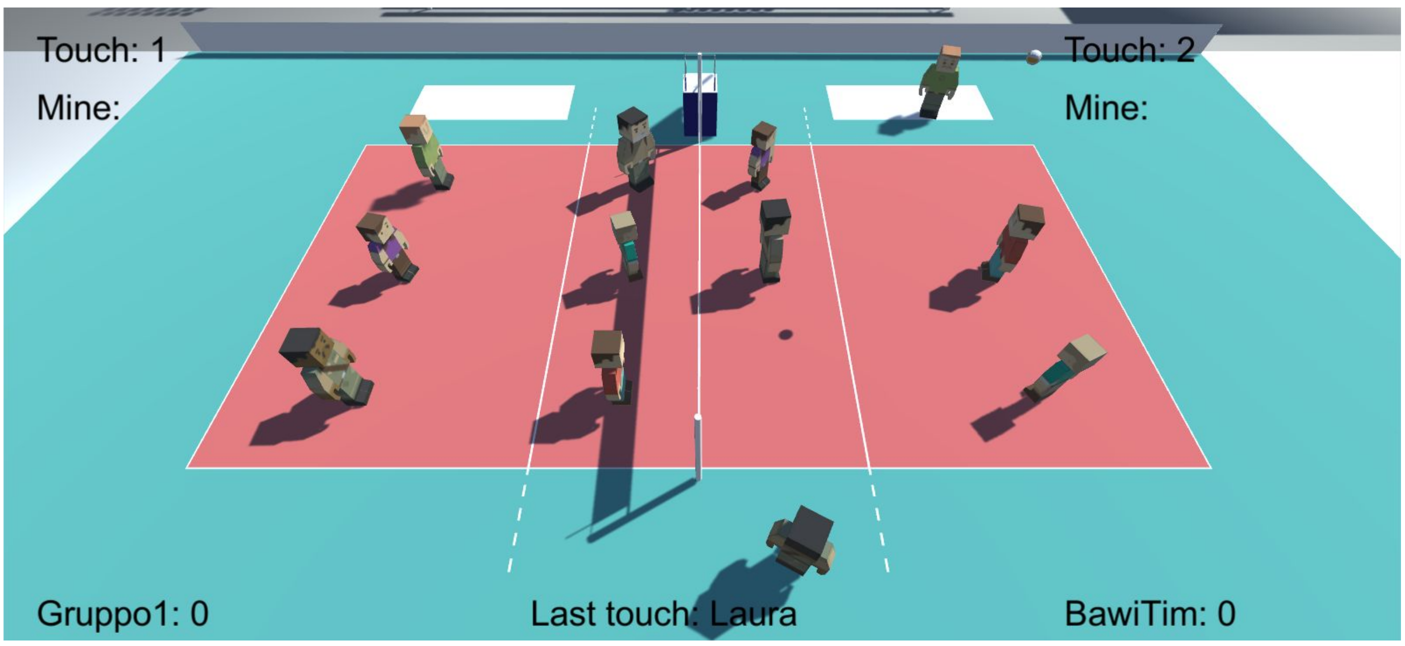
Maione Michele (931468)

michele.maione@studenti.unimi.it

Folie

A library (in C++/CLI) for Unity that simulate intelligences for volleyball games.



Index

Analysis

Volleyball rules

Game rules

A volleyball court is 9m×18m, divided into equal square halves by a net with a width of one meter. The top of the net is 2.43m above the center of the court for men's competition, and 2.24m for women's competition; each assigned to one of the two teams.

Each team consists of a maximum of 12 players, of which 6 are holders; one of the players can be designated as Libero, who has the right to temporarily replace one of the defenders without limitations on the number of substitutions.

The game is divided into sets: a team wins the victory of a set on reaching the 25 points, as long as it has at least two advantages; otherwise, the set continues until one of the two teams gets the 2 points of advantage needed.

The game ends when a team wins 3 sets.

Set rules

Each action begins with the service performed by the player in the position 1, of the team that obtained the right. The action continues until the ball touches the field, or is sent out of it. The team that wins a game action wins a point and the right to serve.

For each game action, the team has 3 touches available (excluding the possible touch of the wall), to send the ball into the opposing field by passing it inside the passage space. After making a wall, a player can hit the ball again without incurring a double-touch foul and making the first team touch.

In the event that the ball touches the net and comes back, it can be replayed within the limits of the touches available to the team[1].

Goals

The artificial intelligence need manage:

- Movement: individuals, realistic, using physical simulation.
- Decision making:
  - Possible actions: set, receive, pass, attack, block, serve, move;
  - Distinct states: looking at the ball, active, take position, take the ball;
  - Behaviour change: ball position, touch count;
  - Need to look ahead: ball destination, touch count.
- Tactical and strategic AI:
  - Characters think for themselves and display a group behaviour:
    - Strategy: serve-receive system, offensive system, coverage system, defensive system;
    - Formation: "4–2", "6–2" and "5–1".

[1] http://en.wikipedia.org/wiki/Volleyball

Future developments

Missing items

Items that are missing:

- An algorithm to calculate all the parabolas of a projectile motion;
- Skills: block, dig;
- Formations: “6-2”;
- Attack by jumping forward;
- Spikes types: hard-driven, off-speed, standing, open hand tip;
- Tipping types: dink, power tip;
- Set types: 1, 2, 3, 4, 5, 6, 7, 9, 10, 32, BS, shoot, flare, slide, iso, tandem, double quick, x.
- Other features: sit on the bench, replace a player, salute, shake hands, interact with the referee, get hurt.

Other engines

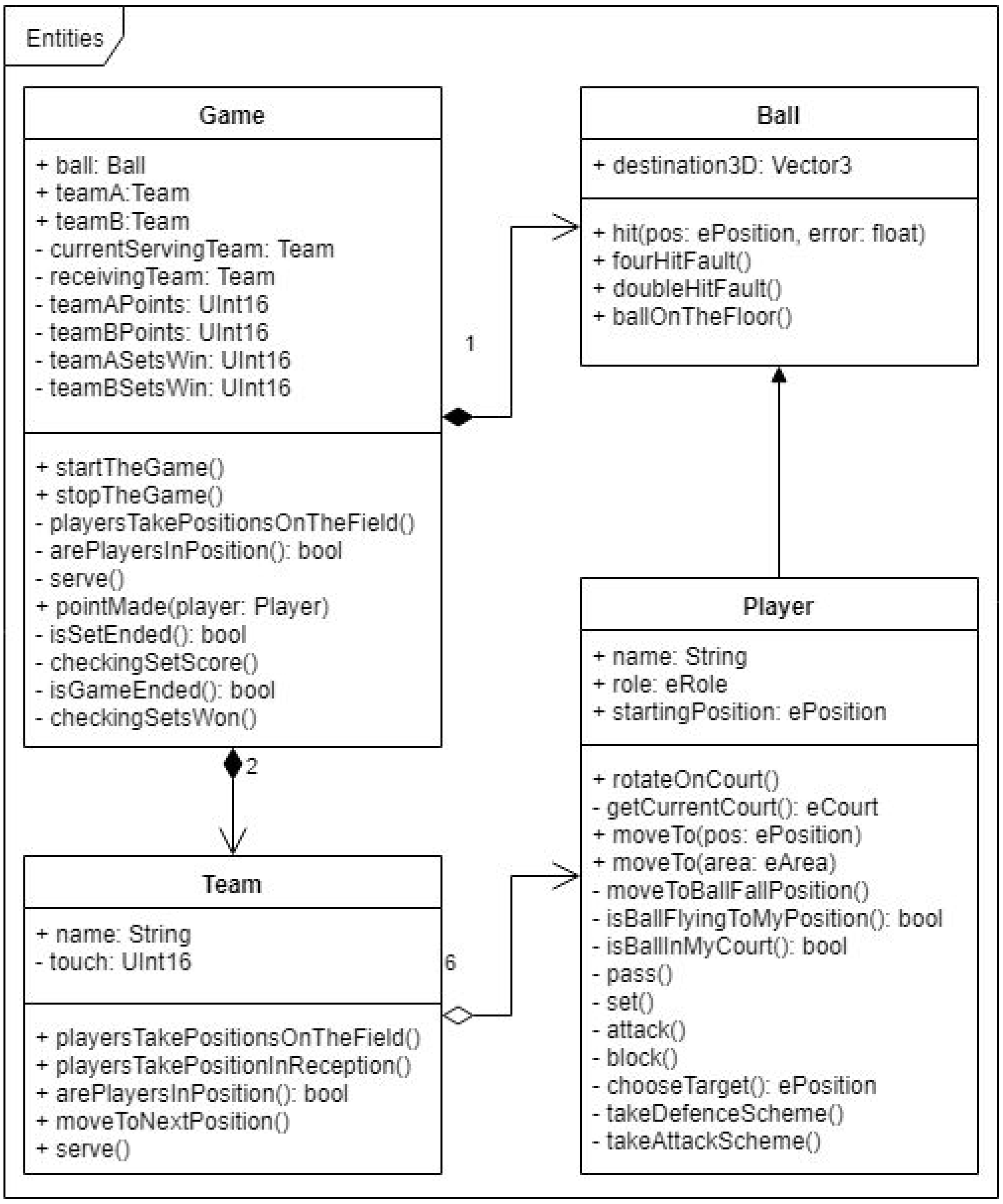
The decision to encode a series of algorithms to simulate intelligences for a volleyball game is due to the fact that there aren't a lot of volleyball games on the market.

