Crematoria

Salvador Roura

January 27, 2026

1 Introduction

Crematoria is a planet in the Igneon System. Many years ago, its surface was not habitable because of its extreme temperatures: one would be incinerated if in the sunlight, or frozen solid at night. Because there was only a small buffer of minutes where travel on the surface was possible, a maximum security prison was built in some caves underground, from where it was almost impossible to escape. Some elevators connected the surface of the planet to the prison.

To increase security and control the population, some huge hellhounds were released in the prison. Those hellhounds were genetically engineered animals, hybrids of dogs and reptiles, who indiscriminately attacked and devoured any inmates that could not avoid them.

It is known that at least one Furyan (Riddick) was temporarily locked inside the prison of Crematoria. Furyans are a deviation of humans with great physical abilities, which make them the most fearsome warriors. Riddick was able to escape from Crematoria, but this would be another story...[1]

[1] This game is based on the film "The Chronicles of Riddick". It's not the best film ever made, but it's wacky and weird, which are obviously good qualities.

Necromongers are the natural enemies of Furyans. Necromongers fanatically believe in the Underverse, their promised land. Therefore, they search for an honorable death in combat. The Necromonger fleet moves from planet to planet, appearing as a comet before arrival.

Necromongers also believe in a philosophy that says "you keep what you kill", meaning that ending another's life entitles you to their property.[2]

Many years passed, and the surface of Crematoria became walkable at night. This fact made the prison less secure, so it was abandoned. Recently, it has been discovered that the sunlight generates valuable gems. Four rival groups of pioneers have decided to establish themselves in Crematoria. The goal of each group is twofold: to conquer underground cells, and to pick gems outside.

Since pioneers are weak, they decided to contract Furyan warriors to "protect" them. This was not a great idea, because now pioneers are the prey of the Furyans of the rival groups. Additionally, Necromongers are attracted by the presence of Furyans at Crematoria, and they land every now and then on its surface. To make things worse, three hellhounds are still in the caves, and they are hungry...

2 Game rules

This is a game for four players, identified with numbers from 0 to 3. Each player has control over a group of Pioneers helped by some Furyans. The goals of the game are to conquer as many cave cells and to pick as many gems as possible.

This game includes a few other units as well: three Hellhounds, and at most 10 Necromongers. Those units correspond to player -1.

The game lasts 120 rounds, numbered from 0 to 119. Each unit of every player, Hellhounds and Necromongers included, can move at most once per round.

The board has two levels (underground and outside), each with 40 rows and 80 columns. Therefore, a position (i, j, k) is identified with three coordinates such that $0 \leq i < 40$, $0 \leq j < 80$, $k = 0$ for underground cells, and $k = 1$ for outside cells. The upper left cell of the underground is $(0, 0, 0)$. The lower right cell of the outside is $(39, 79, 1)$.

The map of Crematoria is circular with respect to the horizontal dimension: to the right of position $(i, 79, k)$ we have position $(i, 0, k)$.

There are 20 elevators (of type Elevator) scattered around. A cell at position $(i, j, 0)$ is of type Elevator if and only if the cell at position $(i, j, 1)$ is also of type Elevator.

[2] Incidentally, this is a basic rule of most games in Jutge.org, where you capture units by killing them.

The rest of cells of level 1 are of type Outside. At level 0 we have cells of type Cave and Rock. Rock cells are not passable by any unit. The rest of cells of both levels are passable by all units. The Elevator cells of level 0 are surrounded by eight Cave cells.

All units can move horizontally, vertically and diagonally. The set of Cave cells is connected. Additionally, Pioneers and Furyans can go Up (from level 0 to level 1) or Down (from level 1 to level 0) when they are on an Elevator.

Half of the cells at level 1 are under the sun, and the other half at night. The sun moves circularly two columns to the right per round: At round 0, columns 40 to 79 are under the sun; at round 1, columns 0 to 1, and 42 to 79 are under the sun; at round 2, columns 0 to 3, and 44 to 79 are under the sun; etc.

With probability 25% at each round, the energy of the sun generates one new gem on any of the Outside cells of the two columns that just entered the night-half of the surface of Crematoria. If the sun reaches a previously generated gem, that gem gets destroyed.

The sun kills all units under it. Also, any unit that moves onto a cell under the sun dies immediately.

Each cell of the board can have at most one unit on it. When a unit tries to move onto a cell already occupied by another unit, this in fact means an attack. A unit that attacks does not move during that round, even if the attacked unit gets killed.

Pioneers cannot attack at all. Furyans cannot attack units of the same group. (This would be an illegal move and would be ignored.) A Necromonger will never attack another Necromonger. Hellhounds do not attack: they just devour everybody around them at level 0. A Hellhound will never be adjacent to another one.

