C++ Tower Defense Game: Documentation

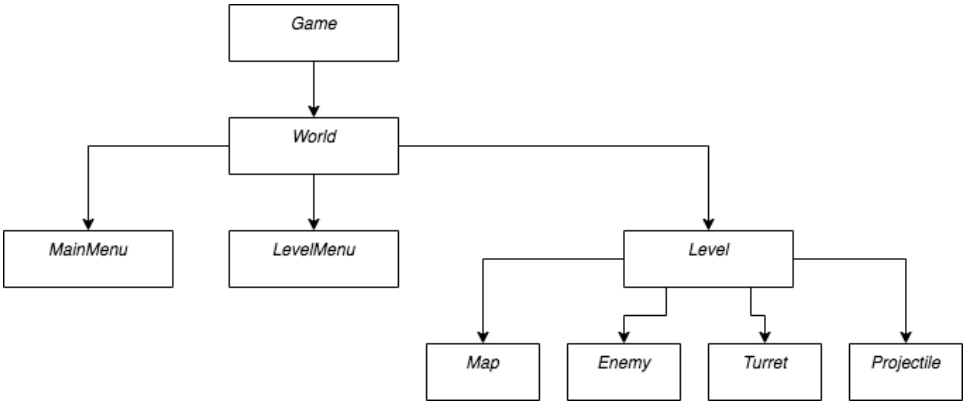
Mark Heidmets, Niko Kemppi, Minh Dinh, Kerkko Konola

OVERVIEW

The piece of software is a video game. Its genre can be described as *tower defense*: the objective of the game is to destroy arriving enemies and not let them reach the end of their path. The enemies are destroyed by turrets that can be bought and placed on the map. The player must manage their money well and consider which turrets are worth buying.

The game features multiple levels. Additionally, the creation of new levels has been made possible by the fact that new levels can be constructed with text files written in a specific form.

SOFTWARE STRUCTURE



The whole class hierarchy starts with main.cpp. Under that we have class Game that encapsulates the framerate and event listening of the game. It also contains the actual window in which all the activities take place. The class Game has one member variable: World, which in turn adds all the scenes and logic to the game. World consists of a reference to the Game window, the actual bounds of the game, the current Mode, background music and a popup message.

The Mode class is an abstract class inheriting from Node. Node, in turn, is another abstract class that inherits from sf::Drawable and sf::Transformable of SFML (Simple and Fast Multimedia Library). It lays the boilerplates for drawing the nodes. Namely, each Node instance consists of a list of children and a parent. In that manner, a tree structure is achieved, which helps immensely with drawing the entities in layers (if there are 3 layers: background, entities, HUD, then each would have their own tree structure and the program would first draw the background, then entities, then HUD, thus ensuring the correct layering of the scenes). Coming back to Mode: this is the class that has the layers vector consisting of pointers to type Node. It contains window reference, buttons, and data about buttons, levels, and fonts. Each scene (i.e., the screen the user sees from main menu to the level itself) is a Mode.

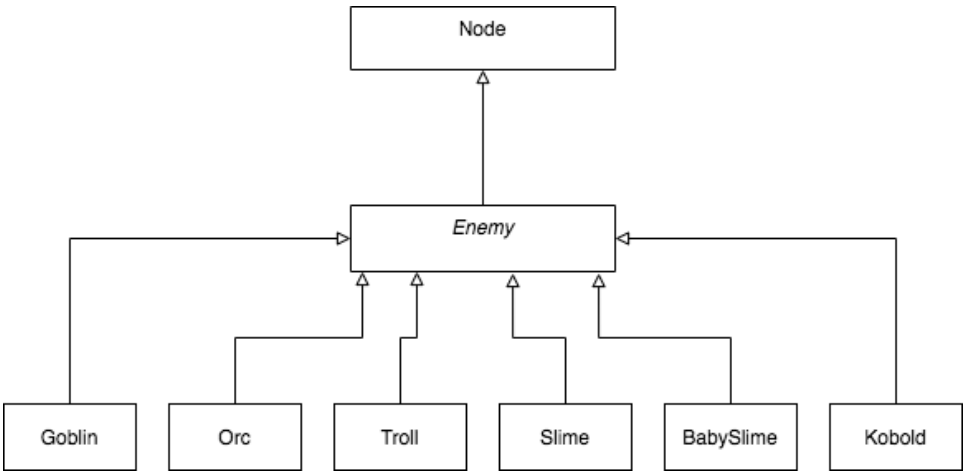
The World class changes the modes by receiving input from Game. When a new Mode is entered, the current mode pointer will be redirected to point to the new Mode instance. There are classes MainMenu, LevelMenu and Level, which all inherit from Mode and are the actual implementations of Mode's pure virtual functions.

