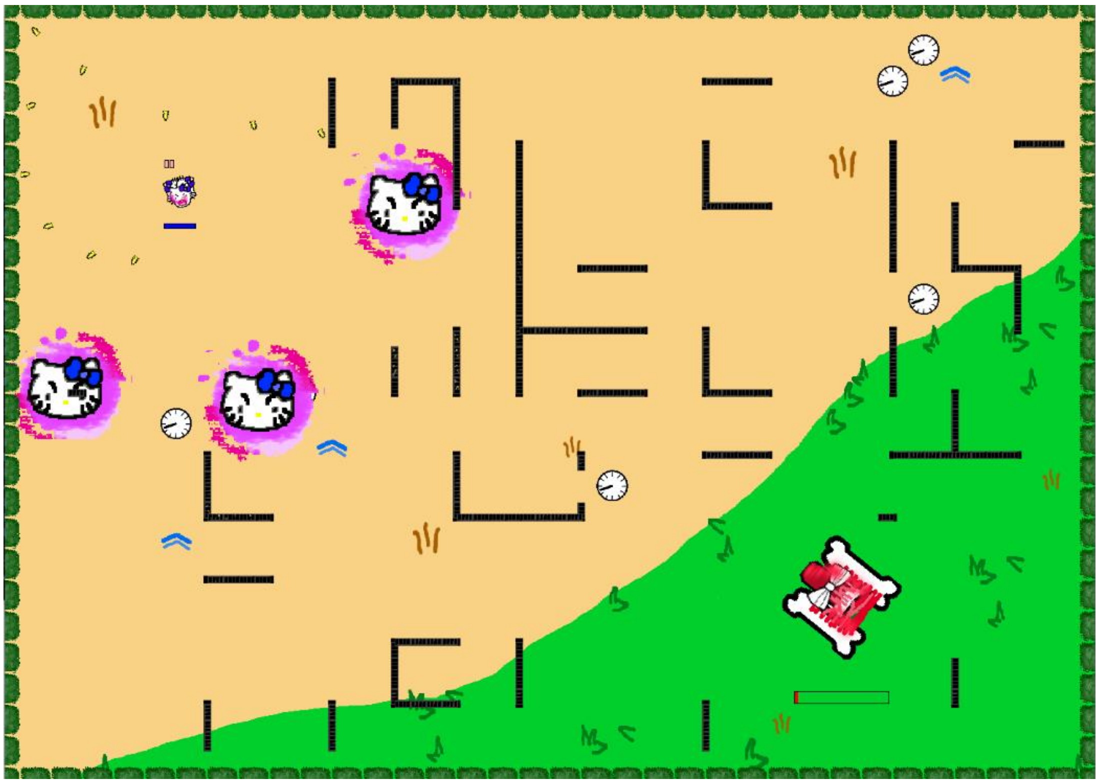
Tank Trouble Remastered



Group 71-72 – Promo 64

Merle Adrian

SIBILEAU Antonin

ROSARD Alexandre

BONNAIRE Léo

INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

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 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 a thread, and update every entity 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 1: 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 2: 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 Graphical User Interface, Playground and Entities clusters in the following document.

I. Graphical User Interface

Remark: every image, sound or animation used in the project (both game and report) have been drawn or recorded by members of the team. We think it is important to highlight as graphical design and identity is a strong part of a project. Even though this is an algorithm course, it took time to make all the animations and sounds of the game and the report to have a fully homemade game as we wanted it to be. We sincerely hope you appreciated it.

The first problem we had to overcome was choosing a library for the graphics. We were taught Java's Swing for graphics interface, which we used. Surfing the web, we found that OpenGL, Libgdx and Graphics2D are mostly used for game purposes. We went with the latter as it was recommended for beginners in this domain but still powerful enough to do what we wanted.

A. Starting menu

In the beginning, a JLabel is created in the JFrame with the animated gif. When it has stopped playing, the starting menu is created with a JLabel containing a background image and a HoverButton "start".

The HoverButton class brings to the project a modified button, that is prettier and has more functionalities. Every button used in the graphical interface will be of this type. It adds a mouseListener which changes the button's Border when the pointer is over it or not and plays a sound when it is clicked.

