Development of an on-line platform for AI programming challenges

| Autor:       | Albert Vaca Cintora       |
| Titulació:  | Enginyeria Informàtica    |
| Director:   | Jordi Petit Silvestre     |
| Departament:| Llenguatges i Sistemes Informàtics (LSI) |
| Centre:     | Facultat d’Informàtica de Barcelona (FIB) |
| Universitat:| Universitat Politècnica de Catalunya (UPC) |
| Data:       | 22 de Desembre, 2013      |
## Contents

1 Introduction ........................................ 6
   1.1 Short history .................................. 7
   1.2 State of the art ............................... 8
   1.3 Goals .......................................... 8
   1.4 Milestones .................................... 10
   1.5 Summary ....................................... 10

2 Understanding the Jutge.org infrastructure ....... 12
   2.1 The correction server .......................... 12
   2.2 The drivers .................................... 14
   2.3 The problems ................................... 15
   2.4 Games as problems ............................. 15

3 Understanding the games code by example .......... 18
   3.1 Code that is independent of the game ........ 18
       3.1.1 SecGame and Game ....................... 18
       3.1.2 In-detail explanation about SecGame .... 19
       3.1.3 Player and Registry .................... 20
       3.1.4 Auxiliary classes and functions ......... 21
   3.2 Code dependent of the game: Implementing Battle Royale .... 21
       3.2.1 Board class .............................. 21
       3.2.2 Action class .............................. 22
       3.2.3 Creating the graphical viewer (HTML) .... 22
       3.2.4 Creating a reference player (C++) ....... 23
   3.3 Problems found .................................. 24
4 Adding support to run our game to Jutge.org

4.1 Overview
4.2 Adding game problems
   4.2.1 Updating the upload-problem script
   4.2.2 Creating a driver for our game
   4.2.3 Testing with Battle Royale
   4.2.4 Changes to the results view
4.3 Implementing the competitions
   4.3.1 Updating the driver to allow matches between players
   4.3.2 Testing the new driver mode
   4.3.3 Defining the competitions use cases
   4.3.4 Changes to the database
   4.3.5 Adding the user interface
   4.3.6 Creating a public web-service API
4.4 Problems found and technical decisions
   4.4.1 Generalizing database to any number of users
   4.4.2 Race condition when running multiple matches at once
   4.4.3 Ownership of the competition matches
   4.4.4 Group players when preparing matches
   4.4.5 Submission IDs could not go over 999
4.5 Testing the competitions

5 Developing a brand new 3D game

5.1 Defining our game
   5.1.1 Requirements
   5.1.2 Technology
   5.1.3 Gameplay
Chapter 1

Introduction

Online teaching has experienced a significant growth in the recent years, with universities as the Stanford University\(^1\) or the MIT\(^2\) offering their own platforms for online education in the context of computer programming. The UPC has always been pioneer providing tools of this kind, like the self-assessment website Jutge.org \(^3\) launched by Jordi Petit and Salvador Roura, or the video-recorded lessons released by Guillem Godoy \(^4\). Within the current project, we want to provide our university and its students a new tool to improve their coding skills.

The main idea behind this project is to create an extension to the Jutge.org site where students will code strategies for videogames. Those strategies will play against each other in a competitive fashion, being ranked in a public leaderboard. Games can range from well known classical games like chess, checkers or even pacman to our own invented games.

It’s important to emphasize that students will not be able to directly ‘play’ the games, but instead they will have to write programs smart enough to play by themselves. The strategies they write should be able to react to any event that happen in the game without any human intervention.

In this project we also aim to create a coupe of games for the platform and document our steps to make it easier for new students and teachers to contribute to the project creating their own.

\(^1\)http://online.stanford.edu/courses - Stanford University online courses.
\(^2\)https://www.edx.org/ - MIT online courses.
\(^3\)https://www.jutge.org - UPC’s Jutge.org.
\(^4\)http://www.lsi.upc.edu/~ggodoy/tc.html - Guillem Godoy’s online lessons.
CHAPTER 1. INTRODUCTION

1.1 Short history

Our university has experience in the field of online teaching, organizing programming challenges and particularly game programming challenges, which are the focus of this project. In this kind of challenges the contestants create a program to play a given game, that will compete against other contestants’ programs.

The year 2007, the former teacher Omer Giménez introduced a game programming competition for the first time in a subject for the degree in Mathematics. Instead of using a classical game like chess or go, he invented the rules and created a game called Dominator. To do so, he used a modified version of an online judge that was already being used for some subjects, as an interface for students to submit the programs, run the games and display the leaderboard.

Since the year 2010, these competitions are used in subjects for both the Mathematics and Computer Science degrees. They are currently maintained by the teacher Jordi Petit, since Omer Giménez left the university. Even though, while the online judge has received major improvements (and is now called Jutge.org), the website for the games has experienced no changes since Omer’s version.

The year 2011 there was an improvement in the games code, aimed to separate the backend that runs the game and the viewer that displays it. This way, the viewer can be written in HTML, allowing to display the games directly in the web browser. This change also simplified the code and made it easier for other people to create their own games, as the teacher Enric Rodríguez did the year 2012 with the game ”Bola de Drac”\(^5\). It also motivated some students to create their own alternative viewers for the games using HTML, introducing jokes and references to the subjects and their teachers\(^6\).

This project has the main goal of pushing the game programming challenges further away, doing a major rewrite of the web interface, integrating it with UPC’s Jutge.org site, and implement some new games. Also, since Jutge.org is open not only to students but to anybody connected to the Internet, we want to allow the game competitions to be open for everybody too.

---

\(^5\)Bola de Drac' competition: https://boladedrac.jutge.org/

\(^6\)An alternative viewer made by the student Otger Roglà: https://github.com/OtgerR/eda-viewer
1.2 State of the art

Apart from our university’s game challenge, there exist several other computer gaming competitions at the time of this writing. Some well known examples of these are:

- The Annual Computer Poker Competition\(^7\), celebrated each year since 2006.
- The Ultimate Computer Chess Challenge, with a prize of $100.000 the year 2007.
- The Google AI Challenge\(^8\), which features their own custom games: In the last edition, the players commanded a colony of ants.

From these, the first two competitions target well-known games, where there is already a lot of research done in the field of artificial intelligence to find the best strategies. We want our games to be new and unique to enforce students to think their own strategies, so in those terms our competitions will better resemble the Google’s AI Challenge.

In our country, the Universitat de Lleida held a Robocode (an AI-commanded robot fighting game) competition the years 2010 and 2011, but is currently discontinued\(^9\).

1.3 Goals

The goals for this project are:

1. **Integrate the web interface with Jutge.org**

   The current web interface was coded for the first version of the competition, and has had no major changes over the years. With this particular goal, we aim to make a big improvement in code maintainability and usability for both students and teachers, integrating the programming challenges in the UPC’s Jutge.org website. The new web interface will fulfill the following requirements:

   - It will allow instructors to organize new competitions for the students of a course and manage them, decide the start and end dates and when the game rounds are played.

---

\(^7\) Computer Poker Competition: [http://www.computerpokercompetition.org/](http://www.computerpokercompetition.org/)

\(^8\) Google’s AI Challenge: [http://aichallenge.org/](http://aichallenge.org/)

\(^9\) Universitat de Lleida’s Robocode competition: [http://robocode.udl.cat/](http://robocode.udl.cat/)
CHAPTER 1. INTRODUCTION

- It will allow students to enter the competition submitting their players for the games, follow the competition results in real time and watch the matches played.

- It will provide a web API for external applications to access information from the competitions, and for instructors to create their own management scripts.

Jutge.org is a platform open to anybody on the Internet (and not only to UPC students), so it was a natural step to allow the same for our competitions. Doing so, our university will provide an even richer and open-to-anybody platform for online education.

2. Port some of the existing games

We want to release the competitions platform with some games already available. To do so, we will rewrite or adapt some of the existing games to be used in the new environment. These old games include the original first game written by Omer Giménez and others developed more recently.

3. Write a new 3D game

In addition to the existing ones, we will develop a brand new game with some innovative aspects over the previous ones. The most important are:

- It will have a 3D representation: Personally, this will allow me to learn one of the latest standards in web development, WebGL, while creating a nice-looking game.

- It will not be board-based: Since the game will be in 3D, the approach of a grid used in all the existing games will not be needed anymore. Instead of this, the game board will be accessed as a graph, being also a nice exercise for students when they write their AI players for the game.

4. Produce documentation

There is little documentation regarding the games and the Jutge.org website, making it difficult for a newcomer to help in its development. This document will serve as a reference to develop new games and will explain the work behind the scenes and knowledge needed to make the game competitions possible. Then, this goal aims to make it easier for new people to join the project and keep improving it.
1.4 Milestones

During the development of this project, we have to meet some delivery dates. These are:

1st April, 2013 Finish a first game, that will be used in the subject of EDA. Due to the early deadline, this first game will be an adaptation of an existing one.

1st October, 2013 Finish a second game, that will be used for both EDA and ALG. This time we will release the new 3D game we defined in the goals.

1st December, 2013 Release the final, stable version of the website. At this point, we should be able to run game competitions in the new platform without issues.

1.5 Summary

This document is organized in the following parts:

Chapter 1. Introduction This chapter. It explains where we are and where we go.

Chapter 2. Understanding the Jutge.org infrastructure Documents the server-side code we found before starting this project. Details how the submitted programs and games are transferred to the correction server, compiled and executed securely.

Chapter 3. Understanding the games code by example As the previous chapter, it also explains the legacy code we started with. This chapter details the structure of a game and does so in a practical way: explaining how Battle Royale, a very simple game, was written. This game was used in the subject of EDA the last year’s (2013) spring course.

Chapter 4. Adding support to run our game to Jutge.org Explains the changes made to the Jutge.org website and the correction server to run the game competitions. This new web interface, integrated in the Jutge.org, deprecates the old website used until now. It allows teachers to create their own competitions from a web interface, and students to join them, submit their programs and see the results in real time.

Chapter 5. Developing a brand new 3D game This chapter explains how Tron 3D, a new game featuring a full-fledged 3D viewer, was created. The game is a 3D implementation of the classical Tron or Snake game. It was used in last year’s (2013) autumn course in the subjects of EDA and ALG with good reception by the students.
Chapter 6. Planning and economic valuation Describes how the project schedule developed and does a valuation of the economical costs of the project.

Chapter 7. Conclusions and future of the project This chapter sums up the work done and proposes new ideas for this project to grow and keep going forward.

