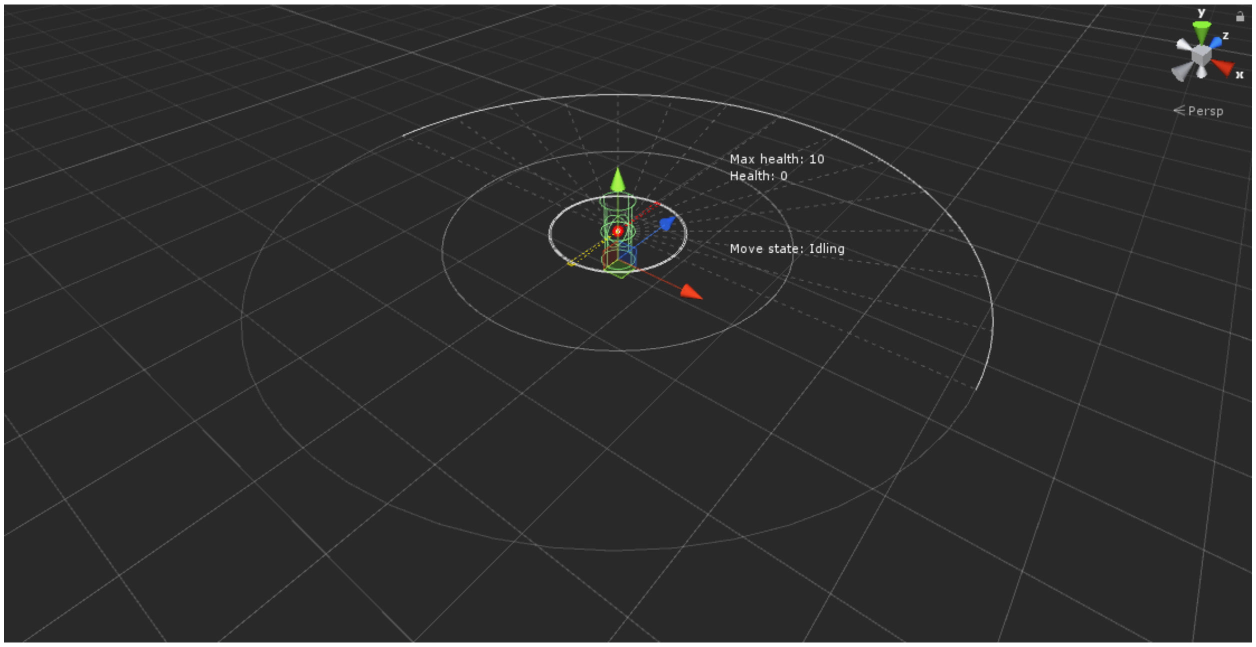
ELIOT MANUAL

Contents

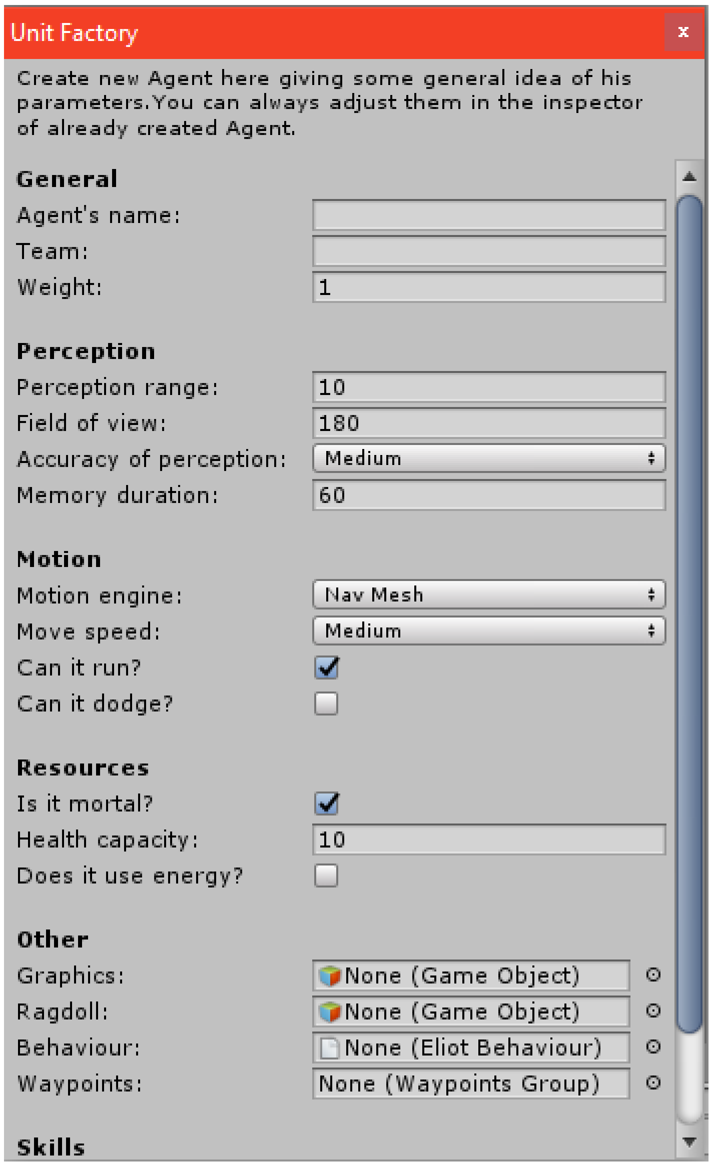
1 Agent



1.1 Description

Agent encapsulates all the components of the behaviour of AI and works as a driver for the Behaviour model defined by user. It is a physical simulation, embodiment of game characters.

1.2 Unit Factory



Create new Agent here giving some general idea of his parameters.You can always adjust them in the inspector of already created Agent.

General

Agent's name:	
Team:	
Weight:	1

Perception

Perception range:	10
Field of view:	180
Accuracy of perception:	Medium
Memory duration:	60

Motion

Motion engine:	Nav Mesh
Move speed:	Medium
Can it run?	✓
Can it dodge?	☐

Resources

Is it mortal?	✓
Health capacity:	10
Does it use energy?	☐

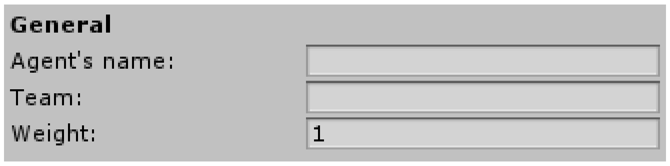
Other

Graphics:	None (Game Object)
Ragdoll:	None (Game Object)
Behaviour:	None (Eliot Behaviour)
Waypoints:	None (Waypoints Group)

Skills

Unit Factory is the easiest way to create new Agent giving some general idea of his parameters. You can always adjust those parameters in the Inspector of already created Agent. Open Unit Factory by choosing *Eliot/Unit Factory* menu item.

1.2.1 General



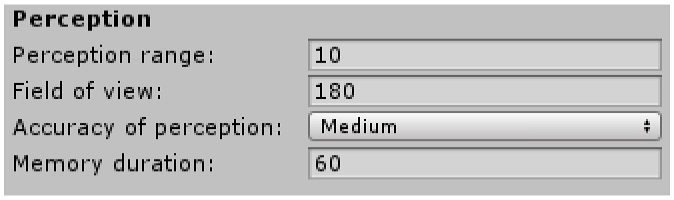
General

Agent's name:	
Team:	
Weight:	1

Agent's name is the **name** of the instantiated GameObject that represents the Agent.