The MainMenu class consists of two buttons, Play and Exit, the first of which leads the user to LevelMenu, whilst the latter merely exits the game. It is the simplest of the Modes. The LevelMenu class

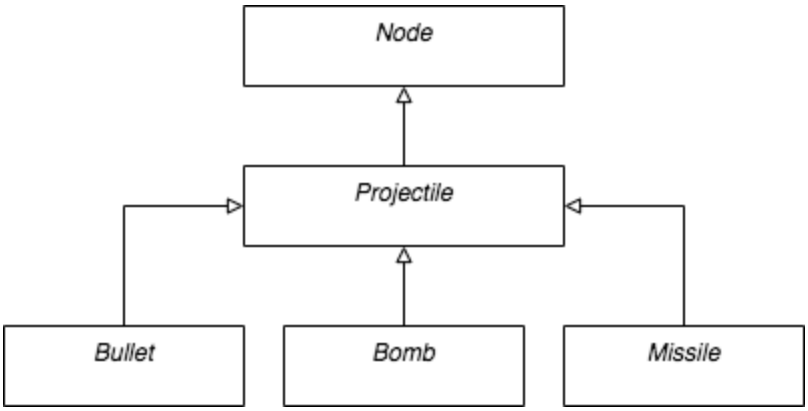
is slightly more advanced, as it has all the levels available for playing. Users start with level 1 and can advance to the next level once they have killed a certain number of monsters (20), which can be also referred to as the player's score. Therefore, at the beginning of the game, the user can only play the first level, but as they advance, more levels will become unlocked. If you try to open a locked level, the World class will notify you with a popup that the level is locked. You can close the message by clicking anywhere but a closed level. There is also an X button in the top-right corner of the scene, which will send the user back to the main menu.

The most complex class of the Modes is Level, which encapsulates all the game logic. The game also has background music that starts playing when the game is first opened. The level itself has its own track. The turrets' shots also play a shooting or explosion sound. The system saves game progress to a file called cache.txt, which is located in the folder *include/auxiliary*. It merely has information about the maximum open level, which it will read when opening a new game instance. It is updated when a new level is completed.

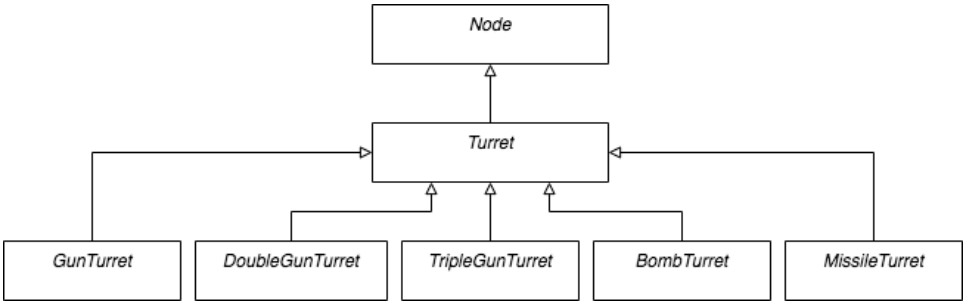
The Map class inherits from the Node class and it's responsible for loading the map design from a text file, drawing the map and finding different types of paths from the road network of the map. The map file specifies 20x15 grid of tiles. There are five different types of tiles. Spawn places (S), Bases (B), roads (#), grass areas (.) and turret places (O). There can be several spawns and bases and the road network may contain branches and form loops. The map offers enemies three different kinds of paths from spawn places to bases. A path always starts from a random spawn place but the route to a base depends on the type of the path. The first path type is a random path that leads to a random base using a random route. The only restriction is that the route may not contain the same road tile twice. The second type is the shortest path that is the shortest route to the nearest base. The third type is the safest path. The safest paths (one for each spawn place) are recalculated always when a turret is bought or removed. The turret radar range, the number of projectiles shot at the same time, the rate of fire and the type of the projectile are taken into account when calculating the maximum damage per second for each road tile.



