1.Page Title

Project Title: Coursework Report

Student Name: Zakaria Boutarfa P2855274

Course Title: Agent Based Modelling and Parallel Computing (IMAT 3721)

Submission Date: 30/01/2026

2. Abstract

This project presents a minimal 2v2 Rocket League – car soccer video game – simulation implemented as an Agent-Based Model (ABM) in NetLogo. The projects aim to compare and find out which type of player has better chances to win over others. The system models autonomous car agents interacting with a ball inside a pitch with goals, wall and boost resources. Each car is controlled and follows a distinct behavioural archetype between the following: Chaser, Bully, Sentinel and Pro-Rotator. Besides some common behaviours and rules applied to all archetypes, each archetype has a different tactical philosophy such as direct pursuit of the ball, enemy disruption, defensive positioning and rotational decision-making.

To find a right balance between fairness, realism and light portability, the simulation separates simultaneous concurrent decision and movement phases from sequential collision resolution. Car agents think and move using concurrent execution, while collisions between cars and between cars and ball are resolved afterward to maintain physical consistency. During development, multiple stability issues were identified and corrected, including a lot of behavioural and environmental bugs such as wall tunnelling, ghost ball collisions, kickoff deadlocks and more. Additional recovery systems were implemented to cover all possibilities of play such as forced unblocking of the ball, kickoff contest resolution and full agent state resets were implemented.

The model also includes performance tracking of archetypes across matches and supports batch experimentation using BehaviorSpace from NetLogo tools for parallel simulations runs which exploit another form of multi-threading running simulations simultaneously on different cores. Results show that different archetype combinations produce distinct tactical dynamics, validating the usefulness of modular behavioural design in multi-agent sports simulations.

4. Introduction

4.1 Problem Statement

Multi-agent coordination and competition are central challenges in many real-world systems, including robotics, autonomous vehicles, team sports analytics, and distributed AI. These scenarios require multiple independent actors to make decisions simultaneously while interacting through shared environments, resources, constraints and goals. This project is a perfect example of these scenarios as it simulates a type of sport which is football in a different simplified situation with players being cars, and a different environment and boost pads that are a resource that improves an agent (car) situational condition. Inspired by Rocket League, autonomous agents aim to win a football game, by interacting with each other through the restricted area of play, while having defined roles that allow them to defend, attack, grab boost. This simulation is a convenient way to test concurrent behaviours and decisions in a competitive environment.

4.2 Motivation and Objectives

Besides its game-like setting that motivates me to choose this specific simulation idea, I believe this problem is worth modelling because it includes different areas of AI in a controlled environment which are decentralised decision-making, distributed agents' roles and real physics interactions. Also, with a clear objective and a simple concept, it makes comparison of agents and their observations easier while still complex enough to highlight tactical difference, coordination failures and emergent dynamics

The main objectives of my project are:

- To successfully model a functioning simulation of a competitive match with working, relevant and well-thought physics, scoring and resource systems.
- To create different unique archetypes of players that sense the world and act differently, which will produce an important number of combinations to experiment in order to find out which combinations and archetypes are better within the experiment conditions.
- To make a stable simulation that considers quite complex physics like collisions between different type of agents and with the environment without creating bugs, agents freeze or deadlocks.
- To implement a relevant way to track archetypes performance across matches and within same games combinations

.

4.3 Importance of Multi-Threading

4.3.1 How lack of multi-threading could have spoiled the whole simulation

One of the most important parts of this project is the implementation of multi-threading and real-time decision-making. Implemented mainly through concurrent agent execution, multi-threading is essential in this model and more generally in competitive simulations for fairness purposes, realism, and scalability. In a fully sequential update order, agents processed earlier each tick are favourited in reacting to the environment leading to biased outcomes, for example in my simulation the car agent with the lowest “who” label (car number 0) would always move first and have an advantage over the others especially in perfectly even scenarios (kickoff case). On top of that, in a competitive simulation with only four agents even small ordering can significantly alter results over many matches.

4.3.2 How Multi-threading was handled

To address this, the model uses concurrent execution for the decision and movement phase of all car agents allowing them to think and act in the same tick without fixed ordering priority. However, collisions and different physics are handled sequentially to keep consistency, this approach improves behavioural equity and fairness while securing stable physics.

While most of the multi-threading was handled by the concurrent execution of agents thinking and acts, large-scale experimentation is supported through batch runs using BehaviorSpace, which distributes simulations across available CPU cores. This enables hundreds of matches to be evaluated in parallel and in couple of seconds which would have taken hours to do manually, and without requiring GPU-level parallel programming.

