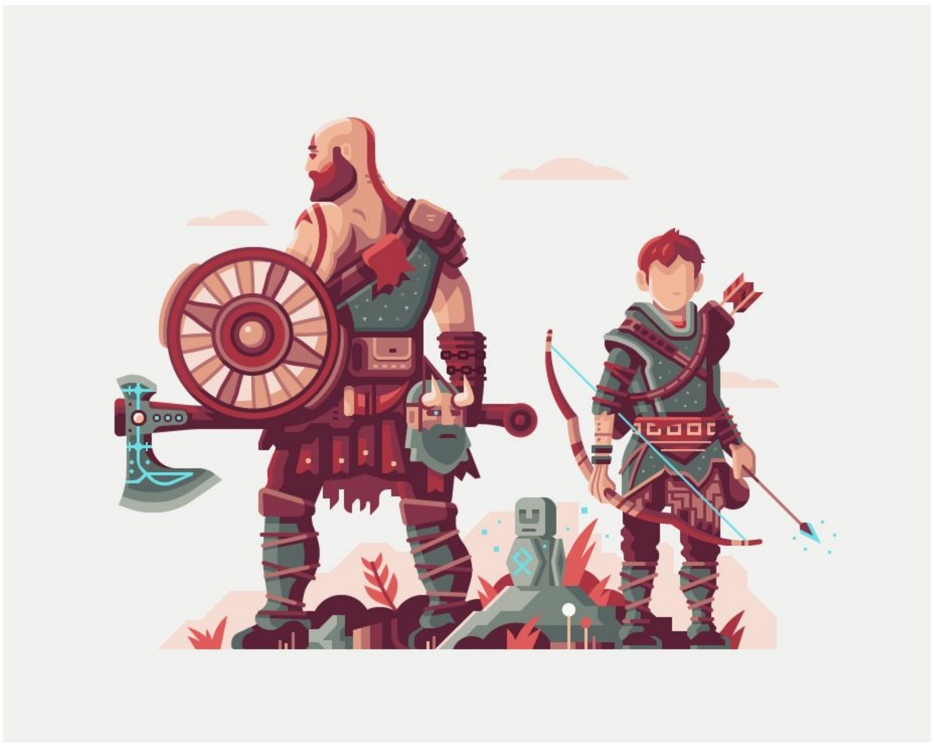
Tiny wars

Artificial Intelligence Documentation



Valentina Tosto - 905749
Victor Untila - 903147

A.A. 2017/2018 - Artificial Intelligence for Videogames

Playlab fOr inNovation in Games

CONTENT

Introduction

Tiny wars is a prototype of a real time strategy game where the player has at his disposal various squadrons of units to command. The idea is to try to give the player the experience of leading an army and living the warfare like in reality. To make this possible we focused mainly on: squad formations, coordinated movement and decision making at a squadron level.

This document is mainly divided in two major parts: Movement and Decision Making. Each part will describe in detail the techniques and algorithms used to obtain the desired behaviour.

All the implementation was done using Unity 2018.2.11f1. For an overview of the project folder consult the *Project folder structure* section.

Description

From now on in this document the term *Agent* refers to a squadron of nine warriors, while overall the term *NPC* refers to both squadrons and turrets. The purpose of developed NPCs is to defend as much as possible their camp from player attacks. In regards with decision making, it is intended that each squadron performs its own decisions as a single independent unit.

There are three types of NPCs in the game: the patrol agent (see fig. 1), that wanders between a set of points looking for a player squad, the defender agent, that is stationary in the camp and attacks the player squads when they try to infiltrate in the area, and finally the turret, which is activated by patrol agents, when they retreat to the center of the camp, and its aim is to attack the nearby player squads.

