ChessTD project documentation

ELEC-A7151

Joel Toppinen, Miikka Åsnabrygg, Petrus Nikoskinen, Petteri Kippo

1. Overview

Our project is a classic tower defense game, in which the core gameplay consists of building towers in order to stop incoming waves of enemies. The theme of the game revolves around board games, with the towers being different chess pieces.

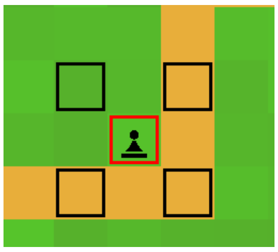
The basic features of the game include different maps with unique enemy paths and wave compositions, being mixtures of different kinds of enemies. Some enemies can take more damage before dying whereas some split into multiple weaker enemies when going down. Map difficulty is adjusted through the path length as well as the strength and number of enemies. Shorter paths and stronger enemies make it harder for the player to take them down.

The gameplay loop consists of destroying enemies with towers, earning money doing so, and then spending the money to build different towers to destroy even more enemies. Stronger enemies award more money to compensate for the additional effort and risk required to take them down. Money can also be used to upgrade existing towers to make them stronger, e.g. have more range or deal more damage. The player has different types of towers to choose from. Towers have different ranges, damage values as well as targeting patterns. These attributes also affect the purchase and upgrade prices of the tower.

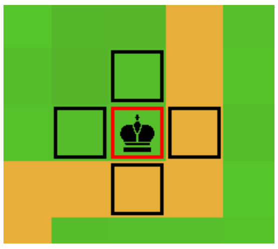
The game’s user interface allows the player to drag and drop towers into suitable positions and select towers by clicking on them. Selected towers can be upgraded using the menu on the right-hand side of the screen. It also has a button for pausing/playing the game. The towers etc. can only be interacted with if the game is paused. The game can be paused manually by the player at any point during the round, and automatically after each wave.

By default, every time a tower shoots, it shoots at the enemy closest to the end (except for towers with special targeting patterns), and gets the possible rewards gained from enemies. Towers do not always shoot when called and have a certain cooldown period after each shot, which can be shortened with upgrades. Tower's range, speed, and damage can be upgraded according to the upgrade table specific to each tower type. In the main game loop, a vector of tower pointers is stored, and all the towers are called through this vector, and new towers are added to it.

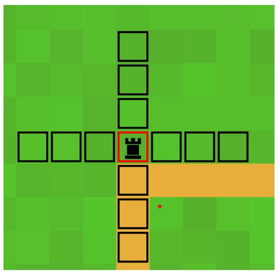
A brief overview of the towers:



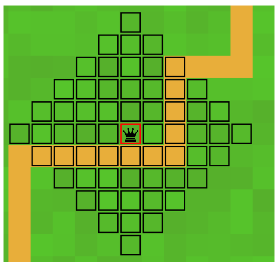
- The Pawn shoots at the tiles cornering the tile where it's in. Basic cheap tower for almost any situation.



- The King shoots at tiles close to it in a flood-fill pattern. The King doesn't exceed in any of the aspects, but the player gets 2x the reward from enemies killed by the King.