At the beginning of the game, each group has 15 Pioneers. They are born with 50 points of health. When a Pioneer moves onto a Cave cell, that cell becomes currently conquered by his group. When a Pioneer moves onto an Outside cell with a gem, the Pioneer picks it. As a consequence, the counter of gems accumulated by his group increases by one.

At any moment, let c be the number of cells currently conqured by a group, and let g be the total number of gems already accumulated by the group. Then, the current number of points of this group is $c + 30g$. The group with the most points after all the rounds have been played wins the game.

At the beginning of the game, each group has five Furyans. Furyans can attack, removing between 25 and 50 points of health from the adjacent enemy unit. Furyans are born with 100 points of health. Furyans cannot conquer any cell, and cannot pick any gems.

Let us now describe the units of player -1. They do not pick any gems, nor they conquer any cells. Note that the decisions made by Necromongers and the Hellhounds do not depend on the groups of the nearby Pioneers and Furyans.

Initially, the board has no Necromongers on it. Afterwards, with probability 50% at each round, one spacecraft with one Necromonger will appear at the sky above any Outside cell (with no gem on it) of the two columns that just entered the night-half of the surface of Crematoria. There is one limitation: the total number of Necromongers on the board will never exceed 10. The Necromonger will land after two rounds. If there were any unit on that Outside cell, it would be killed immediately.

Necromongers are born with 75 points of health. His attack reduces the health of one adjacent enemy unit between 20 and 40 points. They totally ignore the units at level 0. After a Necromonger dies, another Necromonger with the same identifier and 75 points of health may arrive to Crematoria later.

A Necromonger acts this way: If he has an adjacent Pioneer or Furyan, the Necromonger attacks him. Otherwise, he approaches the nearest Pioneer or Furyan on the outside. If there are no such units, Necromongers will move to their right to avoid the deadly sunlight for as many rounds as possible.

The three Hellhounds are immortal. Hellhounds always approach the nearest Pioneer or Furyan at level 0, totally ignoring the units at level 1. Any time, all units horizontally, vertically or diagonally adjacent to any of the Hellhounds immediately get killed. Also, any unit that tries to use an elevator to go down to a cell occupied by a Hellhound will immediately die.

Every 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 program 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 group. For instance, if several Pioneers walk in a single file, and there are not other units around interfering, they all can move one step forward with no collisions among them, by ordering movements from the front of the file to its back.

More precisely, suppose that player 0 gives the commands a_1, a_2 and a_3 in this order, player 1 gives the commands b_1, b_2, b_3, b_4 and b_5 in this order, player 2 gives the commands c_1 and c_2 in this order, and player 3 gives the commands d_1, d_2 and d_3 in this order. Then the commands will be executed following this order: First, a random permutation of $\{a_1, b_1, c_1, d_1\}$; afterwards, a random permutation of $\{a_2, b_2, c_2, d_2\}$; then, a random permutation of $\{a_3, b_3, d_3\}$; and finally b_4 and b_5.

As a consequence of the previous rules, consider giving the orders to your units at every round from most urgent to less urgent.

Each movement is applied on the board resulting of the previous movements. For instance, suppose that one Pioneer and five Furyans (let us call them V, W, X, Y and Z), all of group 0 except for Z who is of group 1, try to move in this order on a cell occupied by a Necromonger. First, the Pioneer will not move because the target cell is occupied. Then, V will attack the Necromonger. Assume that the Necromonger gets hurt but not killed. Then, W will also attack the Necromonger. Suppose that the Necromonger gets killed now. W will not move where the Necromonger was. Afterwards, X will move on the cell now empty. Then, Y will try to attack X. This movement will be ignored because X and Y are of the same group. Finally, Z will attack X.

After all the selected movements of the four players have been executed, it is the turn of the units of player -1. If some unit has several movements that are equally attractive, one will be chosen uniformly at random. For instance, if a Necromonger can attack several adjacent Pioneers and Furyans, he will just choose any of them. Similarly, if a Hellhound has several Pioneers and Furyans at the same distance at level 0, it will approach any of them chosen at random (always avoiding the other Hellhounds).

After all the selected movements of a round are played, the killed Pioneers and Furyans are reborn in random Cave cells not too close to other units.

A Pioneer or Furyan killed by another group will be "captured", so the reborn unit will belong to the attacking group. A Pioneer or Furyan that has been killed by player -1 or by the sun will be assigned to a randomly chosen different group.

At the very end of each round, the units still alive will recover five points of health, never exceeding the maximum of his type of unit.

