Aalto-yliopisto
Sähkötekniikan
korkeakoulu

ELEC-A7151
Object oriented programming with C++

Game: Tower-Defense Group 3 Project Documentation

Daniel Nikkari, 653088

Sani Letchu, 715036

Oskari Kiili, 728890

Vili Nieminen, 593465

Game: Tower-Defense

Daniel Nikkari

Sani Letchu

Oskari Kiili

Vili Nieminen

Table of contents

1. Overview

This project implemented a tower-defense game using C++ programming language. “Tower defense (or informally TD) is a subgenre of strategy video game where the goal is to defend a player’s territories or possessions by obstructing enemy attackers, usually achieved by placing defensive structures on or along their path of attack.” from Wikipedia article *Tower defense* (2021). In TD, enemies move in waves and try to move from point A to point B through paths indicated on the map, and the towers placed by the player try to stop them. The objective of the player is the survival of the base.

Players can add towers to the map by clicking the wanted tower and then clicking an available location on the map. The towers will automatically shoot enemy NPCs who come to their range of fire. Players will have money, and earn more from destroying enemy NPCs, that they can spend on buying upgrades for existing towers or buying new towers. The player loses when his base gets destroyed by the enemy NPCs, i.e. when they reach the end of the path.

The game includes multiple maps with increasing difficulty, and hiscores are kept on each map. In later waves, new types of enemies will come about with fearsome properties. The difficulty of a specific map depends on its layout; the number and hitpoints of enemies are scaled each level similarly in all of the maps. Finally, it is possible to create own maps: there are a wide range of textures available with even some custom-made Christmas tiles.

To be precise on the depth and breadth of our program, we implemented all of the basic features with (at least) the following additional features:

- Non-hardcoded maps (read from file)
- Upgradeable towers
- More different kinds of enemies
- A list of highscores saved per map
- Sound effects

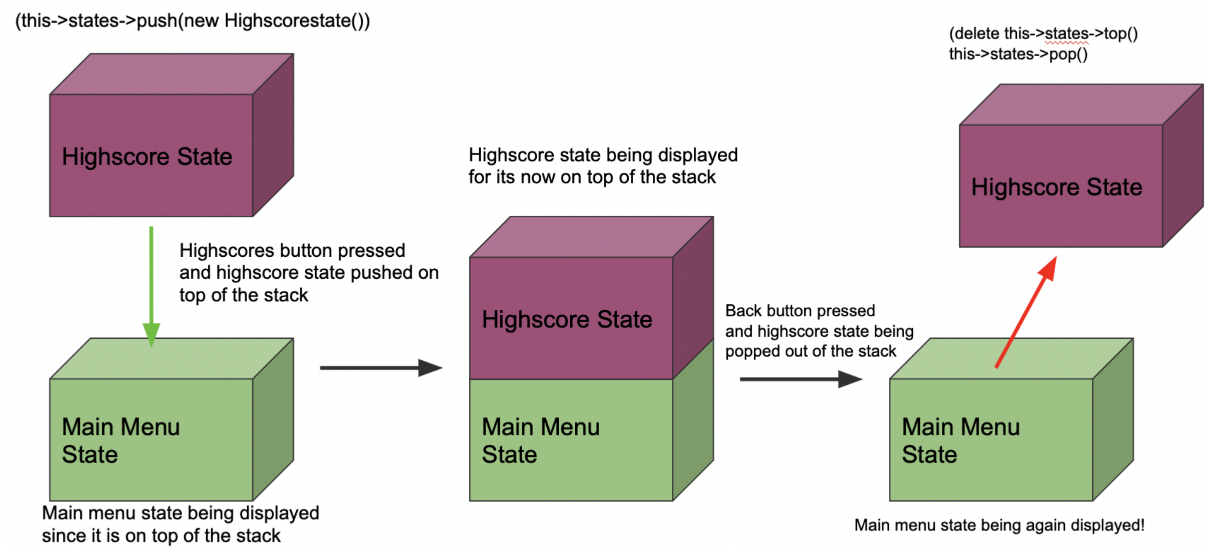
The project documentation proceeds as follows. The software structure with architectural layout and dependencies is given in the second section. This is followed by user instructions; how to build the game and a more detailed overview of game mechanics. In the fourth section, we describe our approach to testing the software. Then, a detailed work log is written in the fifth section.

