TAU - Advanced Topics in Programming, Semester B 2025

Assignment 3 (Link to Assignment 1, Link to Assignment 2)

Note: This document is in DRAFT mode till June-25th 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!

This time the document is NOT opened for comments, all questions on the assignment would be submitted in the relevant forum in the course site!
Changes after the draft period would be highlighted in the doc.
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: August ~~10th~~ 24th, 2025, 23:30.
Note: Submission Instructions appear inside the doc, below!

Assignment 3 - Requirements and Guidelines

In this exercise you shall add a **simulator** that can run **several Tank Games** concurrently, using multithreading – in two modes: comparative or competitive.

Your exercise shall be separated into three parts, each one of them built separately with its own makefile, the parts are:

1. The GameManager (GameManager folder) - for running a single game
2. The algorithmic part (Algorithm folders) - managing Player and TankAlgorithm logic
3. The Simulator (Simulator folder) - running many games in parallel

Each part above may run independently with another team's implementation of the other parts. Parts [1] and [2] would be compiled as shared libraries (.so) that would be dynamically loaded by the Simulation project.

You are required to organize your submission into **5 folders**: 3 folders for 3 separated projects as listed above, with a makefile inside each folder (1-3) and two common folders (4a and 4b):

1. **"Simulator"** folder: includes your Simulator class as well as other helper classes required by the simulator project. The executable name shall be: `simulator_<submitter_ids>` (for example: `simulator_098765432_123456789`)
2. **"Algorithm"** folder: in order to allow us to load different Algorithm ".so" files together, the name of the .so file must be unique, using the ids of the submitters, as follows: `Algorithm_098765432_123456789.so` additionally, all your code shall be located inside a namespace with the unique name `Algorithm_098765432_123456789`
3. **"GameManager"** folder: in order to allow load of different GameManager ".so" files (together or separately), the name of the .so file must be unique, using the ids of the submitters, as follows: `GameManager_098765432_123456789.so` additionally, all your code shall be located inside a namespace with the unique name `GameManager_098765432_123456789`

4. “common” and “UserCommon” (there are no makefiles in these folders!):
    a. **“common”** folder: shall include the common files required by more than one project, published by the course staff as the files that shall be used “as is” without any change. DO NOT add your own files into this folder!
    b. **“UserCommon”** folder: shall include your own common files, files that are required by more than one project. Your makefiles and includes may use those files where needed to avoid duplication. All your code in this folder shall be located inside a namespace with the unique name `UserCommon_098765432_123456789`

The Simulator

The Simulator is able to run several GameManager objects (of the same GameManager class type, or different class types).
Algorithms and the GameManager(s) are loaded dynamically as .so files, as explained below.
The Simulator has two modes of operation:

1. Comparative run:
```
./simulator_<submitter_ids> -comparative game_map=<game_map_filename>
game_managers_folder=<game_managers_folder>
algorithm1=<algorithm_so_filename> algorithm2=<algorithm_so_filename>
[num_threads=<num>] [-verbose]
```
2. Competition run:
```
./simulator_<submitter_ids> -competition
game_maps_folder=<game_maps_folder>
game_manager=<game_manager_so_filename>
algorithms_folder=<algorithms_folder> [num_threads=<num>] [-verbose]
```

Command line arguments for the two modes:

- All command line arguments can appear in any order.
- All command line arguments are mandatory.
- ~~The = sign can appear with any number of spaces around.~~ The = sign should appear without spaces around.
- If unsupported command lines arguments are provided, the program should print a usage with an error message pointing at all the unsupported command lines arguments provided, then finish.
- In case command line arguments are missing, the program should print a usage with an error message detailing the missing command lines arguments, then finish.
- In case a command line argument that should point at a file is pointing at a non-existing file or one that cannot be opened, the program should print a usage with a proper error message, then finish.
- In case a command line argument that should point at a folder is pointing at a non-existing folder or one that cannot be traversed or at a folder that has zero files of the desired usage (at least one that seem to be valid), the program should print a usage with a proper error message, then finish.
- Exact usage description and error message text in all above cases are for your decision.
- The num_threads argument is optional.