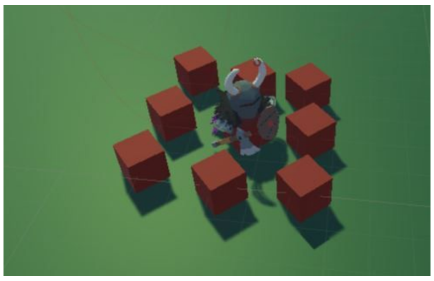


Fig. 1 - Patrol or defender agent

Patrol agent

The patrol agent's purpose is to wander between a predetermined set of patrol points while controlling the presence of a player squads within a specific view range.
This type of agent has an aggressive behaviour, because, when it detects a player squad it will chase it, without getting away too much from the initial point of chasing. Once it has reached the player squad, it will begin to attack it. If in the fighting the NPC agent remains with 30% or less of its forces, it will flee to the safety to the center of the camp.

When they arrive at the camp, they will warn other types of agents, in particular turrets, of the possible player approach. At the center of the camp (see fig. 2) the warriors still alive will start to recharge their health until it is full, while those dead will be replaced by new ones. Once completed the recovery of the health points and dead warriors, the patrol agent will return to its patrol behaviour.

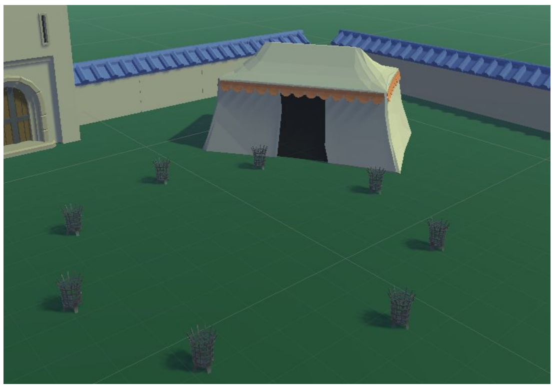


Fig. 2 - Recovery point

Defender agent

The defender agent, as its name says, has a more defensive behaviour than the patrol. Its behaviour consists in defending a certain area and engaging player squads only if they get too close. When it detects an enemy begins to chase it, but after a specific distance it returns to initial position in order to continue the defense of the area.
Unlike the patrol agent, the defender fights until it dies, without to come back in the camp to recharge itself.

Turret

Finally, the turret is a passive NPC (see fig. 3), because there is no movement, but it is designated from the patrol squads just for the alert of the camp and the attack on player squads within the range. When a patrol agent returns in the area, the light of turrets becomes yellow, which means that they are on alert for possible nearby enemies. The light switches to red when the player squad is in the range of a turret , then it begins to attack, shooting a bullet every 2 seconds aiming randomly between the warrior in the player squadrons.

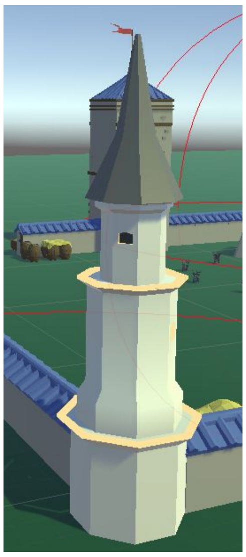


Fig. 3 - Turret

NPC

Movement

made by Victor Untila

Since in this game we have squadrons of warriors, it is needed to move them in a cohesive way, having already made the decision that they should move together. This is usually called formation motion.

Formation motion is the movement of a group of characters so that they retain some group organization. In the simplest form it can consist of moving in a fixed geometric pattern such as a "V" or a square. (see fig. 4)

For this project it was implemented a simple square formation: three lines with three warriors per line.

