PROGRAMMING STUDIO 2 TOWER DEFENSE PROJECT “NIGHT OF FRIGHT”

Nykänen Nuutti 900650

COMPUTER SCIENCE 1ST YEAR 28.4.2021

2. General description

During the course Programming Studio 2, I have chosen 'Tower Defense' as the subject for my project over the course of two months. A tower defense game is a game in which a group of enemies that aim for a target location through a predetermined route. The player has to stop the enemies by hiring recruits next to the path that shoot or in another way defend against the enemy. The recruits are upgradable and the player will experience significant growth in strength as the game progresses. Recruit are either attack or support recruits – attack recruits shoot projectiles that damage enemies and support recruits have a recruit.

Enemies come in clear chunks, waves. It is the player's initiative to start a new wave. Each wave (enemies that they contain) is read from a text file within the project files.

The player can 'level up' through numerous different routes thanks to a pool of towers to choose from and their respective upgrades. The player gains money from defeating enemies and clearing waves. The player loses health when enemies reach the end.

I aimed for an intermediate difficulty from the start and I think I have done this project on that same level. The project does not contain anything too difficult for me, but it was not particularly easy to create.

3. User interface

When the player starts the application, he or she sees a map that contains the enemy travel path. The player also sees a recruit store, which contains all the recruits the player can hire during the game. Below the store, there is space possibly containing more information on a possible selected recruit and a button dedicated to starting the next wave (continuing the game).

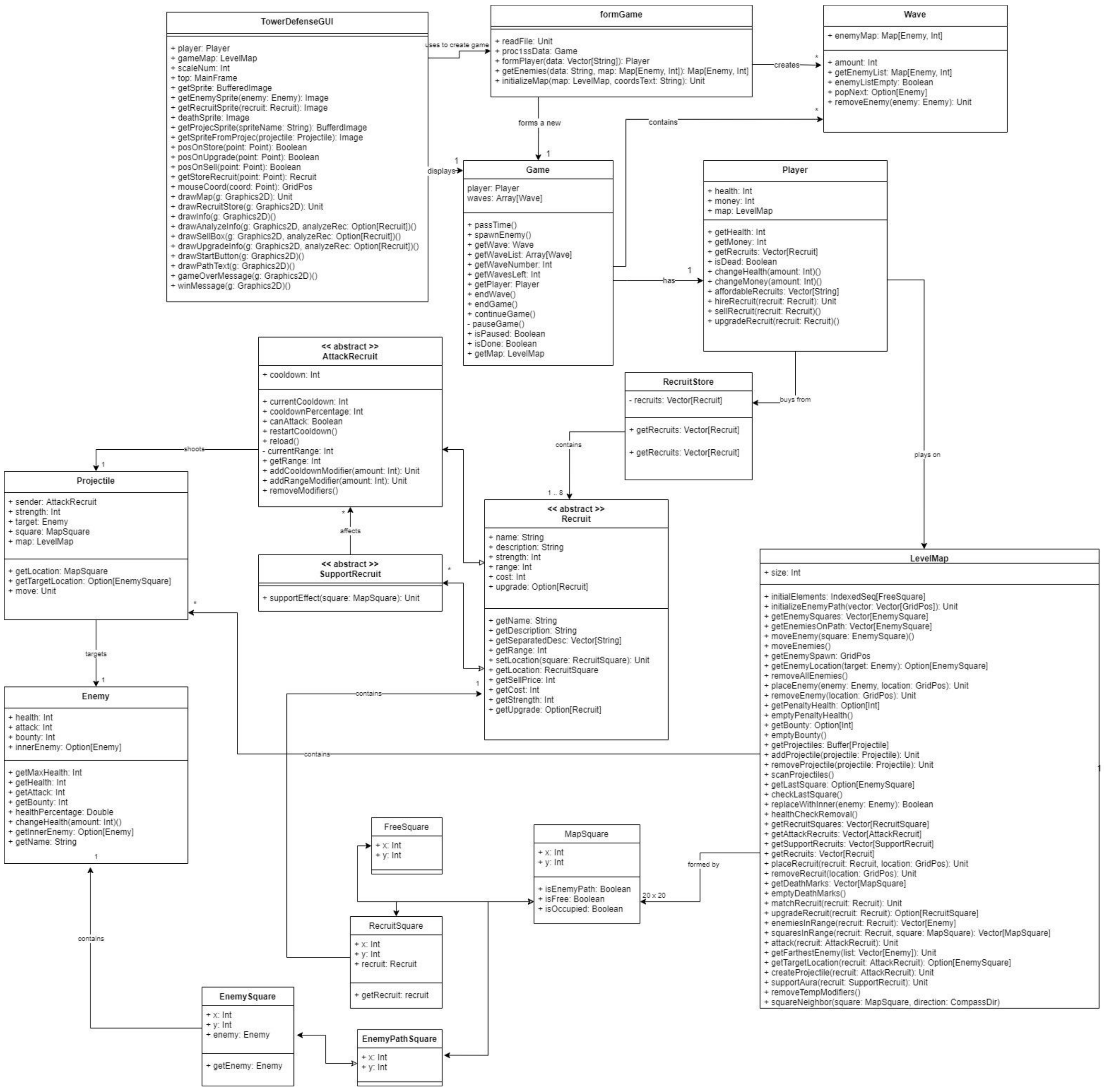
At any point during the game, the player may hire, sell and upgrade recruits given that the player has enough money. All of these functions are accessed from the right side of the GUI. These three functions are the primary ways of playing the game. Engaging with enemies does not require any input from the player – the gameplay is structured solely on placing towers in a desired spot and upgrading them.