2. Software structure

Doxygen-based documentation can be found from the /doc directory. To check the HTML-based documentation, open /doc/html/index.html, and for Latex-based PDF documentation open /doc/latex/refman.pdf.

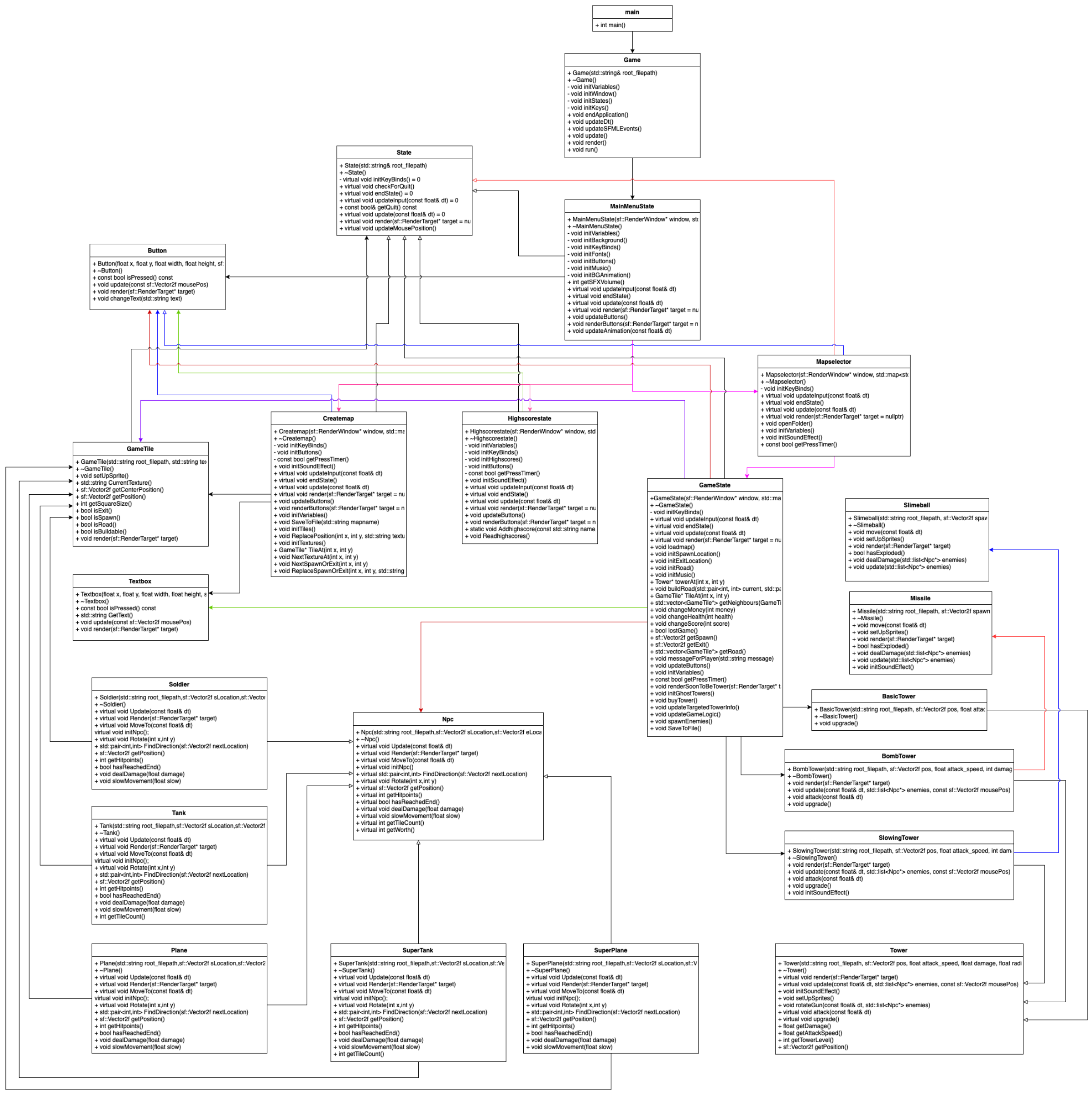
The overall architecture of the program is very reliant on the stack data structure. The main serves as an entry point to the program, which calls the Game class. This runs updates on the state on top of the stack, which can be e.g. main menu state rendering, increasing the number of enemies or pushing another state to the top of the stack.

