ELEC-A7151 - Object oriented programming in C++

Project plan, Real-Time strategy game

Group strategy-game-3

Aleksi Heikkinen, Tommi Salmensaari, Juuso Pulkkinen, Toni Niinivirta

1. Scope of the project

The goal of our project is to create a real-time strategy game, with modern/sci-fi -theme. The basic features implemented will be the following: Basic graphics, non-hardcoded maps, basic AI for the opponent, single player mode and game control with mouse. For the additional features the plan is to create at least better AI for the opponent with adjustable difficulty, various resource types, upgradeable buildings and units and different attacking methods. As advanced features, we have decided to create map editor, and in case we have much time towards the end of the course, also more sophisticated AI for the opponent.

2. Design

The game will open to main menu, from which through buttons the user can either start a game, open map editor or quit the program. When the user decides to start a game, the UI request the user to choose a map from list. In the game state, the window includes the map and the basic UI for the player. Once player selects a unit, building or resource, the basic UI will show the statistics and commands for the chosen item. In the basic UI also the resources, units and building owned by the player are visible.

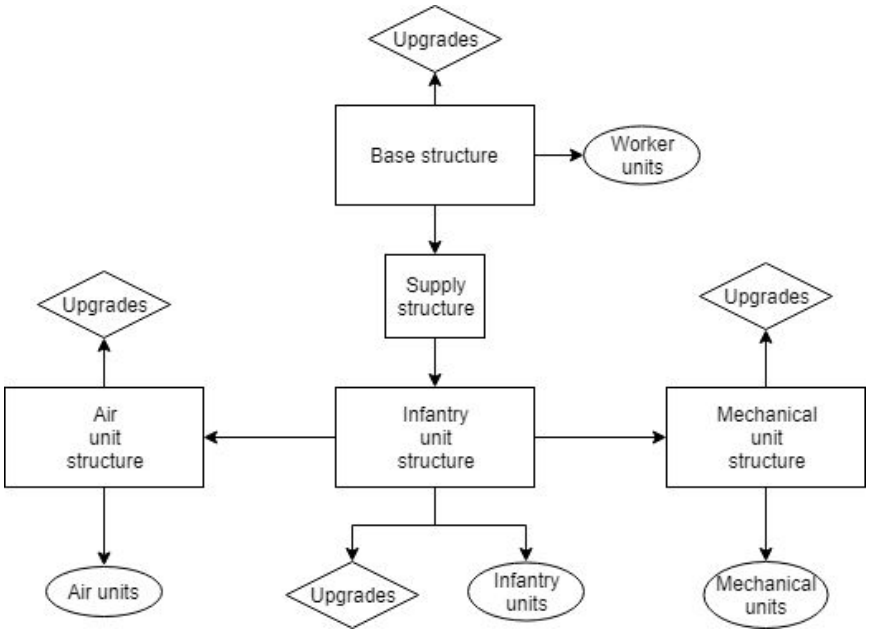

Figure 1: Tech structure

The basic tech structure of the playable faction is shown in figure 1. Basically the whole game revolves around it in a certain type of “rock-scissors-paper” aspect. Different unit types and their upgrades come from different tech paths, thus allowing the player more strategic options. We will start by implementing only a single faction, so only “mirror” matches can be played. If we have time will implement another faction with its own units and tech paths, though this seems unlikely since the faction would require its own graphics and animations and implementing them is very time consuming.

The graphical design of the game will utilize the modern/sci-fi -theme with 2,5D view from above. There will be animations on units and buildings, for example when unit attacks or when the building health level is low. The buildings will also have different graphics indicating the state (ready/being built). Also for each unit and building there will be health bar above indicating its current status. To improve the game experience, there will also be menu/game music as well as sound effects for different actions.

3. Architectural decisions

The game architecture is centered on a GameEngine class which will be run inside an SFML-window. The game’s basic architectural design is illustrated below in figure 2. SFML is simple to use and seemed to have all the graphical, control and sound features we needed for our game project, and thus we decided to use it as our game’s (graphical) basis. The GameEngine class will loop (and the game will run) inside the SFML-window until the window is closed.

