A Multi-Agent System for a Robotic Table Soccer Game

Gianluigi Forte[1], Marco Pometti[2], Domenico Massimo Porto[1], Corrado Santoro[3], Federico Fausto Santoro[3] and Rosa Zuccarà[3]

[1]*STMicroelectronics System Research and Application 95121, Catania, Italy*
[2]*Dipartimento di Ingegneria Elettrica, Elettronica ed Informatica, University of Catania, Italy*
[3]*Dipartimento di Matematica e Informatica, University of Catania, Italy*

Abstract

The paper describes a demo application, that is going to be developed by our research group in collaboration with STMicroelectronics, that consists of a robotic table soccer game aimed at letting a human play soccer against an electronic system. The main feature of the system is that it is entirely composed of boards equipped with microcontrollers (rather than complete PCs) and thus it is demonstration of how a complete intelligent/robotic application can be implemented with such kind of devices. The system is composed of mechanical, electronics and software parts and has been designed ad-hoc to meet the requirements of a table soccer game. A vision system is employed to recognise and track the ball, while a set of eight BLDC (brushless) motors are used to drive the translation and rotation of the poles. The overall system is controlled by a set of interacting agents that, altogether, implement the intelligence driving the playing strategy of the robotic opponent. The paper describes the complete hardware/software system also providing some performance data that help to understand the effectiveness of the designed application.

Keywords

multi-agent systems, robotics, computer vision, EtherCAT

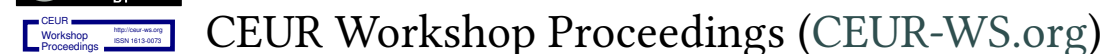
CEUR-WS.org/Vol-3735/paper_16.pdf

1. Introduction

Computer games have emerged as a dominant force in contemporary entertainment and culture, captivating millions of players worldwide. In such games the human counterpart is a software that, exhibiting a sort of “intelligence”, tries to oppose to the human player in a certain way by making moves that are driven by a precise strategy. Often computer games have several (virtual) opponent characters that interact with the environment and sometimes also to one another. In this sense, a computer game can be seen as a multi-agent system in which the opponent(s) are pieces of autonomous software that *(i)* sense the environment and the moves of the human, *(ii)* determine the actions to be done, according to a precise strategy, and *(iii)* execute actions onto the environment.

While the majority of computer games are made of a virtual environment that massively

WOA 2024: 25th Workshop "From Objects to Agents", July 8-10, 2024, Forte di Bard (AO), Italy
gianluigi.forte@st.com (G. Forte); marco.pometti93@gmail.com (M. Pometti); massimo.porto@st.com (D. M. Porto); santoro@dmi.unict.it (C. Santoro); federico.santoro@unict.it (F. F. Santoro); rosazuccara@outlook.com (R. Zuccarà)


© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

exploits computer graphics (CG), there are also computer systems able to play games *in the real world.* In some cases, such a kind of systems are harder to build than CG games because they must take into account the dynamics and the behaviour of the real-world part that is used. Moreover, the interaction needs the presence of some *actuators* (i.e. motors) and *sensors* that must be interfaced, driven and controlled. In this sense, real-world computer games are kind of *robotic systems.*

In the described computer game context, *software agents* are widely used, mainly to implement to "intelligence" that drives the behaviour of the computer characters (for a virtual opponent), and also to determine and plan the actions to be done by the robot(s) in case of real-world games. The project described in this paper goes in this direction: it is a *robotic table soccer game* that has been designed and developed by author's research group in collaboration with STMicroelectronics that, a part of playing a significant role in the design, provided many hardware components and boards that have been used in the implementation.

The system is made of a mechanical structure that hosts the motor needed to drive the poles (rotation and translation), and a set of boards, equipped with high-performance microcontrollers, that have the task of running the software implementing the whole game, from motor driving, ball tracking up to strategy and intelligence. This software is made of multi-agent system that has been designed ad-hoc in order to take into account the requirements of *safety* and *execution speed.*

All of the parts of the project are described in the paper which is structured as follows. Section 2 deals with related work. Section 3 reports the architecture of the system. Section 4 describes the multi-agent architecture of the software explaining the role of each agent. Section 5 reports some details about the implementation. Section 6 ends the paper.

