöön
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Find SFML package (you can adjust the version as needed)
find_package(SFML 2.5 REQUIRED graphics window system)


file(GLOB_RECURSE PROJECT_SOURCES CONFIGURE_DEPENDS src/*.cpp src/*/*.cpp)
add_executable(strategy-game-1 ${PROJECT_SOURCES})

add_custom_command(TARGET strategy-game-1 POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E copy_directory
    ${CMAKE_SOURCE_DIR}/assets $<TARGET_FILE_DIR:strategy-game-1>/assets)


# Link the necessary SFML libraries
target_link_libraries(strategy-game-1 sfml-graphics sfml-window sfml-system)
```

Picture 2, suitable CMAKE

The program uses SFML 2.5.1 and can be run using a CMAKE as seen in picture 2.

User must first run the program, then click play on the menu screen and then place units or spells by first clicking on a card and then suitable grid tile on the map (grass) on their own side. After destroying all the enemy towers or having the enemy destroy all the player's towers the player is greeted by the end screen from which they can choose to return to the menu or quit the game.

Testing

Each module was tested using unit tests and here are the areas covered in the tests and the outcomes:

Units and cards: tested units attacking each other and playing cards that cost more than the available elixir. Both tests succeeded.

Map: tested pathfinding and initialization of the map.

Grid: unit tests to different Grid methods like inbounds, gridToWorld and worldToGrid

Work log

Sprint 0: (29.9. - 13.10.) Plan document

Sprint 1: (14.10. - 24.10.) CMake config Header files with class declarations One meet at the start of sprint to share responsibilities Other “daily” meets when needed

Sprint 2: (25.10. - 7.11.)

Planned: grid, map, units, towers, sprites, main menu

Actual:

Lasse: preliminary version of grid and map , 5-10h

Aleksi: movement and seeking of enemies for units, 5-10h

Oskari: main towers, sprites and working main menu, 5-10h

Juuso: decks and cards, 5-10h

Sprint 3: (8.11. - 21.11.)

Planned: can place units and buildings, fully working grid, playable cards etc.

Actual:

Lasse: grid and map, pathfinding etc, 5-10h

Aleksi: creating units from cards and getting them to show on the map, 5-10h

Oskari: buildings, 5-10h

Juuso: cards and decks, 5-10h

Sprint 4: (22.11. - 5.12.)

Planned: Finishing the features, reading map from file and other additional features

Actual:

Lasse: reading map from file

Aleksi: Documentation

Oskari: refactoring, pause state, path finding fixes

Juuso: AI difficulties