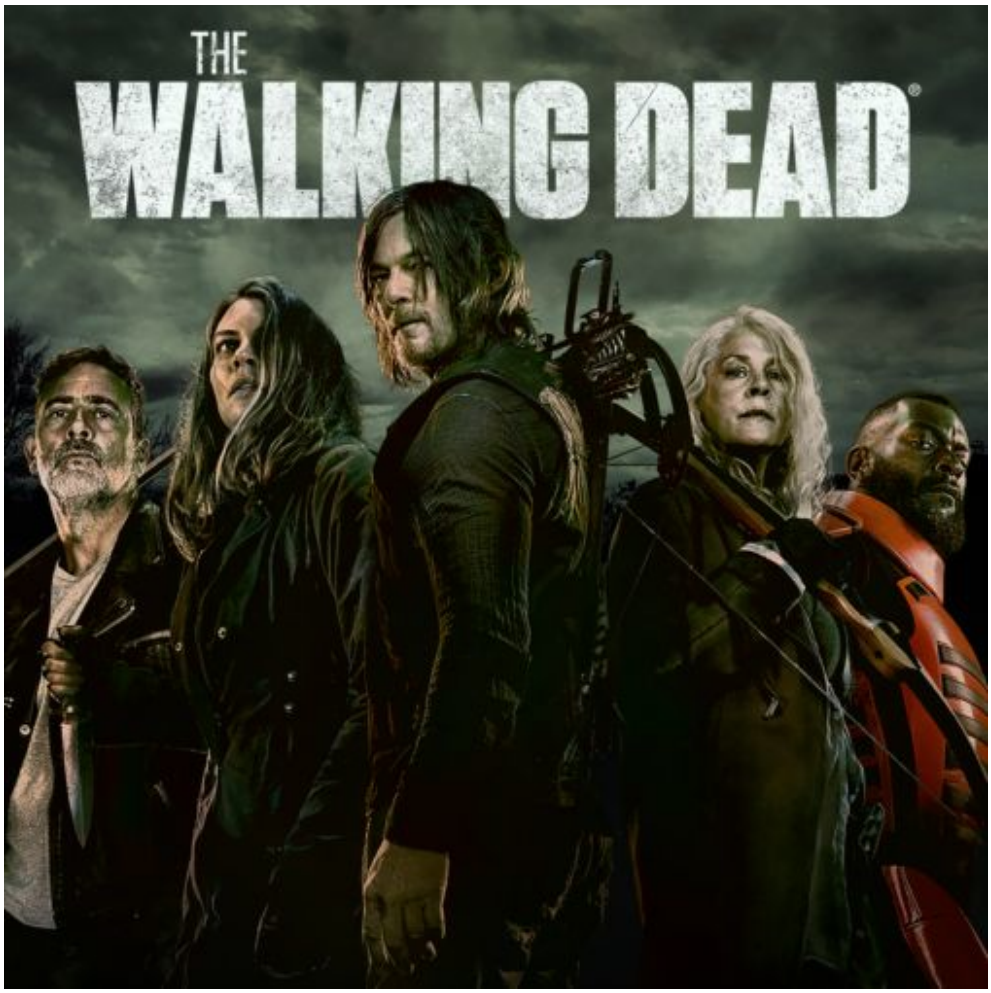
The Walking Dead

Albert Oliveras

November 10, 2022



1 Game Rules

Due to yet unknown reasons, the Earth is going through a zombie apocalypse. The few human survivors fight to stay alive and to gain control of the scarce existing resources.

This is a game for four players, identified with numbers from 0 to 3. Each player has control over a clan of alive units. But there are two additional types of units in the game: dead units and zombies.

The game lasts 200 rounds, numbered from 1 to 200. Each unit can move at most once per round. Dead units, as expected, cannot move. During the rounds, the clans accumulate points and the winner of the game is the clan with the largest amount of points after round 200.

The board game has dimensions 60×60. Units cannot move outside it. A position in the board is given by a pair of integers (r, c) where $0 \leq r < 60$ and $0 \leq c < 60$. The top-left position is $(0, 0)$, whereas the bottom-right position is $(59, 59)$. Hence, r (for row) refers to the vertical axis and c (for column) refers to the horizontal axis. Each cell in the board is either part of a street or is full of waste. Units cannot move on waste and must necessarily move through streets.