I chose to implement it as a library in C++ to make it reusable in the future. The current version was designed to be used in Unity, but was also designed to be adaptable to other engines.

Design

Entities

The project manage 4 entities: game, team, player and ball. Each class inherit from *MonoBehaviour*, so that public properties can be set directly via the *Unity* editor.



Project class diagram

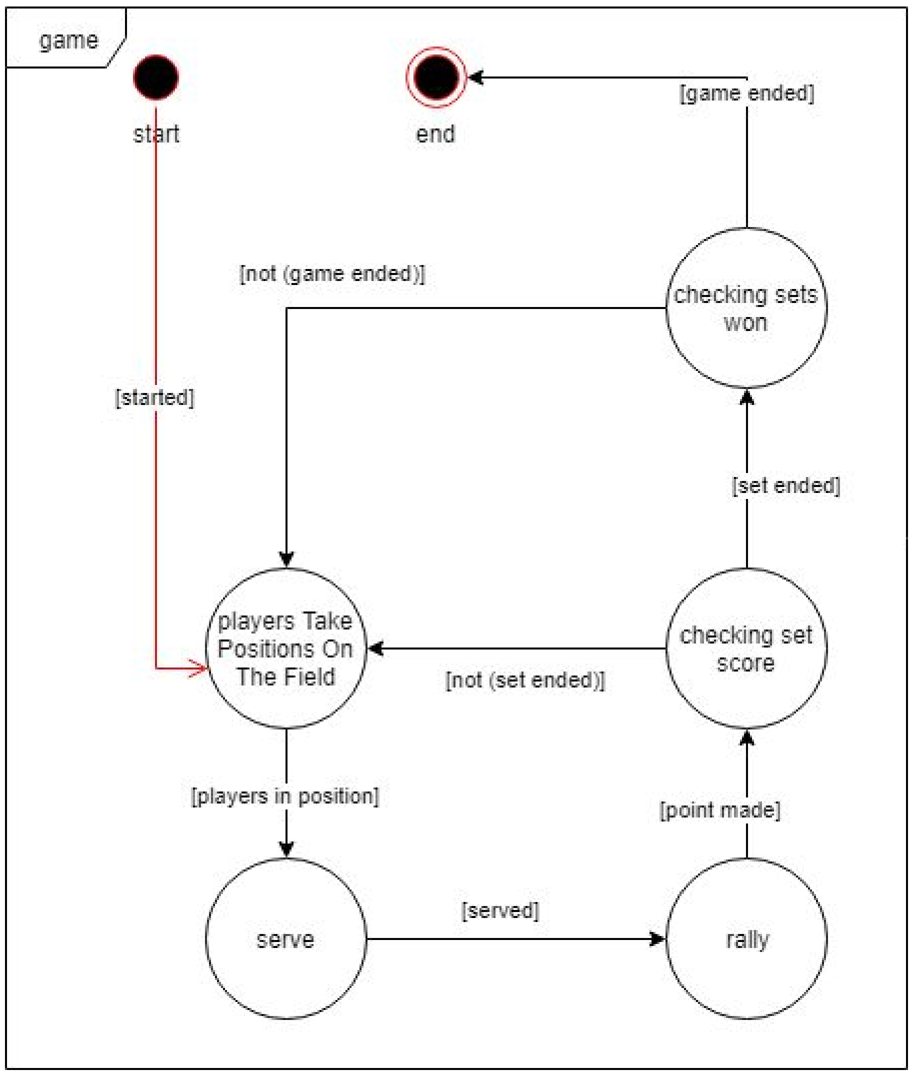
Game

For the game logic it was created the class *Game* that implement a Mealy machine[2].

[2] Mealy, George H. (1955). A Method for Synthesizing Sequential Circuits. Bell System Technical Journal. pp. 1045–1079.

Mealy machine

The implemented version of the finite-state machine used for the game logic is the Mealy machine, for the necessity to use inputs for actions.

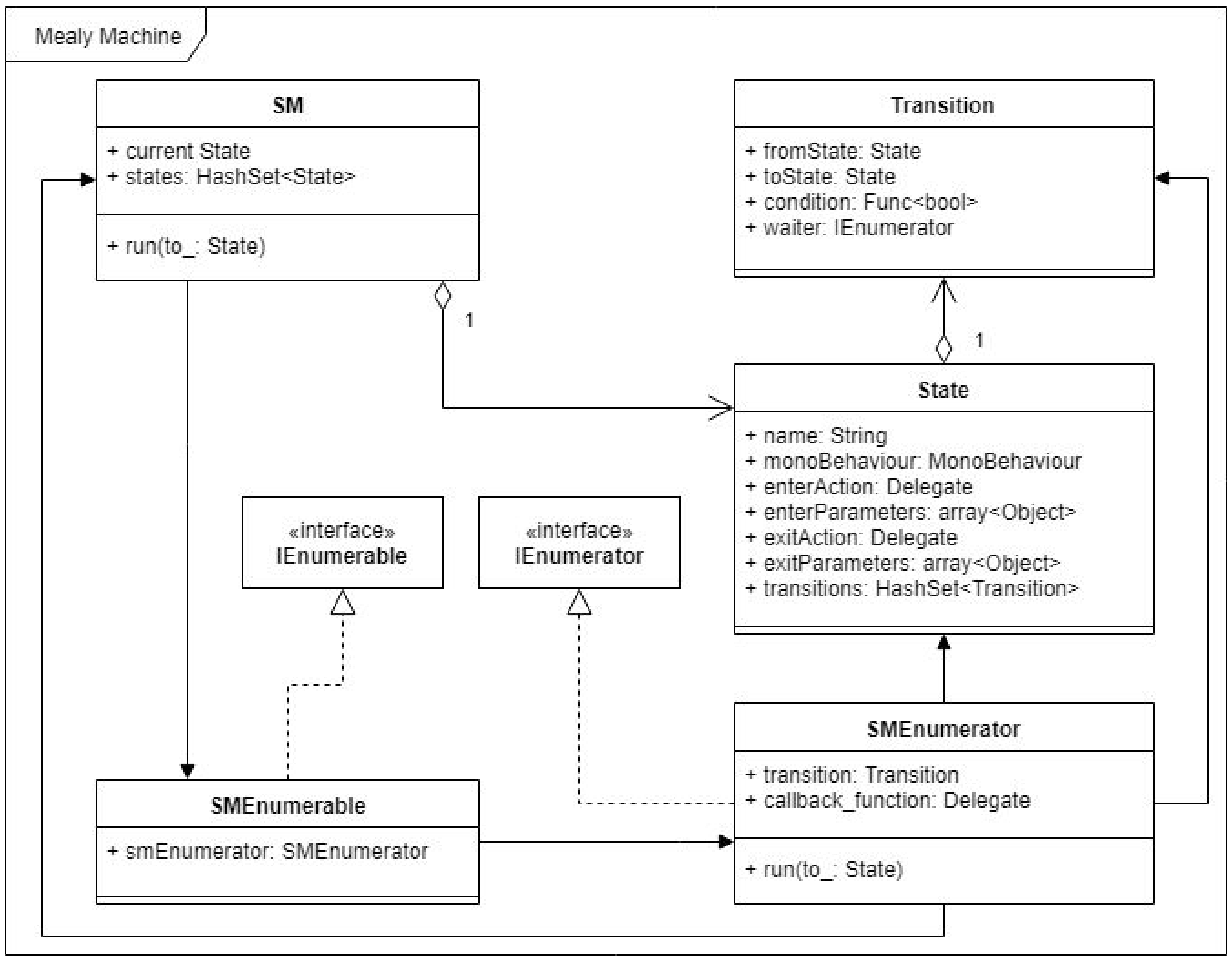


Game F.S.M.

Implementation

There are 3 classes relative to the machine: *State*; *Transition*; *SM*.

And 2 classes that connect the machine to the Unity coroutine system: *SMEnumerable*; *SMEnumerator*.



Mealy machine classes

SM

In the *SM::run* method the class, executes in the following order:

1. Call the enter action;
2. For each transition of the current state:
   - If a *waiter* is present:
     i. Call the coroutine (*StartCoroutine*);
     ii. Call the exit action.
   - Otherwise:
     i. If *condition* is null or true, call the run method (*SM::run*);
     ii. Call the exit action.

State

The only thing to note for this class is the array of input (*enterParameters*, *exitParameters*) for enter action and exit action.