4.4 Project Scope

4.4.1 Project Scope and content

The scope of the project is a two-dimensional, discrete-time simulation of a Rocket League-inspired 2v2 match implemented in NetLogo. The model includes autonomous car agents, a moving ball with simplified physics, boost resources with cooldowns, goal detection, scoring and tole-based behavioural logic. The system also includes agent state memory, collision handling between cars and ball, recovery mechanisms for stuck states, and performance tracking of archetypes across matches.

4.4.2 Assumptions and limitations

Multiple simplifications and assumptions are made. The physics model is intentionally abstracted: cars use scalar speed and heading rather than full rigid-body dynamics, and collisions use impulse approximations rather than continuous contact simulation. The most important abstraction is the choice of a 2D environment which hugely restraints the model performance, this choice excludes one of the most important features of the original game which is the ability to jump. Tactical reasoning is rule-

based rather than learned, and communication between teammates is implicit rather than message-drive. GPU acceleration is not used; instead, CPU parallel batch execution is applied for large experiment sets.

Within these boundaries the project focuses on behavioural design, concurrency control, stability engineering and comparative agent performance rather than graphical and full physical realism.

5. Methodology and Design

5.1 System Overview

The system is designed as an Agent-Based Model in NetLogo simulating a simplified 2v2 Rocket League-style match. The simulation environment consists of a rectangular 100x80 patches pitch with side lines that make each side (orange vs blue) easily recognisable and goals in each side's colour. The pitch also features wall boundaries and green grass in different shade for better visuals; there is also six big boost resource pads and multiple smaller boost pads and a moving ball governed by simplified physics (image 1). The system evolves in discrete time steps (ticks), where all active agents perceive, decide and act according to their behavioural rules.

These main agent categories are defined:

- Car agents: autonomous decision-making entities assigned to teams and behavioural archetypes.
- Ball agent: a dynamic physics-driven object with velocity components and collision handling.
- Boost pad agents: resource nodes that provide temporary speed advantage and regenerate after cooldown.

The ball agent follows a separate physics update process based on velocity vectors, friction, wall restitution, and goal detection checks. Collisions between cars and the ball are handled using vector projection and overlap correction to avoid penetration and ghost contacts.

Boost pad agents work as timed resources. When collected by a nearby car, they increase boost reserves and enter a cooldown state before becoming available again. Boost pad agents exist in two forms

- Big boost pads: there are six of them in the pitch and give car agents full boost reserve (100) and have a cooldown of 100 ticks (which represents ? seconds) they are targeted by car agents when they decide to get boost.
- Small boost pads: There are more of them in the pitch and give car agents only an additional 12 to their boost reserve and have a smaller cooldown of 30 ticks (? seconds) , they are not targeted by car agents to get boost but rather serve as available boost that car agents get passively during their movements.

Car agents are the core intelligent actors. Each car maintains internal state variables including team identity, archetype, boost level, speed, turn rate, behavioural mode, and temporary status flags such as stun timers and unblock targets. Each tick, cars

execute structured decision-action cycle: they update internal state, select a behavioural procedure based on archetype, adjust heading smoothly, accelerate within limits and move through the environment.

In addition to their specialised behaviours, all car agents share a set of common action rules executed every tick before of after archetype-specific logic. This ensures a consistent behavioural baseline across all roles.

Common Actions:

- Stun and Recovery handling: If an agent has recently been bumped by another car agent, a stun timer temporarily disables steering decisions. During this phase the car continues moving forward with reduced control until the timer expires.
- Boost Consumption Model: Boost is consumed dynamically and proportionately to the speed of the car agent. Faster movements result in higher boost drain. Boost values are restricted between a minimum and a maximum to prevent negative or overflow states
- Boost Collection Rules: Gain in boost depending on the boost pad type and triggering the cooldown of boost pads.
- Speed and Movement constraints: maximum base speed, acceleration rate, smooth turning based on turn-rate function.
- Wall collision handling with heading reflection and speed reduction

Decision-Action Structure:

Each tick follows a standard pattern: state update then archetype decision then acceleration then wall collision check.

This keeps behaviour modular while allowing archetype logic to override navigation targets.

Archetype-Specific Actions and Rules

Chaser: Always prioritise ball pursuit and boost over tactical positioning

Special actions:

- Switch between ball mode and boost mode depending only on boost level.
- Attempts to approach the ball from the correct side to avoid own-goals.
- Uses centre-field waypoints when near walls or geometrically stuck.
- Includes wall-proximity escape logic and anti-circle detection behaviour.

Bully: Target and collide with opponents, completely ignoring the ball.

Special actions:

- Selects an opponent target and maintains pursuit for a fixed number of ticks.
- Uses aggressive acceleration and higher speed caps when enough boost is available.
- Consumes boost deliberately to maximise bump impact.

- Uses midpoint interception routing when the target is in the opposite half
- Goes to get boost when boost level is under a certain level and only goes for the ball accidentally.

