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. Overview

Our project is a singleplayer real-time strategy game with a sci-fi theme. The objective of the game is to destroy the enemy's main building while defending your own base and defeating enemy units. Destroying the enemy's main building will result in victory while losing your main building will result in defeat. Building your own units is done from your buildings, each of which produces a unique kind of unit. Building production is done with worker units and requires resources as does producing units. Resources can be obtained by workers from minerals found on the game map and also they can be produced by resource buildings. There are two different kinds of resources: silver and gold, silver is used for producing units and gold is used for upgrading buildings (upgrades shorten unit build times).

The game is controlled with the mouse. Left clicking allows you to select a building or unit and brings up its information and some commands onto the user interface (UI) below. Right clicking is used for commanding units to move, attack and mine resources. Holding down the middle mouse button allows you to pan the camera. No zooming is available. In addition to the mouse the spacebar key focuses the view onto your own base. From esc button the user can end the game and go back to main menu.

The UI in the gameplay is used for some commands, such as producing buildings and upgrading them. There is also a self destruct button in the UI for players units and buildings. The UI also includes a minimap, where complete view of the map with buildings and units is visible.

Units have ranged attacks and will find paths around obstacles (such as walls) when commanded to move to the other side of them.

The game maps are not hard-coded and are loaded from png files. The map generation system draws mineral positions, terrain types and player start positions from the png files. The game also has basic animations for units and buildings. The main menu and the main game state have music tracks playing in the background to provide a bit of ambiance.

The game includes basic enemy to fight against. This enemy sends units to the player base, and when it encounters player units, it will try to destroy them. The enemy sends both soldiers and tanks with different intervals.

The map editor implementation was started, but there was not enough time to finish it. At the moment the user can draw the blueprints for the map, but those blueprints can not be used in the game.

2. Software Structure

The game functions around the main and gameEngine classes (as well as the SFML window the game runs inside of) which set up the main objects and contain the main game loop. They call on many different game states (base class gameState, playState, mainMenuState, endState and mapEditorState) that each function differently and contain different parts of the game architecture. There are also UI classes (base UI, playUI and mainMenuUI) which implement buttons and texts.

The main game state is of course the play state in which the actual game is run. Running the RTS segment of software involves many other classes such as player, map, tile, enemy, buildings, units and resources.

The game map is loaded from a png file. Each pixel on the png file(s) signifies a specific kind of terrain, resource or base start position. The map consists of tiles each of which have a specific terrain type and can contain (a part of a) building or a resource. There is also a coordinate system at use starting from upper left corner. Unit movement is not tied to tiles, but their movement is affected by the passability of tiles (units can't walk through buildings and walls).

Resource manager is used to load all the necessary textures for player and enemy buildings and units. It is also used for creating the animations for each of the units and buildings. The animations are then passed through to player and enemy classes where they are used in unit and building constructors to assign correct sprites and animations to each entity.

Player class tracks all the resources, units and buildings belonging to the player, updates them and draws them on the map. The enemy class handles the spawning, moving and attack commands of the enemy units and buildings, updates them and draws them on the map. The enemy class is basically a reduced version of the player class that doesn't do any input handling. Additionally, we didn't implement resources to the enemy class since it would be kind of redundant for the gameplay at its current state. Enemy units are spawned at set time intervals and are a movement command is issued to the player's base. Enemy units are programmed so that whenever they find a player unit or building in their attack range they start attacking.

The building classes implement buildings which can be constructed to produce units and resources. The main building being destroyed for either the player or the AI ends the game (main buildings can't be constructed but both sides start with one). Buildings are tied to the tiles and cause the tiles they occupy to become impassable for units until they are destroyed. Buildings can produce units (by clicking the buttons in the UI). Each unit has a build time so production is not instantaneous. Build times are reduced by upgrading the buildings. Constructing buildings costs silver (as does producing units) and upgrading buildings costs gold. After a building is built by a worker unit there is a short time window when it is still under construction and won't start producing units or resources yet. Building sprites are loaded from building tileset png files; they also have basic animations when producing units (or resources).

The unit classes implement units which can attack other units and buildings. Units (namely the tank and soldier units) have a ranged attack which damages the unit or building they are attacking. When a unit's (or building's) hitpoints drop to zero it is eliminated. Worker units are exceptional because they are used for constructing buildings and mining resources from the minerals on the game map. Unit sprites are loaded from unit tileset png files and they have basic animations for moving and attacking. Units cannot move through impassable tiles and they have a pathing functionality to find a route around obstacles.