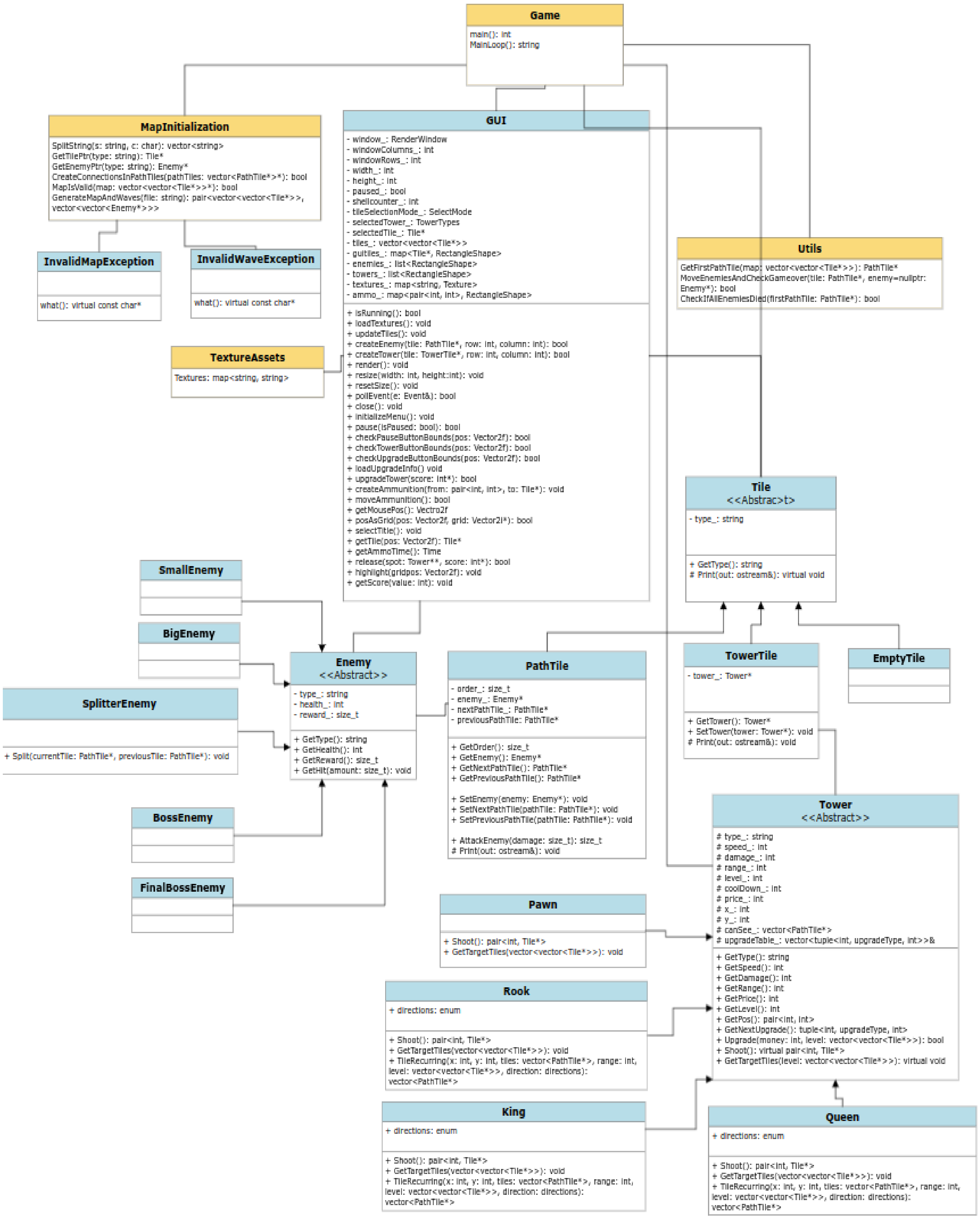


- The Rook shoots at the strongest enemy instead of the first, unlike other towers. The Rook only shoots in the cardinal directions(up, down, left, and right). The "Sniper tower" of the game.



- The Queen is the strongest tower, with exceptional range, damage, and speed. Very expensive, but the price is worth it.