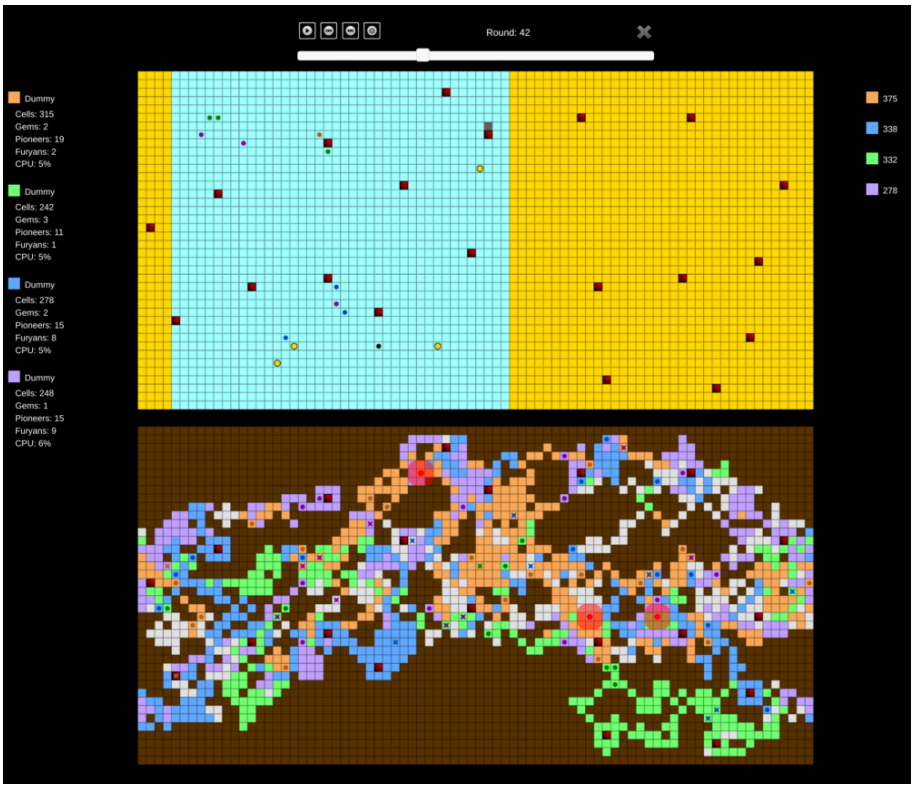
As a result of all the above rules, the total number of Pioneers and Furyans remain constant during the whole game: 60 and 20, respectively.

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. Moreover, an `api.pdf` file presents the main classes and methods that you may need to program your player. A "Null" player and a "Demo" player are also provided.

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 Linux, run:

```
cp AIDummy.o.Linux AIDummy.o
```

```
cp Board.o.Linux Board.o
```

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 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 `Riddick`, copy `AINull.cc` (an empty player that is provided as a template) to a new file `AIRiddick.cc`. Then, edit the new file and change the

```
#define PLAYER_NAME Null
```

line to

```
#define PLAYER_NAME Riddick
```

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 to access the state of the game. Those functions are made available to your code using inheritance, but do not tell your Software Engineering teachers because they might not like it. The documentation about the available functions can be found in an additional file.

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 AIRiddick.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.
- If your program needs (pseudo) random numbers, you must use two methods provided by the game: `random(l, u)`, which returns a random integer in `[l..u]`, and (less frequently) `random_permutation(n)`, which returns a `vector<int>` with a random permutation of `[0..n-1]`.
- Note that the valid directions are `Bottom`, `BR`, `Right`, `RT`, `Top`, `TL`, `Left`, `LB`, `Up`, `Down` and `None`, corresponding to integers from 0 to 10. The circular definition between 0 and 7 can be used to simplify the implementation of your player. See the `Demo` player for some examples.
- 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.

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 we will include in the comparison all programs from previous competitions!). 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.

- Read the headers of the classes that you are going to use. Do not worry about the private parts or the implementation.
- Start with simple strategies, easy to code and debug, since this is exactly what you will need at the beginning.
- Define simple (but useful) auxiliary methods, and make sure that they work properly.
- Before you strat competing with your classmates, focus on defeating the "Dummy".
- Keep a copy of the old versions of your player. Make it fight against its previous versions to measure the improvements.
- As always, compile and test your code often. It is much easier to trace a bug when you have only changed a few lines of code.
- Use cerr to output debug information, and add `assert`s to make sure that your code is doing what it should do. Remember to remove them before uploading your code, to avoid making it slower.
- When debugging a player, remove the cerrs that you may have in others players' code, so as to only see the messages that you want.
- If using cerr is not enough to debug some of your code, learn how to use `valgrind`, `gdb` or any other debugging tool.
- Make sure that your program is fast enough. The CPU time that you are allowed to use is rather short.
- Try to figure out the strategies of other players watching several games. This way, you can try to react against them, or even imitate or improve them in your own code.
- You can, however, share the compiled `.o` files.
- You can submit new versions of your program at any time.
- Do not wait until the last minute to submit your player. When there are lots of submissions at the same time, it takes longer for the server to run the games, and it could be too late!
- Remember: Keep your code simple, build often, test often. Or regret.