The comparative run mode:

1. Should run all GameMangers in the given folder, on the single given game map and with given algorithms. It is allowed if algorithm1 and algorithm2 point to the same .so file.
2. Should use num_threads as described below.
3. Should create an output file directly under the game_managers_folder provided, with the name "comparative_results_<time>.txt". For the <time> part use code that will generate a new number per time to avoid collision with existing files if any.
   (You can see an example for getting time value into a string in this code example).
4. In case the file cannot be created in that folder, write a proper error to screen and print the output to screen instead, right after the error message.
5. The output file shall contain the following:
   a. 1st line: game_map=<game_map_filename>
   b. 2nd line: algorithm1=<algorithm_so_filename>
   c. 3rd line: algorithm2=<algorithm_so_filename>
   d. 4th line: just a new line
   e. 5th line: comma separated biggest list of <game_manager> names which achieved the exact same final result (player won / tie - for the exact same reason, on the exact same round, with the exact same screen final state)
   f. 6th line: the game result message as in assignment 2 (who won / tie and why)
   g. 7th line: round number in which the game finished (just a number!)
   h. 8th line on: full map of the final state of the game
   i. If there are game managers which were not listed so far, add an empty new line and repeat "e" to "h" again.
   j. Repeat stage i as long as there are still game managers which are not listed.

The competitive run mode:

1. Should run the given GameManager on all *K* game maps in the given folder, with a competition of all *N* algorithms, as following: for the kth game map, there would be a competition between the algorithms indexed i against (i + 1 + k % (*N*-1)) % *N*, e.g.:
   For kth=0:   0⇔1, 1⇔2, …, *N*-1⇔0
   For kth=1:   0⇔2, 1⇔3, …, *N*-1⇔1 (this may exist or not)
   For kth=1:   0⇔3, 1⇔4, …, *N*-1⇔2 (this may exist or not)
   [...]
   For kth=N-1:   0⇔1, 1⇔2, …, *N*-1⇔0 (this may exist or not)
   For kth=N: 0⇔2, 1⇔3, …, *N*-1⇔1 (this may exist or not)
   Above creates a total of 2 games per each algorithm, per each map *k*. Note that in the case of k = N/2 - 1 (if N is even), the pairing for each algorithm in both games would be exactly the same. You are supposed to run only one game for each pair in this case.
2. If the algorithms folder has less than 2 algorithms, print usage to screen and do not run competition. It is valid to have a single game map, if it doesn't have any maps at all, print usage to screen and do not run competition.
3. Should use num_threads as described below.

4. Should create an output file directly under the algorithms_folder provided, with the name "competition_<time>.txt". For the <time> part use code that will generate a new number per time, similarly as described for the comparative output file.
5. In case the file cannot be created in that folder, write a proper error to screen and print the output to screen instead, right after the error message.
6. The output file shall contain the following:
    a. 1st line: game_maps_folder=<game_maps_folder>
    b. 2nd line: game_manager=<game_manager_so_filename>
    c. 3rd line: just a new line
    d. 4th line on: a sorted list by winning counter, of:
        <algorithm name> space <total score>

   Number of lines from the 4th line, including, should be the same as the number of algorithms. Algorithms with the same score can appear in any order.
7. Score calculation is as follows: per each win, the algorithm gets 3 points, per each tie each participating algorithm gets 1 point, losing gives 0 points. The total score is the sum of all scores across all games.