2. Related Work

The authors of [1] present a robotic table soccer game designed to let a human play against a computer. The system consists of mechanical, electronic, and software components, specifically designed to meet the requirements of table soccer. It employs a vision system to recognise and track the ball and uses eight brushless motors to drive the translation and rotation of the poles. The overall system is controlled by a set of interacting agents that implement the intelligence driving the playing strategy of the robotic opponent. The work provides a detailed description of the hardware and software components and performance data that help understand the effectiveness of the designed application and identifies three key aspects that make pool challenging for robots, the discrepancies between planned and executed strokes due to human limitations in perception and motor control (modelled using probability distributions), the robot kinematic constraints limiting the space of possible actions compared to a human player and modelling human decision-making and skill to improve the robot's win rate. The game of pool is formulated as a sequential stochastic game with continuous state and action spaces. It is modelled as a MAXPROB MDP, which judges the quality of policies based on the probability of reaching a winning state rather than the expected reward. To develop the human model the authors focused on measuring human stroke difficulty based on factors like distance to the cue ball, distance to the pocket and cut angle (optimised to match human decisions and

rankings from psychological experiments), and human pocket probability as a function of stroke difficulty, derived from a Monte Carlo evaluation of human strokes in pool games. In addition, a human discount factor is estimated using maximum likelihood estimation to model how much the human player considers future table states when planning. The framework is evaluated through simulations and experiments with a real robot playing against a human opponent on a pool table.

In [2] the authors present a novel approach for determining optimal striking points in a ping-pong-playing robot. The method focuses on learning striking points based on a reward function that evaluates the coincidence between the ping-pong ball's trajectory and the racket's movement trajectory. By employing a stochastic policy over the reward given the striking point, the robot can explore a wide range of prospective striking points and adapt its striking accuracy over time. The work highlights the importance of selecting optimal striking points due to factors like the robot's nonlinear dynamics, distance to the ball, and velocity requirements for successful ball interception. The proposed method aims to enhance the robot's performance by considering various factors that influence a successful return in table tennis. The article also discusses the components of robot table tennis systems, such as ball position estimation, ball trajectory prediction, interception point determination, inverse kinematics, and robot trajectory generation, emphasising the complexity and challenges involved in achieving accurate and effective striking points. The authors propose an approach for learning striking points without explicit acceleration analysis and without limiting the search area to a specific virtual striking plane. This approach can result in crucial differences, such as allowing the robot to move the racket along the ball flight trajectory but in the opposite direction when returning the incoming ball, which could reduce the effects of prediction errors and accumulated execution errors and thereby substantially increase the success rate. The method is based on a ball-flight model and a rebound model, which can obtain a set of reachable striking points within the robot's workspace. However, while these striking points are geometrically reachable, their success probability differs substantially due to the robot's nonlinear dynamics, the distance to the ball, the need to reach sufficient velocity as well as the right angle at interception, and non-uniform sensitivity to errors. Thus, a ping-pong robotic system must select striking points well. As a successful ball interception is the result of various factors that cannot be modelled straightforwardly, the authors suggest determining optimal striking points based on a reward function that measures how well the ping-pong ball's trajectory and the racket's movement coincide. The authors use a weighted average technique to obtain the mean of the reward, and the variance depends on the confidence of the mean. The confidence of the mean is defined as the maximum weighted coefficient and the variance should depend on this confidence.

The paper [3] presents Gambit, a robotic system designed to autonomously play chess against human opponents in non-idealised environments. The Gambit system consists of a custom 6-DoF robotic arm with a parallel jaw gripper, a PrimeSense depth camera mounted on the shoulder, and a small camera built into the gripper. Gambit uses a hierarchical driver software architecture and employs a set of interacting agents to implement its playing strategy. The system includes a vision system for detecting and tracking the chessboard and pieces in real time. Gambit can play with arbitrary chess sets on a variety of boards, requiring no instrumentation or modelling of the pieces. It continuously monitors the board state and detects when and what kind of move an opponent has made. The paper describes Gambit's approach to playing