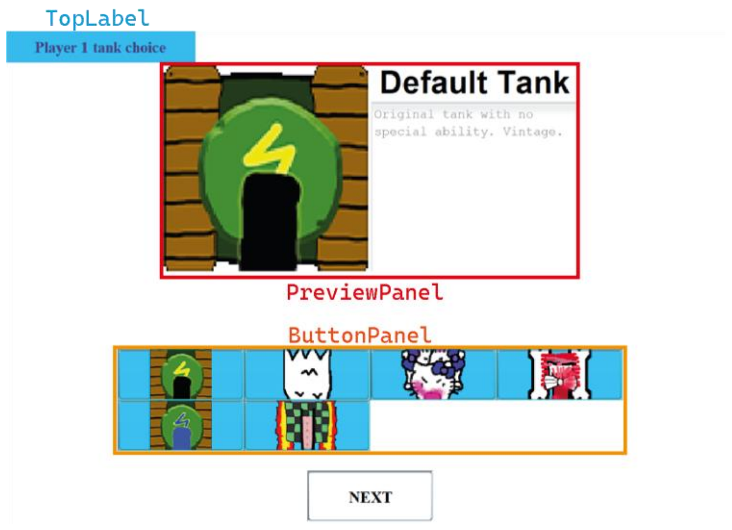


Figure 1: GUI Label and Panel organisation

Then begins the tank Selection. We add a Preview Panel, a JPanel containing the HoverButtons to select the tanks, as well as a JLabel named TopLabel that helps the user in the menus.

The JPanel uses a GridLayout. It has four columns and when adding a fifth HoverButton the layout creates a new line. This adaptative behaviour makes it easy to add new tanks to the interface.

The PreviewPanel is a class that extends JPanel containing a JLabel with the image of the currently selected tank, another JLabel with its name and a JTextArea with its description.

Lastly, the user will choose the number of games. The PreviewPanel and the buttonPanel are removed and two buttons "+" and "-" and a JLabel displaying the number of games are added.

Remark: Two objects may have an icon and use a method setIcon(): JButton and JLabel. Both inherit from JComponent, but JComponent does not have a setIcon() method. Thus, to make a common method for both, it has to take as an input a JComponent and then to avoid Compilation error, I had to check if they were instances of either JButton or JLabel and then downcast to use setIcon().

B. Ending menu

Three JLabels are added to a global JPanel: a big JLabel displaying the winner of the series of games and two JLabels with the scores of each player. Also, two HoverButtons "Quit" and "Play again" lay on the bottom of the frame.

II. Playground

The game is played on 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 walls everywhere, it's hard to touch your opponent). The map is a huge 2D matrix where each component is a tile.

A. Maze generation

To generate a maze, we will first create a paving (Figure 1), 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.

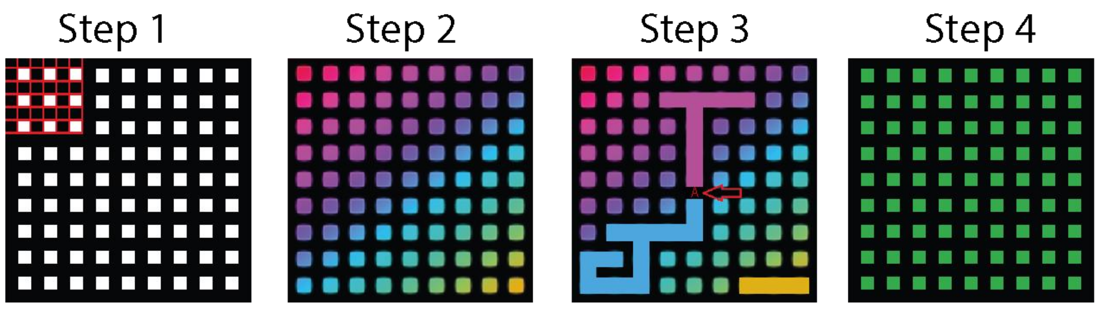


Figure 2: Steps of maze generation

Once the generation of the paving is 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 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.

B. Wall removal

When this process is finished, we remove more walls of the labyrinth to create several possible paths from one 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 cell 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" and 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 are not walls) if there is any. We also replace all of the identifiers of the empty cell with a "0" value. That is it, the 2D matrix is finished!

C. From matrix to Tiles

We now have a matrix with only zeros and ones. All we need is to 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 on if it has any wall neighbour (Figure 3). Hence when passing to the 2D Tile array we have to adapt in case there is an adjacent wall.