Here is an example for the states stack (figure 1). The states stack updates only the top state and if the top state gets a new state pushed over it, the new state becomes the top state and thus is updated. When the top state is popped out of the stack, the state under it becomes the top state and is being updated.


(this->states->push(new Highscorestate())
Highscore State
Highscores button pressed and highscore state pushed on top of the stack
Main Menu State
Main menu state being displayed since it is on top of the stack
Highscore state being displayed for its now on top of the stack
Highscore State
Main Menu State
Back button pressed and highscore state being popped out of the stack
(delete this->states->top()
this->states->pop()
Highscore State
Main Menu State
Main menu state being again displayed!

(Figure 1)

Here is a class diagram of the software, the class name is in the top box and the lower box contains all the public (+) and private/protected (-) functions (the inheritence is shown as a hollow arrow):


main
+ int main()
Game
+ Game(std::string& root_filepath)
+ ~Game()
- void initVariables()
- void initWindow()
- void initStates()
- void initKeys()
+ void endApplication()
+ void updateDt()
+ void updateSFMLEvents()
+ void update()
+ void render()
+ void run()
State
+ State(std::string& root_filepath)
+ ~State()
- virtual void initKeyBinds() = 0
+ virtual void checkForQuit()
+ virtual void endState() = 0
+ virtual void updateInput(const float& dt) = 0
+ const bool& getQuit() const
+ virtual void update(const float& dt) = 0
+ virtual void render(sf::RenderTarget* target = nu
+ virtual void updateMousePosition()
MainMenuState
+ MainMenuState(sf::RenderWindow* window, std
+ ~MainMenuState()
- void initVariables()
- void initBackground()
- void initKeyBinds()
- void initFonts()
- void initButtons()
- void initMusic()
- void initBGAnimation()
+ int getSFXVolume()
+ virtual void updateInput(const float& dt)
+ virtual void endState()
+ virtual void update(const float& dt)
+ virtual void render(sf::RenderTarget* target = nu
+ void updateButtons()
+ void renderButtons(sf::RenderTarget* target = n
+ void updateAnimation(const float& dt)
Button
+ Button(float x, float y, float width, float height, sf:
+ ~Button()
+ const bool isPressed() const
+ void update(const sf::Vector2f mousePos)
+ void render(sf::RenderTarget* target)
+ void changeText(std::string text)
Mapselector
+ Mapselector(sf::RenderWindow* window, std::map<std
+ ~Mapselector()
- void initKeyBinds()
+ virtual void updateInput(const float& dt)
+ virtual void endState()
+ virtual void update(const float& dt)
+ virtual void render(sf::RenderTarget* target = nullptr)
+ void openFolder()
+ void initVariables()
+ void initSoundEffect()
+ const bool getPressTimer()
Createmap
+ Createmap(sf::RenderWindow* window, std::ma
+ ~Createmap()
- void initKeyBinds()
- void initButtons()
- const bool getPressTimer()
+ void initSoundEffect()
+ virtual void updateInput(const float& dt)
+ virtual void endState()
+ virtual void update(const float& dt)
+ virtual void render(sf::RenderTarget* target = nu
+ void updateButtons()
+ void renderButtons(sf::RenderTarget* target = n
+ void initVariables()
+ void SaveToFile(std::string mapname)
+ void initTiles()
+ void ReplacePosition(int x, int y, std::string textu
+ void initTextures()
+ GameTile* TileAt(int x, int y)
+ void NextTextureAt(int x, int y)
+ void NextSpawnOrExit(int x, int y)
+ void ReplaceSpawnOrExit(int x, int y, std::string
Highscorestate
+ Highscorestate(sf::RenderWindow* window, std
+ ~Highscorestate()
- void initVariables()
- void initKeyBinds()
- void initHighscores()
- void initButtons()
- const bool getPressTimer()
+ void initSoundEffect()
+ virtual void updateInput(const float& dt)
+ virtual void endState()
+ virtual void update(const float& dt)
+ virtual void render(sf::RenderTarget* target = nu
+ void updateButtons()
+ void renderButtons(sf::RenderTarget* target = n
+ static void Addhighscore(const std::string name
+ void Readhighscores()
GameTile
+ GameTile(std::string root_filepath, std::string te
+ ~GameTile()
+ void setUpSprite()
+ std::string CurrentTexture()
+ sf::Vector2f getCenterPosition()
+ sf::Vector2f getPosition()
+ int getSquareSize()
+ bool isExit()
+ bool isSpawn()
+ bool isRoad()
+ bool isBuildable()
+ void render(sf::RenderTarget* target)
Textbox
+ Textbox(float x, float y, float width, float height, s
+ ~Textbox()
+ const bool isPressed() const
+ std::string GetText()
+ void update(const sf::Vector2f mousePos)
+ void render(sf::RenderTarget* target)
GameState
+GameState(sf::RenderWindow* window, std::ma
+ ~GameState()
- void initKeyBinds()
+ virtual void updateInput(const float& dt)
+ virtual void endState()
+ virtual void update(const float& dt)
+ virtual void render(sf::RenderTarget* target = nu
+ void loadmap()
+ void initSpawnLocation()
+ void initExitLocation()
+ void initRoad()
+ void initMusic()
+ Tower* towerAt(int x, int y)
+ void buildRoad(std::pair<int, int> current, std::pa
+ GameTile* TileAt(int x, int y)
+ std::vector<GameTile*> getNeighbours(GameTi
+ void changeMoney(int money)
+ void changeHealth(int health)
+ void changeScore(int score)
+ bool lostGame()
