ROBOCUP - Soccer Simulator2D Research Project

COMP 396 – Final Report

Raphael JULIEN under the supervision of Joseph Vybihal

Fall 2023

School of Computer Science
Faculty of Science
McGill University

December 5th, 2023

Abstract

For years now video game developers has been able to simulate highly realistic and efficient soccer matches. However, most of those simulators embed their player AI into a virtual realm, ignoring physical factors and changing rules in order to ensure a better playing experience. The RoboCup Soccer Simulator[1] project aims to bridge the gap between real and simulated soccer by offering an accessible framework for researchers around the world. Following the work done by Melissa Katz[2], this project succeeded in implementing the basis of a fully functional, cooperative team.

1 Introduction

The RoboCup project provides a platform for computer scientists to participate in a global AI competition. The major challenge of this work is to implement a realistic, cooperative, functional AI with a strictly partially-observable environment and limited resources such as stamina. So far, the project only had a basis for computer-server communication, functions to fetch information from the players, the ball, the pitch, as well as an implementation of the Goal Oriented Action Planning[3] system for the AI. In other words, the skeleton was there but the flesh was missing. My objective for this research was to build on what has been achieved so far, in order to facilitate subsequent development as well as experimenting various approaches in order to propose a rudimentary implementation of a team.

2 Getting Started

2.1 Installation

It is required to download additional libraries and utilities in order to run the RoboCup Soccer Simulator. The various libraries and utilities needed to be installed will depend with your OS. The project's README[4] page contains all the steps required for each OS. However, I was only able to make it work on Linux. Therefore I will focus on this implementation, and I recommend future developers to do so as well.
First, go to the RCSoccersim github[5] page and download the latest tar files for the server (rcssserver-<version>) and the monitor (rcssmonitor-<version>). On your machine's CLI, download boost, flex, bison, and qt5. Lastly, cd in the server directory, run *./configure* and *make*. Repeat for the monitor's directory. You should now be able to run able to run:

```
./rcssserver-<version>/src/rcssserver
./rcssmonitor-<version>/src/rcssmonitor
```
[*1*]

If successful, a window with a the simulator's field will pop up.

2.2 Run the code

Now that the server and monitor are up and running, clone the code from the GitLab repository in the same directory as the server and the monitor. You can now run the Java code from the the soccer-tournament directory. Before doing that, it is important to compile each directory using the following commands:

```
javac RCSS2D/src/*.java
javac RCSS2D/src/<directory>/*.java
```
[*2*]

Note: Use the first line for the src directory and the second for any other subdirectory.

Now you may run any runnable code, ie the tester codes (gktester, midtester, teamtester) using the following command:

```
java RCSS2D.src.<tester-code>                                   [3]
```

If successful, players should appear in position. If not, simply Ctrl+C to shut down the code and re-run it.

2.3 Simulator

To run the simulation, go to *Referee>Kick Off*. The left team kicks-off and the simulation runs for 3000 cycles, then the right team kicks-offs and the simulation runs for another 3000 cycles. To interrupt the simulation, simply Ctrl+C the CLI. The simulator offers options to visualize real-time data: by toggling *View>Status Bar*, you can point to any point on field to know the coordinates. This is extremely useful especially to differentiate left and right, up and down. Another set of options is *View>Preferences* (also accessible with Ctrl+V): here you can toggle many visual cues such as the Offside line and the flags.