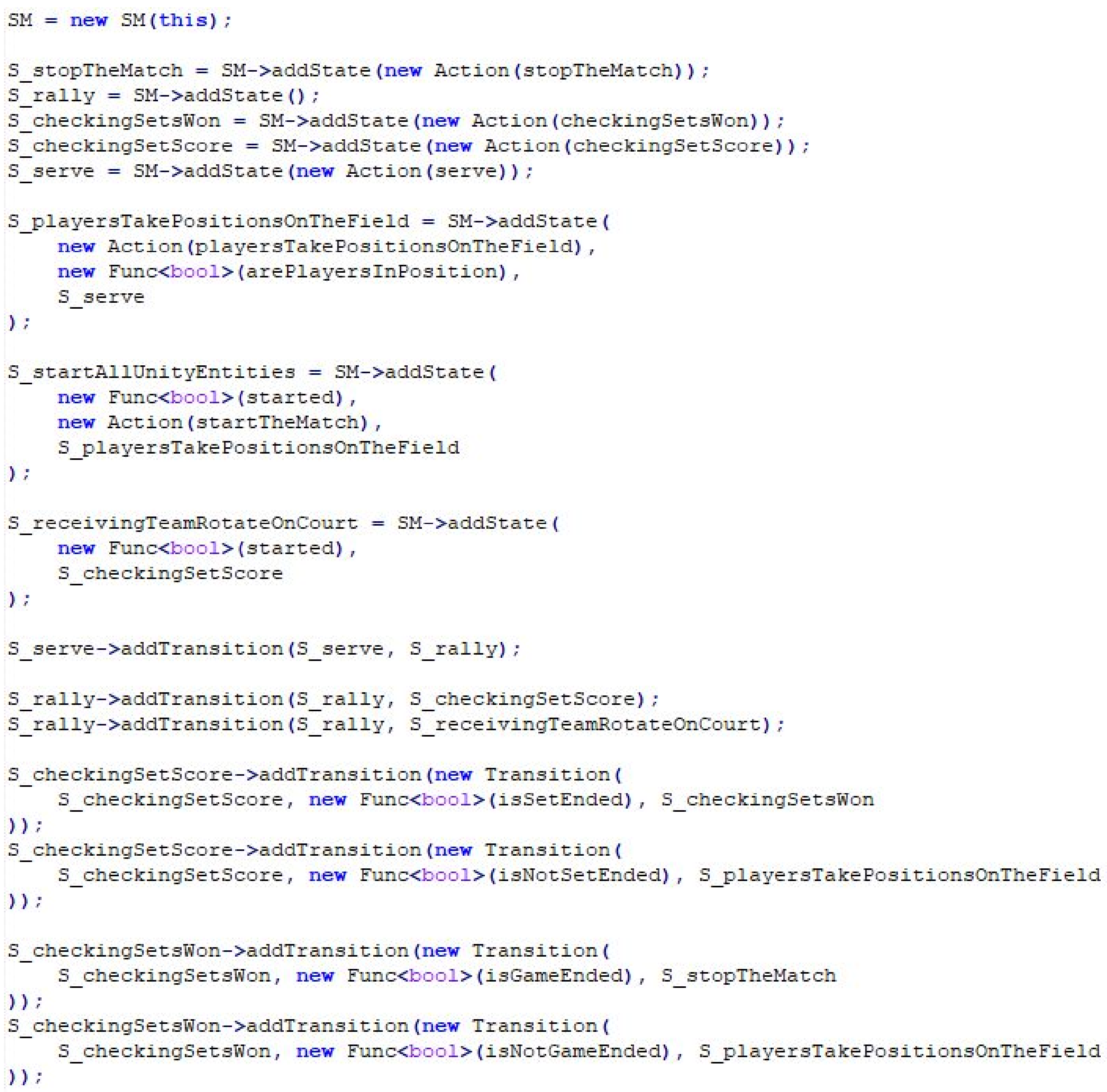
SMEnumerator

SMEnumerator implements *IEnumerator*. When the *IEnumerator* reach the end it call *callback_function* that is a pointer to the function *SM::run*, in this way the machine goes to the next state.

Transition

The important things in this class are:

- In the field *condition* a pointer to a function that makes the machine continue to the next state (*toState*) only if the function returns a positive result;
- In the field *waiter* a pointer to an *IEnumerator* that is the parameter for the function *StartCoroutine*, used to call boolean functions;
  For example: *transition->waiter = new Func<bool>(arePlayersInPosition);*.



```
SM = new SM(this);

S_stopTheMatch = SM->addState(new Action(stopTheMatch));
S_rally = SM->addState();
S_checkingSetsWon = SM->addState(new Action(checkingSetsWon));
S_checkingSetScore = SM->addState(new Action(checkingSetScore));
S_serve = SM->addState(new Action(serve));

S_playersTakePositionsOnTheField = SM->addState(
    new Action(playersTakePositionsOnTheField),
    new Func<bool>(arePlayersInPosition),
    S_serve
);

S_startAllUnityEntities = SM->addState(
    new Func<bool>(started),
    new Action(startTheMatch),
    S_playersTakePositionsOnTheField
);

S_receivingTeamRotateOnCourt = SM->addState(
    new Func<bool>(started),
    S_checkingSetScore
);

S_serve->addTransition(S_serve, S_rally);

S_rally->addTransition(S_rally, S_checkingSetScore);
S_rally->addTransition(S_rally, S_receivingTeamRotateOnCourt);

S_checkingSetScore->addTransition(new Transition(
    S_checkingSetScore, new Func<bool>(isSetEnded), S_checkingSetsWon
));
S_checkingSetScore->addTransition(new Transition(
    S_checkingSetScore, new Func<bool>(isNotSetEnded), S_playersTakePositionsOnTheField
));

S_checkingSetsWon->addTransition(new Transition(
    S_checkingSetsWon, new Func<bool>(isGameEnded), S_stopTheMatch
));
S_checkingSetsWon->addTransition(new Transition(
    S_checkingSetsWon, new Func<bool>(isNotGameEnded), S_playersTakePositionsOnTheField
));
```

S.M. instantiation

Team

For the team it was created the class *Team*, it's called by the class *Game*; It say to all the players:

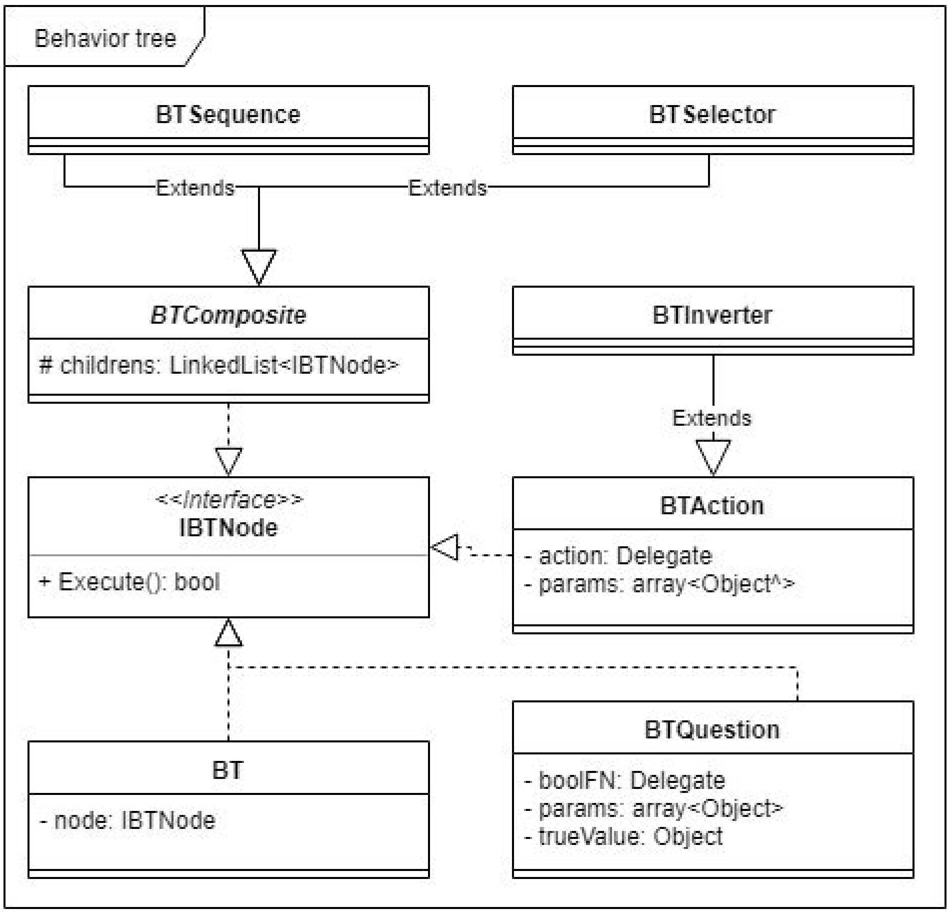
- To take positions on the field;
- To rotate on court.

