"Red vs Blue" - a C++ code for simple battle simulations

July 14, 2023

Abstract

This report contains description and documentation of a C++ code aimed at simulating a clash between two opposing armies, internally labelled as 'Red' and 'Blue', with the core aim being the ability to run the simulation to completion using only simple rules programmed for each army unit. This was accomplished by creating a base class for an army unit, serving as the stem of a tree, containing all specialized concrete unit classes. A separate 'army' class was used as a container class, holding and managing units. A 'battlefield' class was created to manage the simulation itself, serving as a container for army classes as well as the interface responsible for running and stopping the simulation itself. A separate set of functions was defined to allow the user to specify the simulation parameters using a basic configuration file. The resulting program performs reasonably well, however, the simple unit action rules frequently lead to drawing scenarios. Expanding them to more robust pathfinding algorithms would be the logical next step in development.

1 Outline

The main task of this project was to produce a program capable of running and animating a basic simulation of a battle between two armies based on user input in the form of a configuration file. This file would contain the basic parameters needed to run the simulation, such as the width and height of the rectangular battlefield, on which the battle takes place, and maximum number of iterations. The names of the two armies along with their unit composition would also need to be provided as part of the configuration.

The basic structure of the code itself consisted of a class tree of unit classes, with an abstract generic 'unit' class serving as the stem, along with a container 'army' class to hold and manage the unit classes. The simulation itself is managed by a 'battlefield' class, serving as a container for army classes as well as deciding when to stop the battle.

The remaining sections of this report contain a more thorough description of the code design (section 2), a demonstration of it in action (section 3) and a discussion of the code's capabilities and potential improvements (section 4).

2 Design description

2.1 Class tree

The code itself was designed with flexibility in mind, thus all army units were described through classes derived from a single abstract 'unit' class, to allow for an easier adding of extra unit types. A concrete generic 'army' class was used to hold unit classes via a vector of shared pointers to implement runtime polymorphism. A final 'battlefield' class was used as an interface for managing the entire simulation and in turn contained army classes through another vector

