Tanks Commanders

Reinforcement Learning Project

Itay Lavi-Okabi

# Introduction

## Project Design

### System Design

The system is divided to “back” – meaning the ML algorithms & communication, and “front” – a GUI for visualizing the tanks to the client. The client should be able to visualize the algorithm as he sees fit, which is why the back is completely separated from the implementation of the GUI, as long as it follows the API.

The client holds a GUI and connects to the back. In response he gets a finished gameplay – text describing the game. The gameplay includes initializing and positioning the tanks, every command given to the tanks, and the game’s aftermath.

### Game Flow & Algorithmic Plan

A **turn** – one frame. Hence the unit measure is [F]. Assume 25 frames per second [fps].

**One meter** in real life will translate to **3 pixels** [P] on the board.

Controlling the tanks

A tank can do only one of the following actions in each turn:

1. Move forwards/backwards (change position).
2. Turn by x degrees.
3. Turn its turret by x degrees.
4. Shoot. Shooting creates a bullet in the environment, and its movement is one sided and constant in velocity. Once the bullet reaches the shooter’s max range/hits the target it’s removed from the game.

Battle Zone (Reinforcement),

Training Zone (Genetic)

Each tank decides what to do every turn – moving, shooting, or scanning. Once a decision is made (let us call it a **plan**) a list of proper steps for executing it is generated, containing the following options (let us call them **actions**): moving one step forwards/backwards, rotate, rotate turret or shoot. Remember: the step size or rotating angle a tank can make in a single turn is determined by its properties.

#### Genetic Algorithm

Bigger body – Higher Defense, Lower Mobility

Bigger turret – Higher power, Slower to rotate

Higher Cost – Larger Sonar & Better Reliability, Lower Chance to being produce

## Plan

Diagram

Description automatically generated

We start with a basic GUI made with React.js that allows to place, move and rotate Tank objects. After that, we go to building the back – a Node.js server that will cover the Python ML algorithms as mentioned above. Then it’s time to work on what we all gathered here for – an amazing top notch edge of technology Machine Learning. To start off easy, we will actually begin the ML journey with the Genetic Algorithm that will – hopefully – improve the tanks used

# GUI & Tank control (React.js)

Focusing on visualizing and controlling the environment and its objects.