2. Software structure


Game
main(): int
MainLoop(): string
MapInitialization
SplitString(s: string, c: char): vector<string>
GetTilePtr(type: string): Tile*
GetEnemyPtr(type: string): Enemy*
CreateConnectionsInPathTiles(pathTiles: vector<PathTile*>*): bool
MapIsValid(map: vector<vector<Tile*>>*): bool
GenerateMapAndWaves(file: string): pair<vector<vector<Tile*>>, vector<vector<Enemy*>>>
InvalidMapException
what(): virtual const char*
InvalidWaveException
what(): virtual const char*
TextureAssets
Textures: map<string, string>
GUI
- window_: RenderWindow
- windowColumns_: int
- windowRows_: int
- width_: int
- height_: int
- paused_: bool
- shellcounter_: int
- tileSelectionMode_: SelectMode
- selectedTower_: TowerTypes
- selectedTile_: Tile*
- tiles_: vector<vector<Tile*>>
- guitiles_: map<Tile*, RectangleShape>
- enemies_: list<RectangleShape>
- towers_: list<RectangleShape>
- textures_: map<string, Texture>
- ammo_: map<pair<int, int>, RectangleShape>
+ isRunning(): bool
+ loadTextures(): void
+ updateTiles(): void
+ createEnemy(tile: PathTile*, row: int, column: int): bool
+ createTower(tile: TowerTile*, row: int, column: int): bool
+ render(): void
+ resize(width: int, height:int): void
+ resetSize(): void
+ pollEvent(e: Event&): bool
+ close(): void
+ initializeMenu(): void
+ pause(isPaused: bool): bool
+ checkPauseButtonBounds(pos: Vector2f): bool
+ checkTowerButtonBounds(pos: Vector2f): bool
+ checkUpgradeButtonBounds(pos: Vector2f): bool
+ loadUpgradeInfo() void
+ upgradeTower(score: int*): bool
+ createAmmunition(from: pair<int, int>, to: Tile*): void
+ moveAmmunition(): bool
+ getMousePos(): Vectro2f
+ posAsGrid(pos: Vector2f, grid: Vector2i*): bool
+ selectTitle(): void
+ getTile(pos: Vector2f): Tile*
+ getAmmoTime(): Time
+ release(spot: Tower**, score: int*): bool
+ highlight(gridpos: Vector2f): void
+ getScore(value: int): void
Utils
GetFirstPathTile(map: vector<vector<Tile*>>): PathTile*
MoveEnemiesAndCheckGameover(tile: PathTile*, enemy=nullptr: Enemy*): bool
CheckIfAllEnemiesDied(firstPathTile: PathTile*): bool
Tile
<<Abstrac>t>
- type_: string
+ GetType(): string
# Print(out: ostream&): virtual void
SmallEnemy
BigEnemy
SplitterEnemy
+ Split(currentTile: PathTile*, previousTile: PathTile*): void
Enemy
<<Abstract>>
- type_: string
- health_: int
- reward_: size_t
+ GetType(): string
+ GetHealth(): int
+ GetReward(): size_t
+ GetHit(amount: size_t): void
BossEnemy
FinalBossEnemy
PathTile
- order_: size_t
- enemy_: Enemy*
- nextPathTile_: PathTile*
- previousPathTile: PathTile*
+ GetOrder(): size_t
+ GetEnemy(): Enemy*
+ GetNextPathTile(): PathTile*
+ GetPreviousPathTile(): PathTile*
+ SetEnemy(enemy: Enemy*): void
+ SetNextPathTile(pathTile: PathTile*): void
+ SetPreviousPathTile(pathTile: PathTile*): void
+ AttackEnemy(damage: size_t): size_t
# Print(out: ostream&): void
TowerTile
- tower_: Tower*
+ GetTower(): Tower*
+ SetTower(tower: Tower*): void
# Print(out: ostream&): void
EmptyTile
Tower
<<Abstract>>
# type_: string
# speed_: int
# damage_: int
# range_: int
# level_: int
# coolDown_: int
# price_: int
# x_: int
# y_: int
# canSee_: vector<PathTile*>
# upgradeTable_: vector<tuple<int, upgradeType, int>>&
+ GetType(): string
+ GetSpeed(): int
+ GetDamage(): int
+ GetRange(): int
+ GetPrice(): int
+ GetLevel(): int
+ GetPos(): pair<int, int>
+ GetNextUpgrade(): tuple<int, upgradeType, int>
+ Upgrade(money: int, level: vector<vector<Tile*>>): bool
+ Shoot(): virtual pair<int, Tile*>
+ GetTargetTiles(level: vector<vector<Tile*>>): virtual void
Pawn
+ Shoot(): pair<int, Tile*>
+ GetTargetTiles(vector<vector<Tile*>>): void
Rook
+ directions: enum
+ Shoot(): pair<int, Tile*>
+ GetTargetTiles(vector<vector<Tile*>>): void
+ TileRecurring(x: int, y: int, tiles: vector<PathTile*>, range: int, level: vector<vector<Tile*>>, direction: directions): vector<PathTile*>
King
+ directions: enum
+ Shoot(): pair<int, Tile*>
+ GetTargetTiles(vector<vector<Tile*>>): void
+ TileRecurring(x: int, y: int, tiles: vector<PathTile*>, range: int, level: vector<vector<Tile*>>, direction: directions): vector<PathTile*>
Queen
+ directions: enum
+ Shoot(): pair<int, Tile*>
+ GetTargetTiles(vector<vector<Tile*>>): void
+ TileRecurring(x: int, y: int, tiles: vector<PathTile*>, range: int, level: vector<vector<Tile*>>, direction: directions): vector<PathTile*>