Team is the property of Unit component that gets added to a GameObject along with Agent component. **Team** property helps distinguish between friendly and unfriendly Units (which are probably also Agents, but not necessarily).
Weight defines how hard it is to push Agent around (Agents can push other Agents around using Skills).

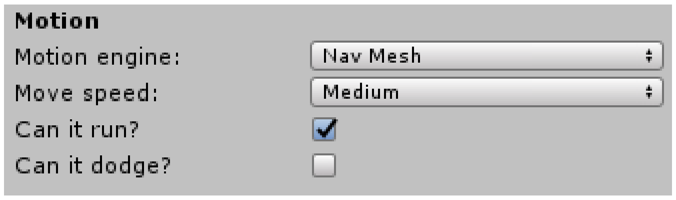
1.2.2 Perception



Perception	
Perception range:	10
Field of view:	180
Accuracy of perception:	Medium
Memory duration:	60

Perception range defines how far Agent is able to see things.
Field of view is the width of the Agent's view angle, measured in degrees along the local Y axis.
Accuracy of perception defines the number of rays that will be casted by Agent within its Field of view. *Low* is 7 rays, *Medium* is 15 rays and *High* is 35 rays.
Memory duration is the duration in seconds for which the Agent is able to remember its targets.

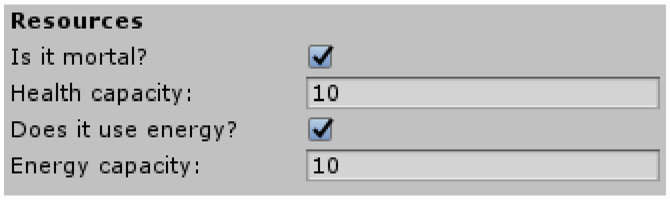
1.2.3 Motion



Motion	
Motion engine:	Nav Mesh
Move speed:	Medium
Can it run?	✓
Can it dodge?	☐

Motion engine defines what system will be used for pathfinding. Can be changed later in the Inspector. At the moment there are three available engines – Unity's NavMesh, A* Pathfinding Project Pro and Turret for static Agents.
If **Should it move** is false, **Agent.Motion.Type** is set to *Turret.* Agent will not be able to move, except from the rotation of the 'head', to imitate turret-like patrolling.
Move speed defines the walking speed of the Agent in case it's motion type is Creature.
Low **Move speed** corresponds with 1.5 walking speed, *Medium* is 3 and *High* is 5.
If **Can it run** is true, the Agent's running speed is set to double the value of its walking speed. Otherwise, it is equal to the walking speed.
If **Can it dodge** is true, **Agent.Motion.DodgeSpeed** is set to 10 and **Agent.Motion.DodgeDuration** is set to 0.1 seconds. These settings enable Agent to move quickly over a short timespan away from the danger or towards a target.

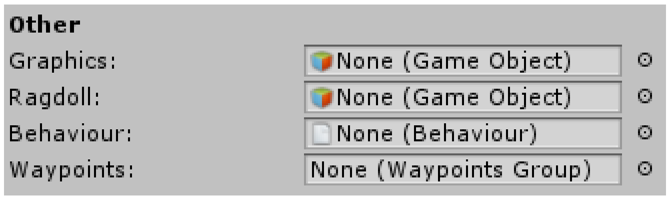
1.2.4 Resources



Resources	
Is it mortal?	✓
Health capacity:	10
Does it use energy?	✓
Energy capacity:	10

If **Is it mortal** is true, it is possible to 'kill' the Agent by decreasing its Health to 0.
Health capacity is the initial amount of health Agent has.
If **Does it use energy** is true, Agent is able to use his Energy for movement or Skills and to regenerate it over time.
Energy capacity is the initial amount of energy Agent has.

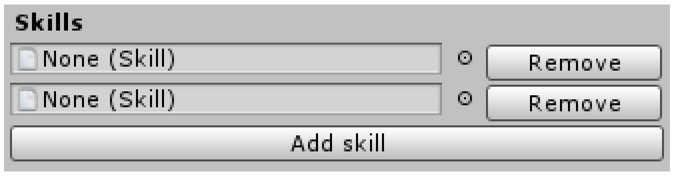
1.2.5 Other



Other	
Graphics:	None (Game Object)
Ragdoll:	None (Game Object)
Behaviour:	None (Behaviour)
Waypoints:	None (Waypoints Group)

Graphics is the prefab that is used to display Agent, with a possibility to link Agent's actions with corresponding animations using Unity's Legacy Animation system or Mecanim.
Ragdoll is the prefab that's going to be spawned upon Agent's death.
Behaviour is the Eliot's Behaviour model, which is used by Agent to define the rules by which it operates in the world. If there are already created Behaviours, you can just select one of them in Unit Factory.
Waypoints is one of the Waypoints Groups already existing in the scene. Defines Agent's home area or a path. Agent can operate well without Waypoints, so it's not a mandatory field.

1.2.6 Skills



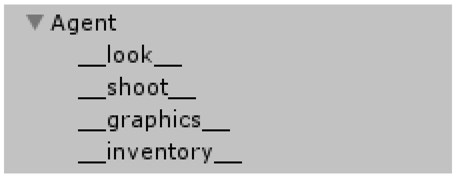
Skills
None (Skill) | Remove
None (Skill) | Remove
Add skill

You can specify Skills that Agent will be able to use by clicking **Add Skill** button and choosing already existing Skills.

1.2.7 Create

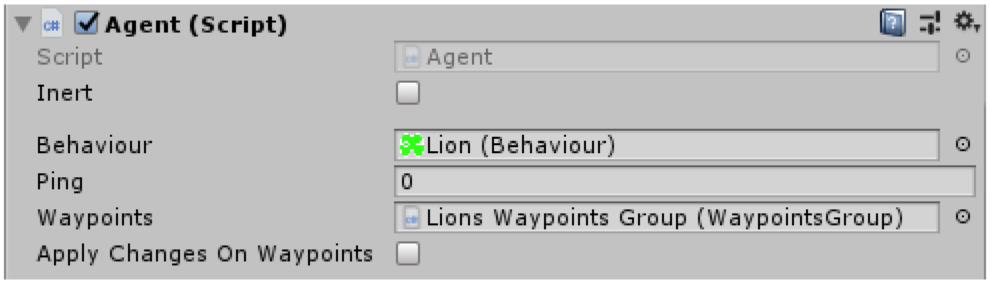
When you are finished filing the form, click **Create** button. This will instantiate a new Agent with predefined parameters.

Agents, created with the help of Unit Factory, have objects that are used as an origin for perception, origin of casting Skills, a container for Agent's graphics and a container for Agent's Inventory already created and adjusted.



```
▼ Agent
    __look__
    __shoot__
    __graphics__
    __inventory__
```

1.3 Properties



Agent (Script)	
Script	Agent
Inert	☐
Behaviour	Lion (Behaviour)
Ping	0
Waypoints	Lions Waypoints Group (WaypointsGroup)
Apply Changes On Waypoints	☐

1.3.1 Inert

Making **Inert** true prevents Agent from displaying any behaviour, leaving active component on the GameObject.
Inert Agents can be used to get access to Agent's API and to make other Agents think that a GameObject is also an Agent without letting it interfere with other MonoBehaviours.