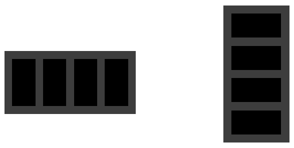


Figure 3: Different wall skin depending on the neighbours

If there's no collision, it means there's no wall and hence nothing is to be drawn. Otherwise, we draw the corresponding rectangles (seen and figure 2). We first draw a small square at the center of the tile to have nice transitions between the rectangles then we draw either the horizontal or vertical image.

D. 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. Using multi-thread computation may also be a solution. Also, printing ongoing processes in the interface would ensure the user that everything's going fine.

III. Entities

A. General Presentation

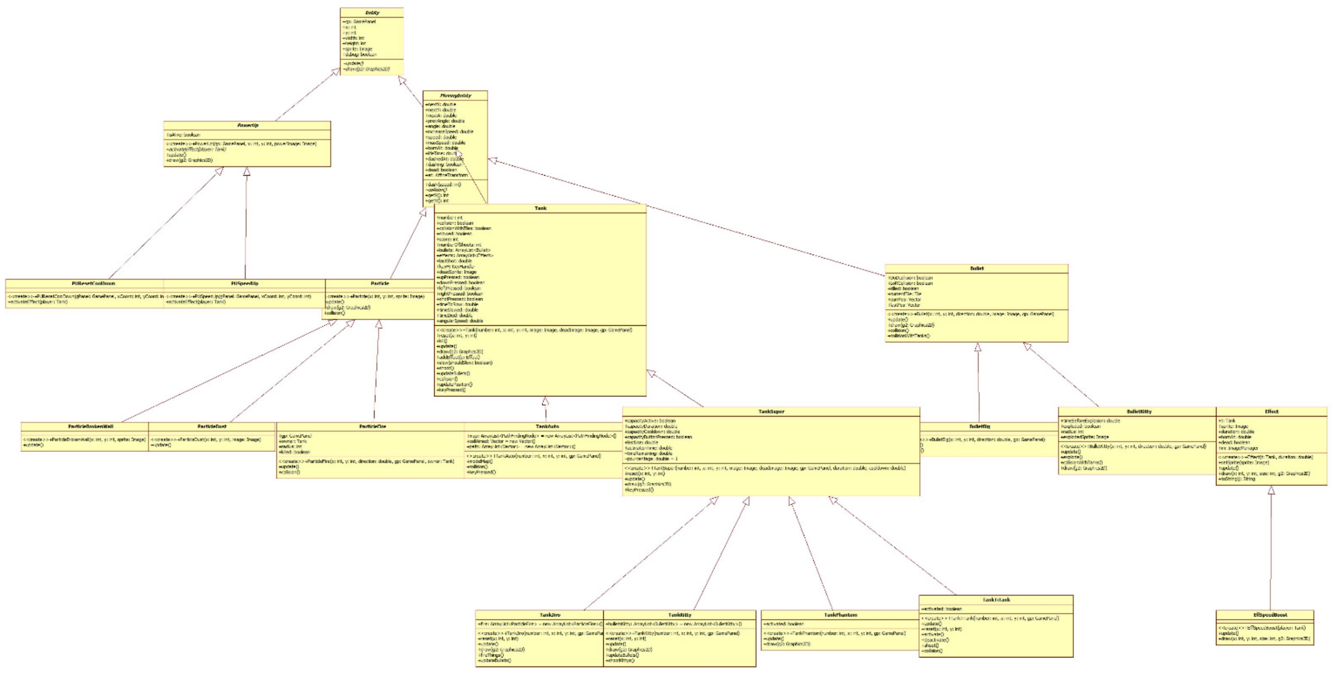


Figure 4: UML Diagram

Here is an extract of the UML diagram with a focus on the entities part.

We have created the Entity class from which any Tank, Special powers, Particle effect are derived... This class contains the basic attributes such as coordinates, image, size and a *JPanel*. From this class comes *MovingEntity* and *PowerUp*. Special effects are much simpler because they don't need attributes related to the movements. The *PowerUp* class has only one more variable that stores if the power has been activated, i.e. if a tank has passed over it. The *MovingEntity* class contains attributes related to the movements, i.e. speed, angle, and new coordinates.