This is a basic UML of our program's class hierarchy. The levels are stored in tiles containing all the relevant information. Game stores the map as 2D Array with Tile objects. PathTile manages the Enemy objects that are on that specific tile. PathTiles are linked to each other like a doubly linked list, so the tile can move enemies to the next PathTile. TowerTiles are used to contain towers.Enemy, Tower and Tile are abstract base classes.

The GUI class acts as an interface to the graphical elements of the game. It uses the SFML library to visualize different game elements and allows users to interact with the game through its methods. They contain functionalities like highlighting a single tile or rendering all of the game elements and are used in the main game loop to visualize the events that happen in-game.

Tower-class does not store any unnecessary information related to the level, but is given a pointer to the map every time possible changes are to be made. It stores pointers to all the path tiles it can see. For the more complicated target areas an additional recursive method for handling them is used. Every time a tower shoots at an enemy, it returns a pair of int and Tile*, int is the possible rewards gained and Tile* is the targeted tile, used for drawing the shooting animation. As cooldown is handled by each tower individually, the towers shoot-method is called every loop of the main game loop but the tower doesn't shoot every time the method is called, and in the cases where the tower does not shoot, the return pairs tile pointer is a null pointer, as a message not to draw a shot.

There are currently five different enemy types: ‘small’, ‘big’, ‘splitter’, ‘boss’, and ‘final boss’ enemies. All of them have different health points and rewards. The Splitter enemy differs from the others in that when it dies, it spawns two small enemies to the path. The reward parameter is the number of points the player gets for killing the enemy.

MapInitialization header contains the tools responsible for generating a new map and its enemy waves from a custom-formatted text file. It also checks whether the map and the waves in the file are appropriate, if not it throws custom errors for them. Map files have specific forms to generate the map: e.g:

```
Map:
e,e,e,e,e,e,e,e
0,1,e,e,6,7,8,e
e,2,3,4,5,e,9,e
e,e,e,e,e,e,10,e
Waves:
1:s,s,s,s
2:s,s,b,b
3:b,b,sp,sp
4:m,m,f
```

“e” means EmptyTile (There is no tower yet) and numbers are pathTiles in order. Waves are at the bottom of the file in the ordered list. “s” is SmallEnemy, “b” is BigEnemy, “sp” is SplitterEnemy, “m” is BossEnemy and “f” is the FinalBoss. Tiles and enemies in waves are separated by comma characters. The GUI can to a certain degree draw different-sized maps but all of the menu elements don’t currently scale in size and creating really small maps might cause some functionalities to not fit on the screen.

Utils contains some general-purpose functions that are used in the Game loop. The functions check some game states or update them, like moving all enemies forward one tile and checking if the game is over. The assets folder contains the pictures used to visualize towers and enemies.