For the pathfinding the A* algorithm was used. The algorithm calculates the total cost for a tile, which consists of the cost from the path to the tile and the estimated minimum cost from the tile to the destination. Then the algorithm generates the adjacent tiles, and calculates the total cost for these. For each adjacent tile, the information of the parent tile is also stored. Then the next tile is determined with lowest total cost of these tiles. The the same procedure is repeated until the current tile is the destination tile. Then the path for unit is created by looping the tiles backwards starting from the destination, and always taking the parent tile of the current tile to be the next tile, until the start tile is reached. This path is then passed for unit, from which it is completed tile by tile.

We used a ready-made SFML AnimatedSprite class for animation implementations (source: https://github.com/SFML/SFML/wiki/Source:-AnimatedSprite). The usage is simple: First an Animation object is created. The Animation class is basically a std::vector of texture rectangles. The required animation frames are pushed to the vector. Then we create an AnimatedSprite object and provide it with the created animation. The AnimatedSprite inherits from sf::Drawable and sf::Transformable so it behaves like a regular sf::Sprite. Animations are updated in the player class every frame by using the deltaTime (frame time) variable. This changes the animation frame at set time intervals.

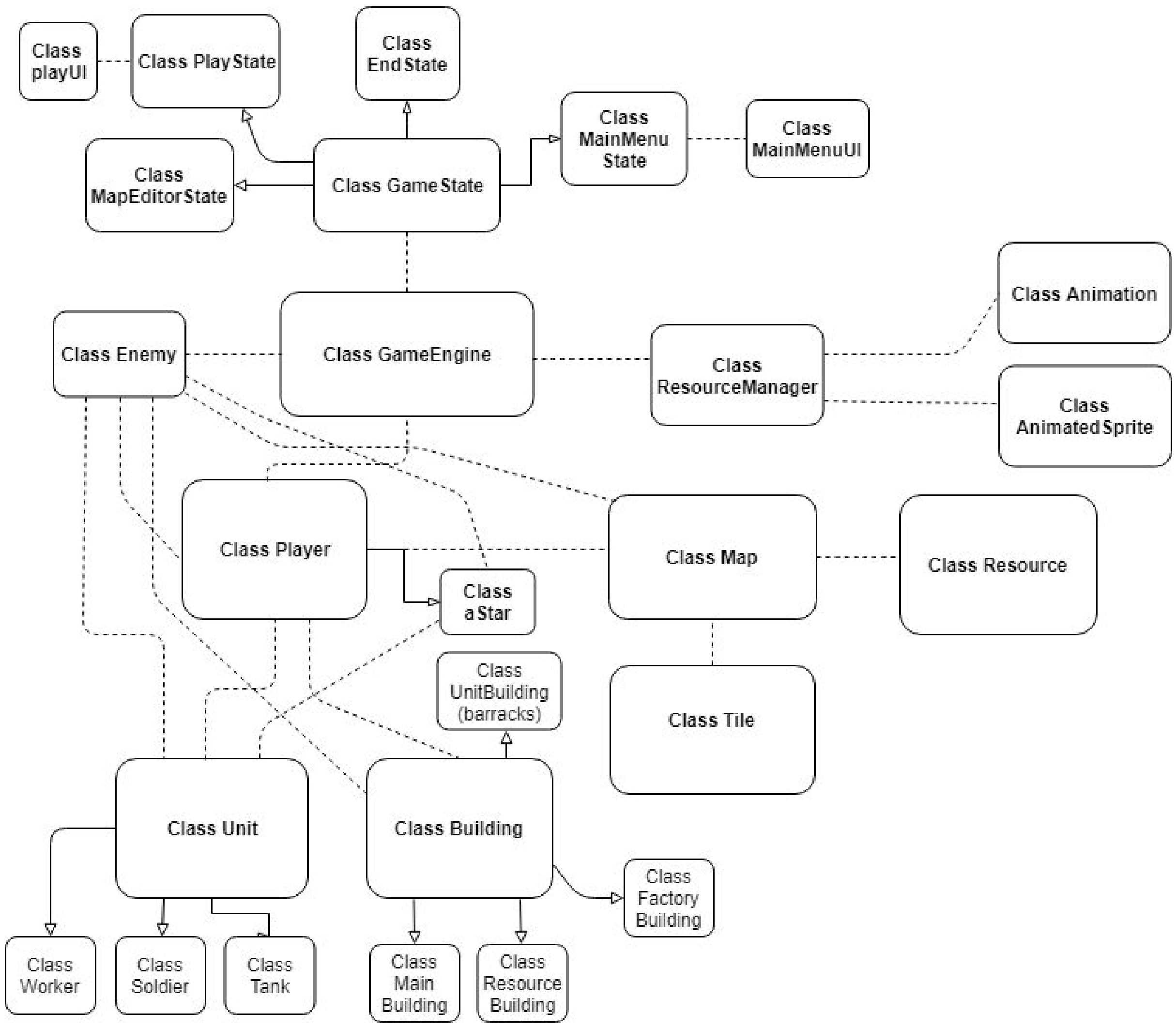

Figure 1: Class structure of the game

3. Instructions for Building and Using the Software

Building the game:

Best odds for success are gained with Ubuntu 16.04 using SFML 2.3.2. Custom installing SFML 2.3.2 is very difficult and time-consuming, so school Linux computers are suggested, since they have it installed already.