Placing a recruit requires the player to click on the desired recruit within the store. The game checks if the player has enough money for the purchase (this is visible via the GUI displaying a darker color on affordable recruits). Clicking on an affordable recruit results in the player being able to place the selected recruit on the grid map. This is visible in the GUI. The actual placement happens by clicking on a free space on the grid map.

Aside from managing recruits, the only other function the player has is starting a wave when the player so desires, given that a wave is not already active. All of these functions are made from the right side of the GUI in their respective, titled buttons.

4. Program structure

UML diagram of this project:



TowerDefenseGUI
+ player: Player
+ gameMap: LevelMap
+ scaleNum: Int
+ top: MainFrame
+ getSprite: BufferedImage
+ getEnemySprite(enemy: Enemy): Image
+ getRecruitSprite(recruit: Recruit): Image
+ deathSprite: Image
+ getProjecSprite(spriteName: String): BufferdImage
+ getSpriteFromProjec(projectile: Projectile): Image
+ posOnStore(point: Point): Boolean
+ posOnUpgrade(point: Point): Boolean
+ posOnSell(point: Point): Boolean
+ getStoreRecruit(point: Point): Recruit
+ mouseCoord(coord: Point): GridPos
+ drawMap(g: Graphics2D): Unit
+ drawRecruitStore(g: Graphics2D): Unit
+ drawInfo(g: Graphics2D)()
+ drawAnalyzeInfo(g: Graphics2D, analyzeRec: Option[Recruit])()
+ drawSellBox(g: Graphics2D, analyzeRec: Option[Recruit])()
+ drawUpgradeInfo(g: Graphics2D, analyzeRec: Option[Recruit])()
+ drawStartButton(g: Graphics2D)()
+ drawPathText(g: Graphics2D)()
+ gameOverMessage(g: Graphics2D)()
+ winMessage(g: Graphics2D)()
uses to create game
formGame
+ readFile: Unit
+ proc1ssData: Game
+ formPlayer(data: Vector[String]): Player
+ getEnemies(data: String, map: Map[Enemy, Int]): Map[Enemy, Int]
+ initializeMap(map: LevelMap, coordsText: String): Unit
creates
Wave
+ enemyMap: Map[Enemy, Int]
+ amount: Int
+ getEnemyList: Map[Enemy, Int]
+ enemyListEmpty: Boolean
+ popNext: Option[Enemy]
+ removeEnemy(enemy: Enemy): Unit
forms a new
contains
displays
Game
player: Player
waves: Array[Wave]
+ passTime()
+ spawnEnemy()
+ getWave: Wave
+ getWaveList: Array[Wave]
+ getWaveNumber: Int
+ getWavesLeft: Int
+ getPlayer: Player
+ endWave()
+ endGame()
+ continueGame()
- pauseGame()
+ isPaused: Boolean
+ isDone: Boolean
+ getMap: LevelMap
has
Player
+ health: Int
+ money: Int
+ map: LevelMap
+ getHealth: Int
+ getMoney: Int
+ getRecruits: Vector[Recruit]
+ isDead: Boolean
+ changeHealth(amount: Int)()
+ changeMoney(amount: Int)()
+ affordableRecruits: Vector[String]
+ hireRecruit(recruit: Recruit): Unit
+ sellRecruit(recruit: Recruit)()
+ upgradeRecruit(recruit: Recruit)()
<< abstract >>
AttackRecruit
+ cooldown: Int
+ currentCooldown: Int
+ cooldownPercentage: Int
+ canAttack: Boolean
+ restartCooldown()
+ reload()
- currentRange: Int
+ getRange: Int
+ addCooldownModifier(amount: Int): Unit
+ addRangeModifier(amount: Int): Unit
+ removeModifiers()
shoots
RecruitStore
- recruits: Vector[Recruit]
+ getRecruits: Vector[Recruit]
+ getRecruits: Vector[Recruit]
buys from
contains
plays on
Projectile
+ sender: AttackRecruit
+ strength: Int
+ target: Enemy
+ square: MapSquare
+ map: LevelMap
+ getLocation: MapSquare
+ getTargetLocation: Option[EnemySquare]
+ move: Unit
affects
<< abstract >>
SupportRecruit
+ supportEffect(square: MapSquare): Unit
1 .. 8
<< abstract >>
Recruit
+ name: String
+ description: String
+ strength: Int
+ range: Int
+ cost: Int
+ upgrade: Option[Recruit]
+ getName: String
+ getDescription: String
+ getSeparatedDesc: Vector[String]
+ getRange: Int
+ setLocation(square: RecruitSquare): Unit
+ getLocation: RecruitSquare
+ getSellPrice: Int
+ getCost: Int
+ getStrength: Int
+ getUpgrade: Option[Recruit]