3. Building and using the software

a. Building the software

The project root contains a CMakeLists.txt file that the CMake build tool can use to create a Makefile for the project executables. Building that Makefile creates two executables: “ChessTD” which is the main game and “Test” which runs the automated set of tests. In order to be able to build the software, an external library SFML has to be installed on the device.

b. Basic user guide

Running the main game opens up the first map and the player is able to place towers on the map by dragging them from the side menu into a suitable empty position. Places towers can also be upgraded by selecting them and pressing upgrade from the side menu. Both of these actions consume points, which can be earned by killing enemies using the towers. A wave can be initiated by pressing the “Play” button in the upper right corner of the window. The same button can also be used to pause the game mid-wave. Note that the player can interact with towers only if the game is paused.

Once the player beats all of the waves on a map, a new game is automatically opened with the next map. If the player fails the same map starts once again from the beginning. This way the player does not lose all progression if he fails later down the line. Once all maps have been cleared the game closes down. The game can also be closed at any time by closing the window using the close button in the upper right corner of the window.

4. Testing

The GUI test is aimed at testing whether the external SFML library works on the machine and whether the GUI class can successfully visualize a basic map. It should open the graphical window and stay waiting until the user closes the window, after which the test concludes.

The EnemyTest creates different types of enemies and tests that all important Enemy class methods are working correctly. These tests also test the utility methods used to e.g. move enemies. It is also tested that there are no memory issues from Enemy or tile classes.

The TowerTest places different towers and enemies on the map and tests that they interact with each other in an intended manner. This includes all important aspects like range, damage, upgrading, and their costs, as well as reward collection. The test results are printed in a practical format and are easy to grasp without extensive knowledge of the program.

5. Work log

Week 1
Miikka: Created Tile and Enemy classes to represent different tiles and enemies and did the tests to them. Tested that SFML library is working on my computer and fixing CMakeLists file. (~5h)

Petteri: Started implementing Tower base class and implementation. Adding Doxygen documentation as we’re moving onward(~6h)

Joel: Studied both the CMake and SFML features. Created the CMake configuration necessary to build the project as well as implemented a basic window that can show tiles. (~9h)

Petrus: Worked through a lot of issues with SFML not rendering the game window, researching the documentation, early tests. (~6h)

Week 2

Miikka: Created ability to read and build maps from a file. As well as a function for moving all enemies forward in path and check if some enemy reaches the goal, it's game over. (~6h)

Petteri: Got the base class and first implementation working. Wrote tests to confirm how it's working. (~5h)

Joel: Implemented the basic UI elements so that it also draws enemies from the tiles. Created the main game file that runs the loop and moves enemies around (~8h)

Petrus: Created placeholder textures and implemented their rendering in the GUI, implemented loadTextures to preload assets at game launch, though functionality is not yet integrated in master (~4h)

Week 3

Miikka: Implemented the wave system that is also read from the map file, so the difficulty of the map can be easily modified through the file. The map now plays through all of its waves. Added the Split() method to SplitterEnemy, when the splitter dies, it spawns two SmallEnemies to its path. (~6h)

Petteri: Added rest of the needed towers, and additional methods needed for firing animation and money collection. (~5h)

Joel: Implemented the in-game menu with functionalities for placing towers, upgrading them, and pausing the game. Also added those functionalities to the main game loop so that the game could be played by the end of the week. Finally added a firing animation for the towers.(~12h)

Petrus: Sprite rendering fully integrated after some initial merge issues, debugging (~3h)

Week 4

Miikka: Created the "map cycling", when map1's all waves are survived map2 will be played next. If it's game over, the same map is played again. Fixed bug in Split() method. Created a new UML diagram. Wrote more Doxygen comments to code. Wrote project documentation and worked on demo slides. (~8h)

Petteri: Added more methods for towers needed for user interface. Added 4 new maps, 2 boss enemies and graphical icons for all towers and enemies. (~14h)

Joel: Implemented the tower range visualization and fine-tuned the coloring of menu elements. Refactored/cleaned/documented previously written code. Wrote project documentation (~8h)

Petrus: Worked on spriting, researched SFML audio capabilities, did a lot of testing and troubleshooting with audio but could not get it to function. Worked on presentation and did some playtesting (~8h)