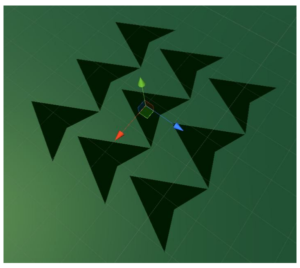
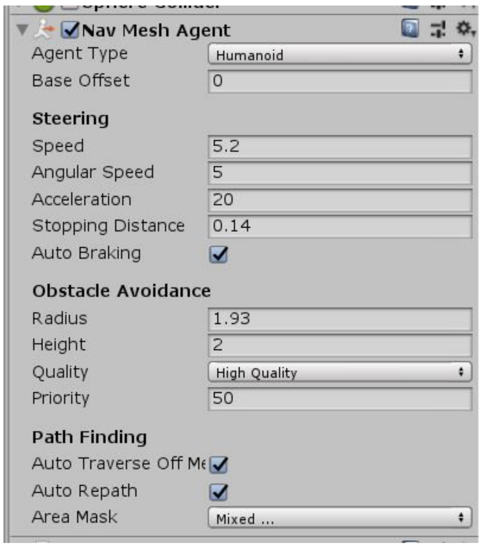


Fig. 4 - Formation pattern

The implementation of the formation motion was inspired by the "two-level formation steering" technique explained by Ian Millington in A.I for Games. The idea is that there is a "virtual leader" that has the purpose of pathfinding and then steers the formation pattern accordingly, while the single character steers to stay in formation when follows its slot in the formation pattern. This system is very flexible, because each character can have their own collision/obstacle avoidance behaviors and any other compound steering required.

In Unity the virtual leader was implemented as follows: an empty game object was created which has as many childs (other empty gameobjects) as positions in the formation we want to add. Each child has a specific coordinate as its function is a "slot" for a character. So for example if the first one is (0, 0, 0) the second will be (2, 0, 0) and so on.

In this way we build the pattern of our formation. In order to make the pattern move, a *Nav Mesh Agent* component was added to the virtual leader *GameObject*.



Nav Mesh Agent	
Agent Type	Humanoid
Base Offset	0
Steering	
Speed	5.2
Angular Speed	5
Acceleration	20
Stopping Distance	0.14
Auto Braking	✓
Obstacle Avoidance	
Radius	1.93
Height	2
Quality	High Quality
Priority	50
Path Finding	
Auto Traverse Off Me	✓
Auto Repath	✓
Area Mask	Mixed ...

An important note about the *Nav Mesh Agent* is that it has a very low angular speed. This is because whenever the formation will change the direction of movement this hasn't we don't want it to "rotate" towards the new direction. Only the characters assigned to the formation will rotate towards the new direction.

This does not represent strictly a functional choice, it was decided to proceed in this way mainly because it looked cleaner and better whenever there was a sudden change of direction in movement for this type of formation (square formation).

For a different type of formation, for example a “V” shape formation has the angular speed parameter sets higher, in this way the front of the formation will be always face the direction of movement.

The virtual leader is the same both for AI squadrons and player squadrons. The main difference is how the movement orders are received: for player squadrons it is decided by clicking on somewhere on the terrain, while for AI squadrons the destination of their movement is set, based on the decision making processes.
About that, player squadrons will have a script attached to their virtual leader called *PlayerFormationManager.cs,* while AI squadrons will have *EnemyFormationManager.cs*.
These scripts are responsible mainly for movement, attack, chasing ecc, and they hold a list of *Warriors* which are in the squadron.

The Warrior
To implement the warrior, a cube *GameObject* was created with a *Rigidbody* and a *BoxCollider* component attached to it.

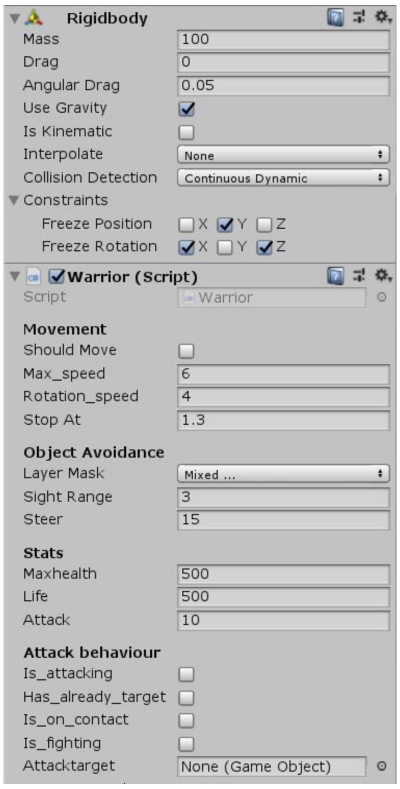
The most important component however it’s the *Warrior.cs* script. This is responsible of the movement of the single warrior and has a basic obstacle avoidance behaviour. Also it contains methods for attacking another warrior and for taking damage.