+ sf::Vector2f getSpawn()
+ sf::Vector2f getExit()
+ std::vector<GameTile*> getRoad()
+ void messageForPlayer(std::string message)
+ void updateButtons()
+ void initVariables()
+ const bool getPressTimer()
+ void renderSoonToBeTower(sf::RenderTarget* t
+ void initGhostTowers()
+ void buyTower()
+ void updateTargetedTowerInfo()
+ void updateGameLogic()
+ void spawnEnemies()
+ void SaveToFile()
Slimeball
+ Slimeball(std::string root_filepath, sf::Vector2f spaw
+ ~Slimeball()
+ void move(const float& dt)
+ void setUpSprites()
+ void render(sf::RenderTarget* target)
+ bool hasExploded()
+ void dealDamage(std::list<Npc*> enemies)
+ void update(std::list<Npc*> enemies)
Missile
+ Missile(std::string root_filepath, sf::Vector2f spawn
+ ~Missile()
+ void move(const float& dt)
+ void setUpSprites()
+ void render(sf::RenderTarget* target)
+ bool hasExploded()
+ void dealDamage(std::list<Npc*> enemies)
+ void update(std::list<Npc*> enemies)
+ void initSoundEffect()
BasicTower
+ BasicTower(std::string root_filepath, sf::Vector2f pos, float attac
+ ~BasicTower()
+ void upgrade()
BombTower
+ BombTower(std::string root_filepath, sf::Vector2f pos, float attack_speed, int damag
+ ~BombTower()
+ void render(sf::RenderTarget* target)
+ void update(const float& dt, std::list<Npc*> enemies, const sf::Vector2f mousePos)
+ void attack(const float& dt)
+ void upgrade()
SlowingTower
+ SlowingTower(std::string root_filepath, sf::Vector2f pos, float attack_speed, int dam
+ ~SlowingTower()
+ void render(sf::RenderTarget* target)
+ void update(const float& dt, std::list<Npc*> enemies, const sf::Vector2f mousePos)
+ void attack(const float& dt)
+ void upgrade()
+ void initSoundEffect()
Soldier
+ Soldier(std::string root_filepath,sf::Vector2f sLocation,sf::Vecto
+ ~Soldier()
+ virtual void Update(const float& dt)
+ virtual void Render(sf::RenderTarget* target)
+ virtual void MoveTo(const float& dt)
virtual void initNpc();
+ virtual void Rotate(int x,int y)
+ std::pair<int,int> FindDirection(sf::Vector2f nextLocation)
+ sf::Vector2f getPosition()
+ int getHitpoints()
+ bool hasReachedEnd()
+ void dealDamage(float damage)
+ void slowMovement(float slow)
Npc
+ Npc(std::string root_filepath,sf::Vector2f sLocation,sf::Vector2f eLoca
+ ~Npc()
+ virtual void Update(const float& dt)
+ virtual void Render(sf::RenderTarget* target)
+ virtual void MoveTo(const float& dt)
+ virtual void initNpc()
+ virtual std::pair<int,int> FindDirection(sf::Vector2f nextLocation)
+ virtual void Rotate(int x,int y)
+ virtual sf::Vector2f getPosition()
+ virtual int getHitpoints()
+ virtual bool hasReachedEnd()
+ virtual void dealDamage(float damage)
+ virtual void slowMovement(float slow)
+ virtual int getTileCount()
+ virtual int getWorth()
Tank
+ Tank(std::string root_filepath,sf::Vector2f sLocation,sf::Vector2f
+ ~Tank()
+ virtual void Update(const float& dt)
+ virtual void Render(sf::RenderTarget* target)
+ virtual void MoveTo(const float& dt)
virtual void initNpc();
+ virtual void Rotate(int x,int y)
+ std::pair<int,int> FindDirection(sf::Vector2f nextLocation)
+ sf::Vector2f getPosition()
+ int getHitpoints()
+ bool hasReachedEnd()
+ void dealDamage(float damage)
+ void slowMovement(float slow)
+ int getTileCount()
Plane
+ Plane(std::string root_filepath,sf::Vector2f sLocation,sf::Vector2
+ ~Plane()
+ virtual void Update(const float& dt)
+ virtual void Render(sf::RenderTarget* target)
+ virtual void MoveTo(const float& dt)
virtual void initNpc();
+ virtual void Rotate(int x,int y)
+ std::pair<int,int> FindDirection(sf::Vector2f nextLocation)
+ sf::Vector2f getPosition()
+ int getHitpoints()
+ bool hasReachedEnd()
+ void dealDamage(float damage)
+ void slowMovement(float slow)
SuperTank
+ SuperTank(std::string root_filepath,sf::Vector2f sLocation,sf::Ve
+ ~SuperTank()
+ virtual void Update(const float& dt)
+ virtual void Render(sf::RenderTarget* target)
+ virtual void MoveTo(const float& dt)
virtual void initNpc();
+ virtual void Rotate(int x,int y)
+ std::pair<int,int> FindDirection(sf::Vector2f nextLocation)
+ sf::Vector2f getPosition()
+ int getHitpoints()
+ bool hasReachedEnd()
+ void dealDamage(float damage)
+ void slowMovement(float slow)
+ int getTileCount()
SuperPlane
+ SuperPlane(std::string root_filepath,sf::Vector2f sLocation,sf::V
+ ~SuperPlane()
+ virtual void Update(const float& dt)
+ virtual void Render(sf::RenderTarget* target)
+ virtual void MoveTo(const float& dt)
virtual void initNpc();
+ virtual void Rotate(int x,int y)
+ std::pair<int,int> FindDirection(sf::Vector2f nextLocation)
+ sf::Vector2f getPosition()
+ int getHitpoints()
+ bool hasReachedEnd()
+ void dealDamage(float damage)
+ void slowMovement(float slow)
Tower
+ Tower(std::string root_filepath, sf::Vector2f pos, float attack_speed, float damage, float radi
+ ~Tower()
+ virtual void render(sf::RenderTarget* target)
+ virtual void update(const float& dt, std::list<Npc*> enemies, const sf::Vector2f mousePos)
+ void initSoundEffect()
+ void setUpSprites()
+ void rotateGun(const float& dt, std::list<Npc*> enemies)
+ virtual void attack(const float& dt)
+ virtual void upgrade()
+ float getDamage()
+ float getAttackSpeed()
+ int getTowerLevel()
+ sf::Vector2f getPosition()

