Project documentation: Tower Defense

ELEC-A7151 Object oriented programming with C++

11.09.2018-14.12.2018

Group #3:
Oskar Tainio 590138
Katariina Tuovinen 561507
Arsi Ikäheimonen 47130M
Esa Palosaari 618573

14.12.2018

Project Overview

Our group's object was to create a simple Tower defense game based on C++ programming language and SFML graphics framework [1]. The game has a simple user interface which consists of three main views; title screen, game menu and the game grid itself.

The execution of the program starts with main function which opens up the title screen. The title screen includes the name of the game and also our group members names. Here user can simply click or press any button and thus move forward to the game menu view.
After the title window is closed, a same sized game menu window will open. Here user can select whether he wants to start or exit the game by mouse clicking. The game menu window also views the top 10 scores of the previous games.

Once the game starts it will load the map from a .txt - file and draw it to the screen with SFML functions. In the game screen user can see the playable map and user interface screen, where the stock of user input happens. UI-screen contains buttons to buy/sell towers, infographics screen and menu options.

Enemies spawn only when a new level starts. There can only be one type of enemies in a single wave. User cannot know the type of the wave beforehand. There is always a limited time between each wave, but the user can start a new level by pressing the start button.

Towers can be bought by clicking the tower icon and then dragging and dropping the object. Towers can't be built on roads. Tower's statistics (such as range and damage) can be seen when user clicks the placed tower. Towers fire different kinds of projectiles based on their type. Projectiles include types like machine guns, rockets and flames. Enemies will lose health points based on each projectile's statistics.

If players homebase reaches zero healthpoints, the game will end. After the health reaches zero, the endgame title will popup. If player has managed to set a new top 10 record, the game will ask the player's name and save the record.