Sentinel: Protect own goal zone and clears ball when threat level is high.

Special actions:

- Computes a shadow position between own goal and the ball (fractional interpolation)
- Enters chase mode when the ball is inside a defined radius in its own half.
- Returns toward centre if overextended in opponent half due to exception trigger.
- Maintains controlled low speed when holding defensive position
- Boost collection is conditional and secondary to defensive positioning

Pro-Rotator: Decide dynamically whether to attack or support based on teammate actions.

Special Actions:

- Uses a priority reporter to determine whether it is “first man” or “second man”
- Applies a tiebreaker using distance thresholds and agent IDs
- Override priority if teammate is wall-stuck.
- Switches behaviour between direct challenge and defensive/ attack support.
- Conditional boost detours when needed, second man and ball is far.

The system is therefore structured as an interaction loop between autonomous tactical agents, a shared physical object and a resource-constrained environment.

5.2 Multi-Threading Strategy

Concurrency is integrated at the agent decision and movement stage to ensure fairness and reduce ordering bias. In a purely sequential agent update model, agents processed earlier each tick would consistently react first, creating systematic advantage. This is particularly problematic in competitive simulations where timing differences strongly affect outcomes.

To address this, the simulation uses concurrent execution for car agents’ decision and movement phase All cars evaluate their behavioural logic and perform movement updated within the same tick using concurrent scheduling. This means perception, decision and movement are computed without a fixed agent priority order.

After this concurrent phase, the model switches back to sequential resolution for interaction-heavy operations: car-car collisions, car-ball collisions, ball physics update, goal detection, boost cooldown updates.

This produces a hybrid model: Concurrent phase for independent agent thinking and motion, where it matters. And Sequential phase for shared physics and collision resolution. This separation is intentional. Decision logic is parallelisable because it depends on current state observations, while collision resolution must be sequential

to maintain physical consistency and avoid race conditions in position and velocity updates.

For large experimental runs, batch simulations are executed using NetLogo's BehaviorSpace tool, which distributes independent simulation runs across available CPU cores. This provides coarse-grain parallelism across matches rather than within a single match step.

5.3 Design Choices

Several key design choices were made to balance behavioural clarity, simulation stability, and implementation feasibility. The system prioritises modular agent behaviour, controlled interaction rules, and simplified but robust physics over full realism. These decisions were taken to keep the model explainable, testable and suitable for controlled experimentation.

5.3.1 Behavioural Archetypes Instead of a Single General AI

Rather than implementing one general adaptive controller, the system is built around four fixed behavioural archetypes. Each archetype represents a distinct tactical philosophy and decision policy. This was chosen for its easy implementation and design, its simplicity in debugging and explaining. Also, it enables controlled comparison between tactical roles, and it allows team composition to become an experiment variable, which was the whole purpose of the coursework.

On top of that, the choice of the four archetypes is thoroughly thought. As the four archetypes are divided into two functional categories:

1. Active ball-pressure roles: Chaser and Pro-Rotator
2. Passive roles: Sentinel and Bully.

It creates natural role complementary within teams and brings a balance and a diversity to games.

5.3.2 Environment and Physics Rules

The environment is modelled as a 2D bounded pitch with discrete ticks and simplified physics. This was a deliberate abstraction choice due to available hardware and a very modest GPU.

A full 3D physics simulation with aerial mechanics, rotations and continuous collision detection would significantly increase implementation complexity and instability risk. Since the primary objective of the project is behavioural and architectural a 2D model was selected.

5.3.3 Interaction Rules Between Agents

Agent interaction rules were designed to be locally reactive but globally stable. Instead of complex negotiation or messaging systems between teammates,

coordination emerges from rule structure and priority logic. Key interaction design choices include:

- Car-car collisions are treated as meaningful tactical events rather than noise. The bully archetype relies on this to influence play without ball contact.
- A special interaction rule resolves between simultaneous ball contact at kick-off to prevent deadlocks and infinite centre pinches
- Role separation to avoid multi-agent convergence, not all agents are allowed to follow the same interaction rule set which creates an interaction diversity to prevent all agents chasing the same object simultaneously, for example the ball.

5.4 System Architecture

5.4.1 Architecture Overview

I built the simulation flow around the main tick loop triggered by the go command after I ran into execution conflicts that were hard to trace. From there I enforced a strict order of operations per tick. Cars evaluate their behaviour and move first. Interaction handling comes next. Ball physics follows. Match variables update at the end. I split these stages deliberately to stop state writes from stepping on each other. Agent decisions run together, shared consequences run later. That ordering did not come from theory. It came from watching earlier versions break when behaviour code and physics updates fired in the same step and produced inconsistent motion and scoring states.