LevelMap
+ size: Int
+ initialElements: IndexedSeq[FreeSquare]
+ initializeEnemyPath(vector: Vector[GridPos]): Unit
+ getEnemySquares: Vector[EnemySquare]
+ getEnemiesOnPath: Vector[EnemySquare]
+ moveEnemy(square: EnemySquare)()
+ moveEnemies()
+ getEnemySpawn: GridPos
+ getEnemyLocation(target: Enemy): Option[EnemySquare]
+ removeAllEnemies()
+ placeEnemy(enemy: Enemy, location: GridPos): Unit
+ removeEnemy(location: GridPos): Unit
+ getPenaltyHealth: Option[Int]
+ emptyPenaltyHealth()
+ getBounty: Option[Int]
+ emptyBounty()
+ getProjectiles: Buffer[Projectile]
+ addProjectile(projectile: Projectile): Unit
+ removeProjectile(projectile: Projectile): Unit
+ scanProjectiles()
+ getLastSquare: Option[EnemySquare]
+ checkLastSquare()
+ replaceWithInner(enemy: Enemy): Boolean
+ healthCheckRemoval()
+ getRecruitSquares: Vector[RecruitSquare]
+ getAttackRecruits: Vector[AttackRecruit]
+ getSupportRecruits: Vector[SupportRecruit]
+ getRecruits: Vector[Recruit]
+ placeRecruit(recruit: Recruit, location: GridPos): Unit
+ removeRecruit(location: GridPos): Unit
+ getDeathMarks: Vector[MapSquare]
+ emptyDeathMarks()
+ matchRecruit(recruit: Recruit): Unit
+ upgradeRecruit(recruit: Recruit): Option[RecruitSquare]
+ enemiesInRange(recruit: Recruit): Vector[Enemy]
+ squaresInRange(recruit: Recruit, square: MapSquare): Vector[MapSquare]
+ attack(recruit: AttackRecruit): Unit
+ getFarthestEnemy(list: Vector[Enemy]): Unit
+ getTargetLocation(recruit: AttackRecruit): Option[EnemySquare]
+ createProjectile(recruit: AttackRecruit): Unit
+ supportAura(recruit: SupportRecruit): Unit
+ removeTempModifiers()
+ squareNeighbor(square: MapSquare, direction: CompassDir)
targets
contains
contains
Enemy
+ health: Int
+ attack: Int
+ bounty: Int
+ innerEnemy: Option[Enemy]
+ getMaxHealth: Int
+ getHealth: Int
+ getAttack: Int
+ getBounty: Int
+ healthPercentage: Double
+ changeHealth(amount: Int)()
+ getInnerEnemy: Option[Enemy]
+ getName: String
FreeSquare
+ x: Int
+ y: Int
MapSquare
+ x: Int
+ y: Int
+ isEnemyPath: Boolean
+ isFree: Boolean
+ isOccupied: Boolean
formed by
20 x 20
RecruitSquare
+ x: Int
+ y: Int
+ recruit: Recruit
+ getRecruit: recruit
contains
EnemySquare
+ x: Int
+ y: Int
+ enemy: Enemy
+ getEnemy: Enemy
EnemyPathSquare
+ x: Int
+ y: Int

The UML differs quite a lot from the plan UML – most notably the LevelMap class has many, many more methods.

The program is split into clear parts made by different components of the game. It all builds into the GUI – TowerDefenseGUI forms a Game through formGame and then knows what game it will display over the course of the program. The GUI gets all info within the game from the Game class, as everything within the game is naturally within the Game class.

The class Game oversees the entire gameplay on a larger scale. It passes time, triggering almost everything within the game (given that the game is not paused, between waves). This means calling enemies to move, projectiles to fly and even checking whether or not the player is alive. Game also keeps track of waves and makes sure they are correctly started and ended in the right order.

A Game has a Player. A Player has health and money, both crucial to the player experience. The Player class is crucial in managing what the player does: buying recruits, managing money, keeping track of health.

The Player can buy Recruits from the RecruitStore class. RecruitStore itself is not a store, but it contains all available recruits to purchase during the game. It does not contains recruits that come from upgrading other recruits – only the "first level" recruits are available to purchase. The class is a way to manage what recruits can be bought and to easily analyze all available recruits (for example, if the Player can afford any of them).