The game has several different states as shown by the GameState class and the game will run differently in each game state. The different states will contain instructions for the game engine (and the SFML-window) on how the game is to be run at each point. The PlayingState is of course the primary one where the users will actually be playing the strategy game in real time, but the other states each provide some additional features to the software as a whole. The MenuState is meant to mainly be the opening state or main menu of the game and connect to some of the different states (namely the PlayingState and MapEditorState). The PauseState will allow users to pause the game in order to take a break or return to the main menu. The WinState and LoseState are used at the end of a game to show the end result of the match. The MapEditorState is one of the advanced features of the game and will allow users to create and edit their own maps that can then be used as backdrops for new game matches. Each game state also has different UI (titles, views and control layouts) which will be implemented in the various sub-classes of the UI class.

The GameEngine class also connects to the classes Player, AI and Map. The Player class is used when the game is being played to keep track of different players: their structures, units and resources. The AI class determines how the non-user-controlled players and their units act. The complexity of the AI is yet to be fully determined until we get a better understanding of the project.

The Map class determines the positions of all buildings, resources and units. The map consists of tiles (implemented by the Tile class) each of which has their own properties. The main properties of a tile are its location, its terrain type (which determines whether it is impassable for some types of units for

example) and its occupant (which can be a unit, structure or resource). The Map and Tile classes allow units to interact with other units, structures and resources when they are close enough in proximity. The Map and Tile classes are also used to draw the game map onto the SFML-window and allow users to click on units, structures and resources in order to select them.

The classes Resource, Structure and Unit (and their sub-classes) determine how they each work and each contain references to their sprites. The Unit class defines how units move, attack and interact with other units, buildings and resources. Different sub-classes of Unit have different limitations on their movement and different interactions with other sub-classes (for example melee units can't attack air units). The different sub-classes of Building can produce different kinds of units. Each unit and building also has a reference to the player who owns the unit or building. The Resource class contains information on the type of resource and the amount of the resource.