Environment Layer

I defined the environment through global variables and setup procedures that build the pitch, walls, goals, goal mouth zones, and boost pad locations. I call all of this in the setup stage and most of it stays unchanged during play. Only the boost pads change state because of cooldown. Also, I use these environment values in every movement and collision check, so cars stay inside bounds and goals get detected correctly. There is no intelligence in this part. It only defines space, limits, and resource points that influence how cars choose where to go.

Agent Behaviour Layer

I grouped all car decision rules inside a single control procedure named car decide and act, then I branch from there toward the chosen archetype logic. Every tick, each car passes through that same entry point. First I process stun timers, then resource and speed changes, then movement limits. Only after those checks do I call the archetype routine such as decide chaser or decide pro rotator. Target selection and steering happen in those modules, while collision handling stays elsewhere. I made that split after earlier versions tangled behaviour with contact code and debugging turned messy fast. If you inspect this layer in isolation, you read motives and movement plans, not impact handling.

Interaction Layer

I handle all contacts between agents and between cars and the ball in a dedicated interaction step after movement. This includes car to car bumps, car to ball impacts, stun timers, and the special kick off contest rule. I run this part in sequence so position and speed updates stay consistent. Earlier versions tried to resolve contacts

during movement and I saw double pushes and overlaps. Moving all interaction rules into one place fixed that. These procedures change heading, speed, and position based on contact formulas I defined.

Physics Layer
I update the ball using a simple physics model with velocity on x and y. Each tick I reduce velocity with friction, predict the next position, check if it crosses a goal, and then handle wall bounces and boundary limits. I also added a small kick when the ball almost stops near a wall because I observed it getting stuck there. I run physics after interaction so collision impulses get applied before the ball moves again. Cars do not use full physics here, only the ball does.

Control and State Layer
I manage match level state at the end of each tick. This includes the match timer, score values, goal flags, resets after goals, and label updates. I also clear agent memory here after a goal, such as targets and timers, because I saw agents keep old intentions into the next kickoff. This layer coordinates resets and phase changes so each new play segment starts clean. It sits at the bottom of the loop and reacts to what already happened in the tick instead of interfering with movement or collisions.

6. Implementation

6.1 Code structure

My code starts by the first block of code which contains declaration of global variables that relate to environmental parameters and different control variables, creation of the different types of agents we will later have such as balls, cars and boost pads and what they own. This is followed by the setup which called different helpers procedures that mainly put into place each field component: nets, pitch markings, boost pads and the ball. We also find in the setup the initialisation of the different control variables such as the scores and the match timer. At the end of the first block of code, we find the helpers procedures used in setup.

The second block of code includes the main procedure used for the simulation. It includes the go procedure which is responsible for setting up the simulation limit through the match-timer variable. It also features the main multi-threading part of the project which is the concurrent execution embodied by the ask concurrent command, which contains the next block within it. The procedure also contains the call for the procedures that handle the car-car collisions and the car-ball-collisions, which are sequential to ensure system stability. In this block and within the same function we find boost resets and goals logic procedures call and the ball physics in the play procedure.

In the third block called from within the second one, we find one of the most important procedures of my code and which is responsible for car agents' behaviour. This procedure named car-decide-and-act starts by implementing the common rules for all archetypes which are the bump logic, the boost consumption and collect logic. Then

there is the specialisation of our car agents' behaviours according to the archetype, for each archetype a procedure named decide plus archetype-name that is detailed at the end of this block of code with its helpers. Before the archetypes procedures and their helpers, we find another set of common actions between agents which are the acceleration after making a decision tailored to the archetype, and finally to close the procedure we check the collision between car agent and the wall to prevent any wall due bugs.

In the last block of code, we find some more procedures that help manage stuck situations such as infinite circle or car agents trying to go through the wall for shorter paths. At the very end of the code, we find the procedure that manage the end of the match which displays the winning team, its archetypes and updates global archetypes scores across matches.

6.2 Key code sections

The 'go' procedure represents the core procedure that manages every part of the play in the desired and defined order. As mentioned above, the procedure starts off by setting 'match-timer' variable functioning which is time elapsing and prepares for the end of match happenings, then comes the concurrent execution of car agents section that involves car friction, shared actions by entering 'car-decide-and-act' and routes all the way through performing car wall checks and dynamic boost consumption and collection logic to the crossroads of archetype behaviours procedures which are 'decide-chaser', 'decide-bully', 'decide-sentinel', and 'decide-pro-rotator' in which targets, state modes and steering choices are set or altered helped by sub-procedures. After the concurrent phase and back into the 'go' procedure, the systems perform interaction checks through designated handlers named 'handle-car-car-collisions' and 'handle-car-ball-collisions' that are ran sequentially which include exceptions such as overlap correction and special kick-off contest to prevent early loops of infinite goals. The 'play' procedure then proceeds to handle the ball interactions with the wall bounce, boundary clumping and goal detection and triggers related to it. At the very end of the 'go' procedure sits the implementation of the variable responsible for triggering 'unblock-game' a procedure that handles game stuck exception and a procedure that updates car labels and the boost level UI. Together these sections form the operational core of the code and define how one full simulation tick is executed.