A Recruit is hired by the Player. Each one has a dedicated cost, strength, range, upgrade. They themselves know their own location on the game map. As the Recruit class is abstract, it functions as a skeleton for the two Recruit types – AttackRecruit and SupportRecruit.

A SupportRecruit inherits from the Recruit class. Each SupportRecruit has their own method for supporting each turn. For example, the DrFrankenstein recruit boosts the range of all recruits within its range. This is carried out for every passing of time while the recruit is alive.

An AttackRecruit also inherits from the Recruit class. It is dedicated to attacking enemies directly by launching projectiles from its location. Each AttackRecruit has their own cooldown, a way of expressing how long it takes for the recruit to fire again after firing. The 'modifiers' within AttackRecruit are given by SupportRecruits. For example, the FatherMerrin class adds a cooldown modifier equal to its strength to all AttackRecruits in its range. The reload() method reduces the currentCooldown by one – the recruit only fires when currentCooldown equals to zero.

Projectiles launched by an AttackRecruit have their dedicated targets decided upon at the moment they are fired. Their strength equals to their sender's strength and they deal damage equal to it when they arrive at the same location as their target. Projectiles approach their target whenever they can – twice as fast as enemy movement.

The class Enemy describes enemies moving on the map. Enemies do not have too much detail as they hardly do anything else than move at a certain pace. Enemy power differences are seen in their health and in what enemy it might contain. For example, the Bat enemy contains Dracula. This means that upon death, the Bat enemy changes into Dracula – a fierce foe indeed. Enemies do have an attack integer, which means how much health they take from the player as they arrive to the end. Enemies also have their own bounty – a sum that is given to the Player upon their defeat.

Perhaps the most crucial class of all, LevelMap, juggles each of these aforementioned game components. A LevelMap is a grid inherited from the class Grid that is formed by instances of class MapSquare. The LevelMap has a lot of functions mostly called by the Game and Player classes. It moves enemies on the determined enemyTravelPath, it keeps track of where every element is (recruits, enemies, projectiles), it launches attacks from recruits and removes enemies if they have died. The LevelMap class contains many methods analyzing its state for further use: enemiesInRange returns what enemies are in a recruit's range, getRecruitSquares returns all locations of recruits within the map, checkLastSquare() returns what enemy has reached the end.

Finally, the elements which LevelMap consists of are MapSquares described by the MapSquare class. They inherit from the GridPos class. Their function is simple – a FreeSquare describes a square in which the player can place a recruit, a RecruitSquare is occupied by a recruit and an EnemyPathSquare is not free and only enemies can exist within in. An EnemySquare inherits from the EnemyPathSquare and, as a parameter, it has the enemy that it contains.

Another implementation could have been to separate Player and LevelMap from each other – to link them both into the Game class directly. This would have been a simpler implementation as they could indeed function on their own and communicate through Game. I went with this implementation as the Player directly calls commands to the map – hiring, selling, etc.

5. Program structure

In this project, what required most work with algorithmic planning was projectile movement. What projectile movement required was to find what grid square in its immediate vicinity was closest to the projectile's target enemy. This was fairly easy to implement with methods already found in the class o1.Grid.

Solving the direction of the next step of a given projectile required me to find if the largest difference in distance was in the x- or y-axis. With each passing of time, a projectile moves in the direction determined with this algorithm.

Creating a projectile happens with the same algorithm – then the program finds the direction for the recruit to spawn the projectile in (the direction towards the enemy).

As the enemies are always on the move, it is important that the projectile is always updated on what direction to take next. However, because the enemies move always at a certain pace, one could only determine the destination for the projectile as one could calculate where the enemy will be in X passings of time. This is efficient, but it won't fit in well with possible expansions to enemy movement (a situation where enemies' speed could change during the game).

Another algorithm that required more planning for me was finding all enemies within a given recruit's range. This took me a lot of time, because I went with a much more complex approach. My first implementation of the enemiesInRange(recruit: Recruit) method used recursion to scan a square's neighbors and then scan their neighbors n times. This was not effective at all and ultimately did not work. This could be a working implementation, but it is hardly an effective one.

I ultimately implemented the enemiesInRange method simply by determining the EnemySquares that exist within a distance determined by the range of the desired recruit. This does call all EnemySquares within a map, which might not be the most effective implementation possible.

Determining the farthest enemy on the enemy path was not too difficult as enemies always move through a Vector of coordinates. EnemyTravelPath: Vector[GridPos] contents are in order of the path – the farthest enemy on the path is simply the one within the location with the largest index in enemyTravelPath.

6. Data structures

I mostly used scala.collection.mutable.Buffer for modifying lists within methods. This allowed for an easy way to create lists piece by piece. In most cases, the program transforms created Buffers to Vectors to ensure that they will not change over the course of the game.