3 Previous Work

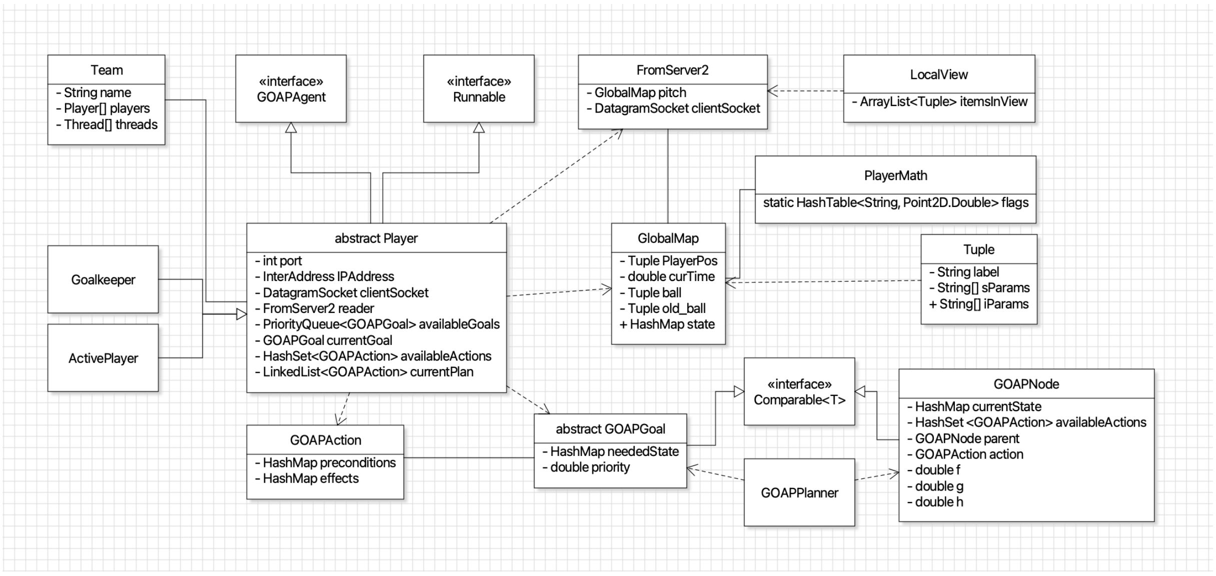


Figure 1: UML Diagram of the previous structure

Figure 1 above shows the overall arrangement of the code. While the structure in itself works well, many underlying components were missing. First, the last two reports on this project were centered around the architectural aspect of the code: the communication with the server, software parallelism, the implementation of the GOAP system for the AI, parsing the sensors, and designing prototype agents. A very basic implementation of a goalkeeper and a player were made, but those lacked any decision making capabilities.

While many features were added during the last research, several were still missing for future development. Namely: the parse sensors were not capturing any visual information about other players in the field of view (FOV); there was no implementation for the auditory information, nor any method to send auditory cues; there were no implementation of the body information (stamina, fouls); several action methods were missing such as Tackle and TurnNeck.

The goal for this research was clear: implement some of the remaining processing methods required for the design of a rudimentary, yet functional team. Several resources were available for me to plan and develop efficiently. First, the RoboCup Users Manual[6] available online, which describes the data sent by the server. Recall that our job is not to create sensory data for our players, but rather to process what is sent to us by the server. Second, the documentation on the current code as well as the previous project report gave me a clear idea on where to start.

4 Classes Overview

It is paramount to understand how the classes interact with each other and what are their respective roles. This section offers a concise summary of each important classes and what to look for in them.

4.1 Players

This directory compiles the Player.java class as well as all its subclasses. The Player.java class defines the various actions any player can make (Dash, Turn, Kick, ...) using the *doAction(String)* method and calculates the ideal Goal and set of actions in order to achieve it. Each subclasses of Player contains all Goals and Actions this particular subclass can perform. In other words, this is where you define what a Player can and cannot do.

4.2 StateKeys.java

The GOAP system uses boolean states to define and influence players' decisions. This enum gather all states to be modified during game time.

4.3 GlobalMap.java

This class updates all players' state every cycle. To do so, it uses 4 methods: *visualUpdate(LocalView), bodyUpdate(LocalView), hearingUpdate(LocalView)* and *positionUpdate()*. The first three use information sent by the server to modify players' state, while the last one relies on the player's coordinates, hence it does not need any input from the server.

4.4 LocalView.java

This class parses the gibberish data sent by the server, and neatly formats it using Tuples. Each information has a label, indicating what object is being oberserved, and a set of information such as time, distance, direction, etc... For instance, formatted visual data should look like this:

```
b time:  0.0|dist:  10.0|dir:  0.0|distChange:  -0.0|dirChange:  0.0| ...  ,
```
[4]

Where b means "ball", and the rest are the information relative to it. Note this message is unique to one player. Note that the format of hear and body messages may differ.

4.5 Goals

This directory gathers all Goals realizable by players. This is where you may define new Goals for your players to perform. Goals are defined using the GOAP system: they require the desired final state, a priority, and a set of criteria for them to be considered.