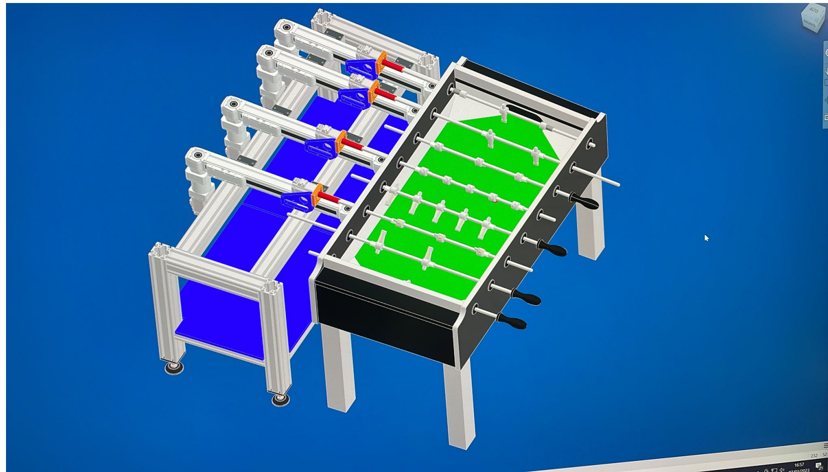


Figure 1: The Mechanical Design of the Soccer Robotic Table

chess flexibly and robustly. It first locates the chessboard and continuously updates its pose relative to the camera. Gambit then learns to recognise chess pieces, which is a necessary step for joining a game at any stage. The actual process of playing a game is detailed, including the manipulation of pieces and the natural language interface Gambit uses to communicate with human opponents. The system's reliability and robustness are highlighted, with most errors being manipulation errors that can be reduced through visual interface.

3. System Architecture

3.1. Mechanics

From the mechanical point of view (see Figure 1), the robotic table soccer game is made by the table itself which, from one side, is connected to a structure made of profiled aluminium that hosts all the component of the system.

The upper part of the chassis mounts four bearing ball skates, one for each pole. Each skate is driven by a belt and a pulley attached to a motor that drives the *translation* of the pole. In the moving part of the skate a mechanical structure hosts the motor that drives the *rotation* of the pole; this motor is directly attached to the the pole axis through a joint.

The lower part of chassis holds all electronics needed to make the system work and is also attached to a support, placed under the table, that hosts the camera needed to capture images for ball detection.

3.2. Electronics

The electronics used in the application is composed of a set of boards equipped with microcontrollers. Since the project is supported by STMicroelectronics, the majority of the boards mount STM32-based MCUs and the boards themselves are designed by STM. The over structure of electronics is depicted in Figure 2 and described below.