Player

For the player it was created the class *Player* that use a behavior tree[3] and three decision trees[4].

Behavior tree

The B.T. is implemented using 7 classes and 1 interface.



B.T. classes

IBTNode

Is the interface that expose the method *IBTNode::Execute* that return a boolean value indicating if the operation was executed successfully.

BT

Is the root node.

BTQuestion

Implement a parameterized function.

BTAction

Implement a parameterized action.

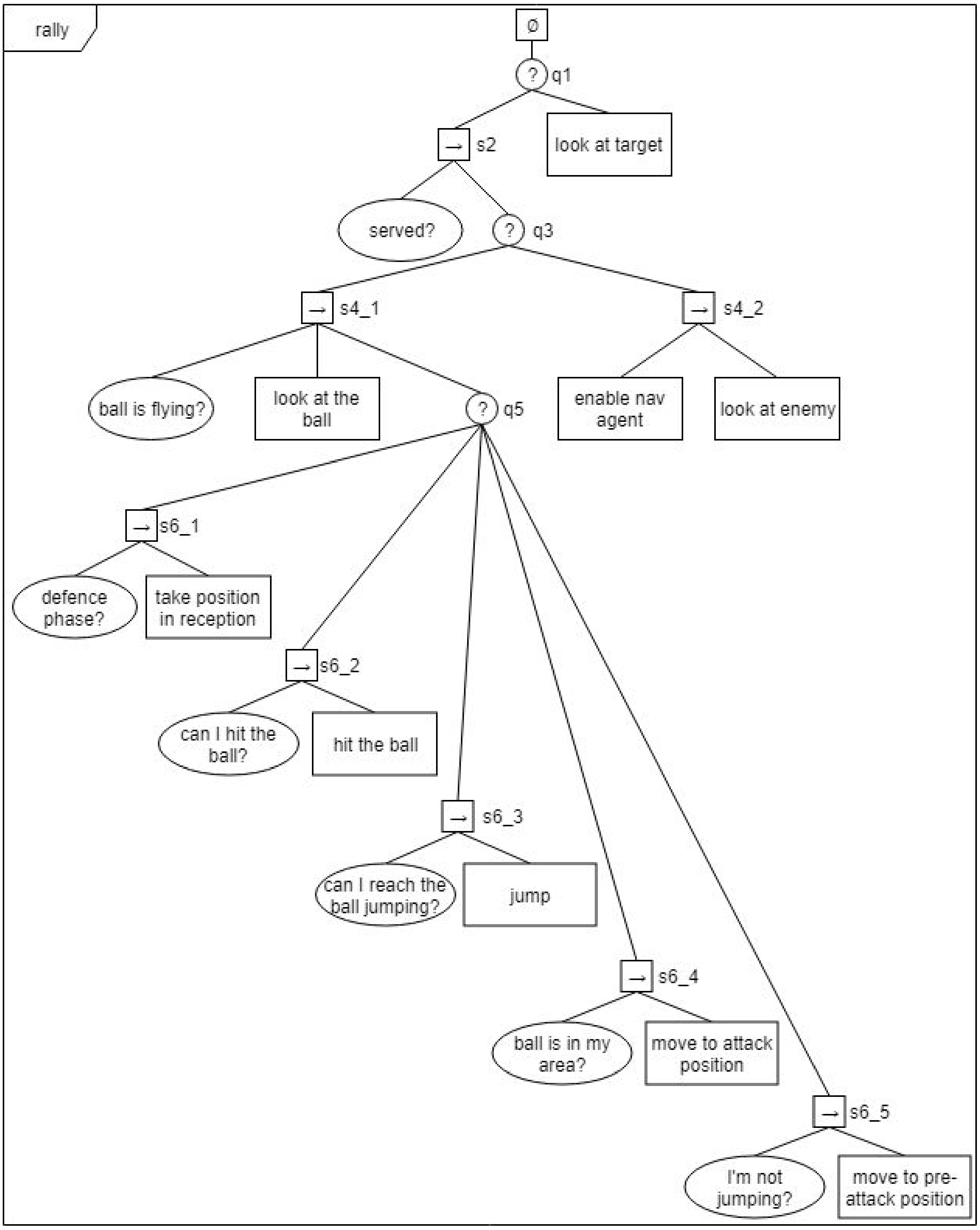
[3] Millington, I., & Funge, J. (2009). Artificial Intelligence for Games(2nd ed.). Morgan Kaufmann.
[4] Kamiński, B.; Jakubczyk, M.; Szufel, P. (2017). "A framework for sensitivity analysis of decision trees". Central European Journal of Operations Research. 26 (1): 135–159

BTComposite

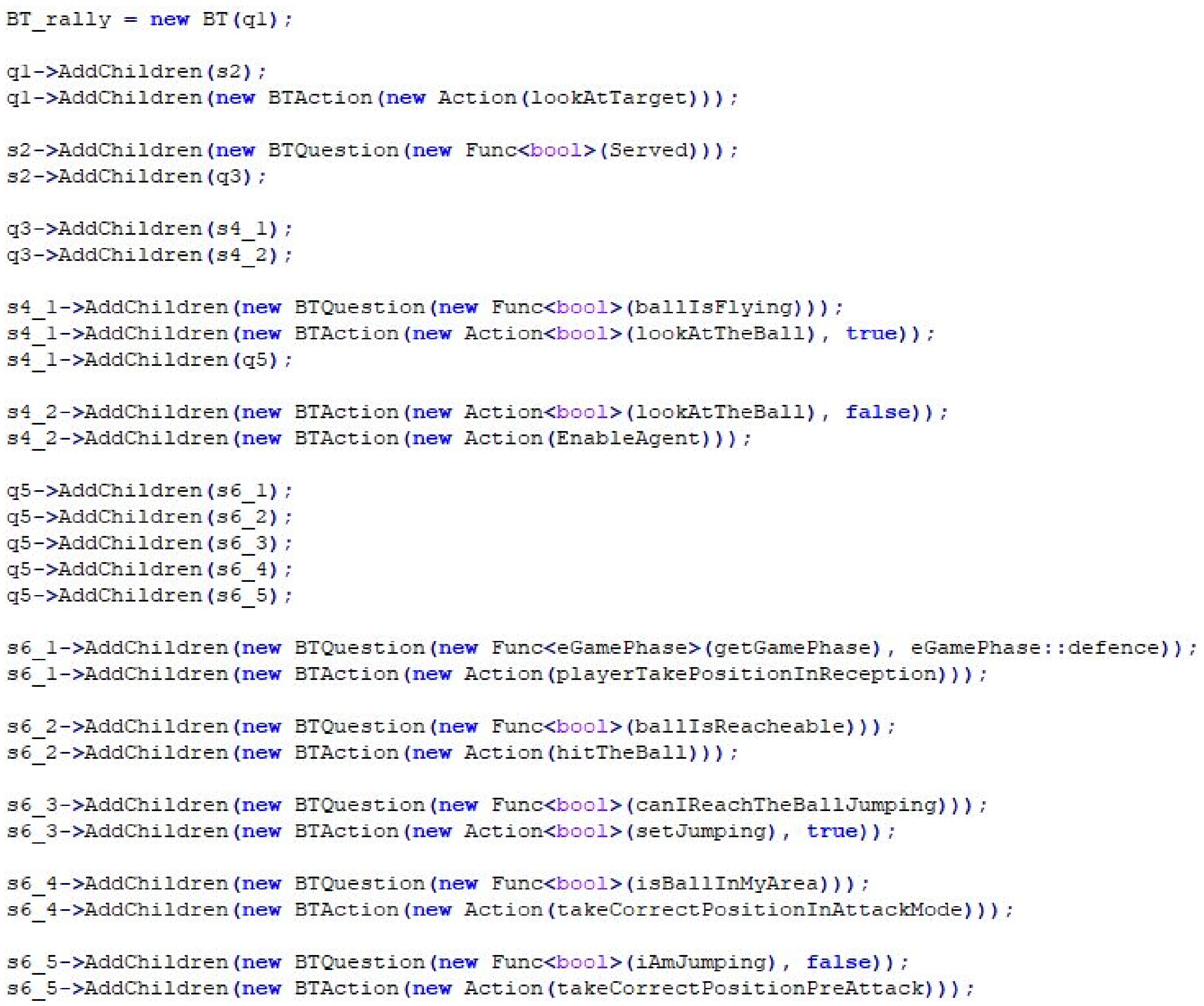
It's an abstract class, it's used to implement control flow nodes. It's extended by *BTSequence* and *BTSelector*.

Rally

This behavior tree takes into account the role and the court in which the player is to take decisions.



Rally B.T.