1.3.2 Ping

Ping is the amount of time in seconds that passes between updates. Can be used for easy adjustment of difficulty of a game without losing features of Agents (like Skills) and without interfering an ingame economy (e.g. amount of health, energy etc.). Also can be used to ease the job for CPU (bigger ping => more FPS).

1.3.3 Behaviour

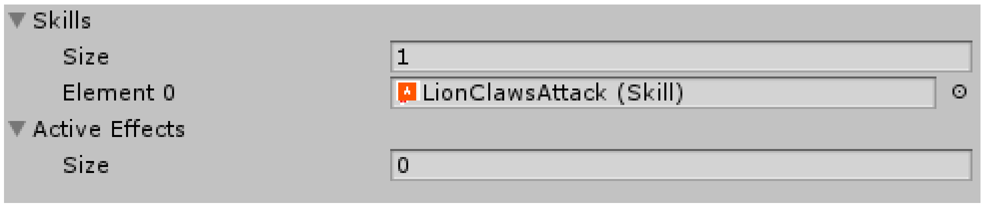
A reference to the file that represents a Behaviour model that will be used by this Agent.

1.3.4 Waypoints

A reference to a GameObject with WaypointsGroup component, which acts as a specific area of Agent, or his path.
Agent can change his Behaviour while following a specified path. This way it is possible to create sequences of actions that Agent will do, to make him, for instance, complete a quest.

1.3.5 Apply Changes On Waypoints

A boolean value that makes Agent agree to be applied changes to, as set by the current configuration of a Waypoint, that Agent passes over. If the value if false, Agent will ignore the request of a Waypoint to change Agent’s parameters.



▼ Skills	
Size	1
Element 0	LionClawsAttack (Skill)
▼ Active Effects	
Size	0

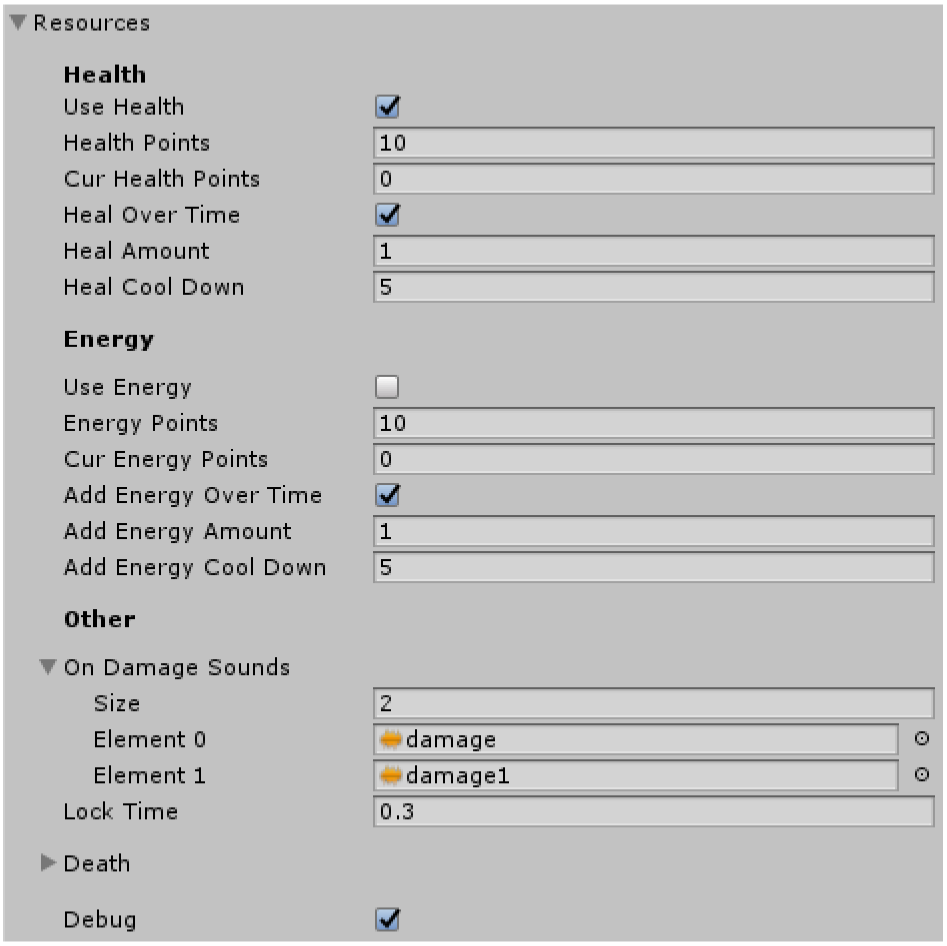
1.3.6 Skills

A list of Skills, that Agent can use. This list can be modified at runtime, for example, when an Agent wields an Item that might temporarily add new Skill to Agent’s disposal.

1.3.7 Active effects

A list of Skills, that Agent is currently being affected by. Skills can be applied to Agents not only with a single call, but they can actually have some duration of action, applying their effect with a given frequency. It allows using skills for many kinds of interaction, from melee attacks and fireballs to buffs, DoTs etc.

1.4 Resources



▼ Resources

Health

Use Health	✓
Health Points	10
Cur Health Points	0
Heal Over Time	✓
Heal Amount	1
Heal Cool Down	5

Energy

Use Energy	☐
Energy Points	10
Cur Energy Points	0
Add Energy Over Time	✓
Add Energy Amount	1
Add Energy Cool Down	5

Other

▼ On Damage Sounds	
Size	2
Element 0	damage
Element 1	damage1
Lock Time	0.3
▶ Death	
Debug	✓

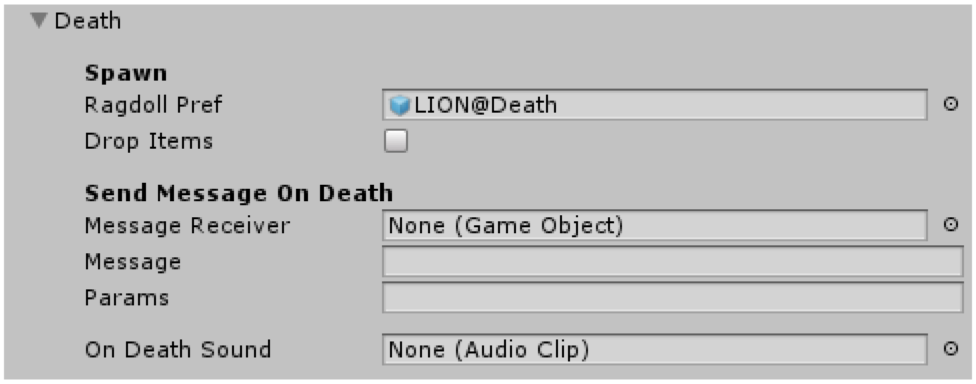
Keep information about finite resources (Health, Energy etc.). Resources usage is optional. Encapsulates Agent’s ability to take damage or to use energy to perform certain actions, like walking, running and using Skills.

If **Use Health** is true, Agent’s health-related economy will be simulated, letting it to be damaged, healed and killed.
Health Points is the maximum amount of health the Agent can have.
Cur Health Points is the current amount of health the Agent has. Its initial value is **Health Points**.
If **Heal Over Time** is true, **Heal Amount** of Agent’s health will be automatically replenished every **Heal Cool Down** seconds.