6.3 Error handling and stability handling

While coding and creating my system, I encountered a number of bugs and errors, some of which I corrected, while others are handled by recovery mechanisms that feel unnatural. The first error handling is the case of a ball being too close or clustered into a wall, and car agents cannot access it because of simplified car agents physics and a rigid marge of turn-rate changes. A solution to this problem was to implement an impulsion in that case within the 'play' procedure for the ball to get into a more accessible area of the field. Another error was also due to car agents turn-rate issue in a specific case of 4 chasers, it would happen that the car would keep orbit around the ball. In Order to fix that, we implemented the 'unblock game'

procedure which picks randomly a victim car that breaks its current state and task in order to unblock the situation by challenging the ball. The last and most persistent error I encountered, was the players that would have a target x and y coordinates and chose the closest way to arrive to the target regardless of an existing wall on its way, which kept the agent to try to penetrate through the walls to arrive from the opposite side. One idea I had was to make them bounce off those walls and changing directions so they would face the right direction through the pitch to the target (whether it was the ball, a car agent, the boost ..etc) however, that did not work out as car agents kept turning back towards the wall. The solution I implemented is to force players in some conditions to go through the middle point of the field for the target to be closest to any other wall. As this solution worked out pretty well, it remains a persistent bug for moving agent targets (Bully case). Minor errors and bugs appear also as syntax error which are a lot easier to fix.

6.4 Optimisation

Most optimisation steps in my model came from watching where the simulation slowed down or produced unstable motion, then simplifying those parts instead of adding more logic. I reduced repeated calculations inside tight loops, especially in collision code. For example, in the car–ball collision handler I store the ball position in local variables before looping over cars, then I use ‘distancexy‘ with those cached values instead of repeatedly querying the ball agent. That made the collision block easier to read and reduced repeated lookups each tick.

I also avoided heavy operations inside patch loops during runtime. The pitch, markings, and zones are drawn once in setup and not recomputed every tick. Only dynamic elements such as cars, the ball, and boost pad cooldown states update during play. This keeps the per-tick workload limited to moving agents and interaction checks instead of repainting the environment.

Another optimisation choice was structural. I separated concurrent car updates from sequential collision and physics updates. When I tested mixed updates in one block, I saw repeated corrections and unstable overlaps, which increased the amount of corrective work per tick. By keeping decision and movement concurrent, then resolving contacts once in sequence, I reduced redundant state rewrites.

For large experiment runs, I rely on BehaviorSpace batch execution across CPU cores instead of adding complexity inside the model itself. That shifts scaling to the run level rather than the tick level. I did not add low-level performance tricks beyond this, since keeping the code readable and stable mattered more than pushing maximum tick speed.

6.5 Challenges faced

During implementation I ran into several issues that came from agent ordering, collision maths, and state persistence. One early problem came from sequential execution of cars, where the same agent kept gaining first contact advantage in symmetric situations like kick off. I changed the car update step to concurrent

execution, so decisions and movement happen in the same tick window for all cars, which removed that ordering bias. Another difficulty appeared in collision handling. At first, cars overlapped and the ball sometimes passed through them because the contact distance formula used incorrect radius and mathematical formula and some direction variables always evaluated to zero. I corrected the distance and normal vector calculations and added overlap push-out so intersecting agents get separated after contact. Another challenge came from behavioural memory. After a goal, some agents kept old targets and timers, which produced incoherent behaviour in the next phase. I fixed this by resetting targets, modes, and timers inside the post-goal reset procedure for every car.

7. Experiments and Evaluation

I ran a set of controlled match batches using BehaviorSpace with fixed parameters and repeated compositions. Each configuration used the same field, same physics, same boost rules, and the same match timer. I only changed the archetype composition. For each setup I ran multiple repetitions and recorded the winning team through the binary win reporters. I focused on win rate rather than raw goal count because it stayed stable across short matches and gave a direct comparison signal.

Across the runs, the pro-rotator composition produced the highest win rate. When I paired two pro-rotators together, or mixed pro-rotator with a support archetype, I saw more consistent control of ball flow and fewer wasted challenges. The pro-rotator logic that checks role priority and distance to the ball reduced double commits and improved recovery after wall rebounds. I saw fewer stalled plays and fewer own-goal situations in those matches.

Sentinel and bully produced similar global results when used as support roles. Both improved team stability compared to double chaser teams. The sentinel held defensive space and reduced easy goals. The bully created disruption through bumps and forced path changes. In direct comparison, bully lineups produced slightly higher win counts than sentinel lineups. The difference showed up in midfield collisions and kickoff contests where the bully altered opponent movement more often than the sentinel.