Clans start the game with a certain amount of strength points. The *strength of a clan* is defined as $\left\lfloor \frac{\texttt{strength points}}{\texttt{alive units}} \right\rfloor$ and is key for determining the winner of the fights that will occur during the game. After each round, the strength points of a clan will be decremented in an amount equal to the number of alive units of that clan. However, strength points are never negative. In order to increment strength points, alive units can collect food. It is easy to see that a clan with a lot of alive units needs to collect a lot of food in order to have a good amount of strength. Thus, some units decide to abandon their clan: at each round, with a probability of 20%, a unit from the clan with the most number of alive units becomes part of the clan with the least amount of alive units. If there are several clans with these properties, one of them is chosen randomly.

Movements of alive units. An alive unit moves in the board following these rules:

- It can only move to adjacent cells horizontally and vertically, never in diagonal.
- If it tries to move to a cell occupied by waste, by a unit of the same clan, o by a dead unit, the movement will be ignored.
- If it moves to a cell with food, the strength points of its clan will be incremented, the unit will occupy the cell, the food item will disappear and the clan will possess this cell. At the end of the round, a food item will

reappear in another position. Note that we will never find a cell occupied by food and by a unit.

- It it tries to move to an empty cell (without food or unit), the movement will be done and the clan will possess this cell.
- If it moves to a cell occupied by a zombie, the zombie will die but the unit will not move. As a result, the clan will receive a certain number of points and, at the end of the round, an alive unit of this clan will reappear in another position, and the zombie will disappear.
- If it moves to a cell occupied by an alive unit of another clan, a fight will start. The unit losing the fight will become a dead unit, and the process of zombie conversion will start: after a certain number of rounds, it will be a zombie. If this unit had already started a zombie conversion process for being bitten by a zombie, the process will restart. The clan of the unit winning the fight will receive a certain number of points, but the unit will not move. The winner of the fight is determined as follows:

  With a probability of 30%, the unit starting the fight will surprise the other unit and hence will win the fight immediately. Otherwise, if the strengths of the clans involved in the fight are N and M, respectively, the first unit will win with probability $N/(N+M)$ and the second unit with probability $M/(N+M)$. If $N = M = 0$, the units will have the same probability of winning.

Movements of zombies. Zombies are not part of any clan and hence are not controlled by any player. Zombies will always move to the closest alive unit, considering that they cannot move through waste. If there are multiple units at the same distance, one of them will be randomly chosen. A zombie will always move following these rules:

- It can move to adjacent cells horizontally, vertically and also in diagonal.
- It will never move to a cell occupied by waste, a dead unit or a zombie.
- If it moves to a cell occupied by food, the food will disappear. At the end of the round, a food item will reappear in a random position. If a clan was possessing this cell, it will stop doing so.
- If it moves to an empty cell (with no food or unit) owned by a clan, this clan will stop possessing this cell.
- If it tries to move to a cell with an alive unit on it, the movement will not be done. However, it will bite the unit and it will start the process of becoming a zombie, that will finish after a certain number of rounds. If that process had already started due to a previous bite, the process will not restart, but rather continue. During the conversion, the unit will behave as an alive one.

As a result of the previous rules, the total number of units is constant during the game.

Object regeneration. Every time a food unit or an alive unit needs to be regenerated, it will always appear in an empty cell *C* with no unit or food in the surrounding cells (the ones with an x in the table):

x	x	x	x	x
x	x	x	x	x
x	x	C	x	x
x	x	x	x	x
x	x	x	x	x

If there is no safe cell in this sense, the object will reappear in an empty cell, with no unit or food in it.

It is important to remark that units have an identifier that never changes, not even after they are regenerated. For example, if a unit with a certain initial identifier becomes a zombie, it will still have the same identifier. If, later on, the zombie is killed and reappears as an alive unit of another clan, the identifier will still be the same.

Score computation. The score of a clan in a given round is given by two components. On the one hand, for each zombie that the clan has killed so far, 10 points will be awarded. For each killed alive unit, 50 points will be given.

On the other hand, for each cell owned by the clan **in this round** 1 point is given. The score of a clan is the sum of these two components. Note that the score can be decremented if a clan loses the possession of a cell. The constants 10, 50 and 1, as well as other constants that define the initial settings of the game, are defined in the input file `default.cnf`. All games will be played with the values given in this file.

Execution of orders. In each round, more than one order can be given to the same unit, although only the first such order (if any) will be selected. Any player that tries to give more than 1000 orders during the same round will be aborted.

Every round, the selected movements of the four players will be executed using a random order, but respecting the relative order of the units of the same clan. As a consequence of the previous rule, consider giving the orders to your units at every round from most urgent to least urgent.