4.6 Actions

This directory gathers all Actions realizable by players. Actions are selected if 1. A player can perform them and 2. if they help realize the player's current goal. Actions require a set of state conditions to be realized, and a set of state effects they produce upon success. Upon success, effects are applied to the player's StateKeys accordingly. This is also where the programmer defines what an action does, using a reference to the Player class and the *doAction(String)* method.

4.7 Graphs and Planning

These directories gather all classes required for the A* system which dynamically selects what Goal/Action a player should select. These classes work fine and should not be touched, unless a change of the AI system is considered.

4.8 PlayerMath.java

This class offers useful methods to calculate angles, player's positions, and flags' positions.

Other classes are either tester classes or classes necessary for the server communication. New programmers are invited to look at them, but modification is not required.

5 New Implementations

My first assignment was to plan out what could realistically be done during a semester, and to come up with ideas for a team implementation. Thanks to my love for european football and my experience with football game simulators (FIFA, Football Manager), I could easily visualize affordable yet viable options. My first instinct was to create different classes of players, each having a distinct set of Goals and Actions.

5.1 New Players

The ActivePlayer class was split into 3 new positions:

- Attacker: The attacker should go forward, wait for a pass/cross from midfielders, intercept ball if close, and kick ball to goal if possible.
- Midfielder: The midfielder should intercept the ball if its does not have it, carry the ball forward until in range to pass/cross the ball to the attacker. If blocked by an opponent, it should pass the ball to a teammate.
- Defender: The defender should stay close to its goal, stay aligned with the ball, and tackle it if an opponent approaches.

Each player has been given a Goal and Action in order to move to their respective position. Their positions are defined by the coloured areas in Figure 2. Some notable design choices include:

- The attacker's position does not depend on the ball. For now, an attacker is in position if he is past a certain x (here +/- 30 depending on team side). In the future, the position should be modified such that an attacker may never be past the offside line.
- The midfielder must be on either side of the ball, and at a certain distance from it in order to avoid clustering.

- The defender's zone ranges from +/- 45 (see Appendix for details on coordinates) to a floating line which depends on the ball's x position. This leaves a space for the goalkeeer to intervene without cluster, and it allows the defender to move backwards if a foe approaches.

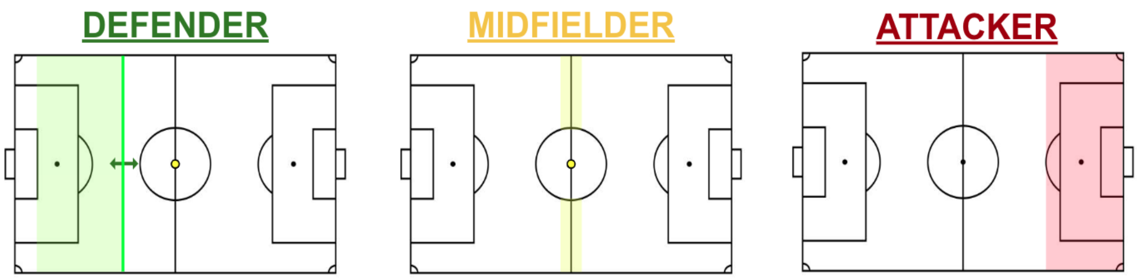


Figure 2: Relative position of each Player on team Left

5.2 Cooperation

Players are now capable of detecting other players and differentiate teammates from opponents. This was done by adding a detector for labels 'p' if the *visualUpdate(LocalView)* method in GlobalMap.java. Visual information about other players is of the form:

```
pt2 time:  0.0|dist:  10.0|dir:  0.0|distChange:  -0.0|dirChange:  0.0| ...  ,
```
[5]

Where p means 'player', t2 means 'team 2'. If the team label corresponds to the player's team, than this is a teammate, else it is an opponent. These tuples can be stored in ArrayLists at each iteration, and decisions can be taken whether or not those ArrayLists are empty, or whether a teammate/opponent is within a certain range.

Another addition was the *doSay(String)* method (using in several Actions), which players can use to broadcast aural messages, heard by all other players. To do so, the method *parseHear(String)* was implemented in LocalView.java. The method takes messages of the form:

```
hear cycle self "<message>"
hear cycle some_number our player_number "<message>"
hear cycle some_number opp "<message>"
hear cycle ref "<message>"
```
[6]

where (self,our,opp,ref) refers to who sent the message, <message> is the content of the message, and some_number is an unknown number. Self means the message was sent by the player itself, our means the message was sent by a teammate, opp means the message was sent by an opponent, ref means the message was sent by the referee. Lines are parsed and relevant information is stored in an array of strings for the player to consult.