Take, for example, the method enemiesInRange in LevelMap.scala. I first create an empty Buffer that I fill with enemies found in the inner method scanRange. At the end, I transform the Buffer into a Vector when modifications have ended. At this point everything the Buffer needs to contain is in the Buffer, so it is only logical to transform it into a Vector to ensure its immutability in the future.

A different usage of Buffer is found in LevelMap.scala, variable projectiles. The variable contains a list of all projectiles on the map. As they are created at a quite frequent pace, I decided on a Buffer that the class can modify over the course of the game. Adding and removing projectiles is quite easy to implement with a Buffer.

Most data structures in this project are Vectors. The program does a lot of scanning during the game and finds, for example, a player's affordable recruits at a given point in time. The program scans this data once for a given point in time and rescans it again when time passes. This means it only needs a result – hardly anything to modify ignoring a few exceptions.

The class Wave uses a scala.mutable.Map[Enemy, Int] to describe the enemies and their respective amounts in a fairly efficient way. Each wave contains a Map that links an enemy type with its amount. This means, for example, that four Zombies in a Wave is expressed like this: new Zombie -> 4. With a Map, the program does not actually need to add four Zombies into an array – n amount of Zombies requires only one instance of it. It is easy to link an enemy into its appearance rate with a Map.

7. Files and Internet access

The program accesses only one file to create the Game in object formGame. The file is found in towerDefense/gameInfo. The file contains text. Here is an example:

EXAMPLE START

```
#PLAYER
HP0100
MN0300

#MAP
1,0:1,7
< 1,7:15,7
< 15,7:15,13
< 15,13:6,13
< 6,13:6,17
< 6,17:14,17
< 14,17:14,20

#WAVE1
Z010

#WAVE2
Z015
C003
B001
M002
```

EXAMPLE END

Each section is separated by ‘#’. In #PLAYER, formGame reads the starting health and money for the Player when forming the game. These require a total of four digits even if the starting health is under 1000.
In #MAP, the enemy path is formed. This allows for a lot of customization. The user can give coordinates for either horizontal or vertical lines that form the map. The path must be uniform, it must not overlap with itself and it cannot be diagonal. Starting from the second coordinate, one has to insert a ‘<’ for differentation.

Finally, in #WAVEN, N representing the wave number, a wave’s enemies are detailed. Each enemy has their respective letter for the text file. One can see them from formGame, but usually they are the first name of the enemy letter. What follows #WAVEN is a list of each enemy that appears in the wave followed by its amount in the wave expressed in three digits.

8. Testing

The program was tested largely by gathering desired data using the command ‘println’. Most of the time I analyzed how different components evolved over time during running the game. For example, for each passing of the time, I requested the program to print both the locations of all enemies and their respective tags given by different instances of the Enemy class. The enemy path was fairly short for the testing period, so it was fairly easy to analyze if the enemies proceeded in the right manner.
I created an app in which I launch a game in different scenarios and see what it prints out. This allowed me to analyze data quite effectively and fix problems. I differed from my original plan in the order of implementation – I spent a lot of time with testing initialization from a text file before moving on to the LevelMap.
What was missing from my test plan was unit tests. I hardly ran any conditional tests to see if they passed. With this project I mostly went with my own analyses as I felt that it worked quite well. I found and fixed many problems one at a time until now the end product works as intended. I have rigorously checked the game over and over again both during and after the development process and thus have eliminated numerous errors.

9. Known bugs and missing features

The GUI could require more work. As of now, the GUI repaints itself over and over again, so it would be much, much more efficient to separate it into pieces. This could be made by splitting the code into multiple components, but I did not realize it in time. This is, as of now, hardly a problem as the program does function at a constant pace.

I originally intended for the map size to be modifiable, but I decided on the map size always being 20x20 for time saving reasons and simplicity. This could be later expanded so that the game supports maps of all sizes.

I have not prioritized game balance for this project. Some recruits are better than others, some enemies' bounties are worth more than others' etc.

10. 3 best sides and 3 weaknesses

BEST

1. Game map initialization

I am proud of the ability to input coordinates in Notepad and thus create an enemy path from them.

2. Managing waves

I think the Map implementation for storing enemy amounts and enemies is quite efficient and it works.

3. GUI showcasing recruit info

I think how the GUI displays recruit info, especially its range when clicked on, is quite great.

WEAKNESSES

1. GUI efficiency

My written code for the GUI is long and it all refreshes when repainting. This could be modified to appear in multiple components, but I did not have the time.

2. Adding new enemies and recruits is cumbersome

If one wishes to add a new enemy, for example, s/he needs to add its class to classes Enemy, formGame, TowerDefenseGUI. This is quite complicated and could be condensed.

3. Not that much unit testing

Instead of using unit tests to test the program, I mostly went with analyzing results on my own. Due to a tight schedule, I wasn't able to write too many unit tests to test the program. However, the program is in my opinion very much functional and I have not encountered bugs while rigorously playing the game over and over again, testing its limits.