Take into account that the every movement is applied to the board resulting of the previous movements. For example, consider the board

x	x	x	x
x	F	U	x
x	V	x	x
x	x	x	x

where F represents food and U, V two alive units of different clans. Let us assume that the player controlling U decides that it should go left, and the player controlling V decides to go up. If the V movement is executed first, then U has no food left, because V has already taken it. Moreover, the subsequent movement of U is an attack to V. If U dies in this fight, in the visualization of the game we will see a transition from the previous matrix to a situation where U has disappeared. Obviously, there could be no attack between U and V in the original configuration because units cannot move diagonally, but the execution order of the movements has made it possible. Take this into account when you do not understand certain situations in the visualization of games.

After the execution of all movements from the players, the movements of zombies are executed. After that, the following processes are executed in this order: units that have finished their conversion process become zombies, killed zombies will reappear as alive units, a unit from the clan with the largest number of alive units might move to the clan with the least number of alive units, food items will be regenerated and scores will be updated.

2 The Viewer

We know describe the viewer:

- In the upper part there are buttons that allow us to reproduce or stop the game, go to the beginning or to the end, enable or disable the animation mode or obtaining a help window with more ways to control the visualization. You will also see the current round and a button to close the game. A horizontal toggle bar shows in which point of the game the current round is.
- In the left column, each player appears with her name and color. Below, we can see the current score, the number of alive units and the player strength. In the game played in Jutge.org, we can also see the percentage of CPU time that has been used (if consumed, a message 'out' is displayed). In the upper-right part we can see the colors of the players, ordered by score.
- Cells have the color of the player that owns them. Otherwise, they are white.
- Waste cells are dark grey.
- Alive units are represented as circles of the corresponding color. If they are becoming zombies, they are squares.

- Dead units are represented by a cross.
- Zombies are represented by a red square with a black outer frame.
- Food units are represented by a red circle with a black outer frame.

3 Programming the game

The first thing you should do is downloading the source code. It includes a C++ program that runs the games and an HTML viewer to watch them in a reasonable animated format. Also, a "Null" player and a "Demo" player are provided to make it easier to start coding your own player.

3.1 Running your first game

Here, we will explain how to run the game under Linux, but it should work under Windows, Mac, FreeBSD, OpenSolaris, ... You only need a recent `g++` version, `make` installed in your system, plus a modern browser like Firefox or Chrome.

1. Open a console and `cd` to the directory where you extracted the source code.
2. If, for example, you are using a 64-bit Linux version, run:

   ```
   cp AIDummy.o.Linux64 AIDummy.o
   ```

   ```
   cp Board.o.Linux64 Board.o
   ```

   If you use any other architecture, choose the right objects you will find in the directory.
3. Run

   ```
   make all
   ```

   to build the game and all the players. Note that `Makefile` identifies as a player any file matching `AI*.cc`.
4. This creates an executable file called `Game`. This executable allows you to run a game using a command like:

   ```
   ./Game Demo Demo Demo Demo -s 30 -i default.cnf -o default.res
   ```

   This starts a match, with random seed 30, of four instances of the player Demo, in the board defined in `default.cnf`. The output of this match is redirected to `default.res`.
5. To watch a game, open the viewer file `viewer.html` with your browser, for example by running `firefox viewer.html`, and load the file `default.res`.

Use

```
./Game --help
```

to see the list of parameters that you can use. Particularly useful is

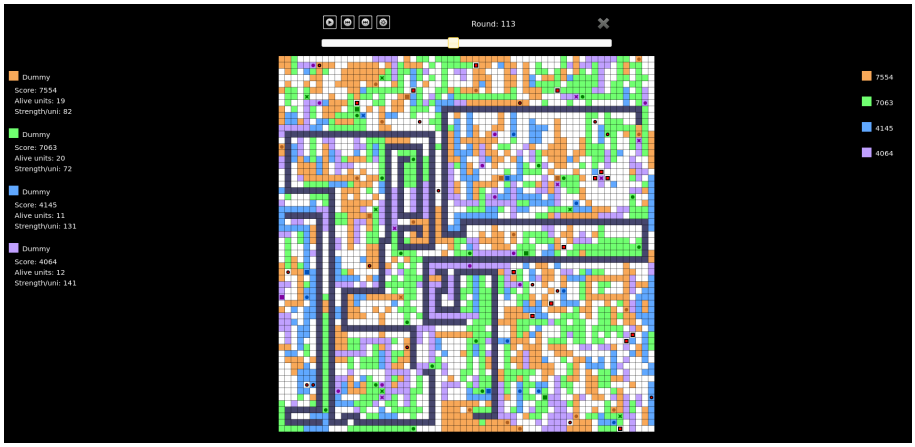
```
./Game --list
```

to show all the recognized player names.

If needed, remember that you can run

```
make clean
```

to delete the executable and object files and start over the build.