Appendices Attached are the statements of the games Battle Royale and Tron 3D, as they were given to the students that took part in the last year (2013) competitions for the subjects of EDA and ALG.
Chapter 2

Understanding the Jutge.org infrastructure

Since we are working on a preexistent project, we first had to analyze and understand the code developed before us. This chapter explains the server-side code, shared by both the old games website and the Jutge.org.

When a user submits a program to Jutge.org to be corrected, it goes through an automated process that checks it and returns the result to the user. Since they both have common roots, game matches are executed by the same backend in a very similar way, even though the web interface is different.

The underlying infrastructure is barely documented: the only written reference is a previous project by the former FIB student Enric Cusell\(^1\). Then, our first mission was to ask about and reverse-engineer how the Jutge.org stores and corrects the programming problems and executes the games. Let’s see what we found, focusing first on regular problems.

2.1 The correction server

The correction server has a number of virtual machines that execute and check the programs, keeping them in a restricted and safe environment. Then, the first step when a user submits a program, is to queue it to be sent to a free virtual machine, and wait for the virtual machine to send a callback to the web server when the correction result is ready.

---

\(^1\)http://upcommons.upc.edu/pfc/handle/2099.1/18641 - Enric Cusell’s PFC
CHAPTER 2. UNDERSTANDING THE JUTGE.ORG INFRASTRUCTURE

Figure 2.1: Screenshot of a program in the queue to be corrected at Jutge.org

Of course, each virtual machine doesn’t have a copy of each problem, so for each problem to correct they have to receive not only the code from the user, but also a correction script (called the "driver") and any other files needed to correct that particular problem (the "problem" files). It is in the main server where all these files are stored.

Figure 2.2: Diagram of the correction process

sub: user submission, pbm: problem code, drv: problem driver, cor: correction output
CHAPTER 2. UNDERSTANDING THE JUTGE.ORG INFRASTRUCTURE

This communication is achieved sending tarballs through SSH connections, each one containing descriptive YAML files. As an example, the next figure shows the content of a submission file:

```
−−−
problem_id: P17652_ca
compiler_id: G++
title: S00084304
description: S003
author: U00000
email: info@jutge.org
priority: 2
callback: https://S0084304:ea6e837e08fcdd212@jutge.org/callback
public_only:
exam_id:
```

Figure 2.3: Sample submission.yml file contents

2.2 The drivers

Each kind of problem is corrected in a different way: the programming problems are compiled, executed with a given input and then the output is compared with a reference; the logic circuit problems are checked for formal validity; and so on.

To do so, there are different correction scripts, called drivers, and each problem has a driver associated. Drivers are just a small python scripts that will run inside the virtual machine with the following input:

- A problem directory - where the original problem files are stored.
- A submission directory - where the user submitted files are stored.
- A correction directory - empty, where the results of the correction have to be stored.

All these and the driver itself will be erased after the execution ends, and only the correction directory will be sent back to the correction server. That directory should contain any files relevant to the correction (for future reference) and a correction.yml file with a particular format that will be parsed by the web server to include the results in the database.
To ease the task of sending a driver to the correction server, I wrote a script called upload-driver.sh. This will be useful later on to upload and test the driver for our games.

2.3 The problems

The problems are directories that are uploaded to the correction server by using a script called update-problem.php. Even though the content of a problem directory can vary (depending on what the driver that will correct it expects to find), there are some required files that this script will need:

- **problem.en.tex** - the problem documentation shown to the users. There can be a file for each language (en, es, ca...).
- **problem.en.yml** - the problem description, that will be added to the web server database. It allows several translations too.
- **handler.yml** - defines which driver has to be used to correct this problem.

2.4 Games as problems

Even though the games have their own web interface, they are still a kind of problem and go through the same correction process, using specific drivers. However, they are stored in a different (files-based, non-relational) database, so they can not use the update-problem.php script. This is due to historical reasons and is something we will fix in following chapters.

The main differences between a game problem and a regular problem are the following:

- The users don’t submit an entire program. They upload only the player class, while all the other code is common for everybody and is contained in the problem files.
- The driver evaluates the programs by running several matches against a reference player, that the program submitted by the user has to defeat.
- The driver also has a ‘competition’ execution mode, used when there is a competition to run matches between several players and get the scores.
- The output file of the execution of the games has to be stored to open it later with the game viewer, to display the replay of the match.
Also, the games website has some game-specific features, like a leaderboard, that we will need to mimic when we implement the games competitions in the Jutge.org website.

Figure 2.4: Screenshot of the games judge (in catalan)
Chapter 3

Understanding the games code by example

As we have seen in the previous chapter, games are like programming problems. The users can submit their code as in a regular problem, it is then executed by the server and they can view the results using the game viewer. In this chapter we will to explain in detail the different parts of the games code, while implementing a simple game called *Battle Royale*: a re-incarnation of the original *Dominator* that Omer coded for the first game challenge the year 2007.

3.1 Code that is independent of the game

The games code has to handle a lot of boilerplate tasks: instantiate the players, communicate the actions between them, handle the secure execution environment... and this code can be re-used with independence of the particular game implemented. In this section, we will explain all the code that is common for every game. All this code is written in C++.

3.1.1 SecGame and Game

These two classes are the entry point of the game and handle its execution. SecGame is used when the game runs in the server, providing a safe execution environment, while Game is distributed to the users to test their players on their computers.

They both receive as arguments a list of players to instantiate, and a configuration file used to initialize the game constants and the board. After doing this, the game loop starts, which has the following steps:
Chapter 3. Understanding the Games Code by Example

1. Print the board state to the output file (for the viewer to read it).

2. Give each player a copy of the board.

3. Execute the players code on those boards (so the second player can’t still see the actions performed by the first one).

4. Receive the actions done by the players.

5. Shuffle the actions and apply them in random order to the original board.

6. Print the actions done to the output file (used by the viewer too).

7. Start over with the resulting board.

This loop in the case of SecGame is slightly more complicated, since each player runs in a separate process (with its own copy of every class) using pipes to receive the actions from the other players.

3.1.2 In-detail explanation about SecGame

When executing SecGame there is a master process that spawns one extra process for each player and connects to their standard input and output with pipes. The only process that can modify the board trustworthy is the master process. To enforce this, the following loop is ran each round:

1. The master process sends the current board to each child process.

2. The child processes create the board from scratch reading the input.

3. The child processes execute their players.

4. The child processes write the actions so the master process can read them.

5. The master process collects the actions from each child process and shuffles them.

6. The master process applies the actions to the board. They won’t be applied if they were not valid.

7. The master process writes the board and actions as required by the viewer.

8. Start over with the resulting board from the master process.
Also, SecGame uses the Linux system call `setrlimit` to limit the CPU time, stack size, number of threads, number of system calls, etc. that the players can use, and kills them if they exceed it. When this happens, the game execution continues but that player units won’t move anymore.

### 3.1.3 Player and Registry

Player is the class that the users will extend to implement their own strategies. It gives access to the Board and the Actions class that they will use to read the state of the game and send commands to their units, respectively. Since the lack of reflection in C++, the Game/SecGame class can not ask for all the subclasses of Player to instantiate them, so a Registry class to keep track of them is used instead. In runtime, each subclass of Player adds itself by name to the Registry, that implements the Factory pattern to instantiate the player objects.

![Game class diagram](image)

**Figure 3.1: Game class diagram**

*Note: Player relation with Board and Action is actually implemented using inheritance.*
3.1.4 Auxiliary classes and functions

Some other functions and small classes are provided too, like a Position class to store coordinates in the board, or a Direction enum with the 4 cardinal directions.

3.2 Code dependent of the game: Implementing Battle Royale

To get used with the existing architecture, I tried to implement a new game based on the original idea that Omer coded for the first game challenge, a game originally called Dominator and later renamed to Battle Royale. In this game, players command an army of knights and farmers. Their goal is to ‘farm’ more tiles of the board than the other players with the farmers, and can use the knights to attack and kill the opponent units.

This game that we implement here, was used in a competition for the students of EDA in last year’s (2013) spring course. The full rules and documentation can be found in the Appendices as they were given to the students.

Let’s go through all the code that had to be changed to implement the game logic. These are the only classes that will need to be modified in to implement new games, in most cases.

3.2.1 Board class

This class stores the current the game state, and provides the read-only functions that the players will use to access it. In our game, the state is represented by a 2D matrix of cells and an array of units.

The public functions in this class will be available to the users when coding their players to access (but not modify) to current board state. In our game, we provide very simple const functions like cell(inti, intj) or unit(intunit_id) that return the corresponding structure containing the cell or unit information.

This class also has to implement some functions to print information for the viewer and to send/receive the state when using SecGame: A print_preamble(ostream) function that prints the parameters of the game (in our game: number of players, units per player, number of rounds....), a print(ostream) function that prints all the content of the board (in our
game, the board matrix plus a list of units) and a constructor Board(istream) that can load
a game state given the output of the two previous functions.

Finally, this class have to implement a move(player, movement) function that applies
a movement to the board. This function can not be called by the players, and it is meant to
be used by Game and SecGame only, with the movements that the players create using the
Action class.

3.2.2 Action class

This class stores the actions that a player performs to the board. These actions are not
executed immediately, instead, they are stored, shuffled (so no execution order should be
assumed by the players) and executed at the end of the round, before the next round starts.
In our game, it contains a single function command(int unit_id, Direction).

For SecGame to work it also needs, like the Board class, a print(ostream) function and
a constructor Action(istream). The former prints (and erases) all the actions contained,
and the later does the opposite, reading actions from a text input.

3.2.3 Creating the graphical viewer (HTML)

The game viewers allow the users to easily see what their program is doing and think how to
improve their strategy. They are implemented as independent programs that take the output
from the actual game and represent it in a visual way. Due to this independence we can also
use a different language than C++, so we use HTML+Javascript to make it cross-platform
and allow the game matches to be displayed directly from the website.

The viewer input is a file that contains the output of the Board’s print_preamble fol-
lowed by successive iterations of Action’s print() of all the players and the resulting Board’s
print() for each round. When the viewer first loads the file, it preprocesses and stores all
the information for each round in a JSON structure, so it can easily access it later.

From the information of the current round, the viewer draws the scores and the board in a
graphical way, by using an HTML canvas. After some milliseconds, it automatically advances
to the next round giving the effect of an animation to the board. Some HTML controls were
also added to pause the animation or seek to a specific round.
3.2.4 Creating a reference player (C++)

The users are only allowed to enter the contest once their players are strong enough to defeat a simple "dummy" player, which code remains private to them. For Battle Royale, this dummy player implemented a simple strategy where the farmers search the closest non-farmed tile and the knights attack the closest unit in a fixed range. When there is no target in that range, they just wander randomly. This player was implemented by Albert Asterisas and Enric Rodriguez, teachers of the subject EDA where the competition of this game was played.