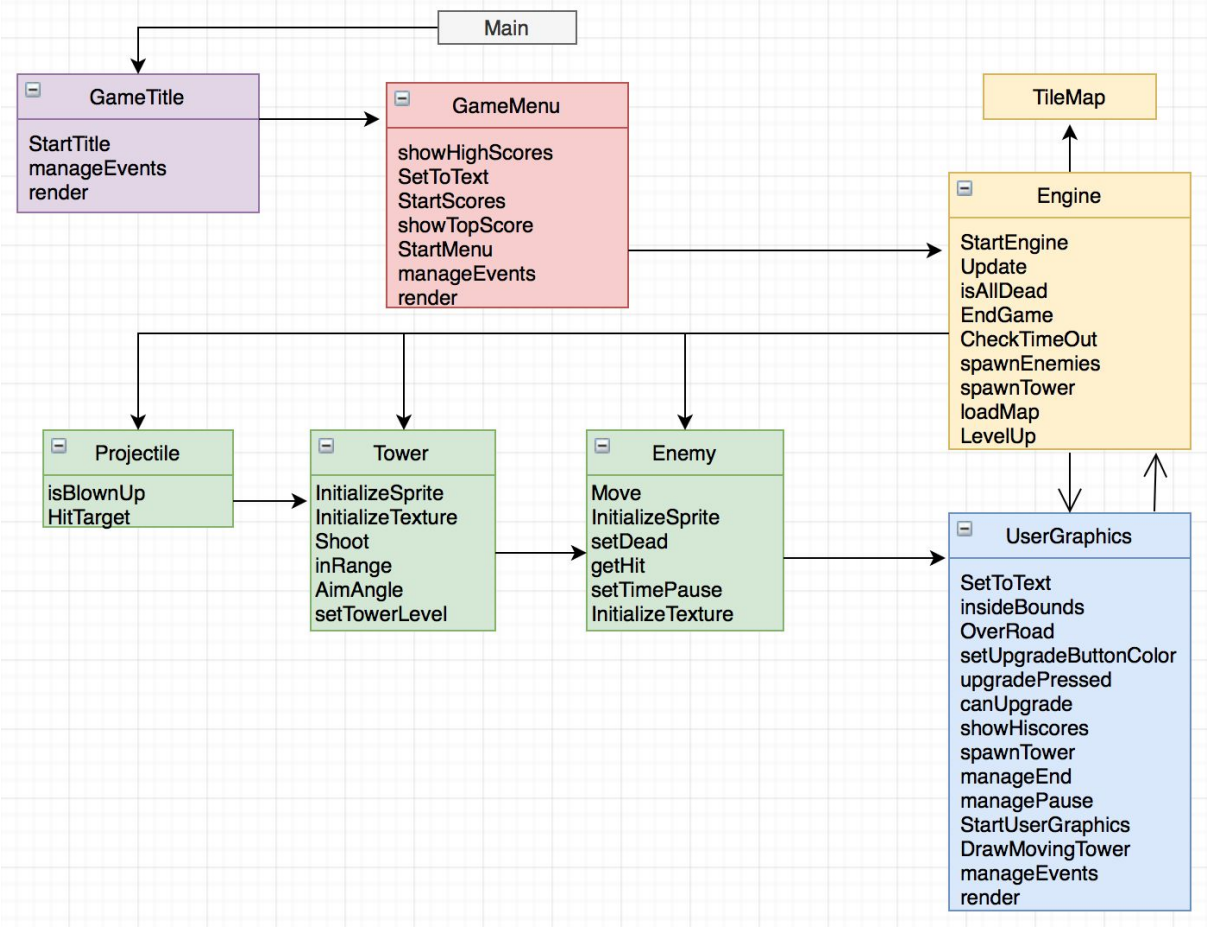
Software structure

Major structure

Program itself is constructed to contain three major phases. Each phase is designed to start another phase in the game and end itself in the process. These phases are called title screen, menu screen and in-game screen. The biggest of the three phases, the in-game phase contains most of the functioning structural classes (Engine, UserGraphics, Towers, Enemies, Projectiles) and therefore most of the user-input and game mechanics related code is stored in these classes. The title screen and menu screen are both managed only by one (yet separate) class, GameTitle for title screen and GameMenu for menu screen. It is therefore notable that Menu screen and title creen are completely separate from the actual game and only interaction between these entities occur through the start-game-function in the GameMenu, which starts the in-game phase of the program.

Engine-class is responsible for maintaining the mechanics and logistics of the game. Engine's Start-function creates a UserGraphics-object, which handles graphics rendering and all the possible user input related events. Possible events which occur during the single frame are all handled inside the UserGraphics-class. After a single frame is handled, Engine's Update-function calls the engine to update everything that happens during the next frame of the game. Update-function basically handles all the possible things related to functionality of the game with the help of the Enemy, Tower and Projectile classes. Update-function is also responsible for maintaining the game during pauses and ending the game is needed. UserGraphics's StartUserGraphics-function loops the in-game window as long as the window created by Engine is open. During each loop it processes the user input and calls Engine's Update-function. At the end of the loop, UserGraphics's render-function is called and it draws the updated screen with updated locations for each in-game object. Program uses SFML-library as an external library to process the window, graphics, sounds, user input and game mechanics such as timing.Game graphics and sound effects were downloaded from https://opengameart.org/ free library [2].

Class Overview



Main

- Contains no additional functions. Starts the GameTitle, GameMenu and Engine separately

GameTitle

- StartTitle: contains a while-loop that takes care of manageEvents() and render().
- manageEvents: Handles user input
- render: Processes window's clearing and drawing

GameMenu

- showHighScores: draws scoreboard to the left part of the screen and sorts the board so that it shows the top 10 players
- SetToText: Creates an in-game button based on the size of the input text
- StartMenu: setup function that starts the menu screen and loops the updating functions (manageEvents and render) until window is closed
- manageEvents: Handles user input
- render: clears window, draws objects and displays everything

Engine

- StartEngine: Creates an UserGraphics-object and starts the in-game clock. Core amount of money for the player is issued here.
- Update: Handles enemy and projectile movement, damage and sorting of enemies. Towers reset their targets and find new ones. Spawning is handled here. Special events are handled
- isAllDead: Checks if all the enemies are dead
- EndGame: Ends the game and handles the user input. Scores are saved to a local text file and sorted depending on the score
- CheckTimeOut: Checks if enough time has passed to start a new wave of enemies
- spawnEnemies: Processes the spawning of enemies depending on the type of enemy received as a parameter
- spawnTower: handles the spawning of a single tower and adds it to the tower vector of the engine class.
- loadMap: loads a specific map from the map-file.
- LevelUp: reset the amount of spawned enemies, increases the level cap, doubles the spawning enemies and makes the game therefore harder.

UserGraphics

- SetToText: Creates an in-game button based on the size of the input text
- insideBounds: Template that is created to check if the target is within the reach of another object.
- OverRoad: Checks if the current object is on the road. For example towers cannot be spawned on the road.
- setUpgradeButtonColor: Changes the upgrade button's color depending on the amount of money player has
- upgradePressed: Handles the event and possible outcomes when cursor is clicked upon the upgrade button
- canUpgrade: checks if the targeted tower can be upgraded. Works closely with the upgradePressed-function
- showHiscores: handles the drawing of Engine's EndGame function's scoreboard at the end of the game.
- spawnTower: Handles graphically the spawning of a tower
- manageEnd: manages user input at the end of the game when Engine's EndGame function is called.
- managePause: Manages user input while game is paused
- StartUserGraphics: loops through majority of the UserGraphics functions and in the end renders the processed frame. Also calls Engine's Update function to update mechanically the current frame.
- DrawMovingTower: handles the re-drawing of a tower when it is moved from user panel to the spawning point
- manageEvents: handles user input such as mouse movement and button pressing
- render: handles clearing of the window and drawing objects. Displays everything at the end.