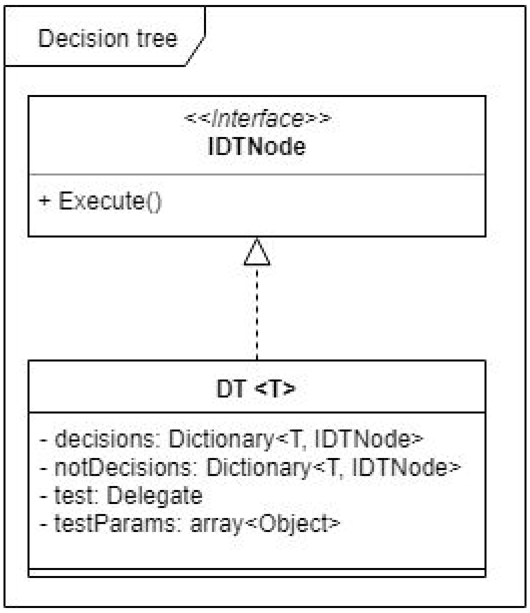


```
BT_rally = new BT(q1);

q1->AddChildren(s2);
q1->AddChildren(new BTAction(new Action(lookAtTarget)));

s2->AddChildren(new BTQuestion(new Func<bool>(Served)));
s2->AddChildren(q3);

q3->AddChildren(s4_1);
q3->AddChildren(s4_2);

s4_1->AddChildren(new BTQuestion(new Func<bool>(ballIsFlying)));
s4_1->AddChildren(new BTAction(new Action<bool>(lookAtTheBall), true));
s4_1->AddChildren(q5);

s4_2->AddChildren(new BTAction(new Action<bool>(lookAtTheBall), false));
s4_2->AddChildren(new BTAction(new Action(EnableAgent)));

q5->AddChildren(s6_1);
q5->AddChildren(s6_2);
q5->AddChildren(s6_3);
q5->AddChildren(s6_4);
q5->AddChildren(s6_5);

s6_1->AddChildren(new BTQuestion(new Func<eGamePhase>(getGamePhase), eGamePhase::defence));
s6_1->AddChildren(new BTAction(new Action(playerTakePositionInReception)));

s6_2->AddChildren(new BTQuestion(new Func<bool>(ballIsReacheable)));
s6_2->AddChildren(new BTAction(new Action(hitTheBall)));

s6_3->AddChildren(new BTQuestion(new Func<bool>(canIReachTheBallJumping)));
s6_3->AddChildren(new BTAction(new Action<bool>(setJumping), true));

s6_4->AddChildren(new BTQuestion(new Func<bool>(isBallInMyArea)));
s6_4->AddChildren(new BTAction(new Action(takeCorrectPositionInAttackMode)));

s6_5->AddChildren(new BTQuestion(new Func<bool>(iAmJumping), false));
s6_5->AddChildren(new BTAction(new Action(takeCorrectPositionPreAttack)));
```

Rally B.T. instantiation

Decision tree

Three decision trees were designed to manage: the defense scheme, the attack scheme, and the game fundamentals (pass, set, attack).


Decision tree
<<Interface>>
IDTNode
+ Execute()
DT <T>
- decisions: Dictionary<T, IDTNode>
- notDecisions: Dictionary<T, IDTNode>
- test: Delegate
- testParams: array<Object>

D.T. classes

IDTNode

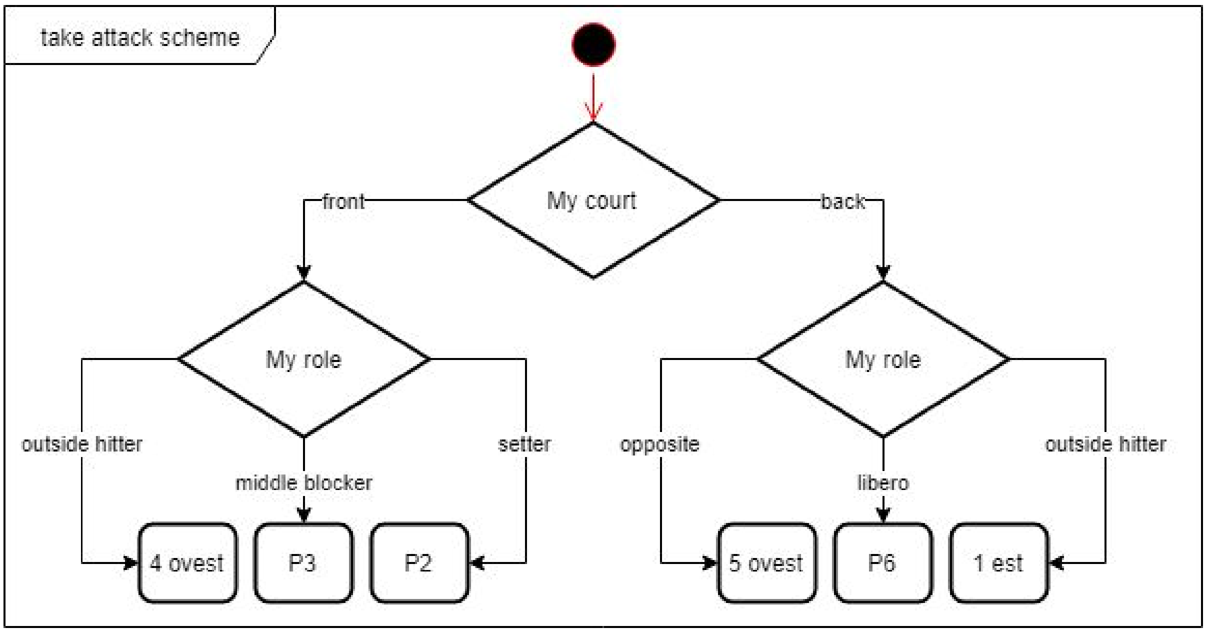
Is the interface that expose the method *IDTNode::Execute* that execute a node.

DT

The *DT::Execute* procedure calls:

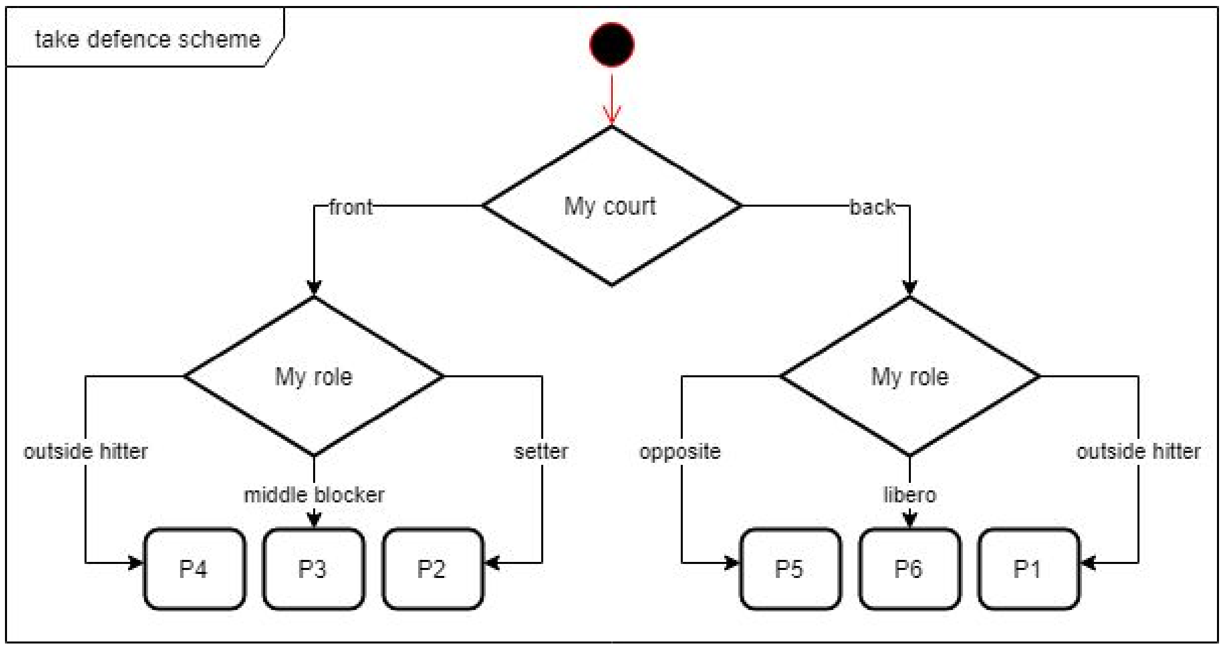
1. *DT::test*: that is a delegate for a parametrized function;
2. If the dictionary *DT::decisions* contain the result *R*:
    a. Call the *IDTNode::Execute* procedure of the *IDTNode* in this dictionary entry;
3. Otherwise:
    a. For each element of *decisions* that is different from *R*:
        i. call the *IDTNode::Execute* procedure of the *IDTNode* in this dictionary entry.