1 http://media.fib.upc.edu/fibtv/streamingmedia/view/2/763 - Battle Royal finals
3.3 Problems found

During and after the development, we found some issues that worth mention:

3.3.1 Using random values when initializing the board

After coding all the game classes described above, we noticed that the game crashed when executed in the one-process-per-player mode (SecGame). We found that the cause of these crashes was on the board initialization: since the units are placed randomly by the Board class at the beginning of the game, each process had a different disposition in its own board!

To fix it we had two choices: send the board pre-initialized to the players processes, or avoid using random at all (thus giving fixed starting positions). In this case we used the former because we wanted to keep the spawn position random, but both would have been valid.

A less correct solution would have been to use a predefined random seed, shared between all the processes. This solution would make the game not parallelizable because the execution order of the different threads could differ, causing the calls to the random function to happen in a different order in each process.

3.3.2 Not checking the input stream when reading the actions (Action class)

After publishing the game, some programs produced unexpected errors when were executed using SecGame. Checking the logs, we were able to spot the cause of the problem.

Since the the memory and CPU time available are limited with the system call setrlimit when executing programs using SecGame, this can cause the execution to stop at any time. The problem was produced because the pipes used to communicate with a particular program can be disconnected at any moment, producing a crash if SecGame tries to write or read from them.

To prevent this problem, it is important to have checks for each read operation in the Action class because this class receives the actions outputted by the player code, that can be terminated at any time.
3.4 Technical Decisions

Several aspects of the game could be implemented in a number of different ways, this sections explains the reasons behind the most important choices made during the development.

3.4.1 To use HTML5

Since the viewer has to be multi-platform, it can not be written it in a language that would need platform-specific code or depend on external libraries. Making it usable from a web browser not only achieves this but also enables the integration with the web interface.

Nowadays there are lots of programming languages capable to run in a browser: Adobe Flash, Adobe AIR, Microsoft Silverlight, Unity 3D, Native Client, Java... but all them require proprietary plugins. So I decided to use the new web standard HTML5 that is available on all the modern browsers, even though it won’t run in outdated (but somehow still used) browsers like Internet Explorer 8.

The following HTML5 APIs were used:

- Canvas (http://www.w3.org/TR/html-markup/canvas): This is the main HTML5 feature used, needed to show the board and units in a graphical way.

- File API (http://www.w3.org/TR/FileAPI/): It is used to load a game file from the disk directly into the Javascript, without going through the server.

- RequestAnimationFrame (http://www.w3.org/TR/animation-timing/): This API is not required but, if present, it is used instead of setTimeout or setInterval to draw the board at 60 frames per second because it provides some improvements over it.

3.4.2 Game file format

Since the game and the viewer are different programs, we need a common file format to store the game information and read it from the viewer. These files are also saved in the server for each match, to be able to watch the replay from the web interface.

The file format consists on a header with the game constants (number of players, number of units per player, number of rounds, board used...) and then, for each round, a description of what happens in the game. To represent this second part, two approaches were taken into consideration:
• Store only the changes made to the board: This minimizes the size of the files stored on the server (and transferred to the players that want to watch a replay), but makes the viewer code more complex because it has to recreate each round.

• Store the whole board for every round: This makes the viewer code simpler but has the counterpart of a greater file size.

Since the viewer code is written in Javascript, and could be too slow if it gets complex, I decided to use the second option. Moreover, most modern web servers and browsers implement on-the-fly compression that can reduce the transfer size transparently when these files are downloaded to the viewer.

3.4.3 The build system

The task of building and distributing the games should be easy for both the teachers and the students, and automatable to ease the integration in the online environment.

The first implementation that Omer wrote used qmake as a build system. This made the compilation completely automated, but required this build system (part of Qt) to be installed. Following implementations relied only on make, a more spread build system, but required the users to manually edit the Makefile in order to add their players.

To improve this, some wildcard macros were added to the Makefile, to remove the need of manual edition with the benefits of both systems:

\[
\begin{align*}
\text{PLAYERS\_SRC} &= $(wildcard \text{ AI*.cc}) \\
\text{PLAYERS\_OBJ} &= $(	ext{patsubst } \% , \% . o , \$(\text{PLAYERS\_SRC}))
\end{align*}
\]

Figure 3.3: Regular expression to find players, matching AI*.cc

Some other improvements were made, like a configuration section that looks like this:

\[
\begin{align*}
\text{OPTIMIZE} &= 2 \# \text{Optimization level (0 to 3)} \\
\text{DEBUG} &= 0 \# \text{Compile for debugging (0 or 1)} \\
\text{PROFILE} &= 0 \# \text{Compile for profile (0 or 1)}
\end{align*}
\]

Figure 3.4: Makefile configuration
CHAPTER 3. UNDERSTANDING THE GAMES CODE BY EXAMPLE

This sets up the appropriate flags with some conditionals later on:

```bash
ifeq ($(strip $(PROFILE)),1)
    PROFILEFLAGS=-pg
endif
ifeq ($(strip $(DEBUG)),1)
    DEBUGFLAGS=-DDEBUG -g -rdynamic
endif
```

Figure 3.5: Usage of the profile and debug configuration values

An interesting fact is that in a Makefile the execution order is not linear but dependency-based, and this applies to the variables too: You can use in the first line of your Makefile a variable that is defined in a posterior sentence, and it will have the value that the posterior sentence sets. A limitation imposed by this behaviour is that the content of a variable cannot be modified or re-assigned.
Chapter 4

Adding support to run our game to Jutge.org

The website of the games used the sources of an old, modified copy of the judge website. This duplicity was harder to maintain and as a consequence the games website was receiving little improvements. Hence, one of the main goals of this project was to include the games competitions in the newer Jutge.org platform, having a single version of the code.

4.1 Overview

In this chapter we explain the changes needed to achieve this integration. Namely they are, in this same order:

- **Additions to the backend** to be able to upload game problems to the website and the correction server and execute them with a new driver, according to the architecture explained in chapter 2.

- **Changes to the database**, to store the new information required for the competitions.

- **New views in the web interface**, for instructors to create and manage competitions and for students to join them and see the results, together with a test interface for administrators.

- **A public web API** to allow external applications to access information about the competitions, and a **private web API for instructors** to create command-line applications to manage them.
CHAPTER 4. ADDING SUPPORT TO RUN OUR GAME TO JUTGE.ORG

4.2 Adding game problems

Since the correction backend was the same for both the games websites and the Jutge.org, I thought it would be easy to start by adding the games as regular problems to the Jutge.org.

This section explains the changes needed to allow the users to access games as a new kind of problem, submit their code, get it corrected by the server and see the results.

4.2.1 Updating the upload-problem script

The upload-problem.php script uploads the problem files to the correction server and adds the needed entries to the database. For this to work for games too, we had to do some changes to this script and define a unified directory hierarchy for all the games, organized as following:

- ./Runner: Contains the game sources with a Makefile that outputs a Game and SecGame executables. Also, it must contain at least a default.cnf game input file.
- ./Obj: Contains .o files of the dummy player for different architectures.
- ./Viewer: Contains the HTML viewer and a sample.out game file.
- ./Doc: Contains the files needed to generate the documentation.

From those directories, the modified update-problem.php script will extract the following information:

- **The problem statement**: Will be copied from ./Doc/main.pdf. If this directory contains a Makefile (e.g. to call a LaTeX to PDF compiler) it will be executed first. The text will also be converted to PS and HTML formats.
- **The material file for users to download**: It is a compilation of the files from ./Runner, ./Viewer and ./Obj. All the player files but AIDemo.cc and all the files related to SecGame from ./Runner are not included, since users should not have them.
- **The problem files** that the virtual machines use to compile the players’ code. Will be a copy of ./Runner.
- **The viewer**: It will be copied to a publicly accessible directory in the web server.
CHAPTER 4. ADDING SUPPORT TO RUN OUR GAME TO JUTGE.ORG

All those but the viewer are resources already used in regular problems, so they only have
to be copied to some existent directories, already in the logic of update-problem.php, for
the website to find them.

For the viewer, we made the script copy it to judgeit/www/gameviewer/game
id, since
anything under www is publicly accessible without adding routes to the site configuration. This
way we can link to it from the web interface with only having to know the game ID, as we
will see later.

4.2.2 Creating a driver for our game

As we explained in the second chapter, the drivers are small Python scripts that take the user
code and a given problem and output a correction. In the previous implementation, there were
specific drivers for each game. Now, we can take advantage of the unified directory structure
we defined to implement a single driver for all games. We called this driver the "game" driver.

At this point we only want our driver to return a verdict (whether the submitted code is
accepted or not) and the executed match scores, so our driver will be very simple:

1. Compile the game source with the player code, return a 'Compilation Error' if it fails.

2. Start the SecGame executable using as input the file default.cnf (both must exist
according to the structure we defined above) and 4 instances of the user’s player. Return
an ‘Execution Error’ if it crashes.

3. Launch the game 4 more times with the user’s player and 3 predefined ”AIDummy”
players. Store the games output. Return a 'Wrong Answer' if the user’s player does not
get the high score in each game.

4. Otherwise return an 'Accepted'.

We are using the same output answers as regular problems because this way the website
will display them properly without changes.

4.2.3 Testing with Battle Royale

After uploading the Battle Royale problem files with update-problem.php and the 'game'
driver with upload-driver.sh (and after some debugging), we could go to Jutge.org, access
our game and even submit a player and get a correction. The following is a screenshot of the Battle Royale statement inside the Jutge.org website:

![Figure 4.1: Statement page for Battle Royale](image)

The **Problem files** section contains links to download the game material and the statement we generated with `update-problem.php`. The same statement is also automatically converted to HTML to be displayed below, so all the content in this page is generated automatically when uploading a game.

Note that we also added a link to a **Sample game** that launches the viewer with the file `sample.out` as input, using the viewer from the location we prepared before.

### 4.2.4 Changes to the results view

Since the correction files contain the game results and output, it was easy to add them to the results interface, too. The correction YAML file was already read by this page to display the results.
CHAPTER 4. ADDING SUPPORT TO RUN OUR GAME TO JUTGE.ORG

From this page, the game output for the several matches can be downloaded or launched in the viewer using the buttons at the left of the scores and CPU usage. Again, this uses the game viewer we stored with update-problem.php.