If **Use Energy** is true, Agent’s energy-related economy will be simulated, letting it to spend energy for movement and Skills usage.
Energy Points is the maximum amount of energy Agent can have.
Cur Energy Points is the current amount of energy the Agent has. Its initial value is **Energy Points**.
If **Add Energy Over Time** is true, **Add Energy Amount** of Agent’s energy will be automatically replenished every **Add Energy Cool Down** seconds.

On Damage Sounds is a collection of AudioClips, a random one of which is played every time Agent gets damage.
Lock Time is the amount of time the Agent's Motion and Skills usage is frozen (or interrupted) upon taking damage.

1.4.1 Death



▼ Death

Spawn

Ragdoll Pref	LION@Death
Drop Items	☐

Send Message On Death

Message Receiver	None (Game Object)
Message	
Params	
On Death Sound	None (Audio Clip)

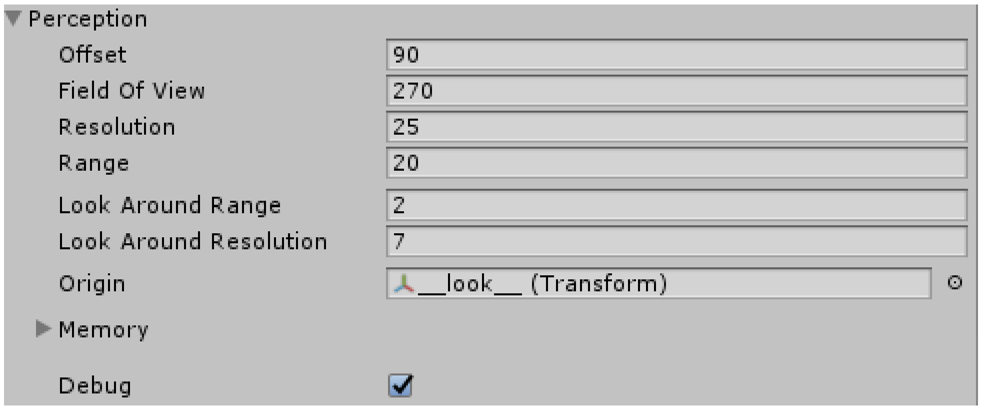
Keeps the actions to be performed upon Agent's death (when his health equals to zero). Can spawn specified ragdoll, drop items from Inventory, send messages to specified objects and play specific AudioClip.

Ragdoll Pref is a prefab that is spawned upon Agent's death. The GameObject that represents Agent gets destroyed.
If **Drop Items** is true, all the Items from Agent's Inventory are dropped upon its death.

Agent can notify other objects about its death by sending a message (using Unity's SendMessage(string message) method).
Message Receiver is the object that should be notified.
Message is a name of a method that should be invoked on **Message Receiver**.
Params is the parameters string that can be passed while sending a message (SendMessage(**Message**, **Params**))

On Death Sound is an AudioClip that is played upon Agent's death.

1.5 Perception


Perception

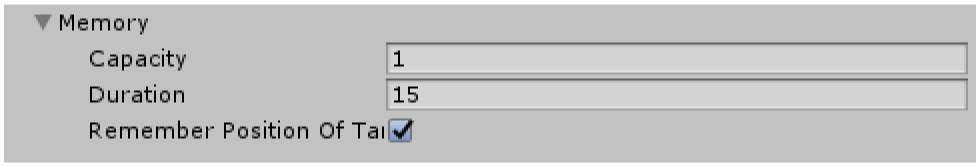
Offset	90
Field Of View	270
Resolution	25
Range	20
Look Around Range	2
Look Around Resolution	7
Origin	__look__ (Transform)
Memory	
Debug	✓

Encapsulates Agent's abilities to understand what is going on around. Does this by casting rays in specified range. User can specify Agent’s field of view, allowing areas where Agent cannot see (e.g. behind his back).
Number of rays influences Agent’s quality of perception, so it can be used to adjust the difficulty of a game and gives room for easy optimization.
Agents see the world in terms of Units (they see only GameObjects that have Unit as one of the components).

Offset is the measure in degrees of the rotation of the group of rays along the Y axis.
Field of view is the width of the Agent’s view angle, measured in degrees along the local Y axis.
Resolution defines the number of rays that will be casted by Agent within its **Field of view**.
Range defines how far Agent is able to see things.
Look Around Range defines how far Agent is able to see things even behind his back (a range of perception where field of view is 360 degrees).
Look Around Resolution defines the number of rays that will be casted by Agent within a 360 degrees field of view on a **Look Around Range** distance.
Origin is the Transform that is used as an origin of rays that constitute Agent’s perception.

If **Debug** is true, handles and gizmos that represent Agent’s Perception configuration will be displayed in Unity Editor.