Attack scheme



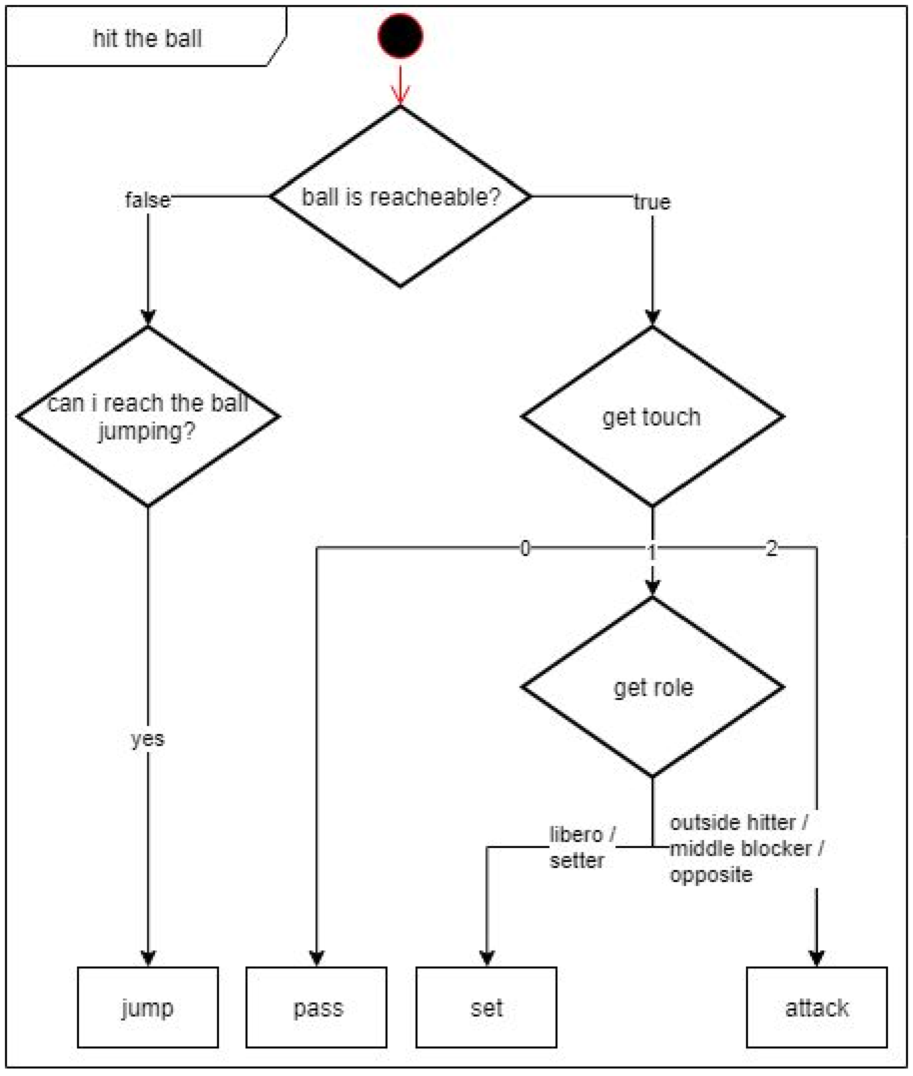
Attack scheme D.T.

Defence scheme

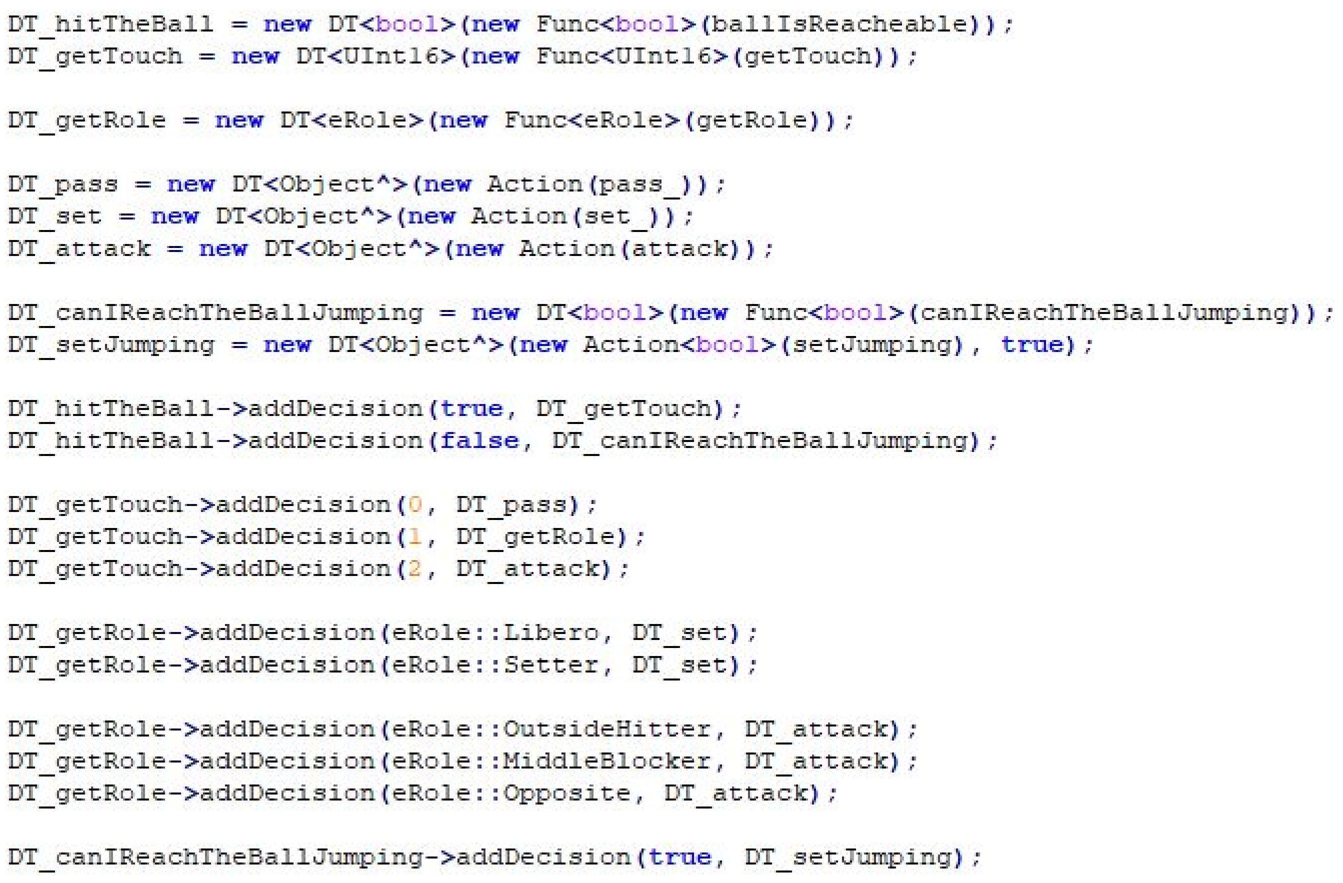


Defence scheme D.T.

Hit the ball



Hit the ball D.T.



```
DT_hitTheBall = new DT<bool>(new Func<bool>(ballIsReacheable));
DT_getTouch = new DT<UInt16>(new Func<UInt16>(getTouch));

DT_getRole = new DT<eRole>(new Func<eRole>(getRole));

DT_pass = new DT<Object^>(new Action(pass_));
DT_set = new DT<Object^>(new Action(set_));
DT_attack = new DT<Object^>(new Action(attack));

DT_canIReachTheBallJumping = new DT<bool>(new Func<bool>(canIReachTheBallJumping));
DT_setJumping = new DT<Object^>(new Action<bool>(setJumping), true);

DT_hitTheBall->addDecision(true, DT_getTouch);
DT_hitTheBall->addDecision(false, DT_canIReachTheBallJumping);

DT_getTouch->addDecision(0, DT_pass);
DT_getTouch->addDecision(1, DT_getRole);
DT_getTouch->addDecision(2, DT_attack);

DT_getRole->addDecision(eRole::Libero, DT_set);
DT_getRole->addDecision(eRole::Setter, DT_set);

DT_getRole->addDecision(eRole::OutsideHitter, DT_attack);
DT_getRole->addDecision(eRole::MiddleBlocker, DT_attack);
DT_getRole->addDecision(eRole::Opposite, DT_attack);

DT_canIReachTheBallJumping->addDecision(true, DT_setJumping);
```

Hit the ball D.T. instantiation

Ball

For the ball it was created the class *Ball* that manage:

- Collisions: via the procedure *OnCollisionEnter*;
- Trajectory: via the procedure *addForce*.