4.3 Implementing the competitions

The old website allowed to run competitions were all the students of a course competed in successive matches, progressively taking the worst players out of until having a champion. This section explains the changes needed to introduce this functionality to our Jutge.org.

4.3.1 Updating the driver to allow matches between players

The competition matches encounter 4 different players, so the first thing we had to do is to change the driver to add this new execution mode. We achieved this by adding an optional field in the submission YAML file that, in the case of a competition match, will contain the names of the players in the match. I decided to allow an input array of any number of players, so in a future we could allow, for example, games that require 1vs1 matches or NvsN.
CHAPTER 4. ADDING SUPPORT TO RUN OUR GAME TO JUTGE.ORG

Also, instead of a single source file, the submission will now include a copy of the source of each player. Since the submission is sent to the correction server in a tarball file, we could make this change easily.

In this mode, the game configuration file will be chosen at random from all the *.cnf files in the game directory. The output file and the scores will be sent back with the correction. Unless there is an internal error, the correction will always be an 'Accepted' because all the player sources in a competition match have been already submitted and accepted before.

Last, the driver also has to support adding Dummy players for those cases when the number of players available to play a match is less than the required.

4.3.2 Testing the new driver mode

To test the changes made to the driver, we added a web interface only accessible from an administrator account to the Jutge.org site. From this interface, we can select 4 players from all the accepted players for a game and make them play. This allowed us to begin testing the code as early as possible.

![Figure 4.3: Screenshot of the competitions test interface](image-url)
4.3.3 Defining the competitions use cases

Until now, to create a new competition it was necessary to access the server through SSH and manually edit several files. However, in the Jutge.org there are instructor accounts that will be able to create competitions from within a graphic interface, as they can already do with courses, problems, documents and other elements. In this section we define this interaction from the instructor and the student point of view through this new interface.

Instructors will be able to add competitions to the courses they own, so all the students enrolled in the course can enter them. To create a competition, they will have to choose a game (from a list of all the available games) and a course, add a title and a description, and set a start date.

Once a competition is created, students can see it inside the course. They can then start submitting players to beat the dummy player of that game, until the start date of the competition. That date, all the users that submitted a player capable to beat the dummy reference player will enter the competition and no more players will be accepted. If a user has more than one submission that defeated the dummy player, he or she will be able to choose which one wants to use in that competition.

Competitions are held in several rounds. When a round starts, players are put into groups of 4 that will play together in a match (dummy players are added as needed to fill the gaps). All the matches for all the groups of players are submitted to the correction server to be executed using the new 'competition' mode of the driver. Once a match ends, the two best players are marked as survivors and advance to the next round. When all the matches are done, all the non-survivors are re-matched again. The same process takes place repeatedly until there is only one non-survivor. That player is marked as out of the competition and a next round will start with all the other players, removing one each round.

If a student wants to keep improving his player after the competition has started, he or she will still be able to make new submissions. Students can then chose the player they wants to use from the competition interface, but the change will not take effect until the next round.

The instructor can, from his private interface, decide when to start new rounds. They can also finish the competition. When a competition finishes, all the survivors are marked as finalists and no more rounds can be played.
4.3.4 Changes to the database

To implement this logic, it was needed to add new tables to the database schema.

The Participant table contains the information about a user in a competition: if he is alive or not, the last round he played and the program that he chose to play (in case he had more than one accepted submission for the same game).

The Competition table stores the competition information described in the previous section, and the state of the competition: whether it has started/finished, the current round number and some other fields needed to store the state.

Lastly, the Match table contains the round and turn in which this match was created, the players in it and, if it has already finished, the scores of those players.
4.3.5 Adding the user interface

This section contains images of the most important view that were added for the competitions.

The students interface

Figure 4.5 shows the competition main page, as accessed from within a course. It contains all the information about the competition that is available to the students: A list with all the played rounds and a list of participants that also details if they are still playing or have been beaten out. From this view, the students can also choose the program they want to use in this competition, from a list of all the accepted competitions. The links in the list of rounds take to the next view, a detail of the matches in that round.

Figure 4.5: Screenshot of the students’ competition view
CHAPTER 4. ADDING SUPPORT TO RUN OUR GAME TO JUTGE.ORG

Figure 4.6 shows the detailed round view. It displays all the matches played and the results, including buttons to download or watch the replay of a match. Once a round ends, it also displays a button to advance to the next round. This view uses AJAX to display the results in real time, without having to reload the web page. Also, we decided to make this view public, so even people that is not taking part in the competition can see the progress of the players in the game.

The instructor interface

Figure 4.7 shows the list of the competitions owned by an instructor. It provides some status information and action buttons to modify them or start a new competition. The form to create a new competition is displayed in figure 4.8. The same view is used to edit the details of a competition too. In the later case, it also contains buttons for actions like finish the competition, cancel an ongoing round or start a new one.
CHAPTER 4. ADDING SUPPORT TO RUN OUR GAME TO JUTGE.ORG

Figure 4.7: Screenshot of the instructors’ competition list view

Figure 4.8: Screenshot of the New competition form
4.3.6 Creating a public web-service API

Some devoted students created applications to display information about the competitions. To do so, they had to resort to parse the HTML of the site to get the information (also known as scraping the site). In this new version we wanted to offer a proper way to achieve this, by implementing a public API.

Since the Jutge.org already provides web services for several purposes, it was easy to add our own to implement the following public API:

- `/competitions/username:competition_name`: Gives information about the specified competition. Returns a YAML document with the competition information and current state.

- `/competitions/username:competition_name/participants`: Retrieves a list of participants in the competition. The YAML document will contain a list of user names and player names for each user playing this competition (that is, users that submitted an accepted program before the competition started).

- `/competitions/username:competition_name/matches/round_number`: Returns all the matches for a given round, grouped by turns, and the names of the players in that match. For each played match, it will also contain the final scores and a link to the viewer for that game.

Besides those, there are other web services for instructors to create courses or start rounds but are only available for users logged in with an instructor account. They are all documented under the section “Documentation/Services” in the website itself.

Even though all the existent web services use YAML, I thought important to provide JSON responses too, for those who want to call our API from a web application. Because of this, we added an optional argument `?format=JSON` that can be appended to any web service URL to get the response in this format.

4.4 Problems found and technical decisions

During and after the development, there arose some issues. Here we detail those which deserve being explained:
4.4.1 Generalizing database to any number of users

Even though we only play 4-players matches now, we wanted to support games with any number of players in a match. Since the Match table has to keep a reference to each user on it, this added an extra problem: we could not simply have 4 columns in the table, one for each player. The solutions we thought were the following:

- **Use PostgreSQL arrays**: They are sent and received as JSON arrays from PHP code and can store an arbitrary number of elements. However, it is not possible to set referential integrity constrains over the elements of an array.

- **Add a join table**: Like those used when normalizing many-to-many relations. This increases the complexity of the database schema, thus making the queries more complex.

Since we wanted to enforce the referential integrity of the database we chose the second option, even though it will have an impact on performance on the queries that will require an additional join.

4.4.2 Race condition when running multiple matches at once

Since the correction server has several virtual machines to run the submissions, it is possible that some matches run in parallel. We observed that when the two last submissions of a round ended at the same time, none of them was aware that it was the last one, so the round was not marked as finished.

To fix this problem, we used a SQL query with the `SELECT ... FOR UPDATE` construction, that creates a mutual exclusion lock between a `SELECT` and the subsequent `UPDATE`. This way, the code path that checked if a match was the last one was only executed by a single thread at a time, avoiding the race condition while keeping everything else still concurrent.

4.4.3 Ownership of the competition matches

The regular submissions are owned by the user that submitted them and can only be viewed by that user. For the submissions of the competition matches (that use programs from several users) we had to decide which user would held its ownership.

Since the scores are already displayed in the round view, that is public to everybody, and there is no need for any user to access the actual submission results page, we decided to assign
them to a specifically created 'competitions' user. As this user does not have a valid login and password, only the administrators will be able to access the competition submissions regular results page.

This solution also allowed us to assign the scores of the Dummy players to this 'competitions' user, being able to keep some integrity constrains of the database that require every program to belong to a user.

4.4.4 Group players when preparing matches

When we make random groups of players for the matches and we have to add dummy players, we try to distribute them to minimize the number of dummies in the same match. According to this, if we have 9 players and we want to make groups of 4, the best solution would be to make 3 groups of 3 players and 1 dummy each. The logic behind this can be a bit tricky and has some special cases: for example, when making groups of 4 from a total of 5 or 2 players, there have to be two dummies per round.

4.4.5 Submission IDs could not go over 999

Since all the submissions belong to the same user, the submission IDs for that user in a given game grown up to very high numbers and this caused an error when it reached the ID 999.

The problem was caused by the IDs being stored as strings with the format S001 in the database, and incremented by this code:

```php
function get_new_id ($user_id, $problem_id) {
    $res = $this->dbc->get_one(
        "SELECT MAX(submission_id) FROM Submissions
        WHERE user_id = " . q($user_id) . "
        AND problem_id = " . q($problem_id) . "");
    $res = (int)(substr($res, 1, 3));
    return sprintf("S%03d", $res+1);
}
```

Figure 4.9: Original submission `get_new_id` code

This code had several problems:
1. **substr was limited to 3 characters.** This could be removed without causing any harm, as the `sprintf`'s field length modifier will work for both shorter and longer numbers (15 becoming S015 and 31416 becoming S31416).

2. **The IDs are stored as strings in the DB,** so when evaluating the MAX they were compared alphabetically. As, alphabetically, S999 < S1000, after reaching the submission 999 we always got that value as maximum.

The final code looks like this:

```php
function get_new_id ($user_id, $problem_id) {
    $res = $this->dbc->get_one(
        "SELECT MAX(substring(submission_id,2)::int) FROM Submissions
        WHERE user_id = ".q($user_id)."
        AND problem_id = ".q($problem_id)
    );
    return sprintf("S%03d", $res+1);
}
```

Figure 4.10: Fixed submission `get_new_id` code

Where the `substr` and the cast from string to integer is done *before* evaluating the MAX, by the database manager itself, resulting in the proper calculation of the maximum value.

### 4.5 Testing the competitions

Once everything was finished and debugged, we did a final test: recreate the last year’s (2013) spring course Battle Royale competition. It involved 132 different programs that played about 3700 matches. Even though we found some problems, this test helped us to fix them and finally we managed to run the whole competition successfully.

After running this test successfully we have the confidence that our platform is solid enough to run large scale competitions and using real-world programs. Those programs sometimes crashed, printed large amounts of debug messages, exhausted the maximum CPU time allowed, etc. without affecting the course of the competition, as we should expect.