3.2 Adding your player

To create a new player with, say, name `Rick`, copy `AINull.cc` (an empty player that is provided as a template) to a new file `AIRick.cc`. Then, edit the new file and change the

#define PLAYER_NAME Null

line to

#define PLAYER_NAME Rick

The name that you choose for your player must be unique, non-offensive and at most 12 characters long. This name will be shown in the website and during the matches.

Afterwards, you can start implementing the virtual method *play*(), inherited from the base class *Player*. This method, which will be called every round, must decide the orders to give to your units.

You can define auxiliary type definitions, variables and methods inside your player class, but the entry point of your code will always be the *play* () method.

From your player class you can also call functions that you will find in the following files:

- `State.hh`: consulting the state of the game.
- `Action.hh`: giving orders to your units.
- `Structs.hh`: useful data structures.
- `Settings.hh`: consulting the initial parameters of the game.
- `Player.hh`: method *me*().
- `Random.hh`: random number generation.

You will find a summary of this information in the file `api.pdf`. You can also examine the code of the "Demo" player in `AIDemo.cc` as an example of how to use these functions.

Note that you must not edit the *factory* () method of your player class, nor the last line that adds your player to the list of available players.

3.3 Restrictions when submitting your player

When you think that your player is strong enough to enter the competition, you can submit it to the Jutge. Since it will run in a secure environment to prevent cheating, some restrictions apply to your code:

- All your source code must be in a single file (like AIRick.cc).
- You cannot use global variables (instead, use attributes in your class).
- You are only allowed to use standard libraries like `iostream`, `vector`, `map`, `set`, `queue`, `algorithm`, `cmath`, ... In many cases, you don't even need to include the corresponding library.
- You cannot open files nor do any other system calls (threads, forks, ...).
- Your CPU time and memory usage will be limited, while they are not in your local environment when executing with `./Game`.
- Your program should not write to **cout** nor read from **cin**. You can write debug information to **cerr**, but remember that doing so in the code that you upload can waste part of your limited CPU time.
- Any submission to the Jutge must be an honest attempt to play the game. Any try to cheat in any way will be severely penalized.
- Once you have submitted a player to Jutge that has defeated the Dummy player, you can send more submissions but you will have to change the

player name. That is, once a player has defeated Dummy, his name is blocked and cannot be reused.

4 Tips

- **DO NOT GIVE OR ASK YOUR CODE TO/FROM ANYBODY.** Not even an old version. Not even to your best friend. Not even from students of previous years. We will use plagiarism detectors to compare pairwise all submissions and also with submissions from previous editions. However, you can share the compiled `.o` files.

  Any detected plagiarism will result in an **overall grade of 0** in the course (not only in the Game) of all involved students. Additional disciplinary measures might also be taken. If student A and B are involved, measures will be applied to both of them, independently of who created the original code. No exceptions will be made under any circumstances.
- Before competing with your classmates, focus on qualifying and defeating the "Dummy" player.
- Read only the headers of the classes in the provided source code. Do not worry about the private parts nor the implementation.
- Start with simple strategies, easy to code and debug, since this is exactly what you will need at the beginning.
- Define basic auxiliary methods, and make sure they work properly.
- Try to keep your code clean. Then it will be easier to change it and to add new strategies.
- As usual, compile and test your code often. It is *much* easier to trace a bug when you only have changed few lines of code.
- Use `cerr`s to output debug information and add `assert`s to make sure the code is doing what it should do. Remember to remove (or comment out) the `cerr`s before uploading your code to Jutge.org. Otherwise, your submission will be killed.
- When debugging a player, remove the `cerr`s you may have in the other players' code, to make sure you only see the messages you want.
- By using commands like `grep` in Linux you can filter the output that `Game` produces.
- Switch on the `DEBUG` option in the Makefile, which will allow you to get useful backtraces when your program crashes. There is also a `PROFILE` option you can use for code optimization.
- If using `cerr` is not enough to debug your code, learn how to use `valgrind`, `gdb`, `ddd` or any other debugging tool.

- You can analyze the files that the program `Game` produces as output, which describe how the board evolves after each round.
- Keep a copy of the old versions of your player. When a new version is ready, make it fight against the previous ones to measure the improvement.
- Make sure your program is fast enough: the CPU time you are allowed to use is rather short.
- Try to figure out the strategies of your competitors by watching matches. This way you can try to defend against them or even improve them in your own player.
- Do not wait till the last minute to submit your player. When there are lots of submissions at the same time, it will take longer for the server to run the matches, and it might be too late!
- You can submit new versions of your program at any time.
- And again: Keep your code simple, build often, test often. Or you will regret.