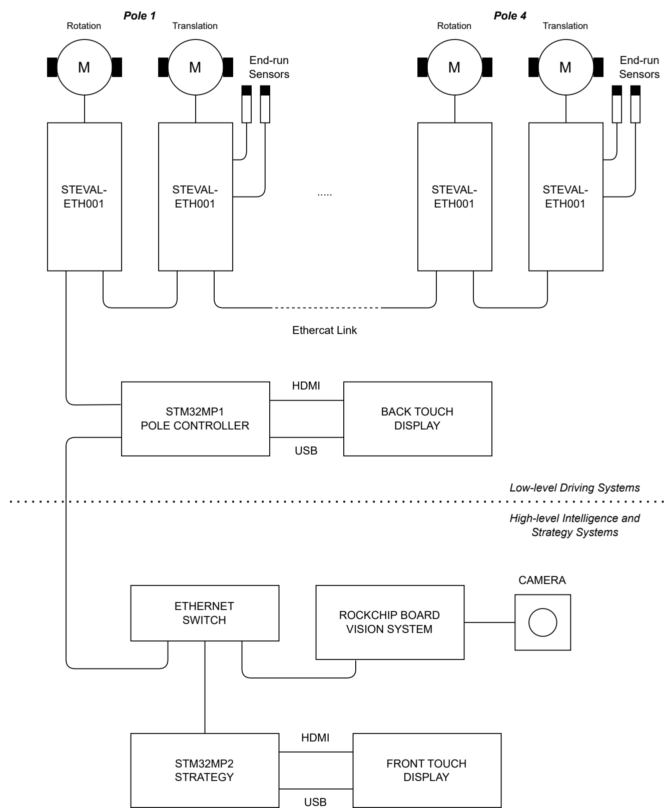


Figure 2: Structure of the electronic parts of the soccer table game

The lowest level of the architecture is made of the motors that drive rotation and translation of the poles; there are eight brushless motor each one attached to a driver board (STEVAL-ETH-001[1]). All motor drivers are interconnected through an EtherCAT [4] link which is a fieldbus communication technology developed by Beckhoff Automation in 2003, and it is an industrial Ethernet protocol widely used in the automation sectors. EtherCAT is a master-slave protocol where a single device has the role of *master* and multiple *slave* devices (up to 65535) are connected to the same bus using, in general, a daisy-chain topology. Independently of the topology, the physical network always forms a logical ring: the protocol is cyclic; the master is the only node that initializes the transfer of process data and at each cycle time it sends a single frame which will pass through all the slave nodes in the network; the frame continues to propagate to subsequent node and the slave reached by the frame extracts the data intended for it and inserts those produced by it upon request of the master; the modified frame continues to the next slave and returns to the master.

The role of EtherCAT master is played by the software running in the board called *Pole*

[1]https://www.st.com/en/evaluation-tools/steval-eth001v1.html

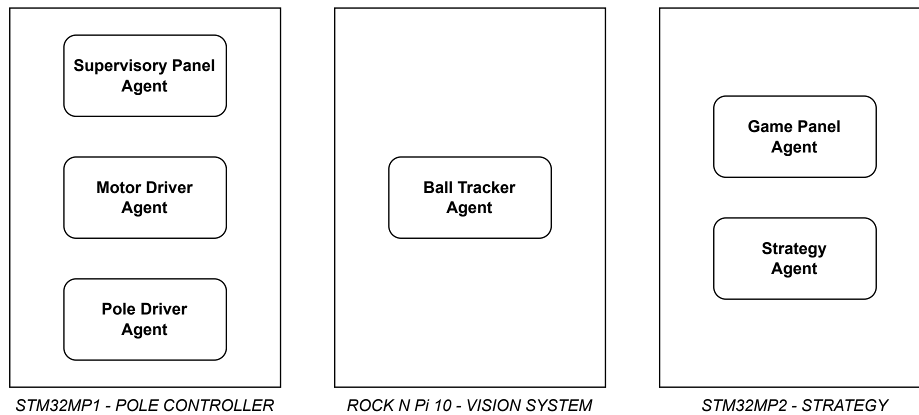


Figure 3: Software Architecture of the System

Controller; it is an STM32MP1 board, i.e. a Raspberry-like board equipped with an ARM32-based MCU and running a Linux-embedded distribution. This board has two Ethernet connections, one for the EtherCAT branch and the other connected to an IP LAN. The board has also a display used as a supervisory panel for the field activities.

The high-level part of the overall system is composed by two boards. A Rockchip N Pi 10 board[2], equipped with an ARM64 MCU, is devoted to the vision system; this is a critical part of the application since it requires to track a ball that can reach very high speeds. For this reason, we employed a camera sensor developed by STMicroelectronics that is able to reach the rate of 200 fps; indeed this is the theoretical speed of the hardware sensor, the actual frame rate depends on the ball tracking algorithm that runs on the Rockchip platform.

The last board used in the application is a STM32MP2 platform, an ARM64 MCU-based hardware that runs embedded Linux. This board runs the agents that manage the game strategy as well as a user interface that shows the evolution of the game in terms of ball position, score, etc.

4. The Agents

In this section we deal with the details of the software architecture describing the role and the working scheme of all the agents composing the application. We refer to Figure 3 that shows the agents and the hardware platform in which each agent runs. We start from low-level up to high-level agents.

4.1. Motor Driving Agent

The lowest level of the software architecture runs the *Motor Driving Agent (MDA)* which not only has the basic task of implementing the EtherCAT master protocol but also manages the

[2] https://wiki.radxa.com/RockpiN10

state machine needed to drive the motors. Indeed, the working scheme of BLDC motors and the STEVAL-ETH-001 driver require certain procedures to start-up the motor and also to manage the positioning commands; in detail, an alignment procedure is needed at the startup in order to match the generation of the magnetic field for the rotors with the field of stators; moreover, a zero-match procedure is also needed for translation motors, since the "zero" and the travel of each pole is different; to this aim, an end-run sensor is placed on the mechanical skate in the position corresponding to the "zero" of the pole and the zero-match procedure runs the motor until the sensor is active. After this initialisation procedure, the agent can manage positioning commands.