We automated the process of starting new rounds by using the web services and a cron task, instead of doing it via the web interface, so we could test the instructor API too.
Chapter 5

Developing a brand new 3D game

To contribute to the growth of this platform and project, one of the goals of this projects was to develop for it a game that looked more like usual videogames and less like a board game. Until now all the games were played on tile-based boards, and this is something we changed.

5.1 Defining our game

5.1.1 Requirements

While doing the next game for our platform, it was my personal goal to use 3D graphics for it. Apart from that, other requirements are given by the fact that the game has to be used to evaluate the students. Mainly these are:

1. It has to encourage students to use programming techniques learned in the subjects where the game will be used\(^1\). This includes graphs and search algorithms (e.g. DFS, BFS, Dijkstra’s algorithm) and data structures from the C++ standard library (queue, map).

2. It should use 4-players matches, as the previous competitions, to run less matches if there are a lot of students.

3. It has to be as fair as possible: minimize the randomness of events in the game.

4. The duration has to be limited: there will be a fixed or a limited number of game rounds.

\(^1\)http://fib.upc.edu/en/estudiar-enginyeria-informatica/assignatures/EDA.htm - EDA content
CHAPTER 5. DEVELOPING A BRAND NEW 3D GAME

5. The game code has to be C++ and the viewer code HTML.

5.1.2 Technology

Hardware acceleration is very important for 3D computer graphics because it allows us to do graphical calculations very fast by using the GPU of our computers. GPUs are faster than CPUs to do those because they take advantage of the common particularities of the graphics calculations:

- Most of the operations are matrix products that can be done on specific hardware.
- There are few data dependencies so most of them are parallelizable. For example, a vertex transformation does not depend on another and, as a result, current GPUs can apply transformations in batches of thousands of vertices at the same time.

HTML5 introduced the possibility of using hardware accelerated 3D graphics in the browser, by adding the WebGL Javascript API that is already present in modern browsers. We will be able to create our game thanks to this new technology.

Currently there are several libraries that implement high level concepts in computer graphics (like "camera" or "scene") on top of WebGL. Examples of those are Three.js, GLGE, SceneJS and PhiloGL. The decision was to use ThreeJS\(^2\) because it provides fall-back renderers in case the browser does not support WebGL, so even students with old computers can run the game. ThreeJS is an open source library mainly developed in Barcelona.

The C++ part will be based on the already existent code used in Battle Royale and previous games, as described in the third chapter.

5.1.3 Gameplay

Jordi Petit brought that a Tron game was made for the subject of EDA some years ago, and it could be a good idea to implement it in 3D (like in the original Tron movie).

In Tron (also known as Snake), each player moves one or more bikes on the board that leave a solid trail behind them, and has to avoid crashing against other bikes or their trails.

\(^2\)http://www.threejs.org - ThreeJS official website
CHAPTER 5. DEVELOPING A BRAND NEW 3D GAME

The original Tron game for EDA had a large map and most of the strategies were about trying to survive without much interaction with the other players.

Figure 5.1: Screenshot of the original Tron game

To solve this we introduced some modifications to the gameplay:

- The map will be much smaller, so encounters between players are more frequent.
- There will be bonus items that will appear in given spots, so players compete to pick them up.

Another important problem to fix was the fairness of the starting positions: In the previous game they were random. We will predefine equidistant starting points for each player’s bikes.

Also, we decided to move away from the tile-based games that were used until now, to move to a game with a 3D board. We will take advantage of the fact that the 3D models are made of polygons, to use the same vertices and edges of the model as a graph for the players explore. This way, while the users will program their movements on a typical graph, we will be able to represent them on a nice-looking 3D board with the camera moving around it.
CHAPTER 5. DEVELOPING A BRAND NEW 3D GAME

Following are the maps we came up with: A cube with rounded edges and a sphere (actually, a subdivided icosahedron, more about this figure later).

(a) Spherical board  
(b) Cube board

Figure 5.2: Two different 3D game boards

Since units can move out of sight behind the board, we also added a map shaped like a plane (where all the units are visible at the same time) to ease the debugging of the strategies. The same algorithms implemented for the plane should work on the other game boards.

Figure 5.3: Plane board

Finally, the bonus items we thought for the game are the following:

- Ghost: the bike can move through another bike’s trail.
- Turbo: the bike can move twice as fast.
- Extra points: increases the score of the player.

With all this in mind, we started the development of the game.

5.2 Development

Following the same structure explained for Battle Royale, this game has a C++ program to run the games and a separate HTML viewer.
CHAPTER 5. DEVELOPING A BRAND NEW 3D GAME

5.2.1 The game runner

The board data structure

We re-used most of the code and only had to change the Board class and Action classes. This time, unlike in previous games, the board was not represented by a grid but by a graph instead. Since our graphs will be quite sparse and the most used operation will be to get the list of neighbours of a given vertex, we implemented it using an adjacency list, where this operation can be done with cost O(1).

Our board is represented with the following data structure:

- A list of players, with their names and scores (and CPU time spent, used by SecGame).
- A list of vertices, each storing if it is empty or contains a bike, the trail of a bike, or a bonus item, plus a list of the neighbouring vertices.
- A list of bikes, each storing which player commands it, its position (for convenience, though it is redundant), if it is alive and if it has some bonus.

Each round all this information is printed to the output file for the viewer to read it.

To initialize the board the first round, we use a map definition file that contains the adjacency matrix and the starting points for each player and for the bonus. This file is created using the game viewer, because it is who knows the topology of the board, and we will explain how later.

The game logic

The logic behind this game is very simple: at each round, the players will generate one action for each one of their bikes. If they don’t move a bike or the action for that bike is illegal, the bike will be removed and its trail will be erased, otherwise, the player will score some points for the movement. The winner is the player with greater score at the end of a fixed amount of rounds.

The bonus items are trivial to implement, with the exception of the turbo. To implement it without adding complexity to the game logic we resorted to the following trick: the game duration is twice as long but normal bikes will only be able to move in odd rounds, while turbo bikes will be able to move in all the rounds, thus doubling their movement speed.
5.2.2 The dummy player

The former EDA student Albert Lobo together with the teacher Jordi Petit contributed to the development of this game by coding the Dummy player. This player executes a BFS search for each bike to find the longest path to the farthest vertex, and moves the bike in that direction.

5.2.3 The viewer

The most interesting part of this game is the development of the viewer, made in Javascript and ThreeJS.

**Setting up a basic ThreeJS scene**

The first step when developing a graphical application is to get to display "something" on the screen, so we can have some visual feedback when coding. In our case, using ThreeJS we only need to instantiate the classes Scene, Camera and Renderer, add a basic primitive to the scene (like a cube) and call `renderer.render(scene, camera)` to draw it on the webpage.

![Figure 5.4: Our first ThreeJS scene](image)

In the above figure there is no light, so all the faces have the same color. To solve this, ThreeJS implements for us the classes AmbientLight, SpotLight and DirectionalLight. This is the result after adding a SpotLight to the previous scene:

![Figure 5.5: The same scene adding a SpotLight](image)
The next step would be to set up an update and render loop, to display the scene 60 frames per second. For this purpose we can use HTML5’s requestAnimationFrame, as we did for Battle Royale. To check if it works, we could add some animation to the previous scene by rotating the cube for a small amount each frame.

Lastly, it’s common to also add the ability to move the camera interactively. ThreeJS provides several camera controls, like the OrbitControls or FlyControls, that can be called in the update step of the main loop and will automatically listen for the input events they need and move the camera accordingly.

Displaying and animating the game

Now that we can render a basic animated scene, it’s time to add the elements of our game.

When the viewer reads the game file, it instantiates and adds to the scene the model of the board (depending on the topology used in that game), the bikes and the bonus items. There is a Bike class and a Bonus class to manage these elements, while the board is just an instance of the ThreeJS Mesh class because it doesn’t need any extra logic.

Each frame, the game loop executes the following logic:

1. Compute the time since last frame. This will be used for animations.

2. Update the camera and lights position, so the light follows the camera.

3. If enough time has elapsed since the last round, advance the round counter.

4. Update the bikes position and trails.

5. Set the visibility of the bonus (they are always in the board, but remain invisible).

6. Update the text with the players’ scores.

To animate the movement of the bikes they move each frame a fraction of the distance between their current vertex and the next vertex, using a lerp function that depends on the time. It is important to make all the animations dependent on the elapsed time for them to look smooth even on a slow computer that could not reach the 60 frames per second (or in a computer with a higher refresh rate). To do so, we compute a delta time between the previous frame and this one, and multiply every constant increment by it.
CHAPTER 5. DEVELOPING A BRAND NEW 3D GAME

As an example of how to use this delta time value, this is a fragment of code used to animate the bonus that are on the board with a simple rotation:

```javascript
Bonus.prototype.animate = function(delta_time) {
    this.mesh.rotateZ(0.002 * delta_time);
}
```

Figure 5.6: Bonus’ animation code

The final result is that the bonus’ model rotates 0.002 radians per second. Without applying delta time, it would rotate 0.002 radians per frame, that is a variable rate.

Figure 5.7: Game screenshot where two bikes and a bonus item are shown

In figure 5.7 you can see the final aspect of the game viewer. Note that the background image is a simple CSS background and that the scores of the players are also written in standard HTML `<div>` elements. The buttons and the slider bar are the same used in Battle Royale, and allow to manually change the round displayed or pause the animation.
Exporting the adjacency list from the model

As we said before, the game runner needs to know the adjacency list for the the vertices of each model that we use as a board. To do so, we added a small function in javascript that, given a 3D model, prints its adjacency list to the browser’s console. Then we can save it as a map file, add to it the starting vertices (that have to be defined in the same file) and load it in the runner.

This way, to add a new map we first have to add its 3D model to the viewer’s code and then export the adjacency file for the runner.

5.2.4 Problems found

During and after the development there arose some issues. Here are the most relevant.

The spherical map

When creating the first map for the game, we wanted the surface to be spherical. Although, we needed it to have discrete subdivisions, or nodes, between which the players will move. We knew that we could use the 3D model’s vertices as nodes for our game, but we found a problem with the spheres. The spheres models’ are already polyhedral approximations that use a finite number of faces, as we need. However, the most used approximation uses a series of vertical triangle strips, like a cylinder stretched at the ends.