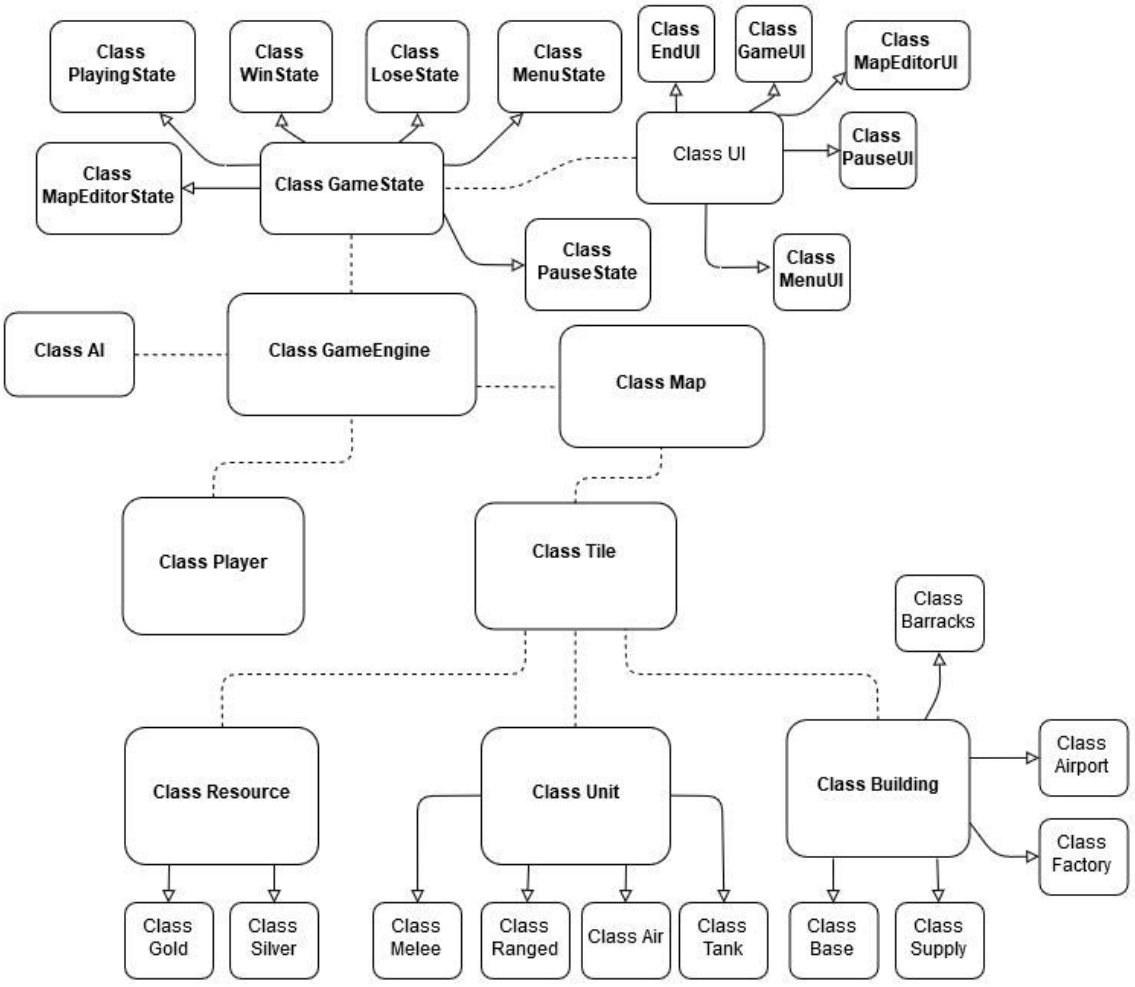

Figure 2: Class structure of the game

4. Schedule

Week 46: a single unit in a map that can be moved

Our aim is to get a rough framework for a game-engine running as soon as possible. This way we can open up the development for different classes which can then be tested independently in the engine. This speeds up the development as each member can fully focus on their assigned tasks. There will be some overlapping functionalities, so we need to pay attention while implementing those.

Week 47: mid-term meeting: all functionality barebones ready

A rough but presentable skeleton model of the game ready for the meeting. This includes things such as map generation, unit production, building production, resource gathering, moving units and attacking enemy units. At this moment there would not be much content in the game, but the point is to demonstrate the mechanics of the game.

Week 48: implementing all basic features

Adding more units, buildings and functionality to the game. This includes things such as upgrades and different types of units, for example heavy or flying units. After this week the player should be able to play a full round to victory or defeat.

Week 49: implementing additional features and testing

Smashing out bugs and balancing the game. Making sure that everything works as intended and the game doesn't crash. If there is some extra time left after the game is finished we will implement any additional features we have time for such as the map editor or better AI.

Week 50: strategy game project ready

This is the final week for the project and ideally the game is already running at this stage. Additional time can be used to play test the game as much as possible, so that we can find flaws or bugs which could be ironed out to improve the overall quality of the game. No new features should be implemented at this late stage as it might cause more harm than good.

5. Roles

Aleksi: Contact person, graphics, resource, (map)
Aleksi will work as the project contact person and he will also be responsible for the implementation of resource classes and graphics. As the contact person he will also be ensuring that the communication works inside the team and that everyone stays up to speed and latest information. Tasks and implementation responsibilities will be divided and assigned by the group as a whole.

Juuso: Graphics, sound, player class (control of units), window
Juuso will be responsible for the initial engine implementation and later on implementing the sounds and player class. He will be responsible for the movement and commanding of the units. Furthermore, he has some background in music, so he was named as the audio responsible.

Toni: Graphics, UI, building, AI
Toni will be responsible for the implementation of the building class, UI class and the enemy AI. He will also implement some of the graphics and help Juuso in making the audio.

Tommi: Graphics, tile, map
Tommi will implement the map generation, map class and the tile class. These are the second most important classes after the main game engine. He will also help in drawing the graphics.

Most of us have been assigned some part in the implementation of graphics. This was due to the common interest in graphics and also because at the same time it is fun but very time consuming. Sometimes it is also almost therapeutic to draw some sprites while thinking why the code doesn't compile.