Other classes

Other classes that were developed are:

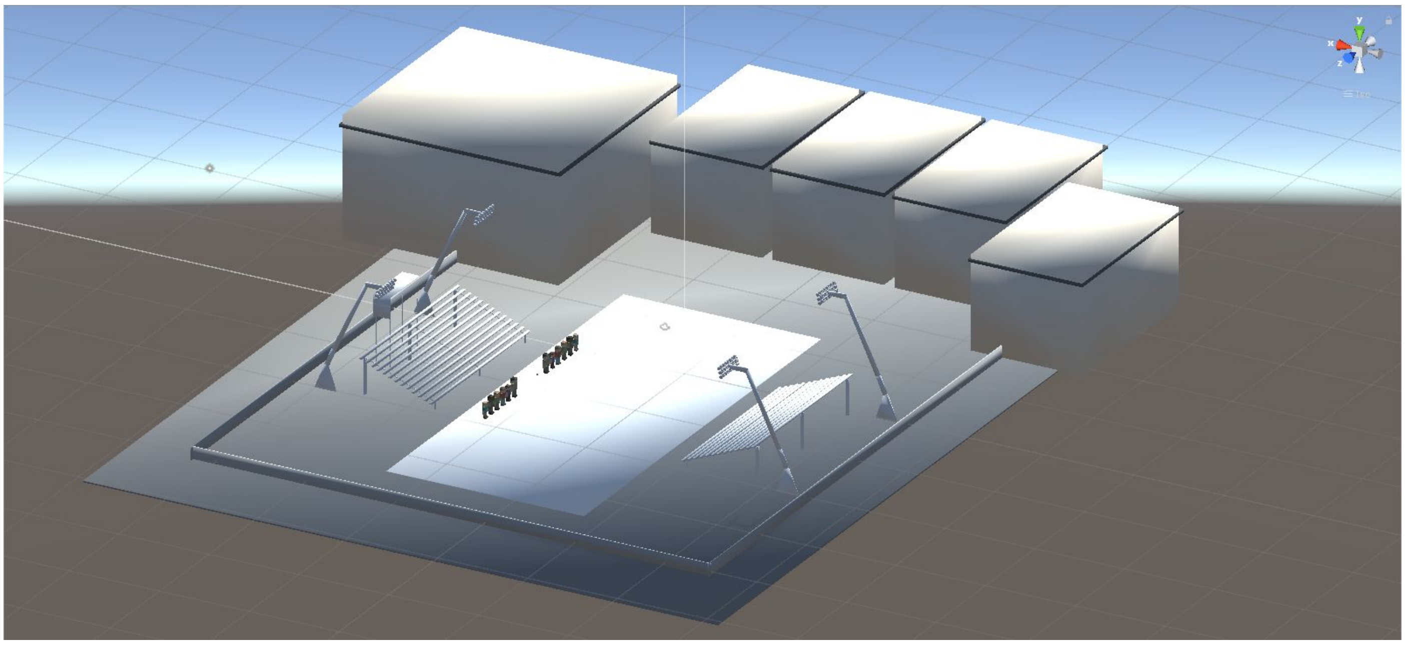
- *GB*: global and general functions;
- *REF*: global static reference to game, ball, teamA, teamB;
- *Enums*: all the enumerators;
- *Job*: a job to enqueue;
- *CoroutineQueue*: for job queuing, via coroutine. It's similar to SMEnumerator;
- *GenericEnumerable*: implements *IEnumerable* and it's used by *CoroutineQueue*;
- *Wait4Seconds*: inherits *CustomYieldInstruction* and it's used to wait for an amount of milliseconds by coroutine.

A demo game in Unity

Level

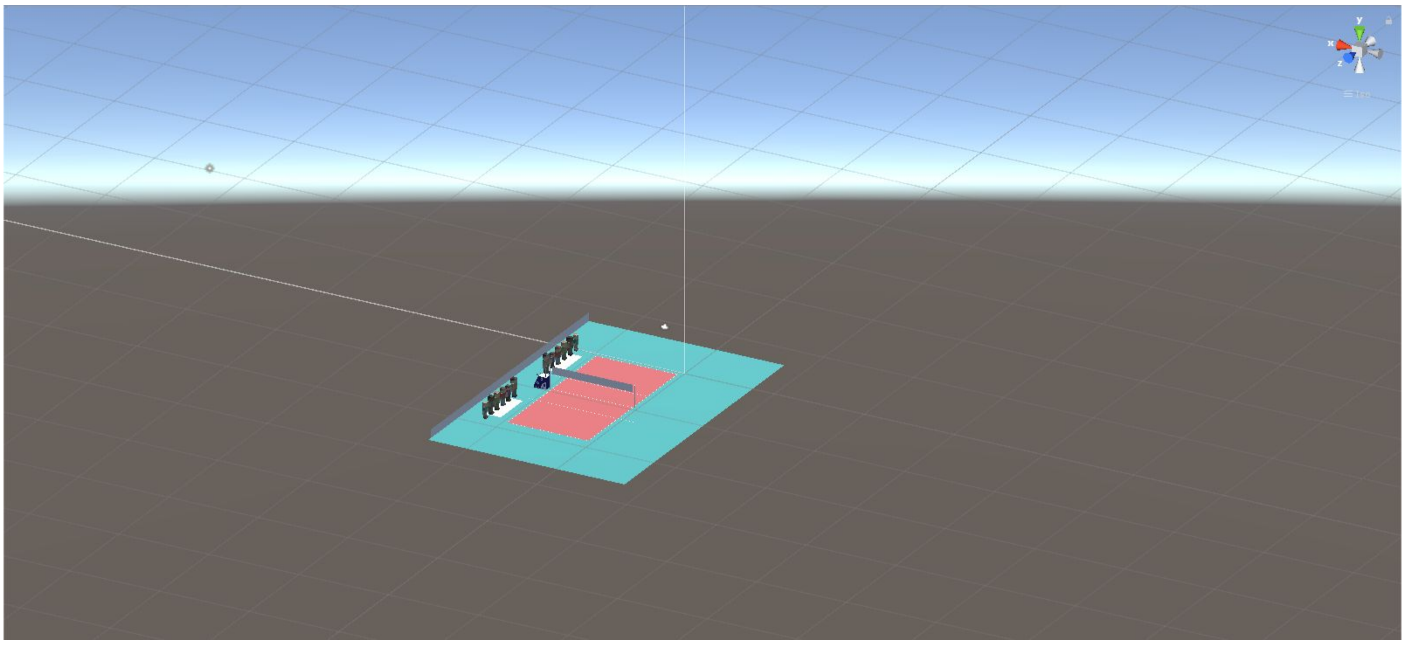
For the design of the level of this demo, different free assets have been downloaded from the internet.

