BEAUTIFUL PRESENTATION PAGE

Tank Trouble Remastered

Everyone has ever played an arcade game online. One of them is particularly remembered in our childhood: Tank Trouble. It’s a funny multiplayer game where you drive a little tank in a maze to try to shoot your friends’ tank. You and your friend are on the same keyboard. That’s why…

We introduce you **Tank Trouble Remastered**!

We have completely re-coded the game and added new features. The original game opposes two players on a map selected from a predefined (finite) set, and you only have one type of tank.

Our first objective was to code the original game. It includes, among others: dealing with collisions (of the tanks and bullets), trajectories of the bullets (with bounces off the walls), acceleration (to make the game more enjoyable to play) and a user-friendly graphical interface. However, we went further with additional features, such as a randomly-generated maze, several super tanks and bullets with special abilities, sound effects, a game menu (to choose your tank and the map), “different game modes”…

We took full use of object-oriented programming in our project since we have similar objects such as entities (tanks and bullets). Therefore, our project archive is divided into several classes (non-exhaustive list):

|  |  |
| --- | --- |
| Class name | Primary features (not exhaustive) |
| GamePanel | Start thread |
| KeyHandler | Key bindings |
| MovingEntity | Abstract daughter class of the abstract class Entity. Contains the basics of entities (movement) |
| Tank | Extends MovingEntity with collisions, displacements, draw, shoot |
| Bullet | Extends MovingEntity with collisions, displacements, draw, shoot |
| Tank\_Super and Bullet\_Super | Extends from Tank and Bullet respectively. Enables to create of entities with special capacities |

Table: Division of the project in classes

We used Git to code in a synchronous way and manage versions. In the beginning, we spent some time getting used to it, but it’s a great way to code together with partners. You can find hereafter the table of involvement of everyone in the project.

|  |  |
| --- | --- |
| Name | Involvement |
| Bonnaire Léo | 25 % |
| Merle Adrian | 25 % |
| Rosard Alexandre | 25 % |
| Sibileau Antonin | 25 % |

Table: Participation of members in the project

Note: The UML graphs presented here are not whole, you can find the entire diagram with the link in our bibliography, at the end of the document.

This document details some of the features implemented in the project. We chose to group classes in packages to talk properly about their properties. We will discuss the Graphics, Playground and Entities clusters in the following document.

# Graphics

## Starting menu

## Entities display

# Playground

The game is played in a different map each time you start it. The playground is a randomly-generated labyrinth, from which we will destroy some of the walls to make it more enjoyable to play (if there are wall everywhere, it’s hard to touch your opponent). The map is only a huge matrix where each box represents either a wall (value = 1) or an empty space (value = 0).

## Labyrinth generation

To generate a labyrinth, we will first create a paving of this form, of the size of our map. This will be the base of the random generation.

Figure: Paving at the beginning of the process

Then, we will attribute to all the white cells (which value is until now 0) a different value. The goal now is to destroy some of the walls in order to have a maze:

* choose a random column, and select in this column a cell that represents a wall
* destroy its wall
* update all of its neighbours to the same value, to record that this process has already been applied here

Repeat this process until all of the non-wall cells have been processed, i.e. have the same harmony value.

When this process is finished, we will remove more walls of the labyrinth to create several possible paths from a place to the other. This will make the game a lot better to play, since the goal is not to find your way through a labyrinth but to try to find and shoot at your opponent.

## Semi-random wall removal

First, we’ll have to ensure that the spawn points don’t contain any walls: otherwise, the tank may spawn in a wall and won’t be able to move at all. We define a 2x2 cells spawn zone at the top left and bottom right and destroy any wall in this area.

As said before, we’ll need to destroy some of the walls: this will be done randomly. As for the generation of the labyrinth, let’s choose a random column, and then a cell within this column which contains a wall, and change its value by 0. Repeat this an arbitrary number of times, chosen such that generally the map is sufficiently empty.

Then, we will destroy any lone wall (a tile where all of its neighbours is not a wall) if there is any, and replace all of the non-wall cells by a 0 value (instead of the “harmony value”).

## From matrix to Tiles

We now have a matrix with only zeros and ones. The only thing to do, is convert the walls to actual tiles, with the properties associated.

Each wall will have a different look and properties depending if it has any wall neighbour. Hence we will check if each wall cell have any wall neighbour, and create a tiles array regrouping these information. This tiles array is also a matrix, of the same size as the previous one, the only difference is that it contains “Tile” objects instead of integers. If the cell is not a wall, it will also create a Tile using the second constructor of the “Tile” class.

IMAGE OF DIFFERENT WALLS

# Entities

## Moving Entity

## Tanks

## Bullets

# Conclusion

Eventually, this project was a great way to widen our knowledge in computer science. Indeed, we never actually did a constantly updating script, which here checks at every run positions, collisions and trajectories. Another challenge was to synchronise our code: we chose to use git, but we had to get used to the command-line interface. However, we did regular meetings to give an update on each one’s advancement of the project, what are the new priorities and to discuss algorithm strategies (for example, how to manage collisions). We definitely understood the need for comments in the code such that others can easily figure out what has been done, and how.

Globally, it was an entertaining project to do even if it was sometimes tedious, for example when we had to identify what was the origin of a bug, and then find a way to correct it. The final program is within our expectations: we have a functional game, with pretty good collisions, cool additional features (as tanks with super capacities). Nevertheless, we can still improve the launch time, or add “loading” widows to ensure the user that the game didn’t crash. Also, bullets collisions may be weird in some specific cases, which must be improved. Maybe we could improve the management of memory, with the use of graphics cards for all the display package (images, animations, interface).

# Bibliography

* RyiSnow [online] [accessed on 2022, March 25th]. How to make a 2D game in Java. Available at <https://youtube.com/playlist?list=PL_QPQmz5C6WUF-pOQDsbsKbaBZqXj4qSq>
* Splash sound effect <https://www.youtube.com/watch?v=nZNR5i9qN4w>
* Pew sound effect <https://www.youtube.com/watch?v=i6DRo6v78yg>