All of the states and the Game class are dependent on SFML. Other external dependencies, like sounds and textures, are brought directly into the src/ directory.

3. Build and use the software

The requirements for the software are fresh SFML, C++ of at least standard 17, and Make. The software can be built easily with Make, and it was successfully built on Linux, Ubuntu 20.04 and macOS. We tried building also on Windows 10 but with problems getting the correct C++ standard this did not succeed without errors. To run the software, just type “make” in project root command line and the software should be built. If one has installed SFML into some other than the default path, one must specify this path as a variable to “DANIELSFMLPATH” in vars.make and use the command “make daniel” instead of just “make”.

After building, the game should launch as a separate window. From there, one can go straight to playing, try to build their own map or check highscores. Selection can be done with left-click of the mouse, scrolling with arrow keys and WASD when necessary, and ESC returns the player to previous state. Pressing ESC in the main menu also closes the game. Let’s cover the three main menu options in order, bolded for easier separation.

To play, choose “PLAY” from the main menu. This opens the map selector, where scrolling works as described. Selecting a map opens the game. The game can be played as intuitively suggested by the UI: select towers from the right-hand side UI with left-click and select a position on the map to place the tower, placing it there with another left-click. Then, the tower can be selected by left-click and upgraded or sold from the UI. The towers automatically target and shoot enemies when they are in its range.