The Enemy class is the parent class of all enemy types. It implements the main features of the enemies, mainly walking along the given path. The child enemy classes specify the main characteristics (texture, hit points, speed, path choice, kill reward etc.) as Enemy constructor parameters. The special features (for example, Slime spawning new BabySlimes when dying) are implemented in the child enemy classes.



The Projectile class is the parent class of all projectile types. It implements the main features of the projectiles, mainly flying and checking if an enemy is hit. Projectile explodes when it hits an enemy or when its flight range has been reached. An explosion causes damage to all enemies within the explosion radius. Bullets have zero explosion radius, meaning that they cause damage only in direct hit to an enemy. The child projectile classes specify the main characteristics (max damage, speed, flight range, explosion radius etc.) as Projectile constructor parameters. The projectile picture and the special features (for example, guided missile locks to the enemy with most hit points) are implemented in the child projectile classes.



The Turret class is the parent class of all turret types. It implements the main features of the turrets, mainly offering different aiming options for child turret classes and asking them to shoot projectiles according to the rate of fire and the aim status. The child turret classes specify the main characteristics (texture, price, rate of fire, radar range etc.) as Turret constructor parameters. The child turret classes implement the creation of projectiles.

BUILD AND USING INSTRUCTIONS

Linux

- Build the program with **make all**
- Run the program with **make run**
- Clean the object files and executable with **make clean**

Windows

First install **MinGW Builds 7.3.0 (32 bit)**. It contains the g++ that is compatible with the Windows SFML library included in the project.

- Add the libs/windows/SFML-2.5.1/bin project directory into the PATH environment variable
- Build the program with **mingw32-make all**
- Run the program with **mingw32-make run**
- Clean the object files and executable with **mingw32-make clean**

How to play

Choose the level you want to play in the level menu view. In the start, only level 1 is available. The other levels are locked. When you complete a level by killing 20 enemies, the next level unlocks. The higher the level the more enemy types you will see, and the faster rate new enemies appear.

Playing the game is quite easy. Simply click with mouse an empty turret base (grey circle) in the map and a buy menu opens. Choose the turret you want to buy, and it is placed in the turret base you clicked. After that the turret works automatically. You can only buy turrets that you have enough gold for.

Upgrading a turret is also simple. Click the turret you want to upgrade and choose “Remove turret”. After that continue similarly to when buying a turret.

Note that the game does not stop when buying or upgrading turrets. So, act quickly! The game ends if an enemy reaches your base. The bases are marked with a red flag.

Turrets

Gun turret is the cheapest base model. It aims at the nearest enemy within radar range and shoots single bullets. The bullets require a direct hit to an enemy in order to cause damage.

Double gun turret is similar to the gun turret, but it has two barrels and thus its maximum damage is double of the single gun turret.

Triple gun turret is the most effective bullet turret. It shoots three bullets at a time.

The bomb turret shoots bombs that explode when they hit an enemy or when they drop to the ground at the end of the shooting range. All enemies within the explosion radius are damaged. The bomb turret aims at the nearest enemy.

Missile turret is the most advanced weapon in the game. It has a relatively slow rate of fire, but the radar and flight ranges are huge. The guided missile locks on to the enemy with most hit points within the radar range and follows its movements. The missile explodes when it hits an enemy or when its

flight range is reached. The explosion is the biggest in the game and all enemies within the explosion radius are damaged.

Enemies

 Orc is the simplest enemy in the game. It is not very clever and thus it chooses a random path to a base. Orcs exist on all levels.

 Goblin is the smaller cousin of orc. It is weaker but on the other hand faster than orc. It shares the path selection logic with orc. Goblins exist on level 2 and later.