Enemy

- Move: Handles the movement of different types of enemies
- InitializeSprite: Initializes sprite objects when they are spawned in the Engine class

- getHit: object loses Hp when it gets hit by a projectile. Also makes changes to enemy's healthbar in the process
- setTimePause: reset the pause timer to adjust the movement of enemies (enemies do not move during pause)
- initializeTexture: initializes textures for each enemy when they are spawned

Tower

- InitializeSprite: Initializes spawned tower's sprite and other tower specific graphics
- ReTarget: resets tower's target
- Shoot: Sets aim to the new target. Aircraft is a priority target and towers are adjusted to target them if possible, otherwise towers target the closest enemy to the final point of the road. Function also checks if the current tower can fire and creates a new projectile-object if time has passed enough to the tower to be able to fire
- inRange: checks if the targeted enemy is in range of the turret
- AimAngle: alters tower's rotation based on the targeted enemy
- setTowerLevel: Upgrades tower to the next level increasing its statistics in the process

Projectile

- InitializeTexture: initializes projectile's textures so that they exist and are drawable
- isBlownUp: checks whether projectile exists or not
- Move: Projectile moves frame by frame straight to its destined target. Moved distance is based on the speed factor of the projectile
- HitTarget: This function is special to every three types of projectiles. In general, this function handles damaging of enemies if they are hit by a projectile

Instructions for building the software

The game is designed to run on a modern Linux distribution (Ubuntu 16.04 or equivalent) and requires the SFML library and CMake to compile [3]. You can get instructions for downloading and installation for SFML from https://www.sfml-dev.org/, and for CMake from https://cmake.org/install/.

Game source code can be acquired from Aalto Version Control System [4] by cloning the repository https://version.aalto.fi/gitlab/elec-a7151-2018/towerdefence-3. Make sure that included `build/` directory is empty (that is, delete everything inside it if there is something) and run `cmake ..` from the directory `build/`. To compile the program, run `make` in the directory `build/`. To start the game, run executable file using command `./TD3` in the `build/` directory.

A basic user guide

The main game menu shows top 10 high scores and gives player two options, start the game and exit game. After starting the game, player can buy new towers to defend the Homebase against invading enemy waves. To buy a tower, first click a corresponding tower icon on the control panel (right side on the game screen). After that, click on any free map location to place the tower. Towers can be upgraded by clicking the designated tower and

upgrade button on the control panel. New enemy waves are generated automatically and new wave can be sended by clicking the *send wave* button on the control panel. If an enemy unit reaches the right side of the game map, player loses hitpoints (HP) according to the enemy type. When HP drops to 0, game ends. The game can be paused or terminated by clicking the *options* button on control panel and selecting corresponding action.

Testing

The software was tested almost completely just by running the program and seeing if it performed as expected. We wrote print statements for cases where there were exceptions (when, for example, files were not loaded) in order to know if and where there were problems. Otherwise, we relied on our own perceptions when using the program (for example: was the map layout on the screen as expected? Did upgrading the towers produce the expected sound?) and on the error messages produced by the compiler. The information was used to debug and fix the problems in the source code.

We used Valgrind [5] to test whether there were memory leaks in the program. Valgrind produced messages complaining about uninitialized references and still reachable memory after closing the problem. Some of the problems were located to rvalue references but we didn't solve them. We found information on the Internet showing that C++ can have memory blocks still reachable even in very trivial programs [6] and on the suggestion of the assistant we did not put more time in making the messages go away. Furthermore, using `-fsanitize=address` and `-sanitize=leak` produced memory leak messages which point only to Linux problems outside the program and which are not really memory leaks [7]

We tested testing with Googletest [8] which is included in the git repository. We produced a minimal working example which tests whether money can be reduced so that it can be negative. The code had a bug allowing the player to have negative sums of money which the test correctly marked as a failure. The test was also correctly passed when the bug in the code was fixed.

Work log

For the project management and organization, we used Aalto Version Control system GIT and Trello web-based project management application [9].

Week 44 (29.10. - 4.11.2018)

- Project planning session 30.10.
- Project topic confirmed 1.11.
- Katariina: Project planning and getting oriented with SFML.

- Arsi: Project planning and getting oriented with SFML.
- Esa: Project planning and getting oriented with SFML and CMake.
- Oskar: Project planning and getting oriented with SFML.