11. Deviations from the plan, realized process and schedule

My week-to-week schedule did not go exactly as planned. Rather than being spread out evenly, my amount of work was quite backloaded, which reduced the overall quality of the project.

I started with creating groundwork for each class – adding methods from the original plan UML and implementing them as prototypes for later modification. During the first two weeks, I focused on reading text files and creating a game from them. This went quite well, but I still was behind on schedule.

For the second checkpoint, I did not do that much. I spent a lot of time trying to import the O1Library to my project to the point that I did not achieve much during this period. This was finally successful (and not too hard). I spent a lot of time doing work for other courses as well.
I did lay out groundwork for both enemies and recruits – at this point I hadn't done much to program a functioning map. In my original plan, a map was one of the first things to do in the project.

For the third checkpoint, I had done a lot more work. I had programmed a functioning map that the player can place recruits on and on which enemies travel almost correctly. This was a huge step forward in implementation.

For the last stretch, I corrected enemy movement, made custom map creation possible, overall made the game function as intended. What took most of my time here was struggling with scala.swing, which was a challenge for me. But, in the end, I got it to work. The last stretch was, indeed, the most work-demading, as I had delayed my commitment to the project until when I had to.

The order in which I created the project was different from what I planned. I formed the map for the third checkpoint, although it was meant to be for the first. It might have been better to start with the map, but when programming, I felt that it was natural for me to start with other classes such as Enemy, Recruit. However, it did not take too much longer for me to create a functioning LevelMap for testing after creating groundwork for other classes.

A major deviation from the plan was after the third checkpoint, when I transferred both projectile and enemy movement to be handled in the LevelMap class. This allowed for simpler implementation of movement methods as, ultimately, the changes are seen in LevelMap.

Some things missing from the original plan are different speeds for enemies and status effects for enemies. I left these out because I decided on a uniform speed for all enemies.

I learnt once again that starting a project early is the key to success. With an early start one can clearly see where a project is headed and how to fix many more problems. What I missed with my limited schedule was heavy testing.

12. Final evaluation

The tower defense game I created functions as intended. The game is, at the moment, fairly simple, but that is exactly what I was targeting. Playing the game does not bring forth any major errors – if any at all. I think the game is well done as it functions as a tower defense game quite well.

The GUI is simultaneously a strong and weak point – it could be more efficient, but I have, in my opinion, laid out information quite well and the display supports gameplay tremendously well. It works as a game and the GUI feels pretty good to interact with. Placing recruits especially feels satisfying.

The shortcomings come in testing – there are bound to be a few bugs within the game due to time constraints. The program could be improved by heavier testing in the future.

The program could be, on a game basis, improved with making enemy movement more versatile – different speeds for different enemies. This could be a challenge due to it being quite confusing in the GUI. This would, however, allow for much more customization within the game. The way I have made this project would require for me to change other things as well – projectile targeting would change and I would need to modify the class EnemySquare.

Otherwise the game can be expanded upon rather easily. One can add different recruits and different enemies as they wish and create even larger maps (contrary to what the default size is at the moment).

The game does work well as a game. It can be played from start to finish with a clear growth in strength, which is something I was looking forward when deciding upon this subject.

If I were to start this project from the beginning again, I would test even more rigorously and create the game so that it could be programmed to a more advanced level more easily. Another thing I would change is definitely the schedule – I would do it in smaller chunks over the course of a longer time instead of in larger chunks over the course of a small time frame.

13. References

https://stackoverflow.com/questions/4515008/are-there-a-good-examples-of-using-scala-swing

https://www.iitk.ac.in/esc101/05Aug/tutorial/essential/threads/timer.html

https://docs.oracle.com/javase/tutorial/uiswing/misc/timer.html

https://store.steampowered.com/app/960090/Bloons_TD_6/

https://play.google.com/store/apps/details?id=com.ironhidegames.android.kingdomrush&hl=fi&gl=US

https://www.reddit.com/r/gamedev/comments/zl0ti/i_am_releasing_my_full_reference_implementation/

Halloween (1978)

Ghostbusters

The Thing (1982)

Suspiria (1977)