Another activity performed by the MDA is fault handling. Motor drivers generate *faults* on the basis of exceptional conditions that may occur during the game; they are: activation of the emergency button, overcurrent in one of the motors, motor locked condition. In all of these cases, the involved driver generates a fault signal handled by the MDA that stops any communication with the driver and also halts the game by forwarding the alarm signal to all the other agents of the system. When the fault is resolved, the MDA is able to detect that normal working conditions are resumed but, just for safety reasons, the game must be restarted manually from the game panel.

4.2. Supervisory Panel Agent

The *Supervisory Panel Agent (SPA)* is a quite simple agent that has the task of reporting the status of the motors in a screen display. It communicates with the MDA in order to acquire the current information of the motors and in particular:

- status of the motors (running, aligning, fault, etc.);
- current position;
- current speed;
- status of the EtherCAT link.

A graphical user interface is being developed that, apart of displaying the said information, lets the user perform few activities like halting and restarting drivers.

4.3. Pole Driver Agent

Position and speed of motors are managed by the MDA in terms of rotation angle (in radians) and angular speed (in rad/s); however, while for the pole rotation such units make sense, for translation the use of *linear* units should be a better choice: indeed, the translation motor is attached to a 5:1 gearbox[3] and then to a pulley of 2 cm of radius, thus five revolutions of the motor correspond to 4π cm of translation. Since the Strategy Agent uses linear units a proper conversion is needed. Moreover each pole has different linear run sizes, depending on the number of puppets and the position of two springs that have the task of establishing pole limit.

The knowledge of the geometry of all the poles is given to the *Pole Driver Agent*; it reads the data (i.e. pole position and run size) from a JSON file and perform the following tasks:

[3]This is needed to increase the torque, since translation movement is done at a very high acceleration.

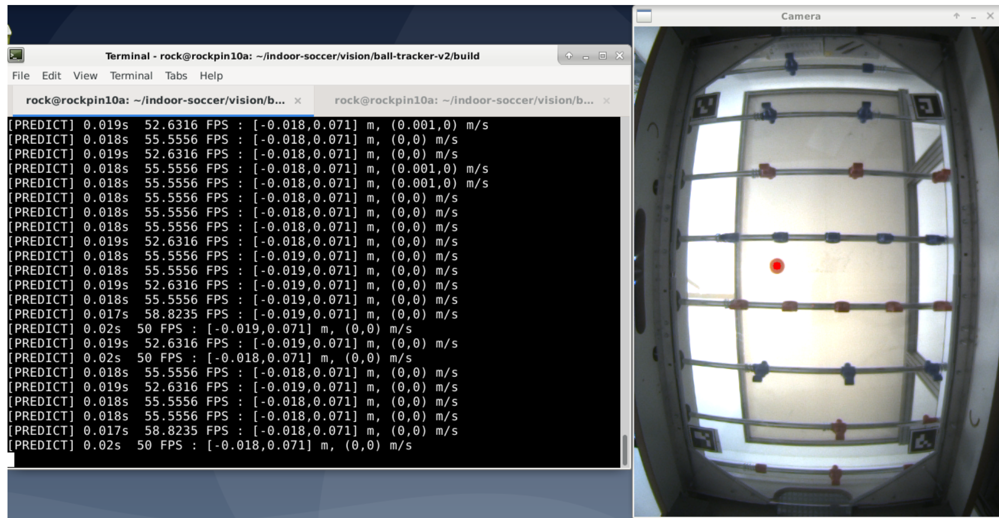


Figure 4: Screenshot of the Ball Tracking System

- it triggers the "zero" procedure of each pole each time the motors are (re-)started; in doing this, it computes an offset since the origin of the reference system of the strategy is placed at the center of the playing table;
- it performs unit conversion by transforming the value of translation of each pole from meters (as computed by the strategy) into radians as needed by the MDA.

Since the PDA has the knowledge of the size of poles, it also checks that the position sent by the Strategy Agent is within pole limits: even if two end-run sensors are placed in the skates that drives the poles, double-checking position data implies to increase the degree of safety of the system thus avoiding any possible mechanical damage that could also be dangerous for the human player.

