TAU - Advanced Topics in Programming, Semester B 2025

Assignment 1

Note: This document is in DRAFT mode till April-8th 23:00. While in draft mode, it may have changes without highlighting them. Even though it is still a draft, you may (and should) start working on it!

During the draft period: you are mostly welcomed to add questions directly into this document, as comments. Please assign your comments and questions to kirshamir@gmail.com so I will get an email when you add them. Then please follow my answer and if it resolves your comment or question please close it.

After the draft period: commenting on the document will be closed and all questions on the assignment would be submitted in the relevant forum in the course site! Changes in the doc, if any, but then they would be highlighted in the document. Note: if a change creates more work for you (e.g. you already implemented things differently), please raise the issue in the assignment forum before rushing to change your code!

Submission deadline: April-27th, 23:30.
Note: here are the Submission Instructions - please make sure to follow them!

Assignment 1 - Requirements and Guidelines

You are required to implement a program that simulates a tank battle game.
Note: This exercise is purely for educational and programming practice. It does not promote real-world conflict. The tank battle scenario is simply in a game-based context.

In this first assignment the requirements are a bit open, you will have to select your API, class design and input file format. Note that in the next assignments a common API and file formats will be dictated and you will have to refactor your code accordingly.

Game Board Input File - TEXT file

The program should get an input file that represents a game board.
The file shall indicate the dimensions of the game board (width and height).
The game board itself is consisted of:

- Two tanks, standing somewhere on screen, one tank per player. Each tank has a cannon, pointing to one of 8 possible directions (U, UR, R, DR, D, DL, L, UL)
- Mines, located in different places on screen.
- Walls, creating the maze.

The following characters shall be used in file:

#	To represent a “wall”	
1	A tank that is owned by player 1	Note: in exercise 1 there can be only one tank per player, in next exercises there can be multiple tanks for each player.
2	A tank that is owned by player 2	
@	Mine	

The tank's cannon is considered as part of the tank's size and doesn't occupy any additional position in the game board. Position of the tanks' cannons on start would be *Left* for player 1 (thus shooting left) and *Right* for player 2. The cannon direction also sets the forward movement direction of the tank. In case the cannon is in a diagonal direction (i.e. UR, DR, DL, UL) then the movement of the tank would be diagonal (i.e. ±1 on both the x and the y axis).

Actions

The following actions can be performed by the tanks, each takes the corresponding steps count:

Action	**Steps count and other restrictions**
Move forward	1 per game step.
Move backward	A backward move request, would do nothing for 2 game steps, then the tank will perform the move backward in one step (3rd step). Immediate additional requests to step backward, right after the 3rd step, as long as the tank didn't stop and didn't perform any other action, would take a single game step. A backward move action that was not performed yet, can be canceled by a forward move action request, in which case the tank will stay(!) in place, ready for any other action in the next step. Other action requests while waiting for backward to kick-in are to be ignored (including additional backward requests).
Rotate 1/8 Left or Right	1 per game step.
Rotate 1/4 Left or Right	1 per game step.
Shoot	1 per game step. However, after shooting, the tank cannot shoot again for 4 game steps.
(Artillery shall movement)	The tank artillery shell movement is in the direction of the cannon when shot, and in pace of 2 per game step (±2 for either or both x and y, depending on direction, without skipping squares).

Each tank has 16 artillery shells when the game starts. There is no recharge option.
If both tanks used all their artillery the game continues for an additional 40 steps, if no one wins by then, the game ends with a tie.

The boundaries of the game board are considered as an invisible tunnel to the other side of the board (from top to bottom and vice versa and from left to right and vice versa). This applies for both the tanks and the shells (i.e. both tanks and shells may continue their movement to the other side through the invisible tunnel).

When a shell hits a wall, the wall is weakened, so after two hits the wall will "fall" (disappear from the game board). Shells cannot hit mines. If a shell hits another shell, both explode.
If a shell hits a tank, the tank is destroyed. If a tank steps on a mine both are destroyed.
If a tank hits another tank (both or one drive on the other) they are both destroyed.
In the first assignment, as there is only one tank per player, if a tank is destroyed the other player wins. If both tanks are destroyed at the same time, it's a tie.
See more on the above in moodle: here (move order) and here (multiple objects collision).

Algorithm

You should implement two different algorithms for the tanks of each player.
The algorithms shall decide on the actions of the tank. All decisions must be deterministic (without any randomness).
In this assignment the algorithms may have full read access for the information of all the objects in the game (position of walls, tanks, shells being shot, mines).

The code MUST be written in a way that the algorithm is being requested for an action and gives it back as a result, but the algorithm itself doesn't change anything in the game. All changes in the game should be performed by some "GameManager" that would interact with the algorithms. The "GameManager" should also handle cases in which the algorithm requests an illegal move (e.g. stepping into a wall, or requesting an invalid move such as reshooting without waiting 4 game steps), it should ignore it but write it to the output file as a "bad step".

It is quite complicated to build a good algorithm. However, in the first assignment you are not required to build a "winning" algorithm, only a reasonable one.
A reasonable algorithm should be able to:

1. Try to shoot the opponent's tank when relevant.
2. Try moving its tank when a shell is chasing it.
3. Avoid stepping on a mine (location of mines is known).

At least one of the two different algorithms that you are required to submit should try to chase the opponent's tank (using DFS or BFS algorithm, try not to calculate the entire path for each step!). You can decide that the other algorithm is only triggered for moving a tank if being chased by a shell.
Make sure that you actually implement two <u>different</u> algorithms.

<u>Output File - TEXT file</u>
The output file should include:

- all the steps performed by each player, including "bad steps"
- the result - win (who) or tie, and the reason

<u>Error Handling</u>
You should reasonably handle any error:

- Program shall not crash, never.
- You should better recover from input file errors (e.g. if a player has more than 1 tank, take the 1st, ignore the rest; if dimensions of the board do not fit the declared dimensions in file: ignore rows and columns beyond the declared dimensions, fill in spaces for missing chars and lines; treat unspecified chars as space). You should write a short description of all recovered errors found in the input file, into input_errors.txt file. Create this file only if there are errors.
- In case there is an unrecoverable error, a message should be printed to screen before the program finishes. No need for input_errors.txt file in this case.
- You should not end the program using the "exit" function or similar functions, even in error cases. Program shall end by finishing main, in all scenarios.

<u>Running the Program</u>

tanks_game <game_board_input_file>

Additional required documents (MANDATORY - part of the assignment grade!):

Add a short high level design document (HLD) as PDF, containing:

- A UML **class diagram** of your design.
- A UML **sequence diagram** of the main flow of your program (only main flow).
- Explanations of your design considerations and alternatives.
- Explanations of your testing approach.

Bonuses

You may be entitled for bonus points for interesting implementation.
NOTE that implementing a smart algorithm will not be entitled for a bonus, as this would be part of your next assignments.
But, the following may be entitled for a bonus:

- adding logging, configuration file or other additions that are not in the requirements.
- having automatic testing, based on GTest for example.
- adding visual simulation (do not make it mandatory to run the program with the visual simulation, you may base the visual simulation on the input and output files as an external utility, it can be written in C++ or in another language).

IMPORTANT NOTE: In order to get a bonus for any addition, you MUST add to your submission, inside the main directory, a *bonus.txt* file that will include a listed description of the additions for which you request for a bonus and how to check them.