Stadium

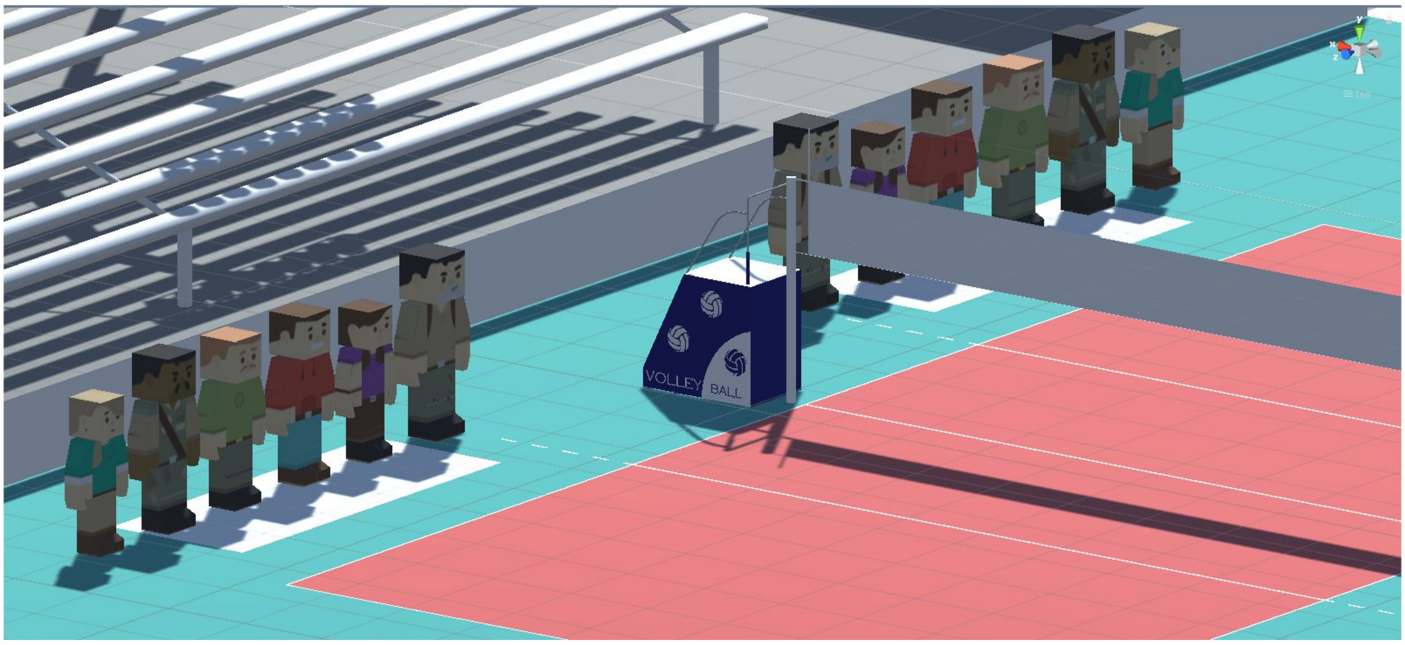


Stadium

Field



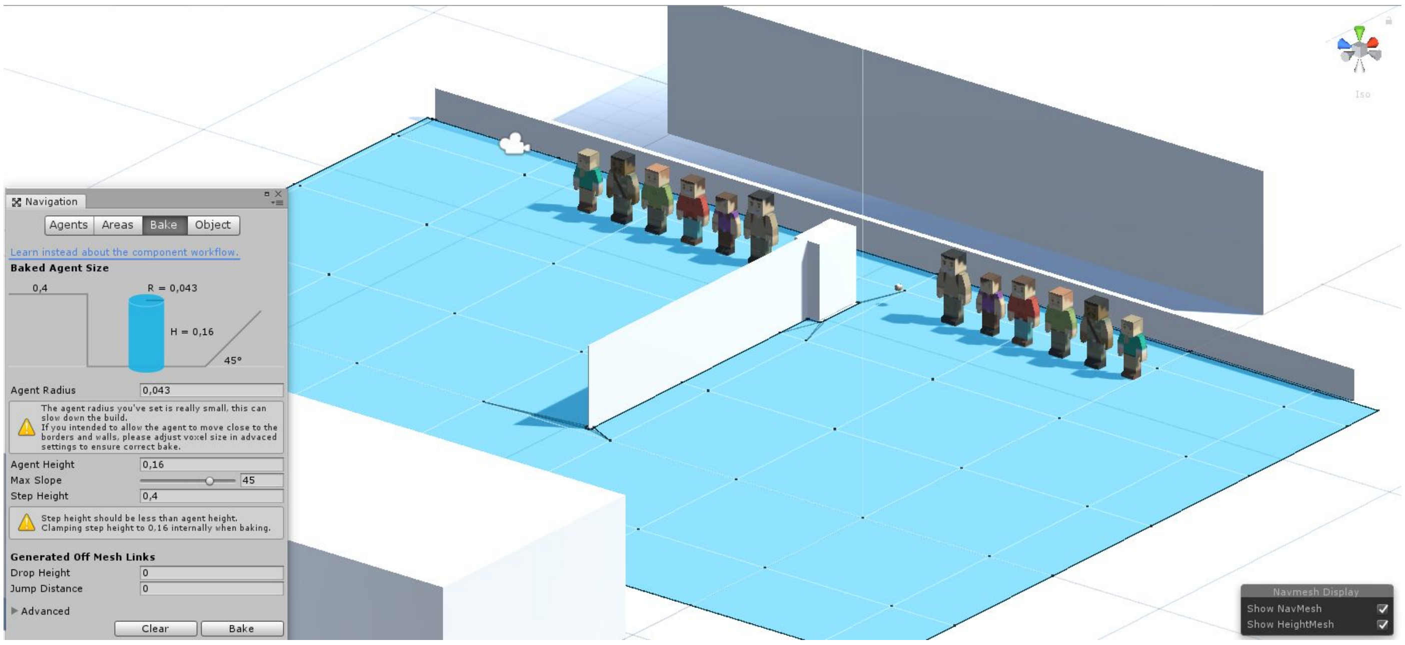
Game field compared to the size of the stadium



Players lined up off the court

Navigation mesh

To move the players in the field the Unity navigation system (NavMesh) was used.



Baked Agent Size

Agent Radius 0,043

The agent radius you've set is really small, this can slow down the build.
If you intended to allow the agent to move close to the borders and walls, please adjust voxel size in advaced settings to ensure correct bake.

Agent Height 0,16
Max Slope 45
Step Height 0,4

Step height should be less than agent height.
Clamping step height to 0,16 internally when baking.

Generated Off Mesh Links

Drop Height 0
Jump Distance 0

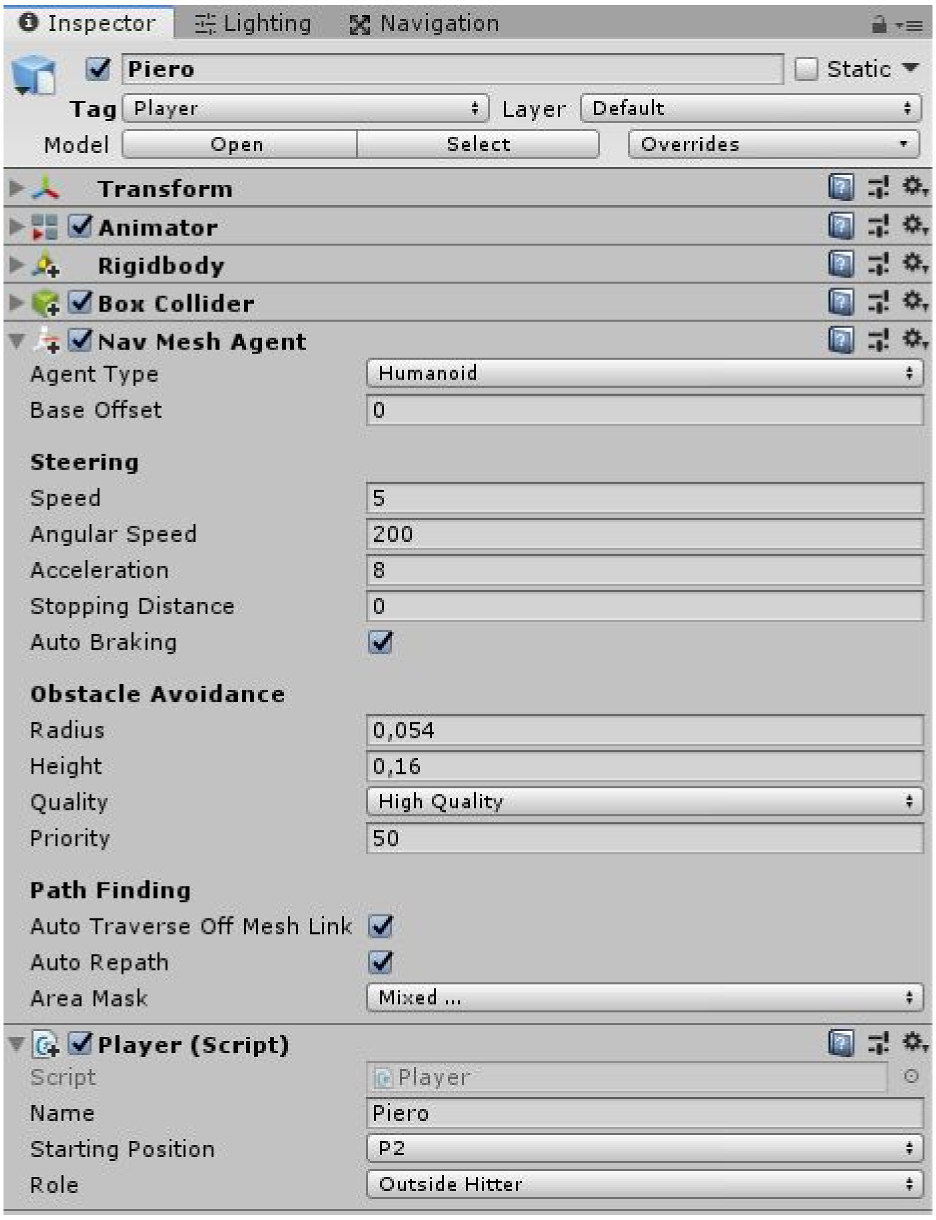
▶ Advanced

Unity NavMesh

Players

By importing the Folie plugin into the Unity project it’s possible to set all the public properties for the game, the team, and the players.

The most important ones are those of the players. Based on the roles of the players, Folie implements different game strategies.



✓ Piero	Static
Tag	Player
Layer	Default
Model	Open / Select / Overrides

Transform

✓ Animator

Rigidbody

✓ Box Collider

✓ Nav Mesh Agent

Agent Type	Humanoid
Base Offset	0
Steering	
Speed	5
Angular Speed	200
Acceleration	8
Stopping Distance	0
Auto Braking	✓
Obstacle Avoidance	
Radius	0,054
Height	0,16
Quality	High Quality
Priority	50
Path Finding	
Auto Traverse Off Mesh Link	✓
Auto Repath	✓
Area Mask	Mixed ...

✓ Player (Script)

Script	Player
Name	Piero
Starting Position	P2
Role	Outside Hitter

Player properties

Version control

The web-based hosting service for version control that was used is GitHub.

For changes and access requests contact the administrator.

Information

Url: https://github.com/mikymaione/Folie

Administrator: Michele Maione (mikymaione@hotmail.it)