The game uses a custom Makefile for building it. Building the game is done by opening the master directory in terminal and typing “make run”. It builds and runs the game. If for some reason the game needs to be built again, first run “make clean” and then “make run” again (there is also “make all” which builds the game without running it).

The game will open up to the main menu. The game is begun by pressing the start game button (this is immediately followed by map selection after which the main game will start). The main game can be exited by hitting the Esc key. The quit button in the main menu exits the software.

More information about playing the game can be found in the readme file.

4. Testing

Testing of the game was mainly done through trying out various scenarios in the game. For example building a large amount of resource buildings and noticing that it is possible to get an extremely high amount of resources causing the UI text to grow too long (and maybe even cause int overflow eventually). As a result, number of resources was capped to 9999. Unit and building behaviour was observed and fixes were made accordingly when issues were found.
Another example of testing leading to bug fixes: we tried putting many units on a building's build queue at once and noticed that the build progress bar of the buildings in the bottom UI didn't update after the first unit was built from the queue. This glitch was then fixed and now the progress bar functions for every unit in a build queue.

5. Work Log

Tommi: building classes (building, mainBuilding, unitBuilding, factoryBuilding, resourceBuilding), sprites for barracks and enemy buildings, a bit of work on tile and map, searching out some bugs and glitches

Juuso: Sprites for various buildings and units. Implemented animations. Created the player class that handles inputs, draws and updates the player's assets. Implemented Unit classes. Created the enemy class and coded the aggro mechanics of the enemy. Implemented music to the game.

Toni: Creation of the UI:s for both main menu and gameplay. Created possibility to move the view in the window, and also minimap. Implementation of the A* pathfinding for the unit movement and implementation of the enemy movement. Drawing of worker and tank sprite textures. Partial implementation of worker class. Implementation of the enemy movement and attack interval. Also started work on map editor.

Aleksi: Responsible for the initial setup of the GameEngine and GameStates class. Upgrading the Map and Tile class and implementing resource class so that the map generation can be implemented. Implemented map generation, resource placement and starting base placement on map. Improved building class and main building class so that they could be placed on the map. Made it possible to produce units from buildings and upgrade the buildings. Managing passable and not passable tiles in map.

Weekly log

Week 45 (November 5 - November 11)

Tommi: first project meeting and working on the project plan - 6 hours

Juuso: project meeting, working on the project plan, SFML testings & game engine research - 6 hours

Aleksi: Project meeting, project plan, SFML 2.3.2 installation and writing a guide for other group members, GameEngine, stateMachine, mainMenuState, Makefile - 14 hours

Toni: Project meeting, project plan, SFML installation and testing - 5 hours

Week 46 (November 12 - November 18)

Tommi: setting up Linux virtualbox and SFML 2.3.2 and creating some basics of tile and map classes - altogether 7 hours (most of which was setup)

Juuso: Unit sprites, implementation of basic movement, input handling - 10 hours

Aleksi: Finalized map and tile class, added map generation from image file, drew maptileset - 6 hours

Toni: Mainmenu UI, programming and drawing of the required textures - 6 hours

Week 47 (November 19 - November 25)

Tommi: began working on building classes (mostly building and unitBuilding) - 6 hours

Juuso : building sprites, player class updates, various bug fixes - 10 hours

Aleksi: Resources class, added resource spawning to mapgen, build instructions for the game - 4 hours

Toni: Implementation of A* pathfinding algorithm. Started creating the workerUnit class. Draw sprites for workerUnit and Tank. Created movable view and minimap. Figured out some issues with GIT. -15 hours

Week 48 (November 26 - December 2)

Tommi: continued working on building classes (for example implementing time management inside building classes) and mid-term meeting - about 7 hours

Juuso: Unit selection implementation, animations, sprites and player class updates - 6 hours

Aleksi: Modified Building and MainBuilding classes, added starting base spawn to mapgen - 10 hours

Toni: Fixing of multiple bugs in unit movement. Added healthbar to UI. Added collision detect to unit movement. - 7 hours

Week 49 (December 3 - December 9)

Tommi: continued working on building classes (for example implementing unit spawning) - about 4 hours

Juuso: Animations, sprites, attack command implementation, various bug fixes - 10 hours

Aleksi: Added worker unit spawning to main building and chased down a bad compiling error - 16 hours

Toni: Created unit/building specific UI:s and small fixes to UI. Fixed a mistake in A* algorithm. Implementation of the resource gathering. - 10 hours

Week 50 (December 10- December 16)

Tommi: creating factory building, enabling resource building resource production, creating sprites for barracks and enemy buildings, testing game, writing documentation, final project meeting - 15 hours

Juuso: final project meeting, animations, enemy class implementation, enemy aggro mechanics, music implementation, various bug fixes and updates, winning/losing condition implementation, documentation - 18 hours

Aleksi: Added cost to buildings units, drew sprites for resource building, building upgrades, enemy starting base, ability to build Barracks, factory, resource building, soldier and tank. Minor bug fixes. Final documentation. - 18 hours.