This improvement of visual and aural parsing has been critical for the cooperation aspect of the simulation. The following StateKeys have been added and are continuously modified by those information:

- has_ball: Player has the ball (only non-addition)
- team_has_ball: A teammate has the ball
- oppo_has_ball: An opponent has the ball

- teammate_in_FOV: Teammate in field of view
- oppo_blocking: Opponent is in front (only for Attackers)
- oppo_attacking: An opponent has the ball (only for Defenders)

5.3 Goals, Actions, States

A plethora of new Goals, Actions, and States were added in order to fulfill various actions for each player class. You are invited to consult each Player's class to see what Goals and Actions they can perform. The tables below show all the content added to each player class:

Attacker			
GOALS	StateKeys	Effect	Actions
FocusOnBallGoal	ball_centered_in_FOV	true	GetBallInFOV CenterBallInFOV
KickBallGoal	kick_performed	true	BringBallToGoal KickBallToGoal
PassBallGoal	kick_performed	true	ScanForTeammates PassBallToTeammate
GetInAttPosition	in_att_position oppo_blocking	true false	MoveToAttPosition GetUnmarked
ResetPositionGoal	in_new_position	true	TeleportAction

Midfielder			
GOALS	StateKeys	Effect	Actions
FocusOnBallGoal	ball_centered_in_FOV	true	GetBallInFOV CenterBallInFOV
PassBallGoal	kick_performed	true	ScanForTeammates PassBallToTeammate
InterceptBallGoal	has_ball	true	MoveToBall
GetInMidPositionGoal	in_mid_position	true	MoveToMidPosition
ResetPositionGoal	in_new_position	true	TeleportAction

Defender			
GOALS	StateKeys	Effect	Actions
FocusOnBallGoal	ball_centered_in_FOV	true	GetBallInFOV CenterBallInFOV
ClearBallGoal	has_ball oppo_attacking	false false	MoveToBallAction TackleAction
GetInDefPosition	in_def_position aligned_with_ball	true true	MoveToDefPosition AlignWithBall
ResetPositionGoal	in_new_position	true	TeleportAction

What does the tables above represent? For each GOAL, the target state is represented by the StateKey and its Effect. In other words, when the player selects a goal to achieve, it will plan out a path of actions in order to satisfy the StateKey. For instance, an **Attacker** who choses the Goal **PassBallGoal** will select a path of actions such that **kick_performed = true**. These actions will be (among other) **ScanForTeammates and PassBallToTeammate**.
Note that other actions might be performed while achieving the Goal. This is because some of the Actions might only be available if intermediate actions are performed.

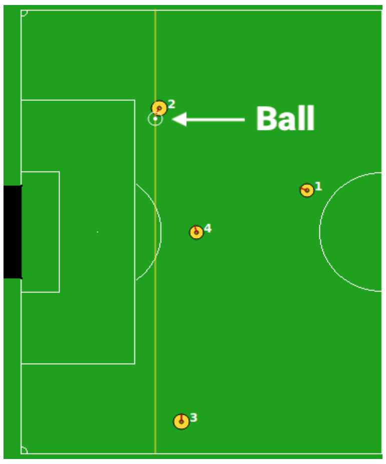
6 Advancement

6.1 State of the Code

All these additions to the code bring us a step closer to a fully realistic soccer simulator. This research project focused on building upon the infrastructure inherited from past researchers, and has succeeded in making instilling realistic behaviour using the GOAP system and Artificial Intelligence.

The code is now capable of utilizing 2 of the 3 sensory (visual, aural) data provided in order to guide behaviour and make game-playing decisions; while a 4^{th} sensory source, that is positioning, has been designed. This means players can rely on more ways to capture information for itself, and broadcast information for others. Not only can players achieve a variety of goals towards scoring, they can now adapt their objectives dynamically, and rely on teammates if opponents are in the way. The code plays with the StateKeys enumerated in part 5.2 in order to adapt behaviour, yielding highly satisfiable and consistent decisions. Additionally, several measures have been taken to prevent recurrent and troublesome issues:

- Clustering: Clustering refers to when players congest all around the ball and stumble on each other. This was most recurrent with Midfielders because of the InterceptBallGoal, and Attackers with KickBallGoal. To avoid this, priority of InterceptBallGoal depends on where the ball is. Midfielders will only go for the ball if it is in the same side of the field as their initial position. Plus, when carrying the ball forward, Midfielders run towards a point above or below the goal, in order to avoid clusters on the middle line.