Chaser based teams ranked last. Double chaser compositions overcommitted, stacked on the same path, and left open nets after rebounds. I observed repeated corner traps and circular chasing near walls. Even when chasers scored early, they conceded more goals later in the same match due to poor rotation and boost usage.

From these runs I ranked the archetypes by match impact as follows. Pro-rotator first. Bully next. Sentinel close behind bully. Chaser last. This ranking matches what I observed during live runs and replay checks, not only the win rate table.

8. Conclusion

I built this model to test how different behavioural archetypes interact inside a competitive agent based match setting. I structured the system around separate behaviour, interaction, and physics steps because early mixed versions produced unstable contacts and inconsistent movement. After I split those responsibilities, the simulation stayed predictable enough to run repeated batches.

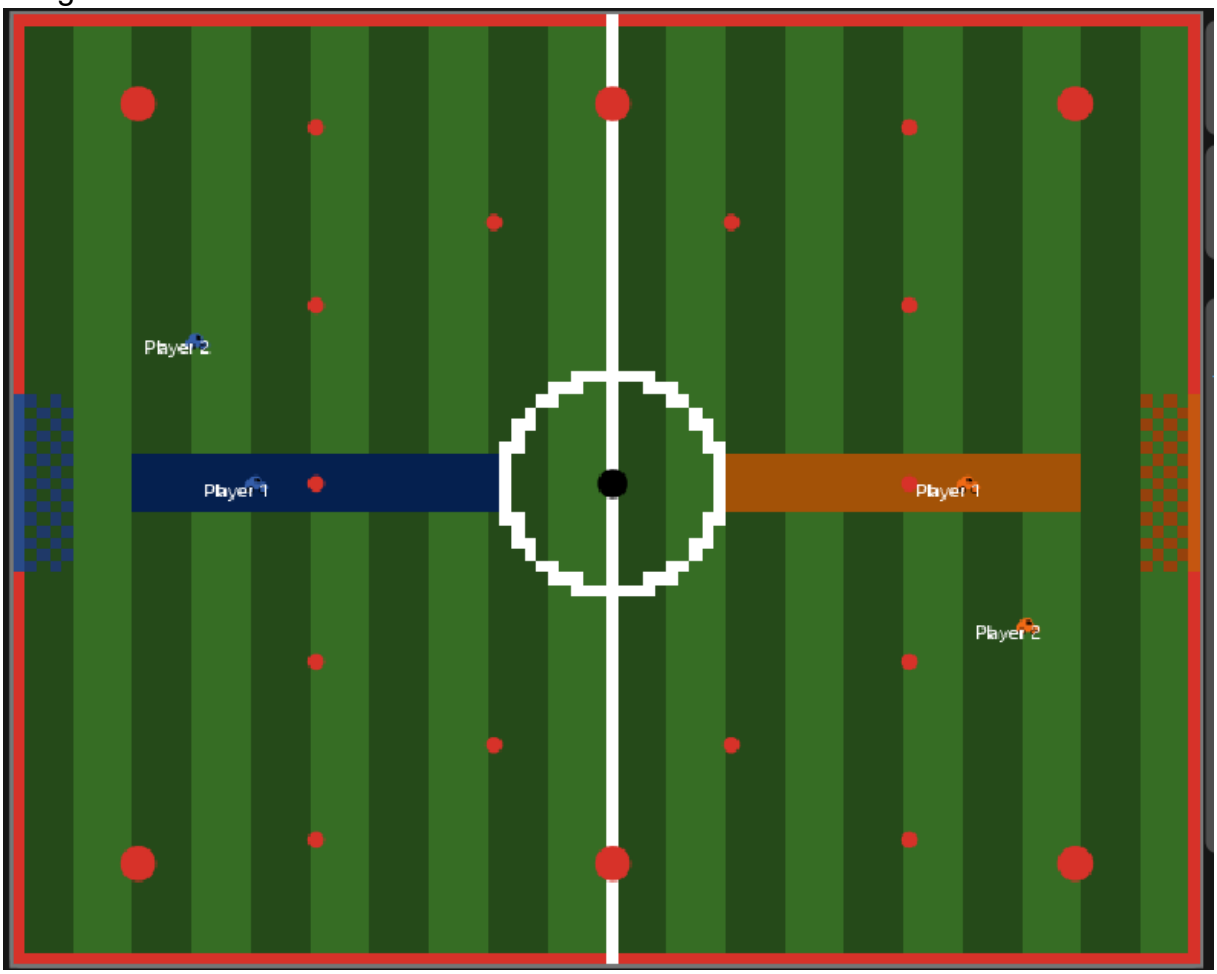
From the experiments I ran, the pro-rotator archetype produced the strongest results. Teams that included it showed better role distribution and fewer positional conflicts. I saw fewer double commits and cleaner recovery paths after rebounds. The bully and sentinel filled similar support roles. Both improved team balance compared to purely aggressive lineups, although the bully produced slightly better outcomes due to its direct interference with opponents. The chaser ranked last. Its direct ball pursuit rule looked active but often broke team spacing and exposed the goal.