A wave or level of enemies begins either after 30 seconds or when “SKIP” is pressed to start immediately. The enemies start marching forward from the start of the path and try to reach the end, the base, after which the game ends if any of them succeed. Destroying enemies increases both score and available bank, from which the player can solidify his or her defense.

There are five different types of enemies: soldiers, tanks, planes, super tanks and super planes. The first are one shots while the tanks spawn three soldiers after they are destroyed. Planes are faster enemies with moderate hitpoints. Super tanks are massive enemies with huge hitpoints, and they spawn three tanks after they are destroyed. Finally, super planes are the fastest enemy that spawn three planes.

There are also three different types of towers: basic tower, bomb tower and slime tower. The first is a balance between one shot damage and speed. The bomb tower deals greater damage on its blast radius, but shoots slow bombs in longer intervals, so it cannot really reach the faster enemies. The slime tower shoots similarly slow slime balls that slow down the enemies that get stuck in them.

The hitpoints of the enemies differ and scale per wave number. Further, the upgrades of the towers increase exponentially in cost such that the difficulty stays until later rounds.

To save the score to highscores, the player can write their name in the textbox in the UI. Then, after the game ends the score is automatically written to highscores under the corresponding map. The player is then returned back to the map selector state.

The map builder can be accessed by pressing “CREATE MAP” from the main menu. Map builder UI consists of a textbox, text input for the map name, “SAVE” button, saves the currently displayed map with the current name to maps folder, and the map built from multiple tiles. Changing tiles happens by just right clicking a specific tile. There are two types of tiles: road tiles and nonroad tiles. Towers can only be built on nonroad tiles. There is no way for the player to know what is what, but for example grass is nonroad tile and mud is road tile. Road for enemies cannot branch and the road must be surrounded by nonroad tiles. Lastly, when the map is complete the player needs to specify where is exit and spawn for enemies. By using left click texture appears on top of the tile specifying if it is an exit or a spawn. Texture looking like x meaning exit and tool meaning spawn. These need to be placed anywhere at the edges of the map. Finally, the player can save the map. However, saving doesn't check if the map is viable so the game will likely crash if the player tries to play an unviable map.

Highscores can be accessed and explored from the corresponding button. They are displayed per map, and one can try to claim their top spot by playing and remembering to fill the textbox in the UI with their name. Scrolling and returning works as before.

4. Testing

The chosen testing framework was Catch2 due to its simplicity. However, due to the heat of intense coding, no proper testing suite was ever implemented. This lack of formal test coverage was compensated by hours of concentrated in-game testing. Each of the features was thus tested manually from the game GUI. Even more so, the team spent a great deal of time balancing the game mechanics, nerfing some upgrades and boosting end-game difficulty, such that the gaming experience would be the best available.

We also tested for memory leaks with valgrind, but with our limited experience, and motivation, this still led to some memory leaks leaking into the end product. These were confirmed by monitoring system memory usage during longer gaming sessions.

5. Work log

Every team member did all kinds of jobs from planning, environment setup, design and development. Here is the overview of the work, more detailed descriptions can be found from the Meeting-notes.md file. Each box of work description is prepended with the hours of work.

Week (#) / Member	**Daniel**	**Sani**	**Oskari**	**Vili**
1 (44)	(8) Project plan / environment setup	(8) Project plan / environment setup	(8) Project plan / environment setup	(8) Project plan / environment setup
2 (45)	(20) Stack framework and base states	(8) Highscore and map builder foundation and gametiles	(6) Further setup implementation	(7) Devised Git flow and test framework
3 (46)	(6) Improved menu	(5) Finished map builder	(3) Created entity foundation	(11) Created Makefile and and Doxyfile
4 (47)	(1) Small tweaks	(30) Textbox class, map selector state. All towers and missiles	(25) NPC class, memory management	(0)
5 (48)	(3) Scrolling improvements	(30) Gaming state UI, money system, tower placing and wave system slimeballs	(4) NPC subclasses	(0)
6 (49)	(25) Game balancing, memory management, sound effects and music, additional textures (supertank, superplane, etc.)	(15) Game balancing	(15) Game balancing	(25) Game balancing, high score saving, Doxygen
Total hours	~65	~90	~60	~50

On top of this work, each team member attended around 10 hours of meetings.