1.5.1 Memory

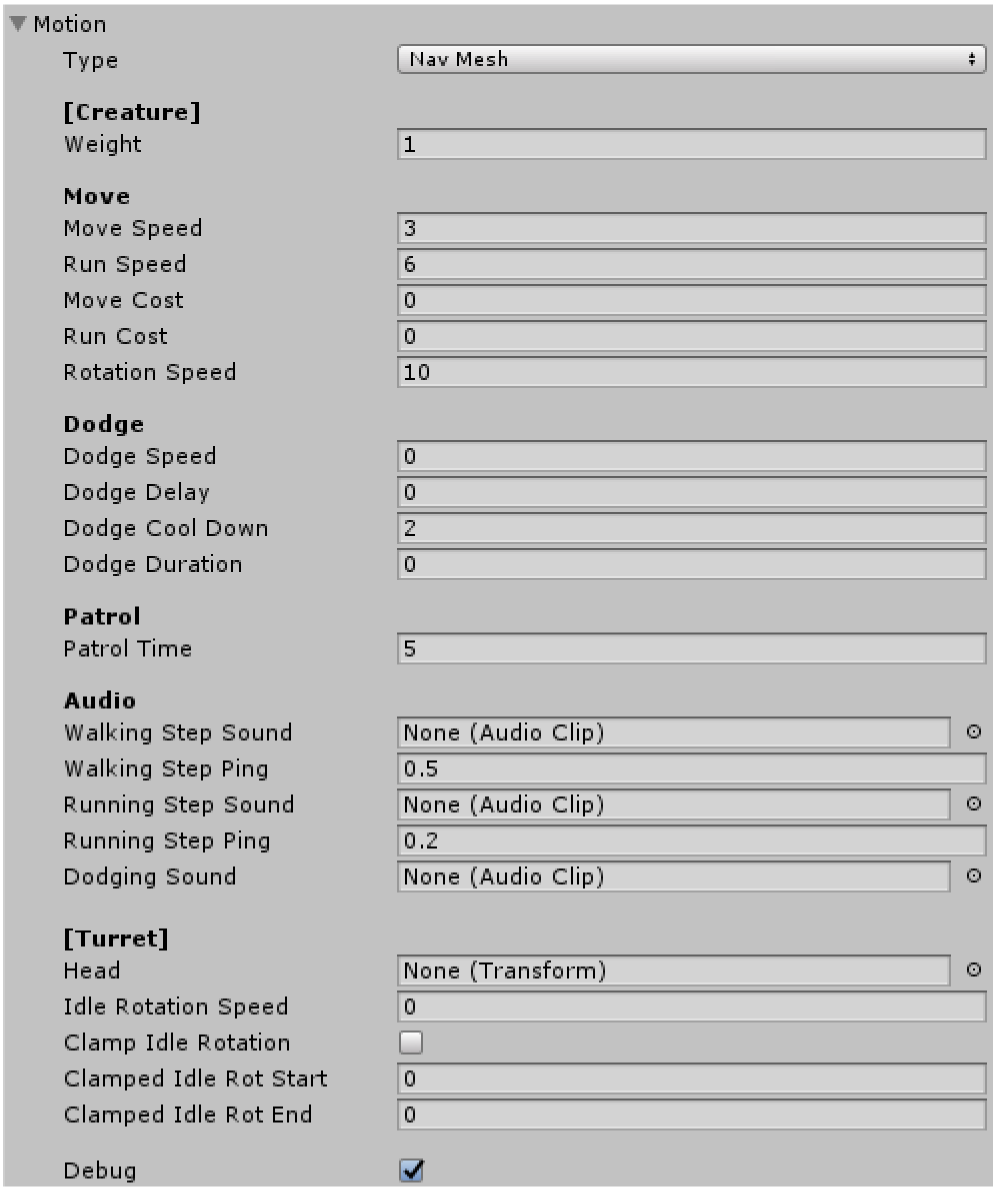

Memory

Capacity	1
Duration	15
Remember Position Of Tai	✓

Cache for Agent’s representation of the world. Agent sees the world in terms of Units. When he sees a Unit, he remembers it to further access the object itself or the position where Agent has seen it last time (this depends on the value of **Remember Position Of Target**).

Capacity is the maximum number of Units that Agent can keep in memory at the same time. When a new Unit is spotted and there is no place in memory, the first remembered Unit in the memory's stack is dropped to free the space for a new one.
Duration is the duration in seconds for which the Agent is able to remember its targets.
If **Remember Position Of Target** is true, Agent can access only the position where it has spotted the remembered target the last time. Otherwise, Agent can access the real position of a remembered Unit.

1.6 Motion



Motion	
Type	Nav Mesh
[Creature]	
Weight	1
Move	
Move Speed	3
Run Speed	6
Move Cost	0
Run Cost	0
Rotation Speed	10
Dodge	
Dodge Speed	0
Dodge Delay	0
Dodge Cool Down	2
Dodge Duration	0
Patrol	
Patrol Time	5
Audio	
Walking Step Sound	None (Audio Clip)
Walking Step Ping	0.5
Running Step Sound	None (Audio Clip)
Running Step Ping	0.2
Dodging Sound	None (Audio Clip)
[Turret]	
Head	None (Transform)
Idle Rotation Speed	0
Clamp Idle Rotation	☐
Clamped Idle Rot Start	0
Clamped Idle Rot End	0
Debug	✓

Encapsulates Agent's abilities to move around in space. Uses one of the available pathfinding engines. At the moment there are 3 engines available – NavMesh, Astar and Turret.

The first one operates in the world by setting target position of NavMesh component. The second one works just like the first one, but uses A* Pathfinding Project Pro instead of Unity's NavMesh. The third one is meant to be static with only one moving part (head), which can be configured to rotate with specified parameters.
Creature mode (NavMesh and A*) allows configuring the cost in energy points of walking and running. If an Agent doesn't have enough energy to run, he will walk even if the Run method is called. If an Agent doesn't have enough energy to walk, he will idle even if the Walk method is called.
Motion has two related interfaces - MotionActionInterface and MotionConditionInterface.

Type defines the Motion Engine that the Agent uses. Can be *NavMesh*, *Astar* or *Turret*. If the **Type** is *NavMesh* or *A**, the parameters under the "[Creature]" header are taken into account. If the **Type** is *Turret*, the parameters under the "[Turret]" header are taken into account.
If **Debug** is true, handles and gizmos that represent Agent's Motion configuration will be displayed in Unity Editor.

1.6.1 Creature

Weight defines how hard it is for other Agents to push this one around.

Move Speed is the walking speed of the Agent.
Run Speed is the running speed of the Agent.
Move Cost is the amount of energy needed for one second of walking. While Agent is walking, his energy will be reduced every second by this amount. If there is not enough energy for walking, the Agent idles.
Run Cost is the amount of energy needed for one second of running. While Agent is running, his energy will be reduced every second by this amount. If there is not enough energy for running, Agent walks.
Rotation Speed is the rate at which Agent rotates towards its target when asked to do so.

Dodge Speed is the speed of Agent's movement while dodging.
Dodge Delay is the amount of time in seconds that passes between calling the Dodge function and the apparent movement of an Agent.
Dodge Cool Down is the amount of time in seconds that passes after the last dodge before it is available again.
Dodge Duration is the amount of time in seconds that the Agent continuously performs the dodge.

Patrol Time is the amount of time that Agent spends standing while performing patrolling. (Patrolling is performing a repeated sequence of actions that includes walking to a random point in Agent's area, which is defined by Waypoints, and standing at place for specified amount of time before picking the next random point).

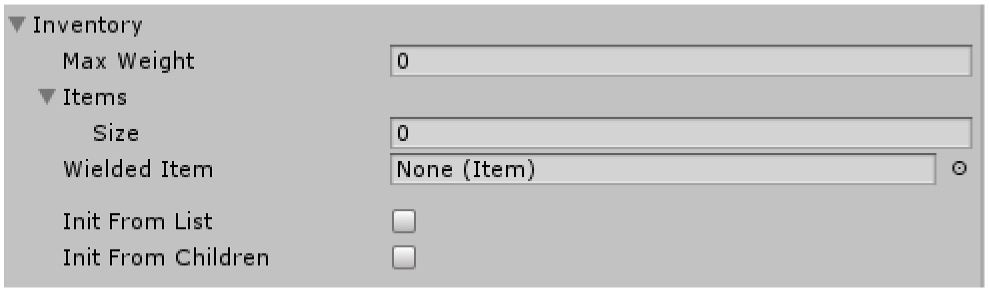
Walking Step Sound is an AudioClip that is played every time Agent makes a 'step' (that occurs every **Walking Step Ping** seconds).

Running Step Sound is an AudioClip that is played every time Agent makes a 'step' (that occurs every **Running Step Ping** seconds).
Dodging Sound is an AudioClip that is played every time Agent performs a dodge.

1.6.2 Turret

Head is a Transform that imitates a moving part of a turret.
Idle Rotation Speed is the rate at which the turret's head rotates around (or from side to side within specified range of degrees).
If **Clamp Idle Rotation** is true, the rotation of the turret while idling is clamped between **Clamped Idle Rot Start** and **Clamped Idle Rot End**.