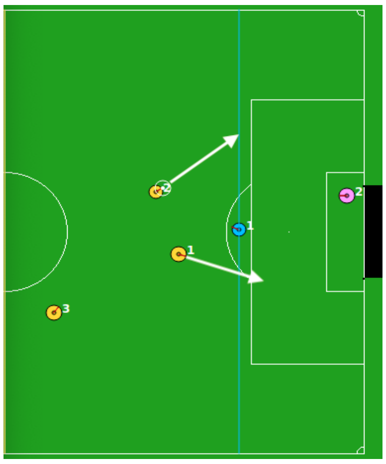


Figure 3: (Left) Only player 2 is intercepting the ball

Figure 4: (Right) Players run towards goal on either side

- Debugging: Several print statements have been left for debugging purposes. StateKeys, Actions, Goals, Plan, and other relevant information can be printed from the *planAndExecute()* method in Player.java. These print statement are not bothersome, they appear on the CLI and can be commented out if desired. They have been critical for debugging and understanding behaviour, therefore I decided to leave them.

- Testers: Several tester methods are available in order to test out different configurations of players. They are useful in order to analyze decision-making in precise conditions.

6.2 Schedule and Deliverables

I started working on this project in September. However, the first month has been dedicated to making it work: documentation is often lacking, incomplete or simply not maintained. This is why I decided to work on Linux, as my attempts on Mac and Windows were unsuccessful. That meant setting up a VM, downloading all dependencies and softwares required. This VM setup would come in extremely handy as I could work on the project from anywhere, however it did fail on me as my disk space ran out towards the end of the project, costing me a few days of development. October was dedicated to understanding the structure of the code, the classes, and planning out my objectives. The remaining 2 months resulted in everything found in part 5 (New Implementations), documentation, and bug fixes. All the work was done by me, and deliverables consist of the updated code, a demo of the various goals, and this report.

7 Future Work

While all these additions allow for pertinent behaviour from player in a game match, many obstacles have prevented me from reaching all the objectives desired initially. Namely, the code is for now not capable of performing all the Goals set to perfection: Midfielders will carry the ball forward until the opponent's box but will not cross the ball for the attacker. This last step could not be implemented correctly because of time constraints.

Found below is a non-exhaustive list of axis of development for future research projects on RoboCup, in no particular order:

- Add more StateKeys, Goals, Actions in order to perform more complex sequences of actions. This would include fixing the issue mentionned above, as well as adding/spliting current Actions and Goals. For instance, the *PassBallGoal* should probably be split into two disctinct Goals: *CrossBallGoal* (Midfielder up on the field spots an Attacker, passes the ball for him to score), and *PassBallGoal* (any player finds a teammate to pass as this would be more beneficial than carrying the ball forward).
- Implement the remaining Actions Models[7], and utilize the TurnNeck action for ball localization for instance, as it may be more efficient than turning the whole body.
- Implement the Body Sensor[8] using the *bodyUpdate(LocalView)* from GlobalMap.java. This would mean the simulator now takes into account stamina, viewQuality, arms, and fouls. This might be a lower-priority challenge however.
- Generalize the system for anti-clustering. The reason it works so far is because it is designed for a system with 2 midfielders, one on each side of the field. Coordinating players such that only one is selected to execute a task is tricky, as information cannot be centralized: it has to transit only through the various sensory detectors. The use of the *doSay(String)* could be the solution, where a player broadcast a message stating that HE and only he will intercept the ball.
- Add more Player types and adapt their behaviours accordingly. Soccer is a complex sport, and there are more positions and roles than GoalKeeper, Defender, Midfielder and Attacker. You will find in the appendix a guide on ALL possible positions a player can take in the game Football Manager 2020. While you are not expected to implement all of them, it might be useful to differentiate a Midfielder from a Winger, as those two roles are completely different.

