Towerfull

Authors :

* Thomas Vuilleumier
* Sebastian Diaz
* Lionel Pollien

Une image contenant bâtiment, maison, pixel, tour

Le contenu généré par l’IA peut être incorrect.

Source : [Isometric tower defense game tower sprite on Craiyon](https://www.craiyon.com/image/TE1RQ3J6R-a4ZzwBaKe8hA)

# Project context

During the HEIG-VD’s MCR course, we are asked to create a project using a certain prototype, one per group. Our model is Prototype, and aims to create instances of a class by cloning another instance. To exploit this, we chose to create a tower defense we chose to call Towerfull.

The use of Prototype in our project resides in two main parts:

* The saves: in the rest of the report, you’ll hear about the GameArea. It’s the center of our game, it’s the conceptual board, and it’s a Prototype. We clone the gameArea, storing it as a save, which we can load later. As such, these aren’t copies that are saved to the disk.
* The factories: our enemies and protections are standardized, and as such we store their instances in factories and clone them when we need them.

This project was made using the library libGDX, coming from a recommendation from the teacher.

# Launching the project

**Clone the repository**

$ git clone [git@github.com:Cobora2001/Towerfull.git](mailto:git@github.com:Cobora2001/Towerfull.git)

**Go in the folder**

$ cd Towerfull

**Launch the game**

From an IDE:

- Open *./lwjgl3/src/main/java/io.github.towerfull.lwjgl3/Lwjgl3Launcher* in your IDE

- From there, there is a **public static void main(String[] args)** method that you can use. This will launch the game.

From the command line:

- Launch the command: $ ./gradlew lwjgl3:run

- If needed, you can do a clean build: ./gradlew

# UML

To be fixed with the current one

Une image contenant capture d’écran, texte, Rectangle, carré

Le contenu généré par l’IA peut être incorrect.

# Understanding our code

## Path

We define our path in the Level. It’s designed as an oriented graph (PathGraph), composed of Nodes. Our Nodes store their successor. These graphs should never have cycles. Any point of the graph that doesn’t have predecessors is a spawn point, any point that doesn’t have a successor is an end point, that the player needs to defend. This method via a graph was done so that we could have multiple paths possible. Our generated levels don’t support it yet.

## Monster

A monster is an entity that can be killed and will attack in our game. They have life, speed, damage, a reward for when we will them, a path they follow (an Array<Vector2>, which is one of the possible routes that our graph describes).

## Wave

A wave is a tool that allows us to define when a monster will spawn. It contains WaveEntries, which serve as storage for both a MonsterType (which allows us to go and get that monster from the factory) and a time at which the monster can spawn relative to the wave.

## Scenario

A scenario is a storage of waves and times that defines the entire timeline of monster spawning. For each wave, we define a starting time. A monster from that wave will be spawned at the time its wave starts plus the time stored in its WaveEntry.

## Tower and BuildSpot

A Tower is a defense mechanism that shoots monsters. A BuildSpot is a place where a tower could be placed, or is placed. If no tower is built in a BuildSpot, it will be available as a button to be built upon. If a tower is there, the tower can be clicked to give the player the option to destroy it, getting back a portion of the money invested, as well as the space available for a new build.

## Level

A level is a structure that defines dimensions for the playing board (GameArea), as well as the path described above. It contains places to install BuildSpots, a Scenario, the starting gold and life for the level, and the appearance of the different elements (Background, which defines how the path looks, how the spawn points look, and so on).

## GameAssets and PrototypeFactory