![Common sphere approximation](image)

Figure 5.8: Common sphere approximation

Probably because it’s easy to code, it has been used historically and nowadays it’s even a primitive in several 3D toolkits like Direct3D and GLUT. The problem with this approximation is that the vertices are not evenly distributed as the smaller strips near the poles have much higher density than those near the equator. Since we want to use those edges as our game map, this was a problem for us and we had to find a different representation.
CHAPTER 5. DEVELOPING A BRAND NEW 3D GAME

The final approximation we used is based on a regular icosahedron figure. An icosahedron is a polyhedron with 20 triangular faces, 30 edges and 12 vertices, and a regular icosahedron is convex, with congruent faces of regular triangles and the same number of faces meeting at each vertex.

![Icosahedron and its successive subdivision iterations](image)

To make an icosahedron look like a sphere, we add more faces using the Charles Loop face division algorithm. On each iteration, this algorithm creates 4 faces from each of the already existing ones (starting with 20, we get 80, then 320...) as shown in figure 5.9. Doing so we end with the 12 initial vertices that are still connected by 5 edges, and a number of new vertices with 6 edges each. Even though this figure is not a regular polyhedron anymore, it has a more uniform vertex density than other sphere approximations and suits better the requirements of our game.

**Programs being terminated in the secure environment**

One of the limitations that games have when executed in the secure environment with SecGame is that they can only open a limited number of files (the required minimum to open the shared libraries and the game file). We found that our game processes crashed at start when executed in SecGame mode. It took a while to realize that we had to increase the number of open files for this game: in this game we have an extra `fopen` call to read the file with the topology of the board.

**Ensuring support for old browsers**

Some browsers don’t implement the WebGL API to render 3D graphics. For those cases we had to use an alternative renderer provided by WebGL, the CanvasRenderer, that computes the projection from 3D to 2D using Javascript and draws it using 2D primitives on an HTML 2D canvas.
This is, however, much slower than using actual WebGL calls, so in this case we had to simplify the representation of the game to the minimum needed: we changed the bike’s model to a box, disabled the lights and changed the MeshLambertMaterials to MeshBasicMaterials.

![Board when WebGL is present](image1.png) ![Board without WebGL](image2.png)

Figure 5.10: Differences between WebGL and compatibility modes

**Performance issues with big maps**

It was a priority to make the viewer run smooth even when there is no WebGL present and or if it runs on old hardware, so we could not add too many bikes or make the board too big, as doing so affected the performance. Bigger maps would make the game more fair, but we had to adjust to this limitation and keep the maps small.

### 5.3 Conclusions

This game was used for the subjects of EDA and ALG during last year’s (2013) autumn semester and we got good feedback from the students. They were positively motivated to play the competition and were following the results of the matches in real time. The audience for the presentation of the final matches at the FIB’s auditorium was remarkable too.

Also, during the course of the competition we got several contributions from students, like a browser extension that notifies you when your program is playing a match (by David Da Silva), a website to bet for your favourite player (by Albert Lobo), and a web application to test your player by running 1000 matches in a row (by Dario Niewenhuis).
Taking all this into account, we can conclude that Tron 3D was successful in its purpose. However, we think that the Dummy player was probably too easy to beat (almost every student did it), and it should be improved for future competitions.

Figure 5.11: Screenshot of a match between players made by FME students
Chapter 6

Planning and economic valuation

For every project, it is important to be able to estimate the cost in time and resources of its development. This chapter explains the time schedule and economical cost of the presented work.

6.1 Planning

The different ideas behind this project grew up as we developed it, so we didn’t have a strict planning beforehand. Also, the lack of knowledge in some areas (and, thus, the need of research) made it more difficult to estimate the time requirements.

We had two deadlines to meet, though, because we had to finish the games in time for the students of EDA and ALG to play them, so that was our main goal in terms of planning (Battle Royale in April, Tron 3D in October). Apart from that, we developed the different parts of this project in successive iterations as they were needed.

![Gantt diagram of the project dedication](image)

Figure 6.1: Gantt diagram of the project dedication
The longest tasks are the development of the game Tron 3D and the website, as they are the biggest parts of the project. Also, working with an already existent code base is notably time-consuming because it requires some time to understand it before beginning to add new features.

Once the games were finished, we also had to provide support for them and fix the bugs that the users found. Although, those tasks overlap with others because they do not require entire dedication. Most of the research and reading (e.g. about ThreeJS or about advanced SQL), was also done at the same time than other tasks.

I expected the documentation process, altogether with the writing of this report, to be more time consuming than it finally was. This had a positive effect on the results of this project, as I could use the extra time to polish some minor details both in the code and the documentation.

The time dedicated to the project was not constant. In some stages it was difficult to combine my job with the project, and this explains some steps taking a little longer than they could have causing the project span to expand to one year. The break during the summer is also due to the same reason.

6.2 Economic analysis

The main costs of this project are evenly distributed in analysis (including design and planning) and development time. To calculate a total amount of money that this project could cost, we will assume the following salaries for each role:

- Analyst costs: 35 €/h
- Developer costs: 25 €/h

I will not consider the cost of the involved computers and servers because we didn’t require any new or extra hardware for the development of this project.

Also, since we only used free software both in the development environment and in the servers, the cost in software licenses is zero.
The following break down includes an approximation of the time spent in each task and sums up the total cost according to the previous amounts:

<table>
<thead>
<tr>
<th>Task</th>
<th>Profile</th>
<th>Time</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reading and research</td>
<td>Analyst</td>
<td>80 h</td>
<td>2 800 €</td>
</tr>
<tr>
<td>Code analysis</td>
<td>Analyst</td>
<td>100 h</td>
<td>3 500 €</td>
</tr>
<tr>
<td>Design</td>
<td>Analyst</td>
<td>80 h</td>
<td>2 800 €</td>
</tr>
<tr>
<td>Web implementation</td>
<td>Developer</td>
<td>180 h</td>
<td>4 500 €</td>
</tr>
<tr>
<td>Games implementation</td>
<td>Developer</td>
<td>160 h</td>
<td>4 000 €</td>
</tr>
<tr>
<td>Games support</td>
<td>Developer</td>
<td>40 h</td>
<td>1 000 €</td>
</tr>
<tr>
<td>Documentation</td>
<td>Developer</td>
<td>40 h</td>
<td>1 000 €</td>
</tr>
<tr>
<td>Project report</td>
<td>Analyst</td>
<td>40 h</td>
<td>1 400 €</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>720 h</td>
<td>21 000 €</td>
</tr>
</tbody>
</table>

So we can conclude that the total cost of this project would be about 21 000 €.

To be fair it is important to note that the analysis time has been lowered by the fact that I had help from the project director, which wouldn’t be possible out of an academic environment.
Chapter 7

Conclusion and future of the project

Looking back at the goals we defined at the beginning, we can say that we have more than accomplished them: our university now has a functional and very complete web platform that includes both the programming problems it already had and the game challenges we developed.

Apart from the two games we created in this project we also ported another two, allowing us to publish the result of this project meeting the expected deadline and with 4 games ready: Tron 3D, Battle Royale, Bola de Drac and Pacman. Also, everything has been documented as well, making it easier for others to keep creating games and making the project future-proof.

Moreover, we have met the milestones in time for the subjects of EDA and ALG to use our games. This provided us a test in a real-world environment, that has great importance in a project of practical application like this, and has proven our software to be rock solid. As we have already remarked at the conclusion of the chapter 5, the feedback we got from the students of these subjects was also good, which gives us more confidence in our results.

Personally, this project has also been useful to me to learn in deep about some technologies that I had never used before, as are HTML5, WebGL, ThreeJS and even LaTeX. This knowledge, together with the experience of working in a large scale application as Jutge.org, is by itself a valuable gain I got from working in this project.

7.1 Future of this project

Now that we published the results of our work and people can begin to use it, we want it to keep going forward. Ideally, it would need some marketing and promotion to spread the voice
so it can get the critic user mass it requires. If enough people is competing and a community emerges around the games, then forums and other community tools should be created to manage it properly.

For those people that want to get involved even more in the project or develop their own games, this report should contain enough information to get started. This point is very important for the future of the project, and that’s the reason why to generate documentation was also one of the goals of the project.

Meanwhile, the games will be used as a grading tool for the subjects EDA in the Faculty of Informatics and ALG in the School of Mathematics and Statistics. To keep the competition interesting each semester, students of greater courses and teachers should be encouraged to write new games that can be used for the forthcoming semesters. This way a fresh new game will be used each time, avoiding that the students copy code from previous years and adding a surprise factor that will make it more fun.

Those are some ideas we got for the project but had to drop off due to the lack of time:

- Be able to start matches against your friends (e.g. connecting your Facebook account), out of a competition.
- Add support for games with a different number of players in a match. At least, games with 1vs1 matches, like chess, are really common and should be supported.
- Add support for games with a single ”all vs all” match, and create a game using it.
- Add support for games in other languages than C++. For example, we could include Robocode competitions, an AI-commanded robots fighting game written in Java.

7.2 Access to the code

The public sources and documentation of the games can be found in the following URLs:

**Battle Royale:** https://battle-royale-eda.jutge.org/?cmd=material

**Tron 3D:** https://tron3d-fib.jutge.org/?cmd=material

We want to keep the secure environment and the server-side code private, but access will be granted for any reasonable purpose at the discretion of this project’s director Jordi Petit (jpetit@lsi.upc.edu).
7.3 Gratitude to free software

This project would not have been possible without free software. Every piece of software used was free software: from the web server (Apache) to the documentation production (Latex), the operating system, compilers...

Every engineering project, and specially in the computer engineering, relies on previous work done to build on top of it and keep. I think that free software is a fundamental part of the development of our field because it makes very easy to reuse and extend the knowledge that other have created, and for that reason I want to give credit to it in this project.

Initially I thought in publishing this project sources as free software too, but the authorship of the sources is shared with several people that contributed to the project in the past, and I was not able to contact them all asking them to yield their ownership. For future reference, some of the copyright holders of the non-free sources used in this project are Jordi Petit, Salvador Roura, Omer Giménez, Albert Atserias, Enric Rodriguez, Enric Cusell, Albert Lobo and Mario González.