There are limits in what I measured. I reduced match length to speed up batch runs and I focused on win rate instead of deeper event metrics. I did not model 3D physics, spin, or aerial control, so some behaviours stayed simplified. Even with those limits, the runs still separated archetype quality clearly enough to compare them.

If you extend this model, you should add richer ball physics, role memory across possessions, and event logging per tick. That would let you measure rotation errors, possession time, and challenge success instead of only final wins. The current version still works as a controlled testbed for behavioural agent design and interaction rules in a small competitive system.

Appendix

Image 1:




```
to setup
clear-turtles clear-patches  clear-drawing   clear-output ; clear all besides the performance plot.
setup-constants draw-pitch draw-side-bands-diagonal draw-center-markings draw-goals-and-nets place-big-boostpads place-small-boostpads
set ticks-since-last-touch 0 set match-timer 150 set blue-score 0 set orange-score 0
ask balls [ die ]
create-ball place-cars-2v2 label-cars assign-2v2-archetypes blue-arch-1 blue-arch-2 orange-arch-1 orange-arch-2
set start? true

tick-advance 0
end
```

```
to create-ball
ask balls [ die ]
create-balls 1 [setxy 0 0 set shape "circle" set size 2.5 set color black set vx 0 set vy 0 ]
end
to go
;; --- 1. CHRONOMÈTRE ET FIN DE MATCH ---
if match-timer > 0 [set match-timer match-timer - 0.05]
if match-timer <= 0 [set match-timer 0 display-final-results stop]
;; --- 2. ACTIONS INDIVIDUELLES (MULTI-THREADING) ---
;; On utilise ask-concurrent pour que les 4 voitures réfléchissent et
;; bougent en même temps, évitant que le Player 1 ait toujours l'avantage.
ask-concurrent cars [
;; Friction (Rolling resistance)
set speed speed * 0.98
if speed < 0.01 [ set speed 0 ]
;; IA + Mouvement + Check Mur (Tout ce qui est individuel)
car-decide-and-act
check-car-wall-collisions
]
;; --- 3. COLLISIONS INTER-AGENTS (SÉQUENTIEL) ---
;; On traite les bumps APRÈS que tout le monde ait bougé pour une physique juste.
handle-car-car-collisions
handle-car-ball-collisions
;; --- 4. GESTION DES BOOSTS ---
ask boostpads [
if cooldown > 0 [
set cooldown cooldown - 1
if cooldown = 0 [ set color red ]
]
]
ask small-boostpads [
if cooldown > 0 [
set cooldown cooldown - 1
if cooldown = 0 [ set color red ]
]
]
;; --- 5. PHYSIQUE DE LA BALLE ET BUTS ---
```

```
;; --- 5. PHYSIQUE DE LA BALLE ET BUTS ---
if start? [
set start? false
set goal-scored? false
]
play ;; Déplacement de la balle (friction/rebonds)
;; On gère les collisions balle/voiture APRÈS le mouvement
;; Utilise ici ta version "Multi-hit" pour éviter les balles fantômes

if goal-scored? [
update-score
reset-after-goal
]
;; --- 6. SYSTÈME ANTI-BLOCAGE ET UI ---
set ticks-since-last-touch ticks-since-last-touch + 1
if ticks-since-last-touch > 200 [ unblock-game ]
update-car-labels
tick
end
to update-score
;; On incrémente le score selon le côté où la balle est entrée
if goal-side = "left" [ set orange-score orange-score + 1 ]
if goal-side = "right" [ set blue-score blue-score + 1 ]
;; Affichage dans la console pour le debug (utile pour ton rapport)
print (word "BUT ! " scoreboard)
end
```

```
to play
let friction 0.90 ; multiply speed each tick (lower = more slowing)
let restitution 0.70 ; bounce energy retention (lower = “softer” bounces)
let stop-eps 0.01 ; stop when speed is tiny
ask balls [
; ---- apply friction (slow down over time) ----
set vx vx * friction
set vy vy * friction
; Si la balle est "quasiment" dans le mur, on lui donne une impulsion vers le centre
;; --- LE "KICK" DE SÉCURITÉ (Balle à l'arrêt contre un mur) ---