1.7 Inventory



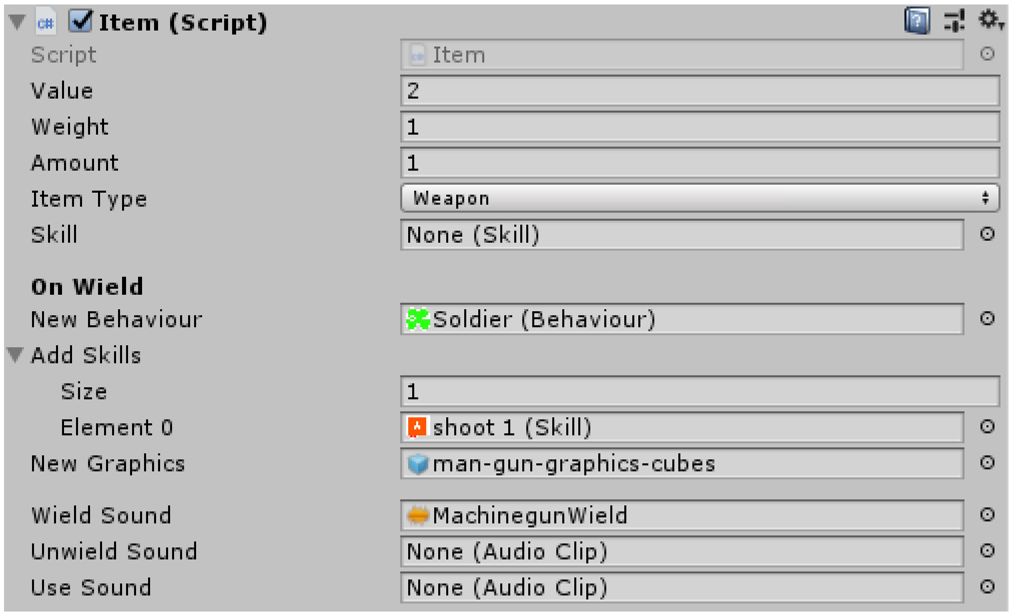
▼ Inventory	
Max Weight	0
▼ Items	
Size	0
Wielded Item	None (Item)
Init From List	☐
Init From Children	☐

Encapsulates behaviour related to interaction with Items (keep, use, wield, throw away, etc.). Items are kept as deactivated children of a parent container, whose default name is "__inventory__".

Max Weight is the maximum amount of weight Agent is able to carry in the Inventory.
Items is the list of Items that are currently present in Agent's Inventory.
Wielded Item is the Item that Agent is currently using as a tool. The field's purpose is to let user track the work of an Inventory, so modifying the value of the field by hand in runtime is not going to do anything. It should be changed using **Item.Wield(Agent)** or **Item.Unwield(Agent)** methods.

If **Init From List** is true, all the Items in the **Items** list will be configured properly at the start of the play mode. Configuration includes transforming, setting parent and visibility of each Item etc.
If **Init From Children** is true, all the Items that are children of Agent's items container, which goes by the name "__inventory__", are going to be properly configured at the start of the Play mode.

1.7.1 Item


Item (Script)

Script	Item
Value	2
Weight	1
Amount	1
Item Type	Weapon
Skill	None (Skill)
On Wield	
New Behaviour	Soldier (Behaviour)
Add Skills	
Size	1
Element 0	shoot 1 (Skill)
New Graphics	man-gun-graphics-cubes
Wield Sound	MachinegunWield
Unwield Sound	None (Audio Clip)
Use Sound	None (Audio Clip)

Represents an item that an Agent can carry in his Inventory. Items can hold Skills and some Agent's characteristics so that they can be changed when Agent uses the item. Skills let Items being used as potions, and other options let Items change Agent's Behaviour, his graphics components etc.

Value represents how valuable the item is. It can be used by Agent to easily understand which Item is better to pick up and use, or which one to throw out if the Inventory is overburdened.
Weight is the amount of space the Item takes to be situated inside an Inventory per one Item. When the actual **Weight** of Item is calculated, it is multiplied by **Amount**.
Amount is the number of times the Item occurs in the Inventory. Usage of the Item as a potion reduces the **Amount** by one.
Item Type is the value that helps Agents more easily decide whether the Item should be treated as a *Weapon* or as a *Potion*.
Skill is applied to an Agent upon using the Item as a potion.

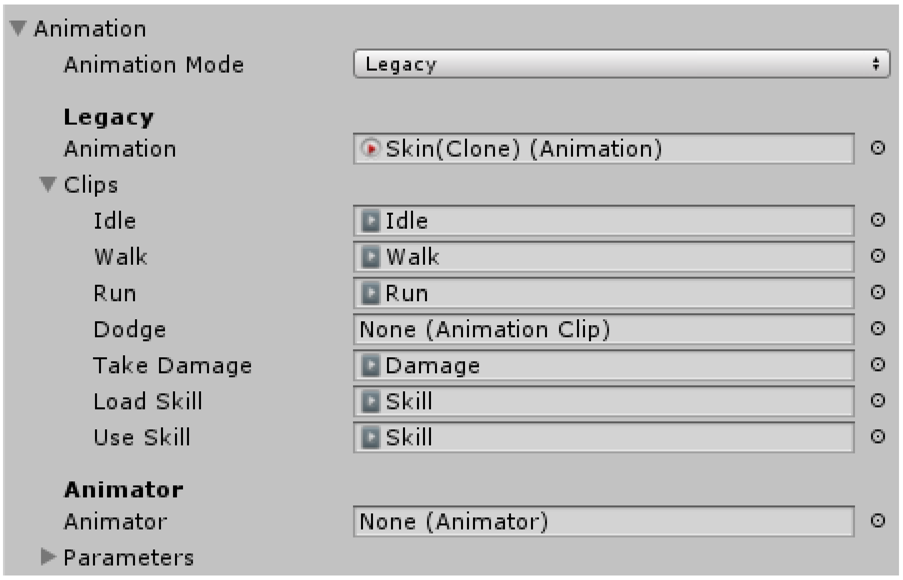
New Behaviour is a Behaviour model that is applied to Agent upon wielding the Item as a tool. This is useful, for example, when Agent, that used beat, picks up and wields a rifle. It is probably helpful to make him behave as if he is wielding a rifle now.
Add Skills is the list of Skills that an Agent is able to use once he wields the Item. Agent loses an access to these Skills upon unwielding the Item.
New Graphics is a prefab that is used to replace current Agent’s graphics upon wielding the Item. Agent’s graphics is reset to a default one (which is the one that is present at the start of the level) upon unwielding the Item. If **New Graphics** prefab has an AgentGraphics component, some settings of Agent are reconfigured upon wielding the Item. These settings are a link to Animation or Animator component in **Agent.Animation**, new positions of __shoot__ and __look__ Transforms (children), and a message can be sent to an Agent’s gameObject (using Unity’s SendMessage(message) method).

Wield Sound is an AudioClip that is played when Agent wields the Item.
Unwield Sound is an AudioClip that is played when Agent unwields the Item.
Use Sound is an AudioClip that is played when Agent uses the Item as a potion.

1.8 Agent Animation

The Animation component helps easily integrate Agent’s actions with the graphical output associated with them. Supports Legacy Animations and Mecanim. The animation engine is setup with **Animation Mode**.