Week 45 (5.11. - 11.11.2018)

- Project Plan submitted on 8.11.
- Oskar: Implementation of core engine functions (update, spawning system, level system). Also implemented user input functions to UserGraphics class. (14h)
- Arsi: Preliminary versions of tower, enemy and projectile classes (12h)
- Katariina: UserGraphics-class constructor and implementation of basic user input (12h)functions (manageEvents and tower spawning). Debugging.
- Esa: Creation of CMake-files and TileMap header files. Debugging. (10h)

Week 46 (12.11. - 18.11.2018)

- Oskar: Implementation of additional engine-class and UserGraphics functions that are related to game mechanics (such as rendering). Implementation of pause and move algorithm for enemies. Cleaning of code. Debugging.(14h)
- Esa: Creation of a proper map for the game and testing of functions. Debugging. (12h)
- Katariina: New map tiles, implementation of special projectiles and related functions. (12h)
- Arsi: Tower-related functions. Shooting algorithm and hitting targets. Cleaning of code.(12h)

Week 47 (19.11. - 25.11.2018)

- Oskar: Added special types of enemies (bosses and minibosses). Balanced the game so that it is playable. Implementation of tower upgrading and related functions to it. Created a proper user panel for the tower menu. (14h)
- Katariina: Debugging (8h)
- Arsi: Graphics debugging (6h)
- Esa: Learning about SFML graphics and creating sprites. (8h)

Week 48 (26.11. - 2.12.2018)

- Project meeting with group 4. 28.11.
- Oskar: Implementation of highscores and top scoreboard. Created flying type enemies to the game. Created a txt-file based system that stores score to a file that can later be uploaded to game. (10h)

- Arsi: Implementation of game menu and related functions. General debugging. (10h)
- Katariina: Implementation of game menu and related functions.(6h)
- Esa: Reading about boost and serialization for saving and loading the game. (6h)

Week 49 (3.12. - 9.12.2018)

- Oskar: Fixing some major bugs related to user interface and adjusted tower menu to appear correctly when upgrading towers. Balancing the game. (4h)
- Arsi: Game title and menu graphics Vietnam style, sound effect and music (8h).
- Katariina: Sound effects and debugging (6h)
- Esa: Memory leakage testing with Valgrind (4h)

Week 50 (10.11. - 16.11.2018)

- Project demo 13.12.
- Final game version and project documentation submitted on 14.12.
- Esa: Program testing with Googletest and Valgrind. Project documentation. (12 h)
- Oskar: Debugging and testing the game manually. Fixed move-algorithm for enemies. Project documentation (4h)
- Katariina: Program testing with Valgrind and Googletest. Code commenting and cleaning. Project documentation. (8h)
- Arsi: Implementation of final sounds. Fixed also a bug related to some sounds not appearing correctly. Project documentation. (8h)

Future implementation

Our project currently fulfills the needed requirements, but there are couple of extra features which could in the future be implemented. The game map could be randomly generated for example with a depth-first search. Thus we wouldn't need to have any .txt - files for the maps and the map would always look different. Also a possibility to save the game would have been useful, but perhaps not that easy to implement. Other improvements could have been updating the game graphics and sounds for more immersive effect.

References

1. SFML. 2018. *Simple and Fast Multimedia Library*. [online] Available at: https://www.sfml-dev.org/. [Accessed 13 December 2018].
2. Open Game Art. 2018. *Open Game Art*. [online] Available at: https://opengameart.org/. [Accessed 13 December 2018].
3. CMake. 2018. *Installing CMake*. [online] Available at: https://cmake.org/install/. [Accessed 14 December 2018].
4. Version.aalto.fi. 2018. [online] Available at: *https://version.aalto.fi/*. [Accessed 14 Dec. 2018].
5. Valgrind. 2018. [online] Available at: http://www.valgrind.org/. [Accessed 14 December 2018].
6. Stack Overflow. 2018. *Valgrind: Memory still reachable with trivial program using <iostream>* [online] Available at: https://stackoverflow.com/questions/30376601/valgrind-memory-still-reachable-with-trivial-program-using-iostream. [Accessed 14 December 2018].
7. GitHub. 2018. *LeakSanitizer detects memory leak in `/usr/lib/x86_64-linux-gnu/libX11.so.6+0x5010c` #231*. [online] Available at: https://github.com/rdesktop/rdesktop/issues/231. [Accessed 15 December 2018].
8. GitHub. 2018. *Google / Googletest*. [online] Available at: https://github.com/google/googletest/. [Accessed 14 December 2018].
9. Trello.com. (2018). *Trello*. [online] Available at: https://trello.com/. [Accessed 14 December 2018].