Also, we would only be able to publish a part of the work, since there are some parts that we want to keep private: The Dummy players sources to avoid copies and the SecGame and the server-side code to not provide information that could be used in an attack.
Bibliography

http://dl.acm.org/citation.cfm?doid=2157136.2157268


http://dl.acm.org/citation.cfm?doid=2157136.2157268

http://upcommons.upc.edu/pfc/handle/2099.1/18641


http://www.w3schools.com/html/html5_intro.asp

[8] The PHP Group: The PHP Language Reference
   http://www.xajax-project.org/en/docs-tutorials/api-docs/

[10] Oren Ben-Kiki, Clark Evans, Ingy döt Net: *YAML Ain’t Markup Language*
    http://www.yaml.org/spec/1.2/spec.html

    http://threejs.org/docs


  *Online game competitions*

  *Graph data structures*
    http://en.wikipedia.org/wiki/Graph_(abstract_data_type)

  *Icosahedron*

  *C++11*
    http://en.wikipedia.org/wiki/C++11

  *YAML*
    http://en.wikipedia.org/wiki/YAML

  *JSON*
    http://en.wikipedia.org/wiki/JSON

  *Base64*
    http://en.wikipedia.org/wiki/Base64
Appendices
1 Introduction

Hello student! This semester you will compete against your classmates playing Battle Royale. Lucky you!

In this game, programs written by the students will fight against each other in successive rounds. The worst program at the end of each round will be out of the competition. Your qualification (that will be added as up to 2 extra points to your final mark in this subject) will depend on the number of rounds your program stays in the game.

The last 16 programs will get the maximum score and will be classified for the Grand Final, a match to find the greatest Battle Royale survivor. This player will be honored with the entrance to the Battle Royale Hall of Fame and will be crowned with a cheap plastic crown.

To enter the competition and have a chance to win this prize, you only need to write a program capable of defeating our "Dummy" program. This will already add 1 extra point to your mark, and you will probably have fun doing it!
2 Game rules

In this game, each player has control over an army of knights and farmers in a tile-based board.

The goal of the game is to “farm” as many tiles as you can, converting them to your team color. Your score at a given time is the number of tiles of your color at that point, and the winner is the player who has more tiles of his color at the end of the game.

At the beginning of the game, all the knights and farmers of a player are randomly placed at the spawn point of that player: one of the four quadrants of the board.

Each round the players can move each unit one position in one of the 4 cardinal directions.

Moving a farmer to an empty tile will convert the tile to the farmer’s team color, even if it already has the color of a different team. Moving a farmer to an non-empty tile is an invalid move.

Moving a knight to a tile occupied by another player’s unit will make it attack that unit, performing a random amount of damage. If the health of a unit drops to less or equal than zero, that unit will be converted to the attacking player’s team and will respawn at his spawn quadrant with full health. (When this happens, the players can say out loud: Wololo!)

Deliberately not moving a unit will increase its health by a given constant. A unit cannot recover more health than its initial amount. Performing an invalid move will result in that unit not moving, but will not regenerate health.

The boards will have “walls”, obstacles that units cannot go through. Trying to move a unit into a wall is an invalid move.

Other invalid moves are moving a farmer to a tile occupied by another unit, or moving a knight to a tile occupied by a unit of the same team.

If the board does not have walls all around, moving out of the board will make a unit wrap to the opposite side.

The order of the actions performed by players is chosen at random, so you cannot expect any execution order of your actions. A notable case is when two units try to move to the same tile: the unit that happens to move first will be able to occupy the tile, while the second one will not.

A game is defined by a board and the following set of parameters:

- `nb_players`: Number of teams in the game.
- `nb_rounds`: Number of rounds that will be played.
- `nb_farmers`: Number of farmers per player.
- `nb_knights`: Number of knights per player.
- `farmers_health`: The maximum (and initial) health of a farmer.
- `knights_health`: The maximum (and initial) health of a knight.
- `farmers_regen`: The amount of health a farmer will regenerate when not moving.
- `knights_regen`: The amount of health a knight will regenerate when not moving.
- `damage_min`: The minimum amount of damage a knight will inflict when attacking.
- `damage_max`: The maximum amount of damage a knight will inflict when attacking.
- `rows`: Vertical size of the board.
- `cols`: Horizontal size of the board.
3 Programming

The first thing you will need is the source code you can find at the game site:

https://battle-royale.jutge.org

The source code includes a C++ program that runs the games and an HTML viewer to watch them in a nice animated format. Also, a "demo" player is 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 will only need g++ and make installed on your system, plus a modern browser like Mozilla Firefox or Chromium.

1. Open a console and cd to the directory where you extracted the source code.

2. Run `make all` to build the game and all the players. Note that our Makefile will identify as a player any file matching the expression "AI*.cc".

3. Make should create an executable file called `BattleRoyale`. This executable allows you to run a game using a command like:

   ```
   ./BattleRoyale Demo Demo Demo Demo < maze.cnf > game.br
   ```

   Here, we are starting a match with 4 instances of the player "Demo" (included with the source code), in the board defined in "maze.cnf". The output of this match will be redirected to "game.br".

4. To watch the game, open the viewer (viewer.html) with your browser and load the "game.br" file.

   Use `./BattleRoyale --help` to see a list of parameters you can use. Particularly useful is `./BattleRoyale --list`, that will show a list with all the recognized player names.

   If needed, remember 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 player copy the file `AINull.cc` or `AIDemo.cc` to a new file with the same name format (AIWhatever.cc).

Then, edit the file you just created and change the

```
#define PLAYER_NAME Demo
```
line to your own player name. The name you choose for your player must be unique, non-offensive and less than 12 letters long. This name will be shown in the website and in the matches.

Then you can start implementing the virtual method `play()`, inherited from the base class `Player`. This method will be called every round and is where your player should decide what to do, and do it.

Of course, you can define auxiliary methods and variables inside your player class, but the entry point of your code will always be this `play()` method.

From your player class you can also call functions to access the board state (defined in the `Board` class in `Board.hh`) and to command your units (defined in the `Action` class in `Action.hh`). 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 both in the header files of the above mentioned classes (and also `PosDir.hh`), and as a PDF you can download from the game website.

Note that you should not edit the `factory()` method from 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 your player is strong enough to enter the competition, you should submit it to [https://battle-royale.jutge.org](https://battle-royale.jutge.org). 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 (AIWhatever.cc).
- Your code cannot use global variables (use attributes in your class instead).
- You are only allowed to use standard libraries like `vector`, `map`, `cmath`...
- Your code 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.
- Your program should not write to `cout` nor read from `cin`. You can write debug information to `cerr`, but remember that doing so on the code you upload can waste part of your limited CPU time.

### 4 The game website

You can find the game website at [https://battle-royale.jutge.org](https://battle-royale.jutge.org). Here you can download the instructions and source code you need to start coding
your player. Also, this website is used to submit your code, see the results in real time and play matches against your friends.

Do not worry if your browser warns you when accessing the website because we are using a self-signed certificate.

4.1 Submissions

When submitting a new player, it will go through the following process:

- The source code will be built on release mode with -O2. If the compilation fails, your submission will be rejected.
- A game with 4 instances of your player will be started. If any of them crashes or reaches the CPU time or memory limits, your submission will be rejected.
- Your player will play 4 successive games against 3 other “Dummy” players (a simple player with an easy to beat strategy). If your player does not win the 4 games, your submission will be rejected.

If your submission is rejected, you will be able to know the cause to be able to fix it before submitting your player again.

If your submission is accepted, it will be able to start playing matches online. Also, its name will be reserved so that no other program can be sent with the same name.

Note that the code you submit will never be published and you should neither distribute it yourself. Although, it is ok to distribute its object file.

4.2 Friendly matches

From the website you can play matches against your friends. The results of these matches does not have any effect on the classification and are only visible to the owners of the players.

For any program you have submitted, you can choose if it will be private or public. Note that this does not mean to publish your source code!

Any other player can start a match against your public program, and only your friends can start a match against your private players. You can add your friends by email on the same website, but remember that your friends have to add you too before being considered friends. A private program cannot play a match if its owner is not friend with all the other users in the game.

By default an uploaded program will be private, but once made public it cannot be made private again.
4.3 Official matches

When the official matches start (see “Important Dates” below), each day the players will compete in successive rounds, discarding the player that gets worst results in each round. These results and the matches themselves will be publicly available to everybody.

To determine the player that has to be discarded, the following process will be used:

- Random groups of 4 players will be created. If there are less of 4 players in a group, “Tonto” players will be added.
- Each group of 4 players will play a match.
- Any player that gets a better result than 2 of its 3 rivals, will move to the next round.
- The remaining players will be regrouped again in groups of 4 players and start the process over, until there is only 1 player left.
- That player will be out of the competition.

You will be able to choose which of your submitted programs will play in the official matches, or even upload a new one, but the changes will not take effect until a new round starts.

4.4 Issues, news and updates

Any new information will be made available through the game website.

4.5 Results

The results for each round will be made public in real time on the website.

4.6 Forum

You can find a topic for the game in the Racó forum.

5 Tips

- Read the headers of the classes 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) auxiliar methods, and make sure they work properly.
• Before competing with your classmates, focus on defeating the "Dummy" player. This already gives you one extra point!

• Keep a copy of the old versions of your player. When you try to improve it, make it fight against its previous incarnations to measure the improvement.

• As always 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 to output debug information and add asserts to make sure the code is doing what it should do. Remember to remove the cers before uploading your code, because it makes the execution slower.

• When debugging a player, remove the cerr you may have in the other player’s code, to make sure you only see the messages you want.

• If using cerr is not enough to debug your code, learn how to use valgrind, gdb or any other debugging tool, they are quite useful!

• Switch on the DEBUG option in the Makefile, it will allow you to get useful backtraces when your program crashes. There is also a PROFILE option you can use.

• 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 the other players watching the games. This way you could try to defend against them or even improve them in your own player.

• Do not give your code to anybody. Not even an old version. We are using JPlag and other plagiarism detectors to check the programs, not only between them but also against other years submissions.

• You could, however, share the compiled .o files or (the easy way) just use the website to play against your friends.

• You can submit new versions of your program at any time.

• Do not wait to 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 games, and maybe it is too late!

• And again: Keep your code simple, build often, test often. Or you will regret.
1 Introduction

Hello student! This semester you will compete against your classmates playing Tron 3D. Lucky you!

In this game, programs written by the students will fight against each other in successive rounds. The worst program at the end of each round will be out of the competition. Your qualification will depend on the number of rounds your program stays in the game.

The best 16 programs will get the maximum score and will be classified for the Grand Final, a match to find the greatest Tron light-bike rider. This player will be honored with the entrance to the Tron 3D Hall of Fame and will be given a lollipop.

To enter the competition and have a chance to win this prize, you only need to write a program capable of defeating our “Dummy” program.
2 Game rules

In this game 2 or 4 players will compete at the same time. Each player will command several Tron-like bikes, that will spawn in predefined spots of the board.

There are several boards where the games can be played. Each board is just a graph made of connected vertices. The cool part is that those graphs have the topology of 3D models, so the game can be represented in awesome 3D graphics!

