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, and updates every entities at each frame |
| KeyHandler | Key bindings |
| MovingEntity | Abstract daughter class of the abstract class Entity. Contains the basics of entities (movement) |
| Tank & Bullet | Extends MovingEntity with collisions, displacements, draw, shoot, with specifics to the entity |
| Tank\_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

Remark: all of the images and sounds used in the project (both script and report) have been drawn or recorded by members of the team.

## 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 a huge 2D matrix where each component is a tile.

## Maze generation

To generate a maze, we will first create a paving (Figure ??), of the size of our map. The paving is a 2D matrix containing zeros (empty cell) and ones (wall). The general goal is to create one path that links all the white/empty cells.

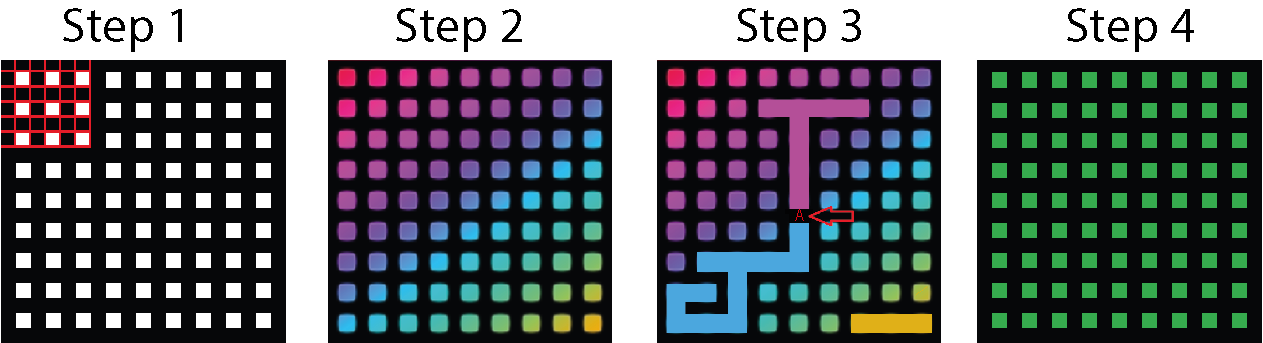


Figure: Steps of maze generation

Once the generation of the paving done, we attribute to all the empty cells (zeros) a new unique identifier. Repeat the following process until all of the non-wall cells have the same identifier:

* choose a random cell that contains a “1” (wall)
* change its value from “1” (wall) to the maximum identifier value of among the neighbours. We define by “neighbours” all of the empty cells already connected (i.e. have the same value) to the cell right next to the wall that we destroyed.
* update all of its neighbours to that same value

This way, we are 100% sure to have one and only one path that connects all the empty cells.

## Wall removal

When this process is finished, we 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.

First, we have to ensure that the spawn points don’t contain any walls. Otherwise, the tank may spawn in a wall and not 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.

Second, we destroy some of the walls. This will be done randomly. To do so, choose a random element of the 2D matrix that contains a “1” change it to “0”. Repeat this an arbitrary number of times, such that the map is sufficiently empty.

Finally, we destroy any lone wall (a cell where all of its neighbours is not a wall) if there is any. We also replace all of the empty cells identifier by a “0” value. That is it, the 2D matrix is finished !

## From matrix to Tiles

We now have a matrix with only zeros and ones. All we need is convert the ones to actual walls, with the properties associated. This is done by defining a Tile class and creating a 2D Tile array.

Each wall has a different look and properties depending if it has any wall neighbour (Figure ??). Hence when passing to the 2D Tile array we have to adapt in case there is an adjacent wall.

 Figure : Different wall skin depending on the neighbours

## Limit

Iterating in arrays takes time. The game will freeze while generating, which can be a little confusing for the user, especially at the first launch. An improvement would be to parallelise tasks: generating the map while the players select their tanks. Use multi-threads computation may also be a solution.

# Entities

## General Presentation

Here is an extract of the full UML diagram, on the moving entities part.

As each player controls a tank. The Tank class contains positions, speed, collisions and images information about the player as its number, score and the number of bullets fired.

The class will also contain the graphics related to the tank, as dust particles.

## Trajectories

To have trajectories, we only need three things: position, orientation and speed. These variables are stored in the Tank class.



Identic variables are used for bullets.

## Collision

## Limits

Since we defined the tank hitbox as a circle to simplify the collisions and allow to rotate even when you are close to a wall, the edges of the tank overlap with the walls which is not realistic at all.

# Conclusion

Eventually, this project was a great way to widen our knowledge in computer science, especially in programming.

It was a first for us as we never made a game before. It was an extremely interesting challenge as such a script needs to constantly update itself (which requires checking every entities positions, collisions and trajectories).It was also a first in terms of project length. Coding over several months really has an impact on how we designed the project.

Another challenge work as a team. To overcome the issue of sharing work we chose to use GitHub. We had to get used to the command-line interface but the benefits justified the cost. We did regular meetings to give an update on each one’s advancement of the project, what were the new priorities and to discuss algorithm strategies (for example, how to manage collisions). We definitely understood the need for comments in the code and in commits such that others could easily figure out what has been done, and how.

Globally, it was an entertaining project to do even if it was sometimes tedious. It took a lot of time and energy to code everything. Especially when we had to identify 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 (such as tanks with super capacities). Nevertheless, we can still improve the launching time, or add “loading” window to ensure the user that the game didn’t crash. Also, tank-wall collisions may be weird in some specific cases, which may be improved. Maybe we could enhance 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>
* UML Diagram <https://drive.google.com/file/d/1X7Sa6L2Nf9PNbINfklYWSefwDBxyWGYT/view?usp=sharing>