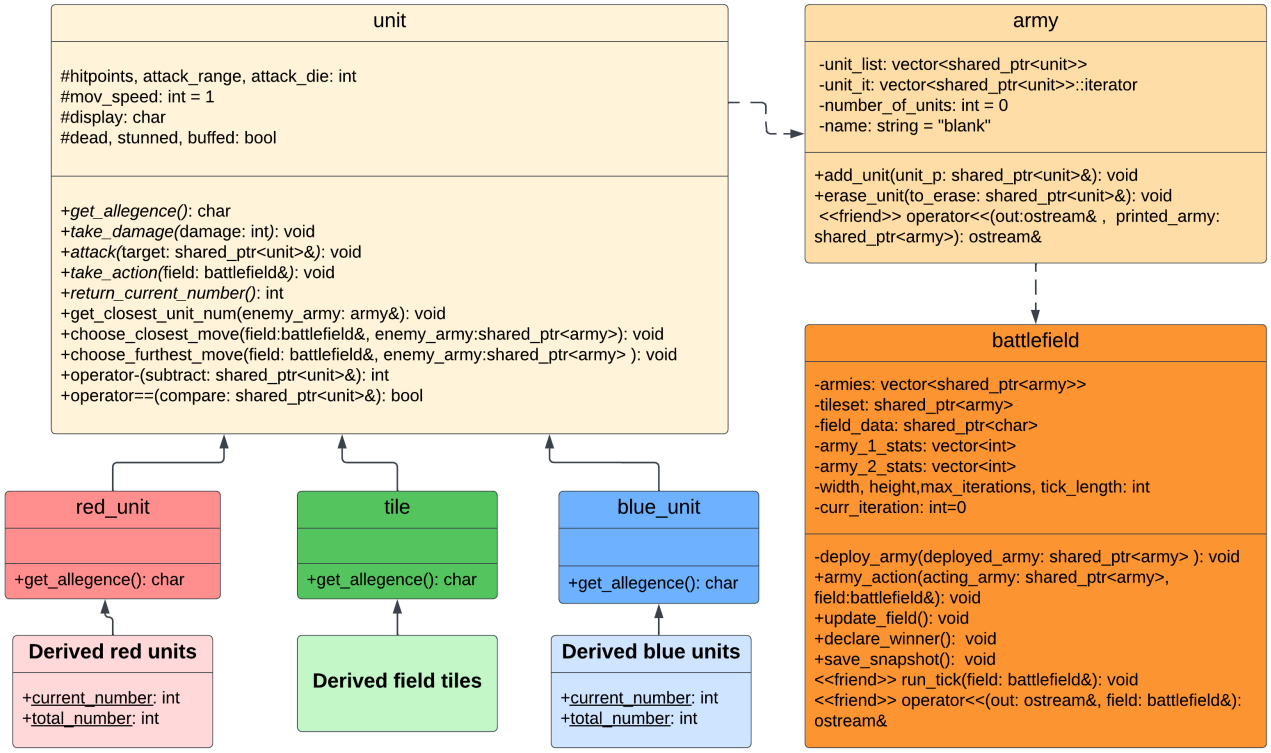


Figure 1: A UML chart showcasing the basic structure of the code that handles running the simulation. Solid arrows indicate inheritance, dashed arrows - dependence. For the purposes of simplification functions simply returning private data members along with constructors/destructors are not listed. For the same reason, unit class variables used for choosing moves and attack targets are omitted as well.

of shared pointers - smart pointers were used throughout the implementation to avoid memory leaks. A schematic display of the class specification is shown in figure 1. As displayed in the figure, the concrete unit classes are implemented through derived abstract classes, which determine their allegiance by overriding the `get_allegence()` function to return a different character - 'r' for red units, 'b' - blue and 't' - battlefield tiles, which have no formal allegiance and serve as environmental hazards.

The concrete unit classes are all implemented by overriding the remaining virtual functions to differentiate their behavior. Each concrete unit class also contains two static variables tracking the number of each type remaining on the field (`current_number`) and the starting number (`total_number`). Both are incremented by the constructor, but only the former is decremented by the destructor. The key function which differentiates unit behavior is the `take_action` function. Below is an example of a `take_action` function for a `red_knight` unit:

```
void red_knight::take_action(const battlefield& field)
{
    if (!dead && !stunned) {
        //Buff to increase attack damage if strength dropping
        if (number_of_red_knight <= 0.5 * max_num_of_red_knight) buffed = true;
        get_closest_unit_num(*field.get_blue_army());
        //Attack if in range, move closer if not
        if (min_dist <= attack_range && min_dist > 1) {
            attack(field.get_blue_army()->get_unit(closest_enemy_num));
        }
        else choose_closest_move(field, field.get_blue_army());
```

```
    }
    else if (stunned) stunned = false;
}
```

The above function first checks the remaining number of `red_knight` units (`number_of_red_knight`) against their starting number (`max_num_of_red_knight`), if the ratio is less than half, the unit is buffed. The exact consequences of a buff manifest differently for every unit, in this case, the damage dealt in the `attack` function is increased. After this check, base class function `get_closest_unit_num` is called to obtain the index of the currently closest enemy unit, the distance between units is calculated as Manhattan distance. Further behavior consists of attacking the enemy if it is in range or moving closer if not. Some other units also check if the enemy is closer than 2 tiles away and choose to retreat instead. Another factor differentiating units is the set of default parameters set by the constructor - these include the display character, printed on screen to showcase the unit's position, the number of hitpoints and the attack die value, used for calculating base damage dealt by the unit using `damage = std::rand()%attack_die + 1`. The basic properties of all derived unit classes are summarized in table 1. Every concrete unit class may be initialized by two parametrized constructors - one simply sets their positions on the field, the other also allows for setting the value of `hitpoints` and `buffed` parameters.

Name	Display	Hitpoints	Hit die	Range	Behavior
`red_sw`	`'S'`	30	12	1	Attacks the closest enemy buff increases attack damage
`red_arch`	`'A'`	15	8	4	Retreats if threatened buff increases attack range
`red_knight`	`'K'`	25	10	2	Attacks the closest enemy buff increases attack damage
`blue_sw`	`'s'`	30	12	1	Same as the red version buff increases health
`blue_arch`	`'a'`	15	8	4	”
`blue_knight`	`'k'`	25	10	2	”
`caltrops`	`'+'`	–	4	0	Damages anyone stepping on it

Table 1: A brief summary of all concrete unit types used in the simulation. Units with capital letters as display characters belong to the `red_unit` branch, others - to the `blue_unit` branch. The only currently implemented unit of the `tile` branch is `caltrops`.

2.2 Configuring and running the simulation

The parameters with which the simulation is initialized can be set via a configuration file `simulation.config`. These parameters include the dimensions of the battlefield, the names of the two armies along with their composition. An example configuration is given in figure 2. As shown in the figure, the army units are inputted by providing their display character, followed by their x and y coordinates separated by ',', parameters for different units are separated by '&'. The coordinates are limited to $0 \leq x <$ `width` and $0 \leq y <$ `height`.