8 Conclusion

In conclusion this project has successfully leveraged the architecture of the code in order to implement Actions, Goals, Players, and Plans using the GOAP system. It is now possible to reproduce human behaviour seen on a soccer pitch, and achieve several goals such as intercepting the ball, passing, scoring, and tackling. Despite obstacles and slow downs during development, the improvement done across a single semester is promising. As the project advances, it also becomes more complicated due to the simultaneous management of dozens of states, actions, and goals. It may be more adequate for a team of several developers to work on the project, rather than a single one.

References

[1] The RoboCup Soccer Simulator official website. URL https://www.robocup.org/

[2] Melissa Katz and Joseph Vybihal, ROBOCUP Soccer Simulator2D Research Project. URL https://gitlab.cs.mcgill.ca/jvybihal/soccer-tournament/-/blob/dev/Reports/COMP_396___Final___Melissa_Katz__updated.pdf

[3] Jeff Orkin. Applying goal-oriented action planning to games. URL https://alumni.media.mit.edu/~jorkin/GOAP_draft_AIWisdom2_2003.pdf

[4] Raphael Julien, Melissa Katz, Joseph Vybhila, Pedro Khalil Jacob. McGill soccer tournament repository, main branch README. URL https://gitlab.cs.mcgill.ca/jvybihal/soccer-tournament/-/blob/dev/README.md

[5] RoboCup Soccer Simulator Github page. URL https://rcsoccersim.github.io/

[6] RoboCup Soccer Simulator Users Manual. URL https://rcsoccersim.readthedocs.io/en/latest/index.html

[7] RoboCup Users Manual, Action Models. URL https://rcsoccersim.readthedocs.io/en/latest/soccerserver.html#action-models

[8] RoboCup Users Manual, Body Sensor. URL https://rcsoccersim.readthedocs.io/en/latest/soccerserver.html#body-sensor-model