The obstacle avoidance consists in raycasting in 3 different directions: in front, 45° to the left and 45° to the left. If one of the rays hits something the direction of movement of the warrior will be recalculated based on the normal vector of the hit ray.

A warrior will be always assigned to a formation and will have its slot, so during movement it will follow the position of its slot (since the pattern of the formation its moving). The movement is done by using the *MovePosition* method.

An important note here is that the speed of movement of the single warrior must be greater than the speed of the virtual leader (which moves the formation pattern).

If the warrior for some reason is far behind its slot in the formation (for example greater than 5 meters) it will increase its max speed to try to keep up with the formation; once it is in his position it will return to the original speed value.



Rigidbody	
Mass	100
Drag	0
Angular Drag	0.05
Use Gravity	✓
Is Kinematic	☐
Interpolate	None
Collision Detection	Continuous Dynamic
Constraints	
Freeze Position	☐ X ✓ Y ☐ Z
Freeze Rotation	✓ X ☐ Y ✓ Z
Warrior (Script)	
Script	Warrior
Movement	
Should Move	☐
Max_speed	6
Rotation_speed	4
Stop At	1.3
Object Avoidance	
Layer Mask	Mixed ...
Sight Range	3
Steer	15
Stats	
Maxhealth	500
Life	500
Attack	10
Attack behaviour	
Is_attacking	☐
Has_already_target	☐
Is_on_contact	☐
Is_fighting	☐
Attacktarget	None (Game Object)

Decision making

made by Valentina Tosto

Regard to decision making, this is a key tool in artificial intelligence, in order to perform an action, based to the behaviour of the agent in response to external events. Decisions can be taken for the single agent or collaborative in a group, but in general movements and actions are carried out by the individual character, while at the group level what is defined is the strategy. In the project it is chosen to use *finite state machines* as decision making technique, because the enemy's behaviour (both stationary and patrol) fits better with a FSM, due to simpleness of design, so it was not necessary extend it with a behaviour tree.
The role assigned to these FSMs is to trigger enemy's actions through boolean values that address the agent to a decision, making the execution more modular and easier to manage.

The following schemes show the FSM of patrol (see fig. 5), defender (see fig. 6) and turret (see fig. 7) NPCs , with a description of *Enter, Stay and Exit* actions (if they are required). For each list is added a specific action that corresponds to a method.

The patrol FSM is composed by six states and eleven transitions:

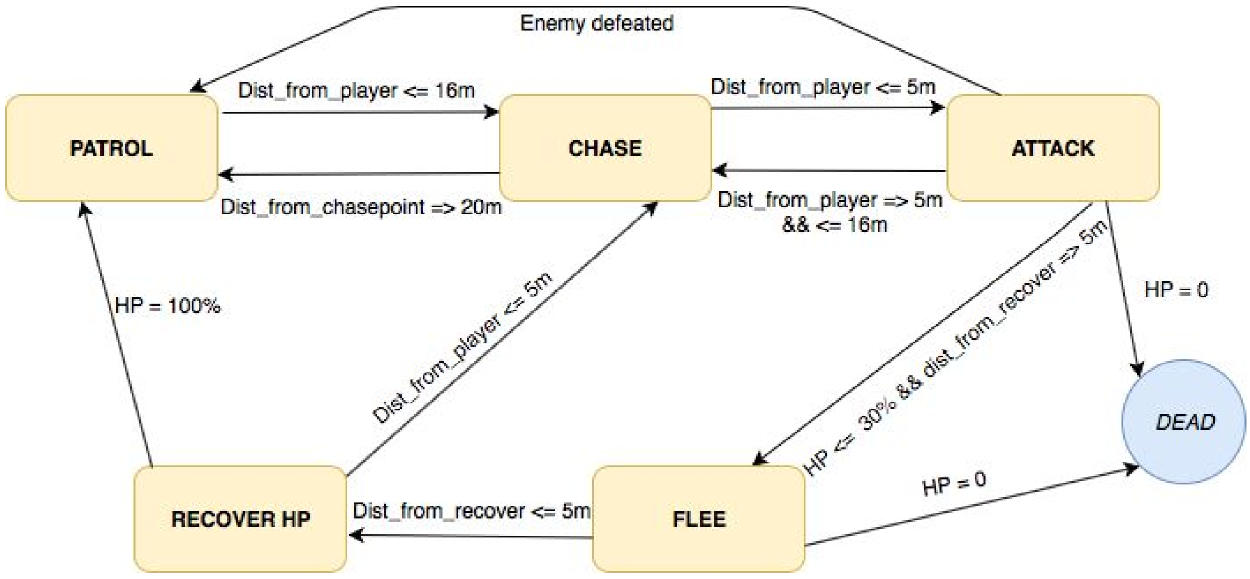


Fig. 5 - Patrol behaviour FSM

- **PATROL** → In this state the agent walks between a specific set of points on the map, while controls the presence of a player squad in a range of 16 meters. There is an *Enter* action (*Set Goal*), where is set a boolean value that triggers the corresponding method related to patrolling; a *Stay* action (*Walk*), where there is simply a debug that shows the action performed by the enemy and an *Exit* action (*StopPatrol*), where the boolean value, set initially, is changed to false in order to pass to the next state.

From the patrol state there is an outgoing transition to the chase state, that triggers when a player squad is present in the range. In the detail there is an array of *Collider*, where is passed the agent's position and range, then each *GameObject* that corresponds to player squad is added to another array.

- **CHASE** → Here the agent chases the player squad up to 5 meters away from him and up to 20 meters away from the initial chasing point. In the *Enter* action (*StartChase*), if the enemies detected are more than one, with a random choice, the agent starts to follow a single player squad, also in this case the action is triggered by a boolean value, that indicates the presence of a target. The *Stay* action (*Chase*) shows only a debug of chase, while in the *Exit* action (*StopChase*) the target is removed, if there are no more enemies in the range.
  From the chase state there are two outgoing transitions. The transition to attack state is verified when the distance from player squad is less or equal than 5 meters; in this case, similarly to patrol state, an array of enemies is filled with the player squads present within the range. Whilst the transition to patrol state is verified when the distance from the initial chasing point is greater than 20 meters.

- **ATTACK** → This is a complex state, where it happens the real combat between agent and player squads, because the enemy has to evaluate more decisions, based to the outcome of the battle. In the *Enter* action (*StartAttack*) the attack is triggered, always with a boolean value, in the *Stay* action (*Fight*) is printed a debug during the fight and in the *Exit* action (*StopAttack*) the array of player squads, that were detected by agent, is cleaned and the target is set to null.
  There are four possible outgoing transitions from the attack state: the transition to chase state is verified when the distance between the agent and the player squad is greater than 5 meters, instead the transition to flee state triggers when the health of the agent is less or equal than 30% (that correspond to 30% of warriors), because the agent tries to escape to recovery point present in the camp when it is weakened.
  The other two cases refer to the outcome of the battle: the transition to dead state triggers in case of defeat, while that to patrol state is verified if the enemy wins the fight, ready to find the next player squad.