The processing of the configuration file and initialization of the simulation is done by a separate suite of functions, defined in `cfg::` namespace. The procedure starts with reading the file line by line, stripping each line of spaces to allow for more flexible syntax, lines containing configuration parameters are split along the '=' sign, the split parts are then stored in `std::map` type containers as keys and values respectively. These are then unpacked and used to initialize

```
#Lines beginning with '#' are ignored
#Enter base simulation parameters, tick_length is measured in miliseconds
tick_length =  250
max_iterations = 100
snapshot_rate = 150

#Enter battlefield parameters (width, height, tileset containing special tiles)
width = 10
height = 15

#variable name must be provided, write 'tileset=' if no special tiles desired
tileset =

#Note: army names currently do not support whitespaces
#Enter army 1 parameters
army_1_name = Red
army_1_units = K, 5, 1 & K, 2, 2 & A, 3, 3 & S, 4, 4 & S, 4, 4 & K, 4, 4

#Enter army 2 parameters
army_2_name = Blue
army_2_units = a, 9, 9 & a, 9, 9 & k, 8, 6 & k, 1, 1 & k, 2, 2 & k 8, 8
```

Figure 2: A screenshot of an example configuration file showing its syntax and the parameters available.

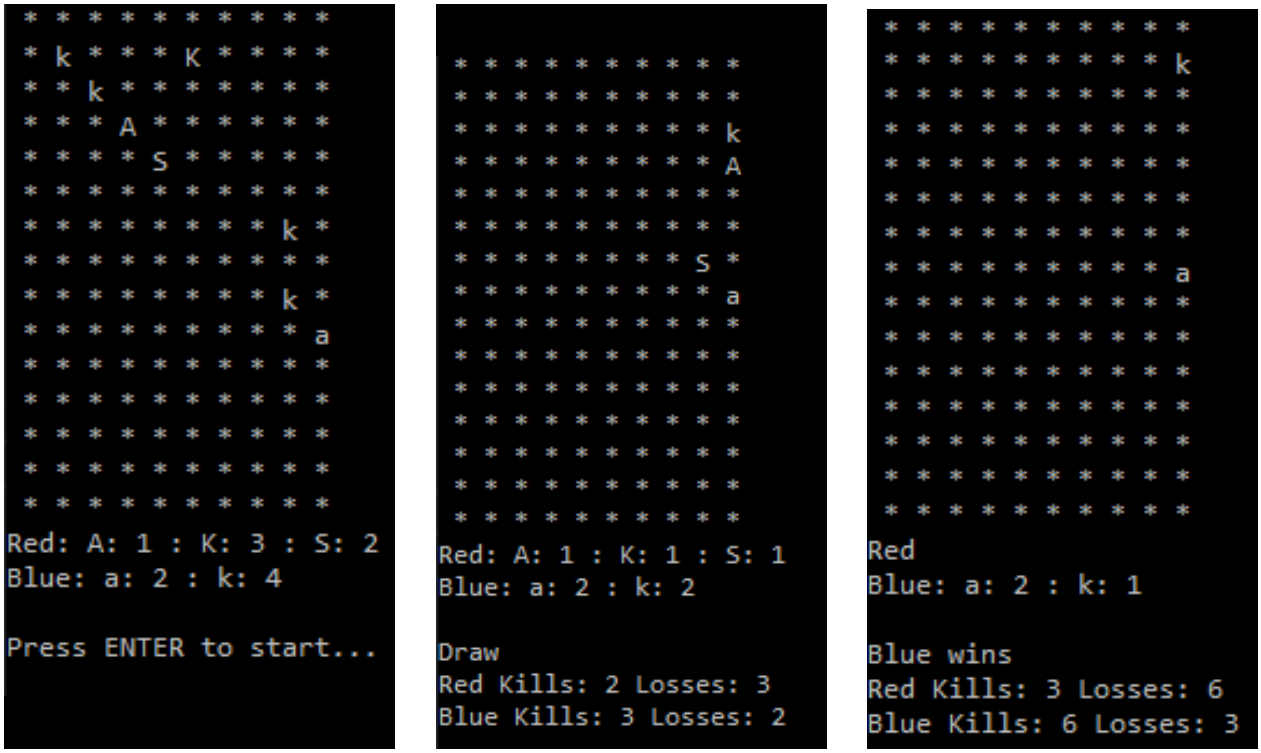
the armies and battlefield. In case of incorrectly formatted lines, invalid or missing parameters the program terminates with an error message.