Threading model

- The num_threads command line argument sets the number of threads for the simulation, as follows:
    - If the argument is missing or if num_threads provided = 1, the program will use a single thread (the main thread).
    - If num_threads provided >= 2, the program will interpret <num_threads> as the requested number of threads for running the actual simulation in addition to the main thread.
    - Above means that the total number of threads will never be 2.
- Note that the exact number of threads may be lower than requested in the command line, in case there is no way to properly utilize the required number of threads you should not open threads which cannot be utilized.
- Note that it is totally OK that your main thread will be waiting for all other threads in a join call (or any other similar blocking wait).
- If you prefer to load all the required .so files ahead it might be a proper solution!
  In case you find a way to avoid loading algorithms / game_managers instances simultaneously but rather load them (once!) only when needed and unload if not being used anymore (but without loading them again!) – you can ask for a bonus for that.
- It is better not to lock if you can avoid locking. But if you need to lock, you should of course lock.
- In case a result table can be known in advance, you can create it ahead (there is no need for a "sparse matrix" for managing results).
- Creating an Algorithm instance / GameManager instance from their factories should be cheap. Do not cache instances, just recreate them using the factories when needed (this is NOT the same as unloading and loading the same .so file, which should be avoided).

API and Abstract Base Classes (Old and New)

To allow common implementation you need to use common abstract classes. The actual header files would be published for use *as-is* and may/shall be in use by any of the three projects: *Simulator*, *GameManager* and *Algorithm*.

---

common/ActionRequest.h

```
enum class ActionRequest {
      MoveForward, MoveBackward,
      RotateLeft90, RotateRight90, RotateLeft45, RotateRight45,
      Shoot, GetBattleInfo, DoNothing
};
```

---

common/TankAlgorithm.h

```
class TankAlgorithm {
public:
      virtual ~TankAlgorithm() {}
      virtual ActionRequest getAction() = 0;
      virtual void updateBattleInfo(BattleInfo& info) = 0;
};

using TankAlgorithmFactory =
      std::function<std::unique_ptr<TankAlgorithm>
            (int player_index, int tank_index)>;
```

---

common/BattleInfo.h

```
class BattleInfo {
public:
      virtual ~BattleInfo() {}
};
```

---

common/SatelliteView.h

```
class SatelliteView {
public:
      virtual ~SatelliteView() {}
      virtual char getObjectAt(size_t x, size_t y) const = 0;
};
```

common/Player.h

```
class Player {
public:
      virtual ~Player() {}
      virtual void updateTankWithBattleInfo
            (TankAlgorithm& tank, SatelliteView& satellite_view) = 0;
};

using PlayerFactory =
std::function<std::unique_ptr<Player>
(int player_index, size_t x, size_t y, size_t max_steps, size_t num_shells)>;
```

common/GameResult.h

```
struct GameResult {
      int winner; // 0 = tie
      enum Reason { ALL_TANKS_DEAD, MAX_STEPS, ZERO_SHELLS };
      Reason reason;
      std::vector<size_t> remaining_tanks; // index 0 = player 1, etc.
      std::unique_ptr<SatelliteView> gameState; // at end of game
      size_t rounds; // total number of rounds
};
```

common/AbstractGameManager.h

```
using std::string;
class AbstractGameManager {
public:
      virtual ~AbstractGameManager() {}
      virtual GameResult run(
            size_t map_width, size_t map_height,
            const SatelliteView& map, // <= a snapshot, NOT updated
            string map_name,
            size_t max_steps, size_t num_shells,
            Player& player1, string name1, Player& player2, string name2,
            TankAlgorithmFactory player1_tank_algo_factory,
            TankAlgorithmFactory player2_tank_algo_factory) = 0;
};

using GameManagerFactory =
std::function<std::unique_ptr<AbstractGameManager>(bool verbose)>;
```

Automatic Registration

To allow simple discovery of the factories, we need to define a common registration process so all loaded classes would register themselves automatically.

Note: we will explain the process in one of the following class meetings or in a zoom meeting or zoom recording. It may seem complicated but it is not too much.