31/08/2022 –

1. Creating an Environment & Training Zone
2. Creating Tank object that can be placed in different positions.

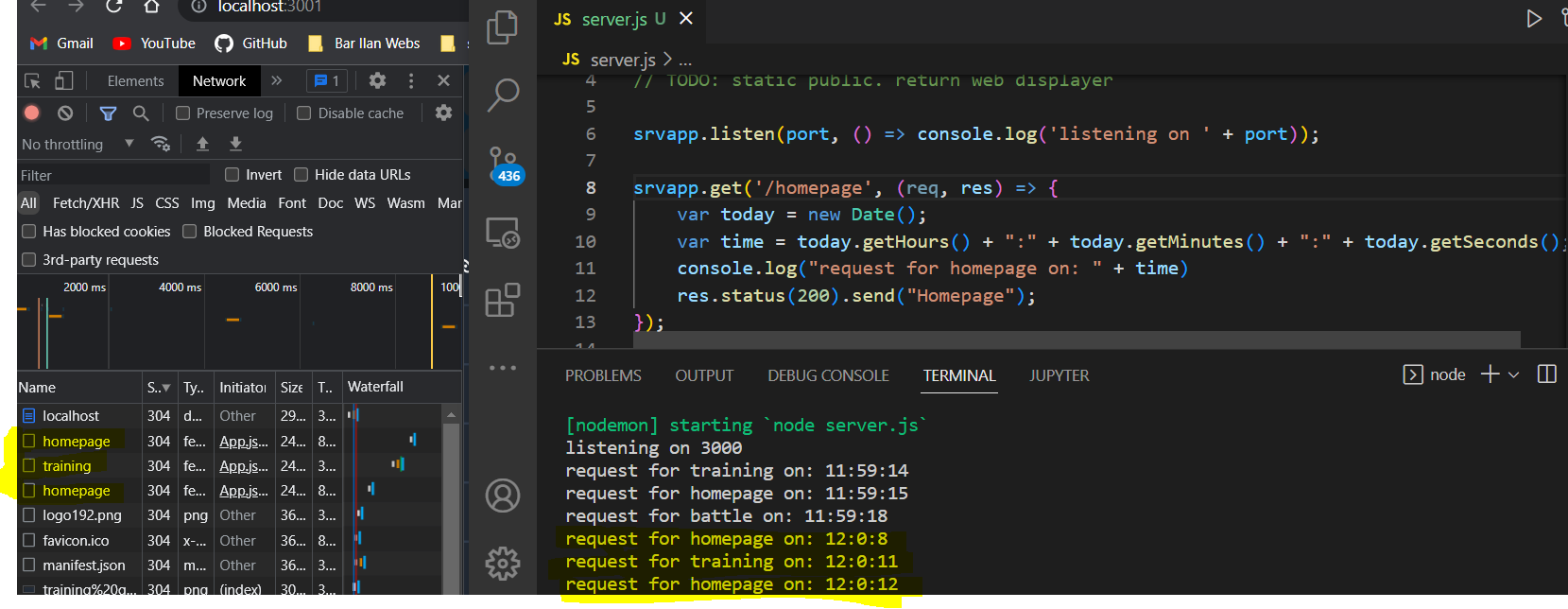
A screenshot of a computer

Description automatically generated with low confidence

## Controlling the game – (Node.js server)

### Creating server

1. Create a node.js server – also install Express.
2. Send & receive message on client side.
3. Create a simple RESTful API for receiving homepage (empty for now), testing request, battle request.



1. Send a temporary file (representing a valid game file) and parse it.

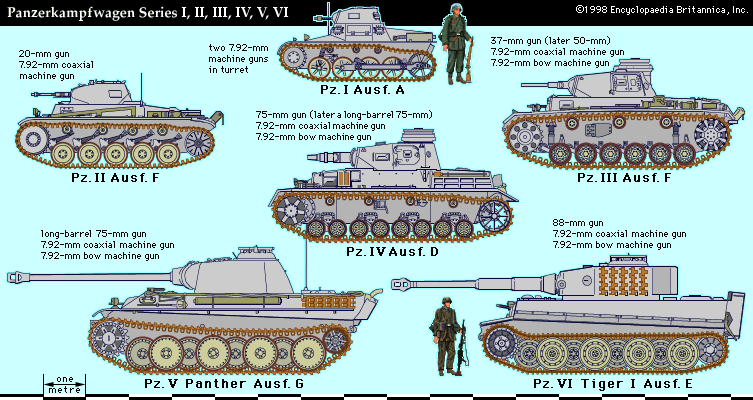
Notable issues at this point:

* New subjects in JS: express, cors, spawn
* API problems (fixed when URL in fetch changed to “http:\\localhost\api” instead of “\api”).
* Parsing the file on the client – had hard time comparing the lines to string “g” (it marks the end of the tanks initialization, and the start of the game itself. Problem fixed by using trim() on the lines read from file).
* Running a python script from the server. A python script is supposed to create the game file and of course run the game itself. So, I already started here to create the python class “BattleGenerator” to handle creating and managing the battle file. Then created two JS modules – API (to contain all actions regarding the communication and API requests) and python\_manager (to contain all methods for handling python scripts).

Controlling the tanks (and animation)

Background:

The tanks dimensions are based on WW2 German tank Panzer V Panther. With 7 meters body length, and about the same turret length. This length is represented in our game as size == 70. To allow wide variety of sizes, the range of the tanks are 7±2. This range also applies to the turret size.



The Panther speed was 45km/h or 45,000m/h or (45,000/3600) m/sec which is 12.5m/sec. Assuming 25fps, it gives us 0.5 m/Frame, or 1 Pixel/F. In summary, conversion from km/h to P/F is:

Let S be the tank size in meters, and V it’s velocity in [m/F], it is reasonable to calculate the tank’s velocity as:

V = (1 – (S/11))

This turned out to be too slow, and after some tests I found that multiplying the result by 7 gives the right velocity.

Notable issues: rendering the tank in different position. It was pretty tiresome to work with conventional React.js methods, so instead I created a Tank class (not a class component) that simply holds the tank details and allows managing the tank and also allows rendering it by calling render().

1. Get basic control on the tank visualization: forward, backward and rotation.

Note: for now, I have a game file that commands three different tanks (sizes: 5, 7, 9) to move forwards for 20 frames, rotate for 20 frames, and go backwards for 20 frames. This sums up to about 3 seconds of game. Yet, the game file is 15KB, which looks too much to me. Currently, the file consists of JSON strings (meaning the keys for each JSON is also present in the game file) for each tank initialization or movement. This allows easy parsing of the file but looks like it takes more than 50% of the file (especially after I reduced the numbers to 3 digits after decimal point).

* 1. Reparse the file after changing the JSON strings of the commands (the JSON strings of the tanks initialization are negligible).

File size dropped to 6KB.

1. Control turret: rotating and shooting

Note: How fast should turret turn each frame? Let us call this parameter Tur\_step. since

Tur\_step = (1 – turret\_size/14)

gives a slow turret rotation, and after some tests, I multiplied the result by 7. But it gave a small difference between the fastest and the slowest turret. So I decided to give more advantage to the smaller turrets by changing the formula to:

Tur\_step = 15 \* (1 – turret\_size/14) 2

This gave both good difference between the turrets, and reasonable speed for the fastest turret.

* 1. Create Environment class
     1. Add position limits.

Note about **Collision**: Checking for each sprite if it has collision with each sprite sounds like a slow approach. I thought about having a 2D array for the board, and mark each ”cell” who’s occupying it.

* 1. Create a bullet class
  2. Shoot bullet