The simulation itself is run and animated in the console window through the following loop in `main()`:

```
    while (!field->is_stopped()) {
        std::cout << *field << std::endl;
        run_tick(*field);
        std::this_thread::sleep_for(std::chrono::milliseconds(tick_length));
        if (!field->is_stopped()) utl::clear_screen();
    }
```

The `tick_length` parameter here controls the speed of animation, `utl::clear_screen()` is used to call either `CLS` if compiled on Windows or `clear` otherwise to clear the console screen for the next animation frame. It is part of the `utl::` namespace, which contains several other helper functions used throughout the code. The program is executed via the `run_tick` function below:

```
void run_tick(battlefield& field)
{
    if (!field.stop_sim) {
        if (field.curr_iteration % field.snapshot_rate == 0) field.save_snapshot();
        //Environmental damage resolves first
        field.army_action(field.tileset, field);
        field.army_action(field.armies[0], field); //Reds move after
        field.army_action(field.armies[1], field);
        field.update_field(); //Redeploy armies once all actions resolve
        field.curr_iteration++;
    }
    if (field.curr_iteration > field.max_iterations) field.stop_sim = true;
}
```



(a) Initial state (b) Draw (c) Blue victory

Figure 3: Screenshots of an example initial state of the simulation along with two of the possible outcomes - a draw when the maximum iteration number is exceeded and a victory for Blue.

The above function simply checks if the iteration should be run, saves a snapshot if it is on the right iteration, then resolves environmental damage done by field tiles. Finally, each army makes their moves and the battlefield is updated. Simulation runs until one army runs out of units or `max_iterations` value is reached, in which case, the battle is declared a draw.

The `snapshot_rate` parameter specifies on which iteration a snapshot of the current state of the field should be taken. Snapshots are saved in separate, numbered files of the form `snapshot_#`, where `#` stands for the file number. Upon each start of the program the user is given the choice to load from a snapshot by specifying a number. The loading of snapshots is handled similarly to that of configuration files, except all loaded units are instantiated using the second constructor - to preserve the number of hitpoints and the buffed status they may have had at the time of the snapshot. It should be noted that static variables are not saved by the code thus only partial recreations from snapshots are possible. Functions to load snapshots are defined under `snp::` namespace to avoid collisions with the configuration reader.

3 Working demonstration

Loading the configuration provided in figure 2, results in an initial state given in figure 3a, with the Reds having 1 archer, 3 knights and 2 swordsmen, the Blues - 2 archers and 4 knights. Evolving from this initial state generally results in two outcomes - in one, the archer units of the opposing sides position themselves in a way that always allows them to dodge enemy attacks as their first action is to retreat if there is an enemy unit less than 2 tiles away, since all units have the same movement speed, it becomes difficult for them to catch up and deal the finishing blow resulting in a stalemate (figure 3b). The other outcome (figure 3c) sees the Blues quickly neutralizing the sole red archer, pinning the remaining Reds with their knights and winning through a numeric advantage. This dual outcome is largely determined by the fact that damage

rolls are random as well as each attack having a chance to stun the attacked, preventing them from taking action for a single tick.

Battles involving simpler compositions of just knights and swordsmen tend to be more decisive, however, in all cases units tend to quickly converge into groups as there is nothing preventing multiple units occupying a single tile and the pathfinding functions are largely shared. The number of these groups, however, tends to be dependent on initial clustering of units. The example in figure 3 tends to result in two clusters emerging due to some blue units being placed behind the red units. Some of this behavior is visible in figure 3b.

4 Conclusions

In conclusion, the program performs the task of simulating a battle between two opposing forces reasonably well despite the general simplicity of the functions determining unit behavior. The procedure of using `std::map` for setting the initial simulation parameters from a file generally works well in allowing for neatly formatted and easily readable configuration file layouts to be used. However, the stripping of whitespaces from each line before reading it means that an army name like `Blue Army` will be read in as `BlueArmy` by the program. This is largely a cosmetic issue, but may be worth fixing in the future to allow for more flexible user input.

As for the simulation itself, the general simplicity of the `take_action` functions coupled with all units having the same movement speed sometimes works against it in producing an abundance of drawing scenarios. While this does not cause technical issues due to the `max_iterations` parameter, it does lead to outcomes that are not particularly interesting. Coupled with units being allowed to pass through each other this also leads to clustering. Changing this condition along with the `take_action` function to, for example, utilize the wave algorithm to compute a cost of traversing to each tile and making decisions based on that may lead to more complex emergent behaviors.