You should use the following header files (the exact header file would be published and you would have to use it as-is) for auto registration:

```
// common/PlayerRegistration.h
struct PlayerRegistration {
  PlayerRegistration(PlayerFactory);
};

#define REGISTER_PLAYER(class_name) \
PlayerRegistration register_me_##class_name \
        ( [] (int player_index, size_t x, size_t y, size_t max_steps, size_t num_shells) { \
                return std::make_unique<class_name>(player_index, x, y, max_steps, num_shells); \
         } );

// common/TankAlgorithmRegistration.h
struct TankAlgorithmRegistration {
  TankAlgorithmRegistration(TankAlgorithmFactory);
};

#define REGISTER_TANK_ALGORITHM(class_name) \
TankAlgorithmRegistration register_me_##class_name \
        ( [](int player_index, int tank_index) { \
                return std::make_unique<class_name>(player_index, tank_index); \
        } );

// common/GameManagerRegistration.h
struct GameManagerRegistration {
  GameManagerRegistration(GameManagerFactory);
};

#define REGISTER_GAME_MANAGER(class_name) \
GameManagerRegistration register_me_##class_name \
        ( [] (bool verbose) { return std::make_unique<class_name>(verbose); } );
```

You are free to implement the .cpp files for the above auto registration classes as you wish but **you are not allowed to change the header files**!

Your concrete implementations for Player, TankAlgorithm and AbstractGameManager will have the following lines in their .cpp, in the global scope:

```
REGISTER_GAME_MANAGER(<class_name>)  e.g.:
REGISTER_GAME_MANAGER(GameManager_098765432_123456789)
```

And similarly:

```
REGISTER_TANK_ALGORITHM(<class_name>)  e.g.:
REGISTER_TANK_ALGORITHM(TankAlgorithm_098765432_123456789_A)
```

And:

```
REGISTER_PLAYER(<class_name>)  e.g.:
REGISTER_PLAYER(Player_098765432_123456789_A)
```

The registered classes are unaware of the actual implementation of the registration process, which happens in the Simulation project (and will be explained to you). Note that the easiest way to implement the registration is to make the Simulation class a Singleton.

Note: the registration headers are in use by both the Simulator project and the Algorithm / GameManager projects. However, the .cpp files of the registration **should be part of the Simulator project only** and their implementation is *on you* (you will be presented with a proposed implementation).

Additional Output and Error Handling

The GameManager shall create output files as in assignment 2, iff -verbose appears on the command line.
The GameManager and Algorithm project may create error logs.
There is no need for the Simulator to handle a crash scenario of GameManager / Algorithm, however in any other case the Simulator shall not crash.

Bonus

There would be a class competition and a bonus would be given for the best algorithms.
Additional bonuses can be requested as in previous exercises.

Submission

You shall submit a zip named ex3_<student1_id>_<student2_id>.zip that contains:

- ***5 folders*** - as described above
- ***4 makefiles*** (or CMakes) - 3 for each of the projects, inside the folder of the project that it builds. Additional one at the root (directly in the zip without any folder), for building all 3 projects.
- ***students.txt*** - a text file that includes one line per submitter: <user_name> <id> (do not include the character ‘<’ nor ‘>’ - directly in the zip without any folder
- ***README.md*** - info and remarks about your implementation - directly in the zip without any folder

DO NOT SUBMIT THE FOLLOWING FILES

- binary files
- external libraries (you may only use standard C++ libraries or libraries which were explicitly approved in the course forum)

<u>Additional Notes and Requirements</u>

1. As in assignment 2, you are not allowed to use *new* and *delete* in your code.

2. As in assignment 2, you should prefer using unique_ptr over shared_ptr – use shared_ptr only if there is actual need for sharing and the lifetime is unknown.

3. Do not forget to unload .so files before the program ends, by using the dlclose command. Make sure not to call dlclose if there are objects related to the .so which are still alive.

4. Note: the minimal requirement for your algorithm are:
    a. Try not to shoot its own tanks.
    b. Try to shoot opponents' tanks when relevant.
    c. Try moving a tank when a shell is chasing it (you can decide that this is the only trigger for moving a tank).

5. [TBD]