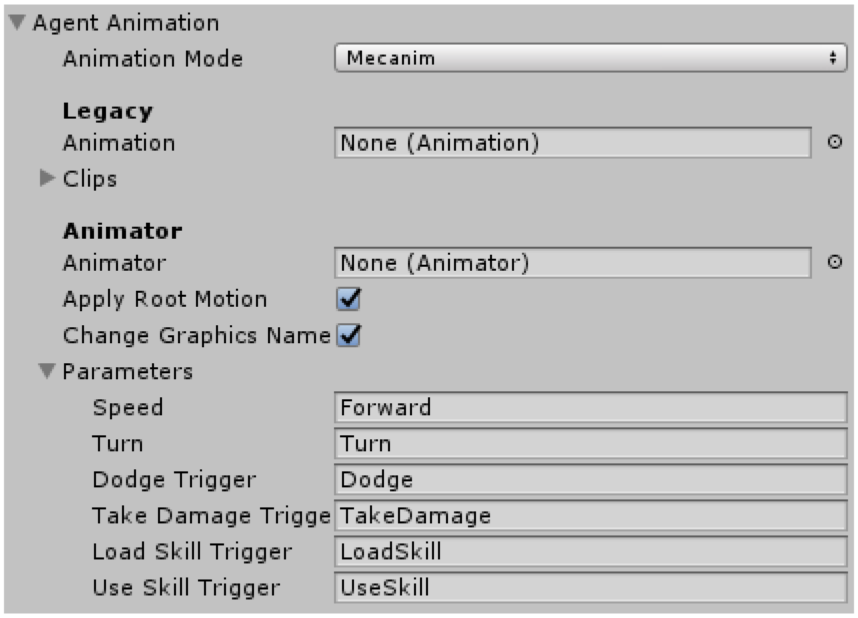
1.8.1 Legacy



Animation	
Animation Mode	Legacy
Legacy	
Animation	Skin(Clone) (Animation)
Clips	
Idle	Idle
Walk	Walk
Run	Run
Dodge	None (Animation Clip)
Take Damage	Damage
Load Skill	Skill
Use Skill	Skill
Animator	
Animator	None (Animator)
Parameters	

Animation is a link to a GameObject in the scene that has Unity’s Animation component ready to play animations on demand.
User needs to specify Animation Clips for related actions. The names of clips are self-descriptive, so I hope there’s no need to go into much detail here.

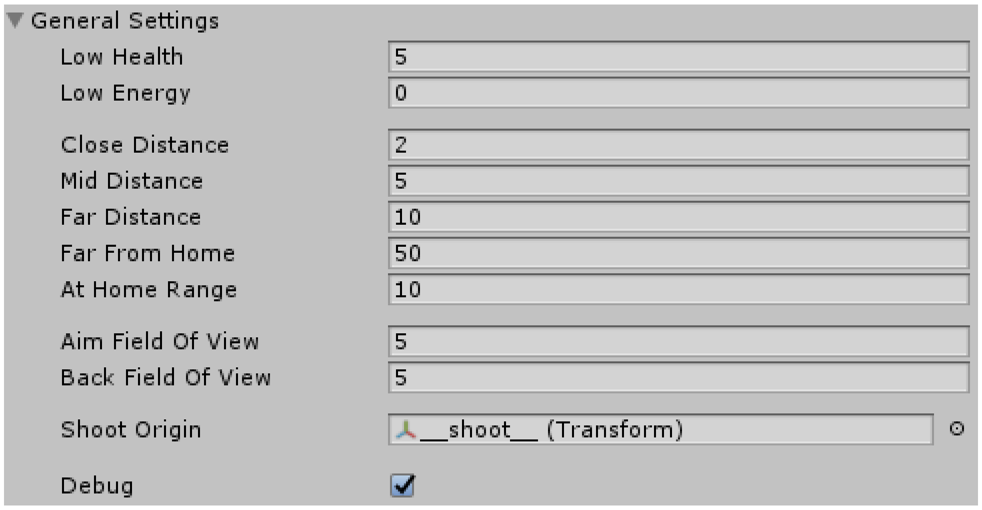
1.8.2 Mecanim



Agent Animation	
Animation Mode	Mecanim
Legacy	
Animation	None (Animation)
Clips	
Animator	
Animator	None (Animator)
Apply Root Motion	✓
Change Graphics Name	✓
Parameters	
Speed	Forward
Turn	Turn
Dodge Trigger	Dodge
Take Damage Trigge	TakeDamage
Load Skill Trigger	LoadSkill
Use Skill Trigger	UseSkill

Animator is a link to a GameObject in the scene that has Unity’s AnimatorController component, configured so that specific messages trigger related Animator states. Trigger messages can be used to animate dodging, taking damage, loading skill and using skill. Skills can have their own trigger messages. Speed is the name of Animator parameter that is used to fade between animations of idling, walking and running. Turn is the name of an animator parameter that is used to fade to animations of turning left and right.

1.9 General Settings



General Settings	
Low Health	5
Low Energy	0
Close Distance	2
Mid Distance	5
Far Distance	10
Far From Home	50
At Home Range	10
Aim Field Of View	5
Back Field Of View	5
Shoot Origin	__shoot__ (Transform)
Debug	✓

Encapsulates the settings that are related to the way Agent should treat current values of any of his parameters or, in fact, define new parameters that are not fit to any other Agent's specific component.
Values of this component can be used to make decisions during the game, for example, to run away from the enemy when the health level is ‘low’.

This component is meant to be added variables and/or methods to, to facilitate the needs of any particular project. For instance, if your game has animals, that are represented by Agents and one of the Agents needs to know what type of the animal is the other Agent, you can create new enum AnimalSpecies and create a new variable of this type inside the Settings component, so that you can assign the type of animal of Agent in Unity's Inspector. Then you can create methods in most appropriate interface (in this case it would be GeneralConditionInterface) to check for the type of an animal to use this value at runtime for decision making.

Low Health is the amount of health that is considered a 'low' amount for this particular Agent.
Low Energy is the amount of energy that is considered a 'low' amount for this particular Agent.

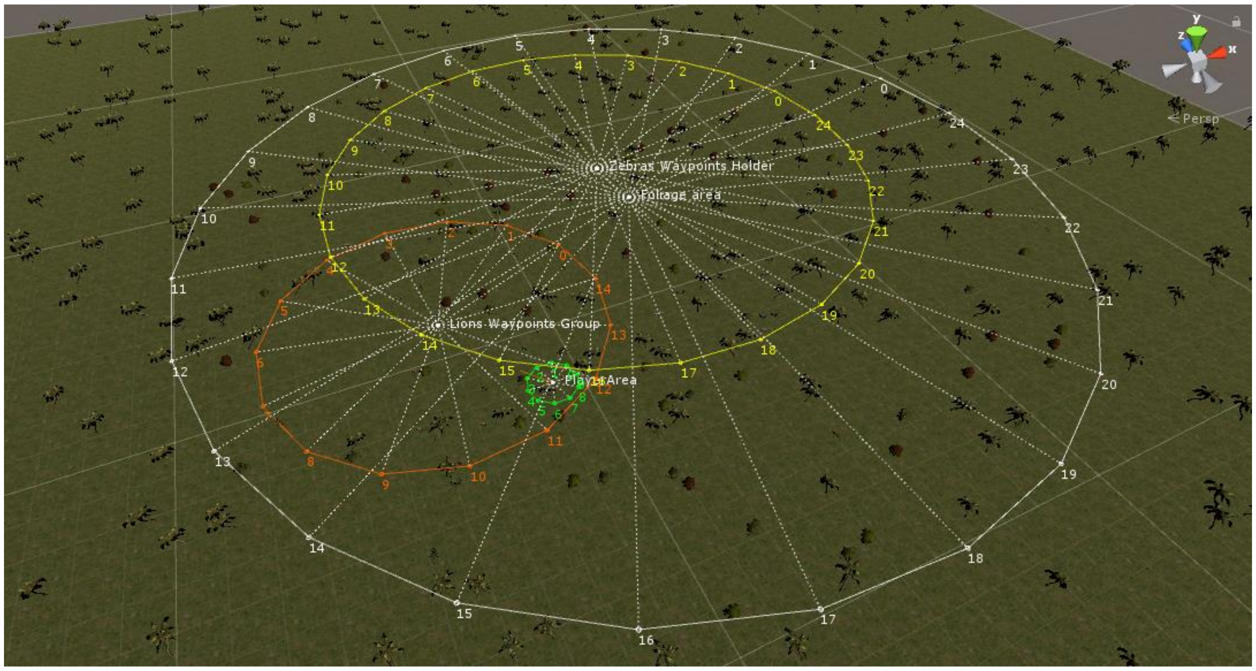
Close Distance is the distance that is considered a close distance to a target for this particular Agent. Most obvious use is as a melee attack distance. However, it is also possible to use Skill's **Range** value for this purpose.
Mid Distance is the distance that is considered a middle distance to a target for this particular Agent.
Far Distance is the distance that is considered a far distance to a target for this particular Agent. Most obvious use is as a range attack distance.
Far From Home is the distance to the origin of Agent's Waypoints Group that is considered a far distance. One of the obvious examples of use is for Agent to chase notices enemies only in certain radius from home area.
At Home Range is the radius of Agent's Waypoints Group's origin where Agent is considered to be at home area. If an Agent has no Waypoints Assigned, his initial position (where he is at the start of a level) is used as Waypoints Group's origin.