Toni: Finished the PlayUI. Created tank class. Implemented movement and spawning of the enemy units. Added support for various new selectables to playUI. Fixed issues with unit stacking and collision detection, and also made the tiles to be se non passable when unit occupies it. Started implementation of the map editor. Multiple bug fixes. - 18 hours

6. Self-assessment

Review form for project strategy-3

Name of project to be reviewed: Arcturus

Names of reviewers: Juuso Pulkkinen, Tommi Salmensaari, Aleksi Heikkinen, Toni Niinivirta

## 1. Overall design and functionality (0-6p)

*** 1.1: The implementation corresponds to the selected topic and scope. The extent of project is large enough to accommodate work for everyone (2p)**

The game functions like a classic RTS game and supports the basic and some advanced features that are commonly included in them. The extent of the project was large enough to accommodate plenty of work for every group member, we even had to leave out some features that we would've liked to implement because we ran out of time. 2p

*** 1.2: The software structure is appropriate, clear and well documented. e.g. class structure is justified, inheritance used where appropriate, information hiding is implemented as appropriate. (2p)**

The game source code is split clearly into different classes and inheritance is used with several class types (buildings, units, game states). Most class variables are protected or private and public functions are used to access these. 2p

*** 1.3: Use of external libraries is justified and well documented. (2p)**

The external libraries that we used were SFML 2.3.2 (as suggested by project instructions) and a SFML class that helps in animation creation. They fit our needs well. 2p

## 2. Working practices (0-6p)

*** 2.1: Git is used appropriately (e.g., commits are logical and frequent enough, commit logs are descriptive). (2 p)**

Git was used appropriately. Some of the commit logs could've been more descriptive especially at the beginning states of the project, but we improved them during the later stage. Commits were done frequently. One thing that we could've used more is the branching feature of Git, but since there was uncertainty on how it works we didn't end up using it too much. 1.5p

*** 2.2: Work is distributed and organised well. Everyone contributes to the project and has a relevant role that matches his/her skills. The distribution of roles is described well enough. (2p)**

Work was distributed evenly and organized well. The role distribution is clear and described well. If someone needed help or ideas on their respective areas support was given from other team members. 2p

*** 2.3: Quality assurance is appropriate. Implementation is tested comprehensively and those testing principles are well documented. (2p)**

The testing was usually done immediately after creating the particular feature. During the implementation, very simple testing was used by adding prints between the code and see where the program stops. These tests were not documented. Also overall testing was done by playing the game to find some more unusual behaviour. 1p

## 3. Implementation aspects (0-8p)

*** 3.1: Building the software is easy and well documented. CMake or such tool is highly recommended. (2p)**

Building the software is done with a custom Makefile. Only one terminal command "make run" is required to build the software. 2p

*** 3.2: Memory management is robust, well-organised and coherent. E.g., smart pointers are used where appropriate or RO3/5 is followed. The memory management practices should be documented. (2p)**

Memory management is implemented where it is required. Vectors of pointers are erased in their respective class Destructors. Smart pointers are used for unit and building vectors. However, RO3/5 is not followed. 1p

*** 3.3: C++ standard library is used where appropriate. For example, containers are used instead of own solutions where it makes sense. (2 p)**

Many features of the C++ standard library are used: for example deque, vector, sstream, string, memory (shared pointer) and utility (pair). Iostream is also used for testing purposes. 2p

*** 3.4: Implementation works robustly also in exceptional situations. E.g., functions can survive invalid inputs and exception handling is used where appropriate. (2p)**

Error handling is not implemented in every single place where it would be appropriate, mainly when textures are loaded. 1p

## 4. Project extensiveness (0-10p)

*** Project contains features beyond the minimal requirements: Most of the projects list additional features which can be implemented for more points. Teams can also suggest their own custom features, though they have to be in the scope of the project and approved by the course assistant who is overseeing the project. (0-10p)**

The project contains all the minimal requirements, and the following additional features from the listed ones:

- Collection of different kinds of resources
- Upgradeable buildings
- Different attacking methods (ranged, melee)

We also implemented our own custom features:

- Minimap view
- A* algorithm for unit movement
- Music

10p