At each round, players must move their bikes. When bikes move, they leave a solid trail behind them. Bikes that crash into those trails will explode, reducing the score of the player who commands them. Bikes that just do not move for one round will overheat and explode, too! When a bike explodes, its trail will disappear.

The goal of the game is keeping your bikes alive for as long as possible, while trying to make the others crash. The players’ score is calculated based in the number of movements their bikes did. Each movement is worth 10 points so the more rounds you keep your bikes alive (and thus moving), the greater your score will be.

When, each round, the players command their bikes, they have no way to know the movements of the other players for that same round. The execution order of the actions of the different players is chosen at random, so if two or more bikes try to move to the same vertex, one at random will be allowed to move, while the others will crash!

2.1 Bonus

To keep the game interesting, we have bonus items too!

In a given round, bonus items will spawn at some vertices of the board (if those are not occupied by bikes or their trails). The bikes that pick up those items by moving to those vertices and will be able to use them to gain super-powers. Note that, until used, items are kept per bike and not per player, and that a bike can only have a single item at a time, i.e. picking up a second item will replace the previous one. The following bonus items exist:

- Turbo: When activated, it allows the bike to move in turbo mode for some rounds. Turbo mode is explained later on this document.
- Ghost: When activated, it allows the bike to go through walls and other bikes for some rounds.
- Extra points: Increases the score of the player by a small amount. This item is consumed automatically upon picking it up. Thus, it will not replace others items the bike could have.
Turbo mode

To implement turbo mode without adding extra complexity, the following decision was taken: The total game duration is doubled, but the normal (non-turbo) bikes are only allowed to move in even rounds, while the turbo bikes can move in both odd and even rounds (thus doubling their move speed). Okay, we know it’s not the most elegant solution on the Earth, but the monkeys we have as interns were not able to find a better one. When writing your strategy, take into account that most of the times you will only be able to move half of the rounds! Also note that it is mandatory for bikes in turbo mode to be moved those extra rounds or they will explode too!

Using the turbo can be dangerous because the bike has to move the extra rounds it is given. However, those extra moves will also give the player movement points.

2.2 Game parameters

A game is defined by a given graph (the game board) and the following set of parameters:

- \textit{nb\_players}: Number of teams in the game (will be 2 or 4).
- \textit{nb\_bikes}: Number of bikes per player (usually 2).
- \textit{nb\_rounds}: Number of rounds that will be played (usually 200, that means 100 without turbo).
- \textit{bonus\_round}: The round where the bonus items will appear (usually 50).
- \textit{turbo\_duration}: The movements a bike can perform in turbo mode after using a turbo item (usually 8, that means 4 extra movements).
- \textit{ghost\_duration}: The movements a bike can perform in ghost mode after using the ghost item (usually 3).
- \textit{score\_bonus}: The points given when picking up an extra points item (usually 50).

All these parameters can be accessed by the players during the game.

Each board also defines a starting point for each bike and a set of vertices where the bonus items will appear, intended to be fair. To know which are those vertices, the players can check \textit{bonus\_vertices} and try to get there before their rivals, but note that if these vertices are already occupied by a wall or a bike, the bonus will not appear.
3 Programming

The first thing you should do is to download the source code.

The source code includes a C++ program that runs the games and an HTML5/Javascript viewer to watch them in a nice animated format. Also, a "Demo" player is 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 will only need g++ and make installed on your system, plus a modern browser like Mozilla Firefox or Chromium.

1. Open a console and cd to the directory where you extracted the source code.

2. Run make all to build the game and all the players. Note that the Makefile will identify as a player any file matching the expression "AI*.cc".

3. Make should create an executable file called Game. This executable allows you to run a game using a command like:

   ./Game Demo Demo Demo Demo < icosahedron.gam > icosahedron.t3d

   Here, we are starting a match with 4 instances of the player "Demo" (included with the source code), in the board defined in "icosahedron.gam". The output of this match will be stored in "icosahedron.t3d".

4. To watch the game, open the viewer (viewer.html) with your browser and load the "icosahedron.t3d" file.

   Use --help to see a list of parameters you can use. Particularly useful is --list, that will show a list with all the recognized player names.

   If needed, remember 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 player copy the file AINull.cc or AIDemo.cc to a new file with the same name format (AIWhatever.cc).

Then, edit the file you just created and change the playername line to your own player name, as follows:

```c
#define PLAYER_NAME Whatever
```

79
The name you choose for your player must be unique, non-offensive and less than 12 letters long. This name will be shown in the website and in the matches.

Then you can start implementing the virtual method `play()`, inherited from the base class `Player`. This method will be called every round and is where your player should decide what to do, and do it.

Of course, you can define auxiliary methods and variables inside your player class, but the entry point of your code will always be this `play()` method.

From your player class you can also call functions to access the board state (defined in the `Board` class in `Board.hh`) and to command your units (defined in the `Action` class in `Action.hh`). Those functions are made available to your code using multiple 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 the header files of each class. If you have Doxygen installed on your system, you can also generate a separate document with all the documentation running "make doxygen".

Note that you should not modify the `factory()` method from your player class, nor the last line that adds your player to the list of available players.

### 3.3 Playing against the Dummy player

To test your strategy against the Dummy player, we provide precompiled `AIDummy.o` object files. This way you still won’t have the source code of our Dummy strategy, but you will be able to add it as a player and compete against it locally.

To add the Dummy player to the list of registered players, you will have to edit the `Makefile` file and uncomment one of the first lines matching your platform (Linux 32 bits, Linux 64 bits, MacOS...). Remember that object files contain binary instructions targeting a specific machine, so we can not provide a single, generic file. If you miss an object file for your architecture, contact us and we will try to add it.

Pro tip: You can ask your friends for the object files of their players and add them to the `Makefile` too!

### 3.4 Restrictions when submitting your player

When you think your player is strong enough to enter the competition, you should submit it to `https://tron3d.jutge.org`. 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 (`AIWhatever.cc`).
- Your code cannot use global variables (use attributes in your class instead).
• You are only allowed to use standard libraries like `vector`, `map`, `cmath`...

• Your code cannot open files nor do any other system calls (threads, forks...).

• Your CPU time and memory usage will be limited when executed on the server. The time limit are 2 seconds for the execution of the entire game. When watching a game evaluated by the game judge, you will be able to see the percent of the total time limit that your program has used.

• Your program should not write to `cout` nor read from `cin`. You can write debug information to `cerr`, but remember that doing so on the code you upload can waste part of your limited CPU time.
4 The game judge

The judge website is used to submit your code, see the competition results in real time and play matches against your friends.

Do not worry if your browser warns you when accessing the website because we are using a self-signed certificate. Wildcard certificates are too expensive for our university!

4.1 Submissions

When submitting the source code of your player, it will go through the following process:

- The source code will be built on release mode with `-O2`. If the compilation fails, your submission will be rejected.
- A game with 4 instances of your player will be started. If any of them crashes or reaches the CPU time or memory limits, your submission will be rejected.
- Your player will play 4 successive games against 3 other “Dummy” players (a simple player with an easy to beat strategy). If your player does not win the 4 games, your submission will be rejected.

If your submission is rejected, you will be able to know the cause to be able to fix it before submitting your player again.

If your submission is accepted, it will be able to start playing matches online. Also, its name will be reserved so that no other program can be sent with the same name.

Note that the code you submit will never be published and you should neither distribute it yourself. It is ok to distribute its object file.

4.2 Friendly matches

From the website you can play matches against your friends. The results of these matches do not have any effect on the classification and are only visible to the owners of the players.

For any program you have submitted, you can choose if it will be private or public. Note that this does not mean to publish your source code!

Any other player can start a match against your public program, and only your friends can start a match against your private players. You can add your friends by email on the same website, but remember that your friends have to add you too before being considered friends. A private program cannot play a match if its owner is not friend with all the other users in the game.

By default an uploaded program will be private, but once made public it cannot be made private again.
4.3 Official matches

When the official matches start (see “Important Dates” below), each day the players will compete in successive rounds, discarding the player that gets worst results in each round. These results and the matches themselves will be publicly available to everybody.

To determine the player that has to be discarded, the following process will be used:

- Random groups of 4 players will be created. If there are less of 4 players in a group, “Dummy” players will be added.
- Each group of 4 players will play a match.
- Any player that gets a better result than 2 of its 3 rivals, will move to the next round.
- The remaining players will be regrouped again in groups of 4 players and start the process over, until there is only 1 player left.
- That player will be out of the competition.

You will be able to choose which of your submitted programs will play in the official matches, or even upload a new one, but the changes will not take effect until a new round starts.

4.4 Issues, news and updates

Any new information will be made available through the game website.

4.5 Results

The results for each round will be made public in real time on the website.

4.6 Forum

You can find a topic for the game in the Racó forum.

4.7 Bets

You will be able to place bets for your favourite players. The website we will made public later on.
5 Tips

• Read the headers of the classes you are going to use. 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.

• Before competing with your classmates, focus on defeating the “Dummy” player.

• Try to keep your code clean, it will be easier to change it and to add new behaviours to your strategy.

• Define simple (but useful) auxiliary methods, and make sure they work properly.

• Keep a copy of the old versions of your player. When you try to improve it, make it fight against its previous incarnations to measure the improvement.

• As always 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 to output debug information and add asserts to make sure the code is doing what it should do. Remember to remove the cerss before uploading your code, because it makes the execution slower.

• When debugging a player, remove the cerr you may have in the other player’s code, to make sure you only see the messages you want.

• If using cerr is not enough to debug your code, learn how to use valgrind, gdb or any other debugging tool, they are quite useful!

• Switch on the DEBUG option in the Makefile, it will allow you to get useful backtraces when your program crashes. There is also a PROFILE option you can use.

• 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 the other players by watching the games. This way you could try to defend against them or even improve them in your own player.

• Do not give your code to anybody. Not even an old version. We are using JPlag and other plagiarism detectors to check the programs, not only between them but also against other years submissions.

• You could, however, share the compiled .o files or (the easy way) just use the website to play against your friends.
• You can submit new versions of your program at any time.

• Do not wait to 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 games, and maybe it is too late!

• Most of the game parameters (number of rounds, duration of bonuses...) won’t change, but if your strategy can adjust to them, you will be extra-safe in case we need to change some of them.

• If you create your own map for the game, send it to us before the competition starts and maybe we will include it! The maps should be OBJ models with a single mesh, made of triangles and/or quads, and have equidistant starting points for the bikes. In case you want to change some game parameters for your map (e.g.: ultra-long turbos, only one bike per player...), the previous point will apply :) 

• And again: Keep your code simple, build often, test often. Or you will regret.