The Phantom Carriage

Castlevania series

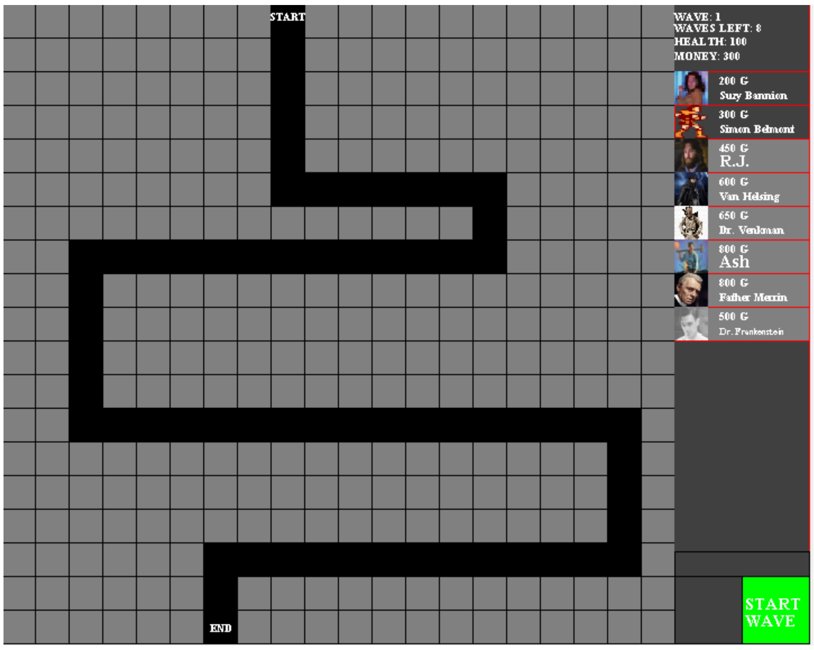
The Evil Dead series

The Exorcist (1973)

Frankenstein (1933)

14. Appendixes

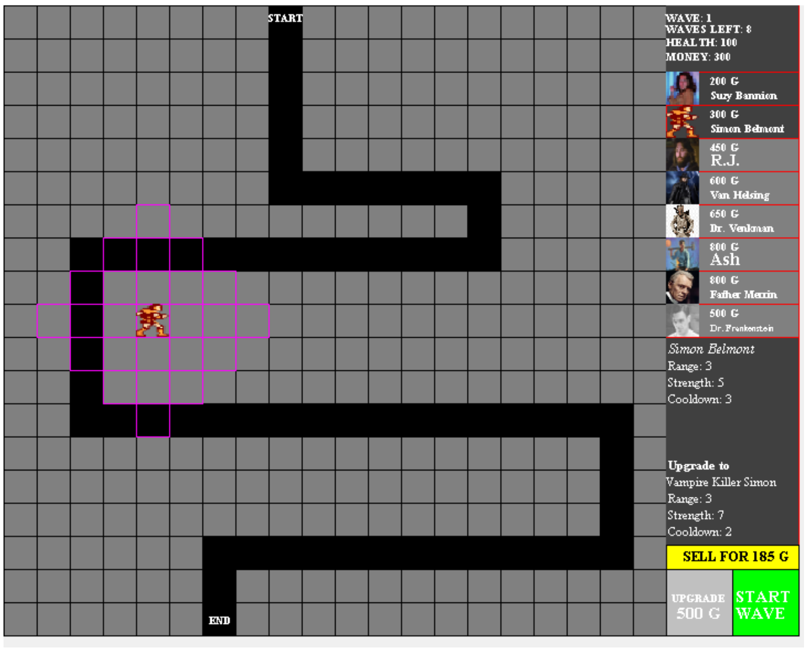
Program screenshots



WAVE: 1
WAVES LEFT: 8
HEALTH: 100
MONEY: 300

200 G	Suzy Bannion
300 G	Simon Belmont
450 G	R.J.
600 G	Van Helsing
650 G	Dr. Venkman
800 G	Ash
800 G	Father Merrin
500 G	Dr. Frankenstein

The game in its initial state. The black line is the enemy path. 'START WAVE' button required to be clicked on to continue, but recruits are placeable in this state.



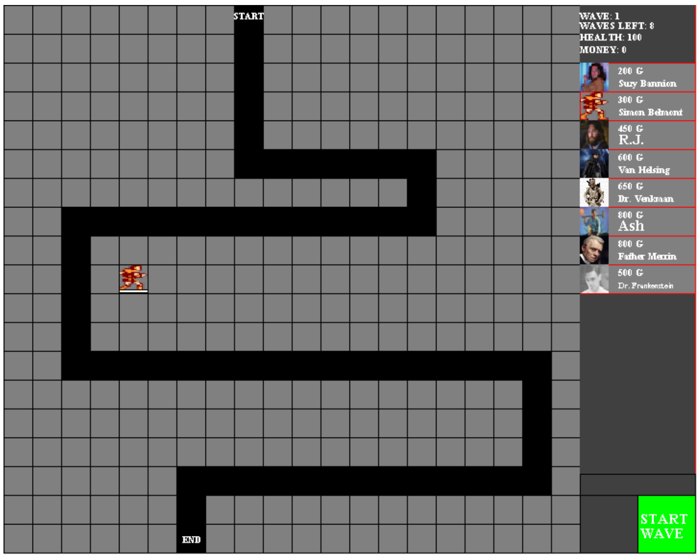
WAVE: 1
WAVES LEFT: 8
HEALTH: 100
MONEY: 300

200 G	Suzy Bannion
300 G	Simon Belmont
450 G	R.J.
600 G	Van Helsing
650 G	Dr. Venkman
800 G	Ash
800 G	Father Merrin
500 G	Dr. Frankenstein