4.4. Ball Tracker Agent

The *Ball Tracker Agent (BTA)*, together with the Strategy Agent, is one of the most important agents of the system from the point of view of performances. Since the ball of the game can reach a very high speed[4], the rate of the detection algorithm must be as high as possible. This is the reason why the BTA runs in its own platform and it is the sole agent running in that hardware.

The task of the BTA is not only to detect the ball but also to compute its physical (real) coordinates (in meters) and to estimate speed and direction; such information are needed by the Strategy Agent in order to estimate ball's trajectory and move the poles accordingly.

The vision algorithm employed performs a binarization of each acquired frame by using a threshold, in the HSV space, according to the color of the ball (orange in our case); the binarized image is then processed in order to detect blobs and then each blob is analysed in terms of

[4]The literature reports that, in professional matches, the ball can reach a speed up to 80 km/h.

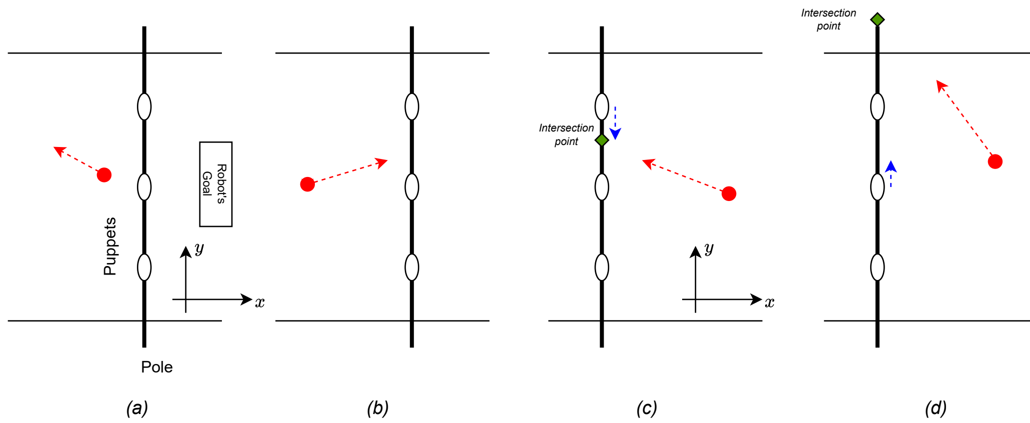


Figure 5: The criteria of the Strategy Agent

area, perimeter, density and eccentricity: these parameters allow filtering of blobs that do not correspond to the ball.

Once the centre of the ball has been detected the next step is to convert pixel coordinates in real-world coordinates; to this aim a former calibration process is needed that implies placing four ARUCO markers [5] in four known positions of the playing table (see Figure 4): the calibration procedure detects marker positions and determines the transformation matrix needed for pixel-to-world translation.

Before sending data to the Strategy Agent, the estimation process is performed. This is done by running a Kalman filter using a state vector that includes ball's position and speed, i.e. $\hat{x} = [x, y, v_x, v_y]$; the estimator receives ball's position and estimates the components of the speed; the state vector is then sent to the Strategy Agent.

The implementation of the ball detection is algorithm has been optimised in order to avoid all possible latencies. Currently the rate reached is in the order of 55 FPS, however we would like to reduce the processing time by profiling the code and try to optimise wherever is possible.

4.5. Strategy Agent

The *Strategy Agent (SA)* implements the game strategy and embeds a behaviour that is based on geometrical and physical assumptions. The action plan is designed in the game environment that considers a two-dimensional Cartesian plane whose point of origin coincides with the centre of the game table, the x axis has the direction along the longer side of the table. Furthermore, each pole has a second reference system, dedicated to rotations, having the ordinate axis perpendicular to the table. Real world objects such as the ball, poles and puppets are modelled using software entities such as points and lines. The centre point of the pole length segment is considered as the pole reference point.

The input data, on which the strategy algorithm determines the criteria and decisions on the sequence of actions to be performed, is information coming from the BTA and concerns the position and velocity vector of the ball. The SA has four parallel behaviours, each one

assigned to one of poles (goalkeeper, defender, centre, attack) and each behaviour runs the same algorithm thus producing a translation and rotation command for the associated pole; therefore, each pole moves independently from one another.