- **FLEE** → The flee state is verified when the agent is strained by the battle with the player, that is whenever the number of warriors is less or equal than 30%. In the *Enter* action (*StartFleeing*) a boolean value triggers the beginning of flee method, while in the *Stay* action (*Flee*), as usual, is printed a debug.
  There are two outgoing transitions: one to dead state triggers if the agent is attacked by a player squad during the escape, in this case it has to fight until the end, whilst the transition to recovery HP state is verified when the distance from the recovery point is less than 5 meters, so the warriors remained start to recharge their health and the dead ones will be respawned.

- **RECOVER HP** → This state is a comfort zone for the agent, because the few remaining warriors can recharge their health by 10% every second, and the dead ones will be respawned every 5 seconds. In the *Enter* action (*StartRecovery*) are set two boolean

values, one to trigger the recharge method once and to start a *Coroutine,* the other to control if the agent is still in the range of recovery point. The *Stay* action (*RechargeHP*) is used only as debugging and the *Exit* action (*StopRecovery*) indicates that the agent is out of the range of 5 meters by recovery point, so it cannot recharge anymore.
Regarding the outgoing transitions, also in this case, there are two: the transition to chase state triggers when a player squad assaults the enemy while it is refilling its health. In particular, as for the patrol and chase state, there is an array of *Collider* with all *GameObjects* within 5 meters, but only ones related to player squad are added to another array, and then the boolean value of flee is set to false. The second transition to patrol is verified when the recharge is completed and the warriors of enemy squad are totally respawned.

- **DEAD** → This is the end state, because there are no outgoing transitions, where all warriors of an enemy squad are defeated during the fight with the player.
  There is only the *Enter* action (*Dead*), because it must be called once, where the *GameObject* of the agent is destroyed.

The defender FSM is composed by five states and eight transitions:

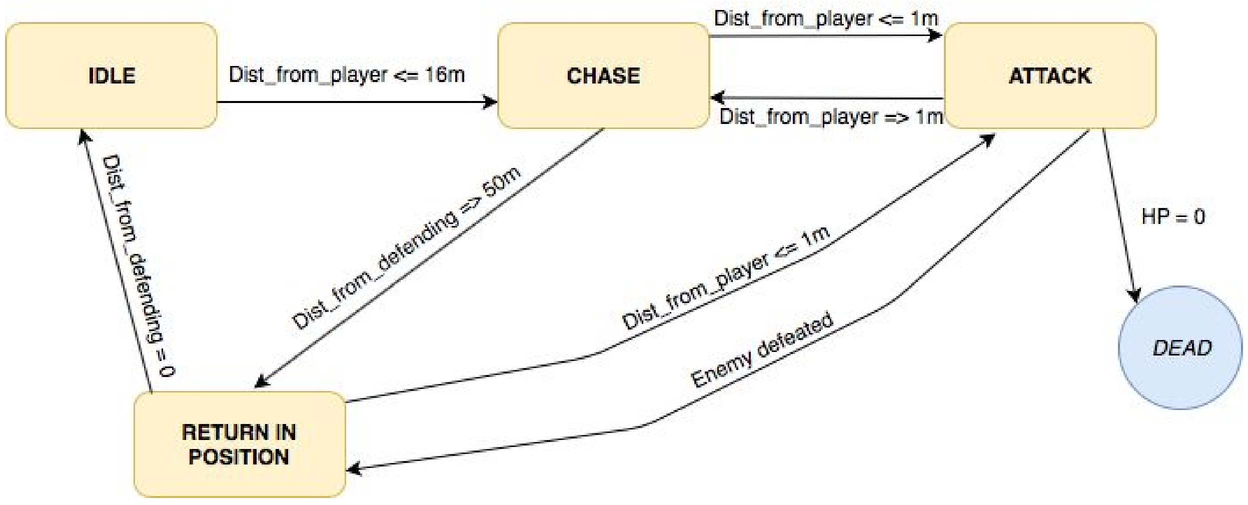


Fig. 6 - Defender behaviour FSM

- **IDLE** → In this state the defender squad stands still and checks the presence of a player squad in a range of 16 meters. There is the *Enter* action (*GoToCamp*) that is triggered when the agent has to come back to initial position as idle, so is set a boolean value related to defending; then in the *Stay* action (*Watch*) there is a debug of its action and in the *Exit* action (*ExitIdleState*), it passes from idle to chase state, changing the boolean value.
  Regard to outgoing transitions, there is only one to chase state, that triggers when a player squad is present in a range of 16 meters. In detail it is used a *OverlapSphere* to

wrap all *GameObjects* in that area, and only their related to player are inserted in a specific array.