;; On vérifie si la vitesse est quasi nulle
if (abs vx < stop-eps and abs vy < stop-eps) [
;; 1. Mur Latéral (Gauche/Droite)
if abs xcor > 47 [
;; On donne une impulsion fixe de 3.0 vers l'intérieur
set vx (ifelse-value (xcor > 0) [ -3.0 ] [ 3.0 ])
set vy (random-float 2 - 1) ;; Petit angle aléatoire pour le réalisme
print "SYSTEM: Balle à l'arrêt au mur latéral -> Projection !"
]
;; 2. Mur Haut/Bas
if abs ycor > 37 [
set vy (ifelse-value (ycor > 0) [ -3.0 ] [ 3.0 ])
set vx (random-float 2 - 1)
print "SYSTEM: Balle à l'arrêt au mur haut/bas -> Projection !"
]
]
;; --- CALCUL DE LA NOUVELLE POSITION ---
; stop completely if very slow (prevents endless micro-movement)
if (abs vx < stop-eps and abs vy < stop-eps) [
set vx 0
set vy 0
stop
]
; ---- predict next position ----
let nx (xcor + vx)
let ny (ycor + vy)
let side goal-side-crossed nx ny
if side != "none" [
set goal-scored? true
set goal-side side
```

```
;; --- CALCUL DE LA NOUVELLE POSITION ---
; stop completely if very slow (prevents endless micro-movement)
if (abs vx < stop-eps and abs vy < stop-eps) [
set vx 0
set vy 0
stop
]
; ---- predict next position ----
let nx (xcor + vx)
let ny (ycor + vy)
let side goal-side-crossed nx ny
if side != "none" [
set goal-scored? true
set goal-side side
die
stop
]
; ---- bounce on vertical walls (left/right) ----
if (nx <= pitch-min-x or nx >= pitch-max-x) [
set vx (- vx) * restitution
; clamp inside so it doesn't get stuck outside
set nx max list (pitch-min-x + 0.1) (min list (pitch-max-x - 0.1) nx)
]
; ---- bounce on horizontal walls (top/bottom) ----
if (ny <= pitch-min-y or ny >= pitch-max-y) [
set vy (- vy) * restitution
set ny max list (pitch-min-y + 0.1) (min list (pitch-max-y - 0.1) ny)
]
; ---- update position ----
setxy nx ny
]
end
```

```
end
to car-decide-and-act
;; --- 1. GESTION DU STUN (BUMP) ---
;; Si on vient de se faire percuter, on "glisse" sans pouvoir tourner
if unblock-timer > 0 [
set unblock-timer unblock-timer - 1
fd speed ;; On continue sur la lancée du choc
check-car-wall-collisions
stop ;; ON S'ARRÊTE LÀ : L'IA ne peut pas reprendre le volant
]
;; --- 1. CONSOMMATION DYNAMIQUE ---
;; Plus on va vite, plus on consomme.
;; Le coefficient 0.4 permet d'avoir une consommation équilibrée.
if speed > 0.1 [
let consumption (speed * 0.4)
set boostInCar (boostInCar - consumption)
]
if boostInCar < 0 [ set boostInCar 0 ]
if boostInCar > 100 [ set boostInCar 100 ]
let effective-max-speed ifelse-value (boostInCar > 0) [ max-speed * 1.6 ] [ max-speed ]
;; --- 2. COLLECTE DES GROS BOOSTS (Si rouge) ---
let big-b one-of boostpads with [color = red] in-radius 4
if big-b != nobody [
set boostInCar 100
ask big-b [
set color grey ;; Devient gris (indisponible)
set cooldown 100 ;; Temps avant réapparition
]
]
;; --- 3. COLLECTE DES PETITS BOOSTS (+12 si rouge) ---
let small-b one-of small-boostpads with [color = red] in-radius 2
if small-b != nobody [
set boostInCar (boostInCar + 12)
ask small-b [
set color grey ;; Devient gris
set cooldown 30 ;; Les petits reviennent plus vite
]
]
```

```
]
;; --- 2. LOGIQUE SPÉCIFIQUE À L'ARCHÉTYPE ---
if archetype = "chaser" [
decide-chaser
]
if archetype = "bully" [ decide-bully ]
if archetype = "sentinel" [ decide-sentinel ]
if archetype = "pro-rotator" [ decide-pro-rotator ]
;; --- 3. ACTIONS COMMUNES (Après la décision) ---
if speed < effective-max-speed [ set speed speed + accel ]
;; Avancer et gérer les murs (valable pour tous)
fd speed
check-car-wall-collisions
end
to decide-chaser
;; Logique de gestion de l'état (Ball vs Boost) spécifique au chaser
ifelse boostInCar < 40 [
if chaser-mode != "boost" [
set chaser-mode "boost"
set target-boost nobody
]
go-get-boost
] [
set chaser-mode "ball"
go-chase-ball
]
end
```