The criteria used by each behaviour are based on the velocity vector and predicted position of the ball in relation to the position of the pole, and team membership. Therefore, the following features are determined: the *direction* of the teams, the position of the ball: *behind* or *in front* of the pole and the direction of movement: *moving away* or *closer*. Selected actions may involve performing specific movements:

move_pole linear movement along the pole to position the a puppet at the point where the moving ball is expected to reach the pole;

passthrough lifting the puppet horizontally backwards by specifically rotating the pole in order to let the ball pass;

kick performing a kick through appropriate rotations.

In relation to the type of team, the criteria for choosing the action to perform are the following (see Figure 5):

1. If the ball is *behind* the pole and goes *away*, no action is applied (Figure 5a).
2. If the ball is *behind* the pole and comes *closer*, the **passthrough** action is performed and possibly also the **move_pole** (Figure 5b).
3. If the ball is *in front* of the pole and *moves away* with a slow motion, the start of the **kick** is evaluated.
4. If the ball is *in front* of the pole and comes *closer*, the possibility of performing the **move_pole** or the **kick** action is assessed.

To perform the **move_pole** action, the intersection point of the ball with the pole has to be first estimated. In the case of fast motion (ball speed >= $0.01\ m/s$), the target point is defined as the intersection between the trajectory of the ball's motion vector and the straight line representing the pole; if the point is inside the table (Figure 5c), the puppet nearest to it is determined and the pole is moved in order to make that puppet reach that point; if the point is outside the table (Figure 5d), the target point is defined as the point of projection of the ball on the straight line of the pole. The same action (Figure 5d) is performed in the case of slow motion of the ball.

The **passthrough** action involves instead the rotary movement of the pole from $0°$ to $-90°$ to implement the move of "lifting" the puppet horizontally backwards.

The **kick** action is made up of three sequential movements which imply the rotation of the pole from $0°$ to $-90°$, then to $50°$, to perform the kick, and repositioning the puppet in the vertical position ($0°$); the rotation speed plays an important role since it determines the power of the kick[5]. The trigger of **kick** depends once again on ball's speed; in the case of a fast ball, the time required by the ball to reach the intersection point with the pole is determined, and this value is compared with the time required to perform kick (weighted by a experimental constant value that takes into account the latency of the system): if travel time is less than kick time, the kick is triggered. In the case of a slow ball, the distance of the ball with the intersection point is

[5]Currently it is set to 3000 $°/s$.