 Troll is a big strong enemy that is relatively slow. It always chooses the shortest path to the nearest base. Trolls exist on level 3 and later.

 Slime is the strongest enemy in the game. It is also the slowest. Like the troll, it chooses the shortest path to the nearest base. Slimes split into five baby slimes when dying. The baby slimes are the weakest enemies in the game. The bomb turret is very effective against them since they walk in a group. Slimes and baby slimes exist on level 4 and later.

 Kobold is the fastest enemy in the game. It is also the smartest one since it chooses the safest path to the most optimal base. Adding and removing turrets changes the routing of the safest paths. Kobolds exist on level 5 and later.

Cheating

For easier testing there are two cheating options available. While playing a level, pressing Q key resets the gold to 9999. Since this can be used several times, it allows players, now cheaters, unlimited resources to buy turrets. The second way to cheat is to unlock all levels by editing the cache.txt file in the include/auxiliary project directory. The file contains the maximum open level and thus changing its content to 6 unlocks all levels.

TESTING

The main bulk of testing took place during the development phase, judging the activities graphically. Now, with SFML, there are no specific texting frameworks that could be used to conveniently assess the correctness of the graphics (this is confirmed by the developers of the SFML library themselves, as we kept in touch with some of them via Discord). To that end, we followed the guidelines of efficient and readable design principles and allocated no heap memory whatsoever without the use of smart pointers. With that in mind: we still have over 3000 test cases run from the one test that is in a file of its own in the tests folder. This test will test the validity of the map text files and can be run with make test. It is only runnable on Linux, as there was no need to test in both systems (these files do not differ in any way across OS's). The framework used for testing was Catch2, which is a header-only framework, making it light and easy to use. The tests check if there is a correct number of rows and columns in each file and whether the files have the correct characters in them. This helps prevent any unwanted graphical discrepancies after editing the levels. It also checks the validity of some of the constant values found in constants.hpp. Last, but not least, it oversees the validity of the data in cache.txt.

Overall, it must be said that automated testing is not a too important part of this project. The nature of the game is such that manual testing is often way more natural and efficient.

WORK LOG

Responsibilities

Mark: Designing the class hierarchy, implementation of nodes, modes, menus, resource manager, sounds etc. Linux Makefile.

Niko: Implementation of the basic classes (level, turrets, projectiles, enemies, map etc.). The core elements of the game. Windows Makefile.

Minh: Design characters (turrets, enemies), interface (play screen, main menu, sub-menu etc.).

Kerkko: Meeting notes, other documents.

Weekly workload and completed tasks

WEEK 1

Mark 5h: Create base classes and resource manager.

Niko 9h: Hardcoded map. Turret and Projectile base classes.

Minh 6h: Learn about the library to understand requirements for graphics. Sketch the originals.

Kerkko 4h: Learned to work with the development tools (unfortunately, this was not ultimately translated to progress project-wise), worked on the project plan.

WEEK 2

Mark 10h: Create Mode and node tree structure for displaying graphics.

Niko 11h: Map class with file loader and single path finder. Enemy base class with ability to follow a path.

Minh 10h: Design the first draft of characters and game menu.

Kerkko 1h: Meeting notes.

WEEK 3

Mark 8h: Refactor the code for further development.

Niko 14h: Turret aiming and shooting. Visible projectiles. Bullet, GunTurret and DoubleGunTurret classes.

Minh 10h: Design the second draft of elements.

Kerkko 3h: Meeting notes, studying the SFML library

WEEK 4

Mark 8h: Implement buy menu, level menu design and inner working of game.

Niko 13h: Multipath finder, shortest path, safest path. TripleGunTurret, BombTurret and Bomb classes. Goblin, Orc, Troll, Slime, BabySlime and Kobold classes.

Minh 7h: Adapt designs to compatible resolution.

Kerkko 1h: Meeting notes.

WEEK 5

Mark 8h: Add the background music and sound effects; add map and cache tests. Documentation.

Niko 16h: Missile, MissileTurret and Explosion classes. Buying and removing turrets. Map design. Documentation.

Minh 7h: Finalize the design resources.

Kerkko 3h: Writing and proofreading the documentation part.