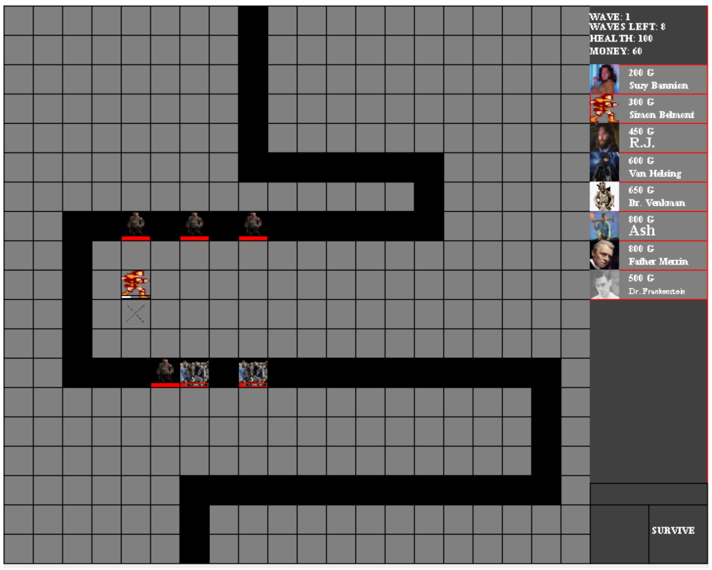
Simon Belmont
Range: 3
Strength: 5
Cooldown: 3

Upgrade to
Vampire Killer Simon
Range: 3
Strength: 7
Cooldown: 2

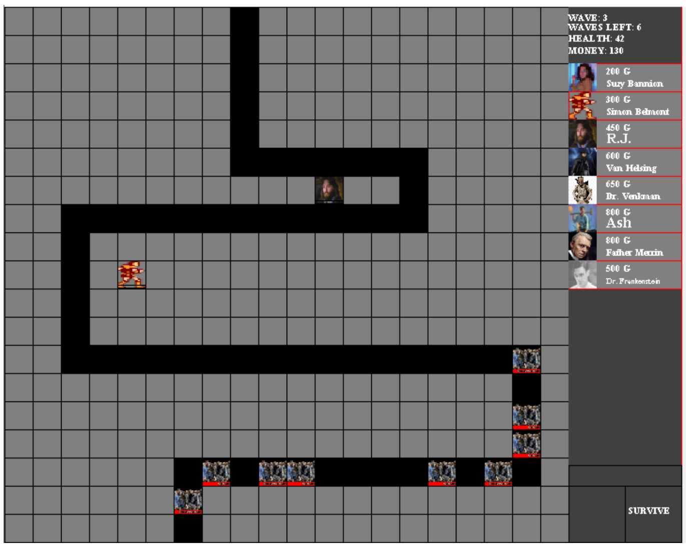
Placing recruit 'Simon Belmont' on the map. More information described on the right and range displayed on grid.



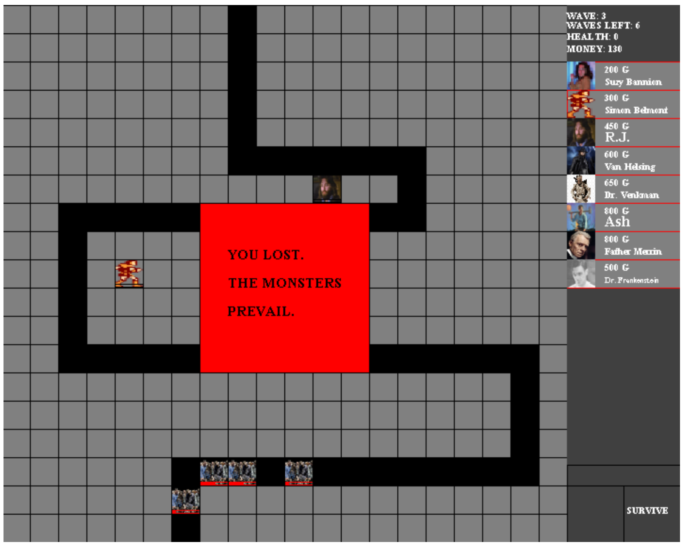
Simon Belmont placed on the map. Cooldown bar visible. Money equals to zero after purchase.



First enemy wave in progress. Recruit Simon Belmont shoots projectiles to enemies. The first enemies have lost HP and Simon Belmont is in a cooldown state as the white bar is not full.


WAVE: 3
WAVES LEFT: 6
HEALTH: 42
MONEY: 130

Two recruits bought. Enemies of the type ZombieHorde have reached the end and previous ones have already taken health.


WAVE: 3
WAVES LEFT: 6
HEALTH: 0
MONEY: 130

YOU LOST.
THE MONSTERS
PREVAIL.

Game over screen. Health equals to zero.