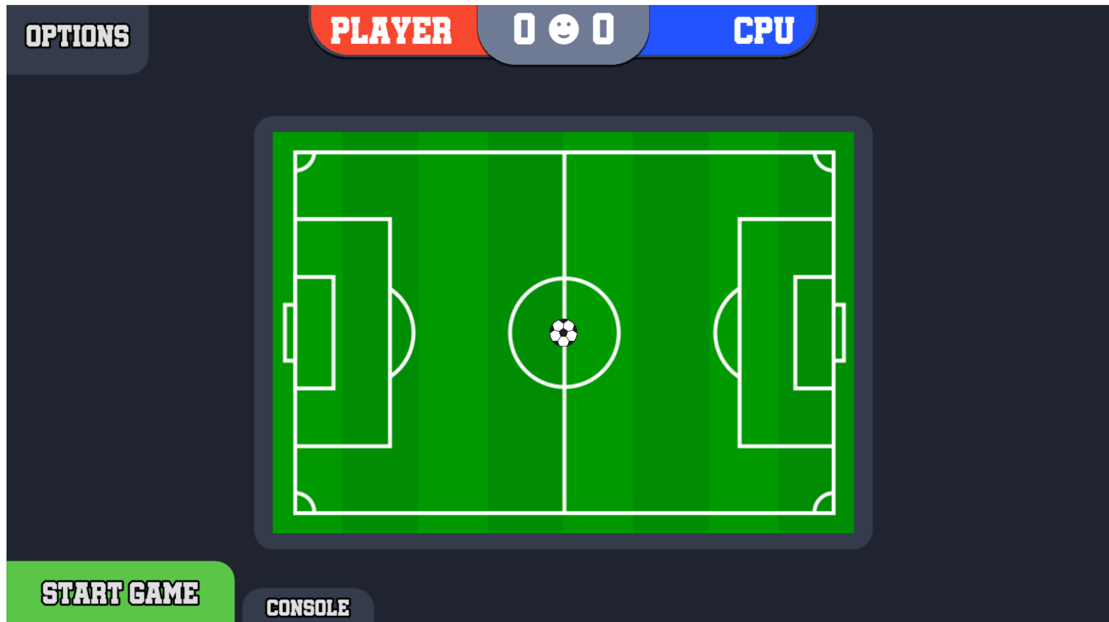


Figure 6: The Game Panel Agent

compared with a `kick_min_distance` constant (that is set experimentally) and, when it is less than that constant, the kick is triggered.

4.6. The Game Panel Agent

The *Game Panel Agent (GPA)* is responsible of the user interface of the game. Basically it displays a panel, shown in Figure 6, that depicts the game field in which is reported also the position of the ball, and the score of the game. Some buttons allow the user to start/stop the game and to configure the vision system in order to tune the thresholds of the ball recognition algorithm, an operation that is needed whenever the light conditions should change.

5. Discussion

To implement the overall system, from the software point of view, we designed everything from scratch using C++ and Godot [6] (for the panel agents). We did not use an agent platform mainly for performance reasons. Indeed there are various tight time requirements in the whole application.

The first requirement is due to the EtherCAT protocol: as it has been reported above, the EtherCAT protocol is based on the periodic transmission of a frame, from the master, that passes through all the slave devices and returns to the master. The period is set to $5\ ms$ and is a strong constraint since if it is not met, the EtherCAT devices recognise a fault condition, therefore the part of the MDA that manages the master protocol must not have any latency; for this reason it is implemented in a thread that runs with `SCHED_FIFO` policy and with the maximum priority.

The second requirement is related to the timing of all the other agents that, as it has been already explained, need to be as fast as possible in order to follow the high speeds that can reach the ball. Indeed, the cycle time of all the agents is dependent of the duration of the ball tracking algorithm which, in the current implementation, is about $15\ ms$ but, since the camera should be able to reach up to 200 FPS, we plan to reduce the processing time by adding more optimisations and/or using a faster computing platform.

These aspects require a native implementation and this is the reason why we chose C++. Also the communication among agents is optimised since it uses raw sockets and data is exchanged in binary form in order to avoid any overhead due to interpretation.

6. Conclusions

This paper has described the hardware/software architecture and working scheme of a robotic table soccer game that lets a human play against a computer. The software part of system is based on a multi-agent system where each agent plays a specific role starting from ball recognition and tracking, robot's game strategy, pole driving and motor control.

From the software point of view, the system has been completely implemented while, as for the mechanics, currently only two poles have been activated, but we plan to complete also the mechanical part in the next few months. Anyway we performed some tests that allowed us to verify the effectiveness of the playing strategy and also to tune some parameters in order to make the strategy more efficient and realistic.

Acknowledgements

The authors thank all the people from SRA Group of STMicroelectronics that are involved in the project and that are giving a significant contribution.

References

[1] T. Nierhoff, K. Leibrandt, T. Lorenz, S. Hirche, Robotic billiards: Understanding humans in order to counter them, IEEE Transactions on Cybernetics 46 (2016) 1889–1899. doi:10.1109/TCYB.2015.2457404.

[2] Y. Huang, B. Schölkopf, J. Peters, Learning optimal striking points for a ping-pong playing robot, in: 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2015, pp. 4587–4592. doi:10.1109/IROS.2015.7354030.

[3] C. Matuszek, B. Mayton, R. Aimi, M. P. Deisenroth, L. Bo, R. Chu, M. Kung, L. LeGrand, J. R. Smith, D. Fox, Gambit: An autonomous chess-playing robotic system, in: 2011 IEEE International Conference on Robotics and Automation, 2011, pp. 4291–4297. doi:10.1109/ICRA.2011.5980528.

[4] The ethercat protocol, 2024. https://www.ethercat.org/it/technology.html.

[5] Aruco marker detection, 2024. https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html.

[6] The godot game engine, 2024. https://godotengine.org/.