- **CHASE** → In the chase state the agent follows the player squad up to 1 meter. In the *Enter* action (*StartChase*), if the enemies detected are more than one, with a random choice, the agent starts to follow a single player squad, also in this case the action is triggered by a boolean value, that indicates the presence of a target. The *Stay* action (*Chase*) shows only a debug of chase, while in the *Exit* action (*StopChase*) the target is removed, if there are no more enemies in the range.
  From the chase state there are two outgoing transitions. The transition to attack state is verified when the distance from player squad is less or equal than 1 meter; in this case, as in idle state, there is an array of enemies that is filled with the player squads present within the range, to control if at least one player squad is so close to attack. Another transition is to return in position state, that triggers when the distance from initial point of defense is greater or equal than 50 meters.

- **ATTACK** → As for the Patrol agent, here it happens the real combat between agent and player squads, but in this case the agent fights till it dies and doesn't escape when its health is low. In the *Enter* action (*StartAttack*) the attack is triggered, always with a boolean value, in the *Stay* action (*Fight*) is printed a debug during the fight and in the *Exit* action (*StopAttack*) the array of player squads, that were detected by agent, is cleaned and the target is set to null.
  There are three possible outgoing transitions from the attack state: the transition to chase state is verified when the distance between the agent and the player squad is greater than 1 meter, then there is a transition to return in position state that triggers when the enemy wins the fight and finally the transition to dead state when the agent loses against the player squad.

- **RETURN IN POSITION** → This state indicates the return of the agent towards his initial point of defense. In the *Enter* action (*ReturnToPosition*) the boolean value related to the return is activated, in the *Stay* action (*Returning*) is shown a debug and in the *Exit* action (*StopReturning*) the agent stops moving in order to attack or stand in his camp. There are two outgoing transitions: one to attack state, when a player squad is in the range of 1 meter, and one to idle state, when the distance from the defense point is equal to zero.

- **DEAD** → As for the Patrol agent, this is the end state, because there are no outgoing transitions, where all warriors of an enemy squad are defeated during the fight with the player.
  There is only the *Enter* action (*Dead*), because it must be called once, where the *GameObject* of the agent is destroyed.

The turret FSM is composed by three states and four transitions:

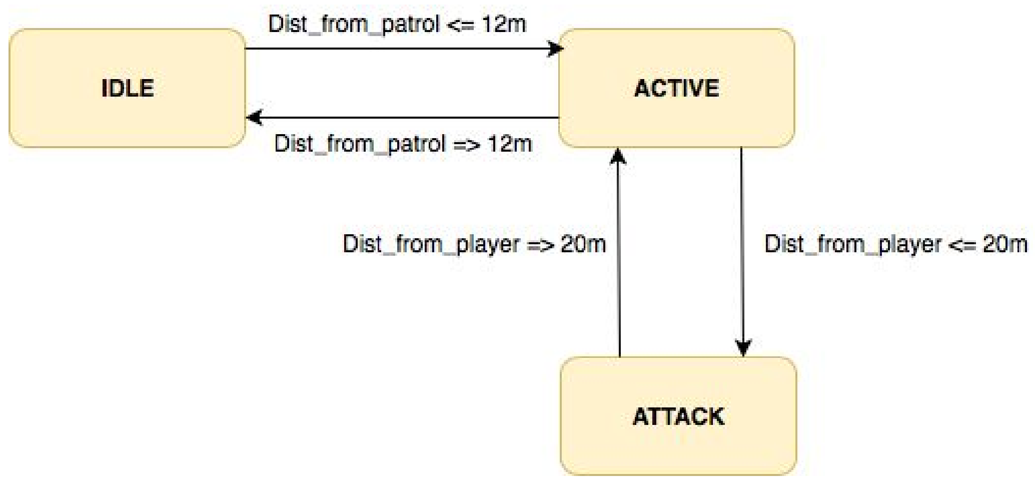


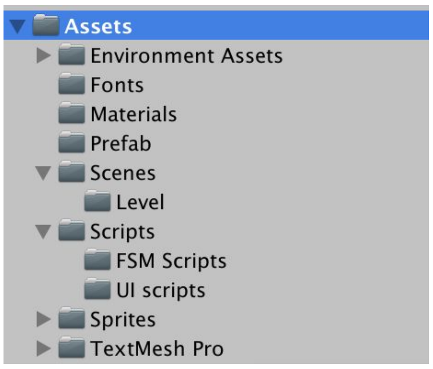
Fig. 7 - Turret behaviour FSM

- **IDLE** → In this state the turret is disabled and just controls if a patrol squad is present in the camp, so that the alarm can be activated. There is the *Enter* action (*Wait*), where two methods are called: one for the yellow light, that indicates alarm off and one for the red light, that indicates alarm on. But initially, when the turret is in idle state, both lights are off. The only outgoing transition is to active state, that is triggered when one of the patrol squads is in the range of 12 meters from alarm range.

- **ACTIVE** → In the active state the turret is allerted (yellow light) and, according to the position of a player squad, it enters in attack state or remains unchanged, until the patrol squad is in the camp. It has the *Enter* action (*Alerted*), where the alarm is set to on and the light becomes yellow, while the *Exit* action (*StopAlert*) disabled the turret. There are two outgoing transitions: the transition to idle state is triggered when there are no more patrol squads in the range of 12 meters, while the transition to attack state controls, always with a *OverlapSphere*, if there is a player squad in a range of 20 meters and, in this case, the light of turret becomes red.

- **ATTACK** → This is the most meaningful state, where the turret attacks the player squad within the range, choosing a random warrior and shooting an arrow every two seconds. In the *Enter* action (*StartAttack*) the light becomes red and the turret begins the real attack on the player squad, in the *Stay* action (*Attack*) is shown a debug, while in the *Exit* action (*StopAttack*) the turret turns off the alarm and returns to yellow, because the player squad left the area. There is an outgoing transition to active state, when the player squad, that it was attacking, is out of range and there are no others.

Project folder structure



The project folder related to *Assets* is divided into eight types of resources, but the essentials are:

- **Prefab** → In this folder there are the prefabs of warriors, bullets, canvas, turrets and formations.

- **Scenes** → There is the scene of level with the navmesh used.

- **Scripts** → The most important folder, there are two subfolders (*FSM scripts* and *UI scripts)* plus scripts with scene camera, player and NPCs movement. The *FSM scripts* folder contains the scripts that implement the decision making system of the NPCs, that are *FSM* (the general class where is built the definition of states and transitions), *FSMDefender, FSMPatrol* and *FSMTurret.* Whilst the *UI scripts* folder contains the scripts to show the health of player warriors (*PlayerHealth*) and to select the whole squad (*SelectionSquad*), clicking on its name, on the UI.
  Outside the subfolders there are the scripts concerning the movement both player and agents squads, as *PlayerFormationManager, EnemyFormationManager, TurretBehaviour, Warrior and Arrow.* Lastly, there is a script relative to scene camera (*RTSCamera*) with a recurrent implementation of the camera in RTS videogames.

- **Sprites** → The folder with the images of UI and arrows used to build the movement's direction of warriors.

The other folders (*Environment Assets, Fonts, Materials and TextMesh Pro*) refer to materials, fonts and sprites used in the scene environment.