That being said, we can talk about tanks, basic elements of the game. The *Tank* class is at the same hierarchical level as the *Particle* and *Bullet* classes which we will talk about later on. The *Tank* class, therefore contains more specific attributes such as the effects applied, the number of the bullet still available or the state of the keyboard keys corresponding to the displacement. It was very useful to use the object-oriented programming aspect of Java. Thanks to OOP, it was very easy to implement special tanks. So we created a *TankSuper* class whose attributes are related to a capacity: *capacityActive*, *capacityDuration*, *capacityCooldown*, *capacityButtonPressed*. Then come the classes of special tanks: *TankJiro*, *TankKitty*, *TankPhantom*... Each special tank has specific attributes concerning its power(s). *TankKitty* for example implements a new method *shootKittys()* corresponding to its special shot.

We have also created a *BulletKitty* class to implement this tank. This class inherits from the *Bullet* class. On the same principle, the *Bullet* class contains the basic attributes of a bullet (if it has killed a player, vectors for the position,...) and its inheritors contain more specific methods like *explode()* of *BulletKitty*. The same principle explains the construction of the *Particle* class and its inheritors *ParticleBrokenWall*, *ParticleDust*, and *ParticleFire*.

B. Trajectories

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

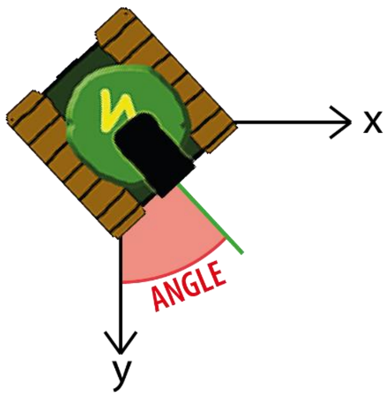


Figure 5: Position and orientation principle of a tank

To calculate the next position of the tank, we simply add at each step the speed to the current position, projected (with the use of the position angle) onto x and y. To make the game more realistic, the speed is not constant over time: It is very small when you start to push the "go-forward" key and increases at each run until it reaches the maximum value. It gives the impression that the tank is accelerating which is prettier on screen.

The same process is used for bullets, except that there is no acceleration (their speed is constant over time). However, bullets have a limited lifespan: they disappear when their Timer goes to 0.

C. Collisions

At first, we chose to use square hitboxes for the tanks, as their sprites are squares. However, the collisions of the edges of the square onto a wall made rotation impossible if you were too close to that wall, which was frustrating when trying to escape from your opponent through the maze. We decided to transform that square into a circular hitbox, which corrected this problem.

The collision detection is the same for both bullets and tanks. In order to achieve it, here are the steps to follow:

- List the surrounding tiles of the object, since collision can only happen there.
- Add these tiles to an array, and add the tiles representing the borders of the map.

- Check every pixel inside the hitbox (a disk) of the object. If any of these pixels is representing a wall, then there is a collision. It will only happen for tiles in the above-mentioned array.

Remark: When a wall Tile is created, we record the surrounding walls (up, down, left, right) in its collision variables.

In order to check if a pixel is representing a wall,

- Verify that the tile containing the pixel does allow collisions, i.e. it's a wall.
- Then we check the orientation of the wall, whether it is an up, down, left or right wall. Then, we verify that the pixel is not where the wall is. For example: if a wall Tile is a left wall, we will check if the pixel is on the left side of this tile. If yes, then there is indeed a collision.

The end of this process sets a Boolean variable named *collision* which tells if a collision occurred and this is handled in a different manner depending on the collision we're dealing with. For example, a collision between a tank and a wall stops the tank, a collision between a bullet and a wall changes the direction of the bullet, a collision between a tank and a bullet kills the player, etc…

Note: We left in the script a conditional loop used to debug, drawing on top of the Tank sprite its hitbox. It is not used when you want to play, but it was clearly useful when we implemented collisions.

D. Limits

Since we defined the tank hitbox as a circle to simplify the collisions and allow it to rotate even when you are close to a wall, the edges of the tank overlap with the walls which are not realistic at all. A bullet may cross one of the edges of a player's tank without any effect. We are aware that it can be very frustrating for players to miss a shot because of the game's hitbox design. We have put a lot of effort into this issue. We searched for different libraries and codes that could allow overcoming this issue. The solution we came up with turns out to be the best of what we can do with the less possible frustration and the best gameplay.

IV. Conclusion

Eventually, this project was a great way to widen our knowledge of computer science, especially 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 entity's 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, and cool additional features (such as tanks with super capacities). Nevertheless, we can still improve the launching time, or add a "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 packages (images, animations, interface).

V. 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

VI. Table contents