Appendix

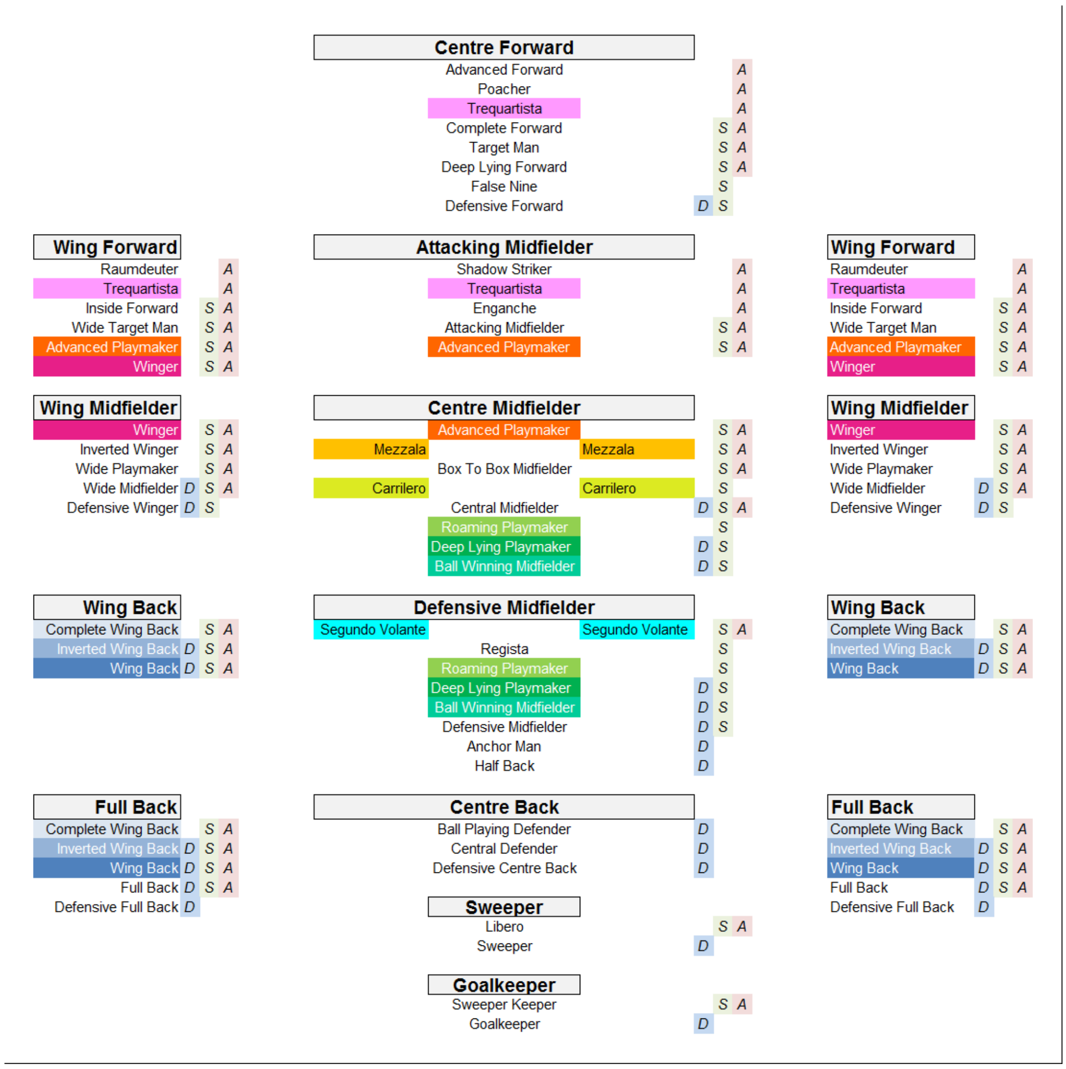


Figure 5: All possible player positions

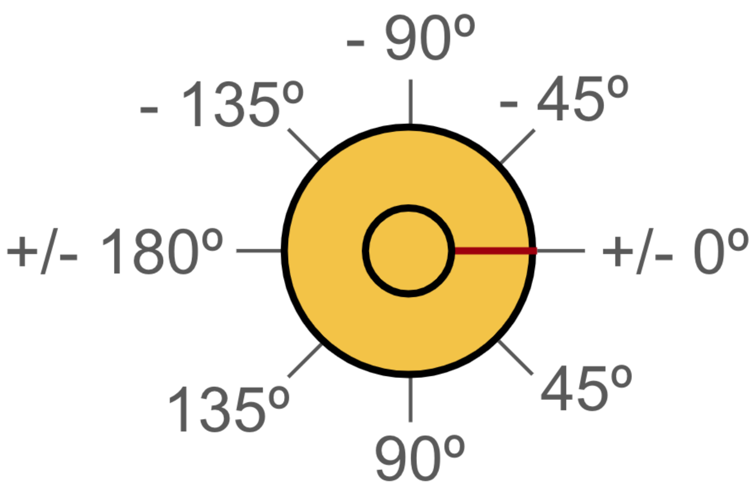


Figure 6: Absolute directions of agents

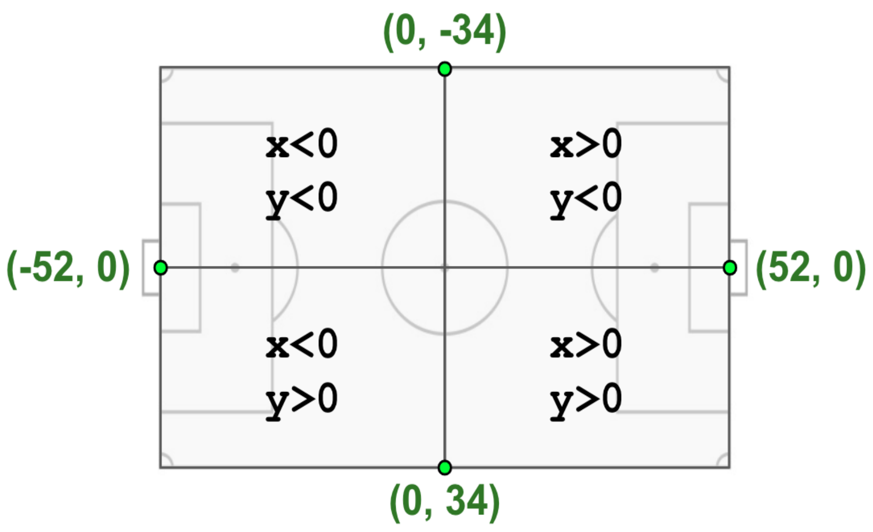


Figure 7: Coordinates system of the simulator