Aim Field Of View is the width of the Agent's view angle, measured in degrees along the local Y axis. If Agent's target is within this field of view, it is considered that Agent is facing the target and is ready to interact.
Back Field Of View is the width of the Agent's view angle, measured in degrees along the local Y axis, facing in the opposite direction from Agent's forward (z) direction. This field of view helps other Agents to understand if they are facing their target's back, for example, for stealth attacks.

Shoot Origin is a Transform that is used to determine the position and rotation at which projectiles are instantiated and rays are casted when the usage of a Skill demands so.

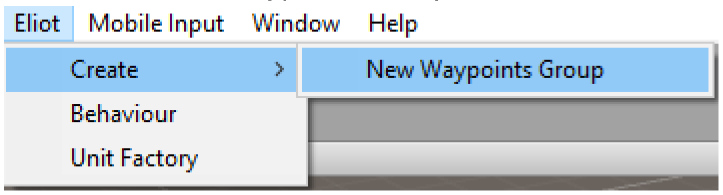
If **Debug** is true, handles and gizmos that represent Agent's Settings configuration are displayed in Unity Editor.

2 Waypoints Group

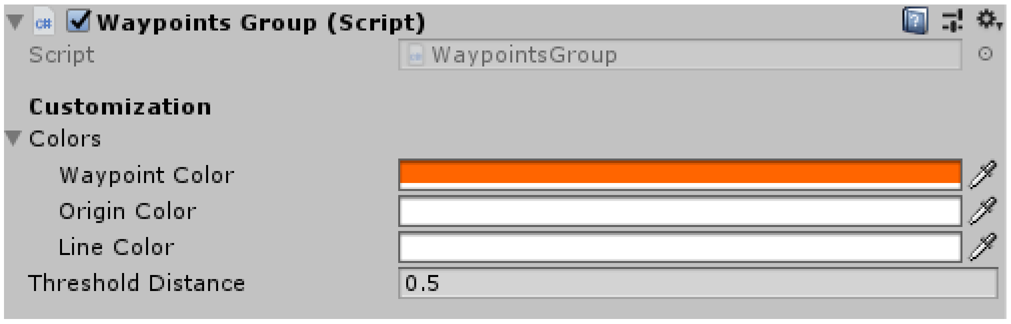


2.1 Description

A controller object for a set of Waypoints that can be used by Agents as an area or a path. To create a new Waypoints Group use *Eliot/Create/New Waypoints Group* menu item.



2.2 Customization



Waypoints Group (Script)

Script	WaypointsGroup
Customization	
Colors	
Waypoint Color	🟧
Origin Color	⬜
Line Color	⬜
Threshold Distance	0.5

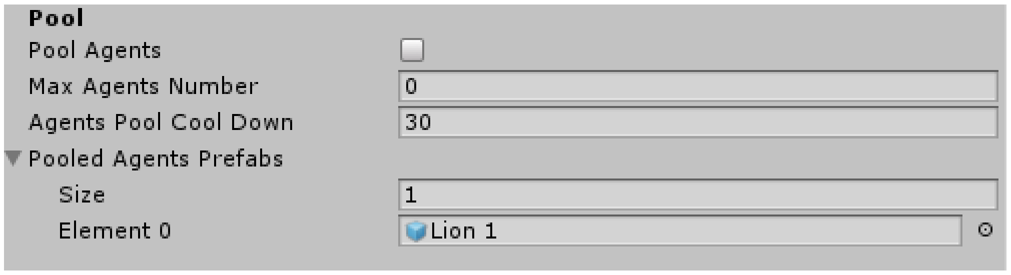
2.2.1 Colors

You can adjust the colors of the origin of a WaypointsGroup, color of Waypoints and color of lines between Waypoints. This feature can be useful to help visually markup different kinds of areas in the level, for example, red for dangerous zones, green for safe zones etc.

2.2.1 Threshold Distance

The value of Threshold Distance is used by Agents to know what radius of each Waypoint in a group is considered a Waypoint's area. For example, if an Agent goes from one Waypoint to another, one by one, the Threshold Distance of the Waypoints Group is a minimum distance between an Agent and a Waypoint that triggers switching target position of an Agent to the next Waypoint.

2.3 Agents Pool



Pool

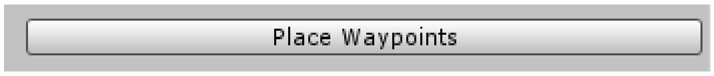
Pool Agents	☐
Max Agents Number	0
Agents Pool Cool Down	30
▼ Pooled Agents Prefabs	
Size	1
Element 0	Lion 1

WaypointsGroup has a capacity to spawn Agents (or whatever prefabs you specify) at runtime. User needs to specify a list of possible prefabs to be spawned (at each spawning operation one prefab from the list will be chosen), maximum number of agents for this particular WaypointsGroup (while current number of Agents equals to the maximum number, no further spawning will occur) and agent pool cooldown (amount of time that should pass after the last spawning operation to make it available again).
This functionality is useful for keeping the amount of creatures on certain area more or less stable throughout the time by respawning the killed ones.

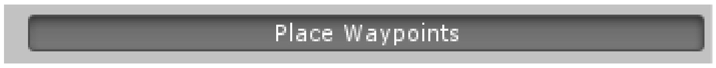
2.4 Project On Plane Button

By clicking this button, the currently selected WaypointsGroup will try to align its vertical position with the surface beneath it, or above (in case there is no surface right under the object). This is useful to save time for doing something more meaningful than trying to put the object right on top of some surface using Unity's gizmos or adjusting its 'y' coordinate.

2.5 Place Waypoints Mode



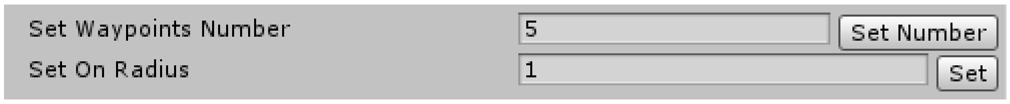
Place Waypoints Mode is Off



Place Waypoints Mode is On

Clicking the button “Place Waypoints” triggers the Place Waypoints Mode, which allows creating and placing new Waypoints on any surfaces in the editor by clicking left mouse button while cursor is over the desired surface. To switch the Place Waypoints Mode off, click the button “Place Waypoints” again.

2.6 Set Waypoints Number And Adjust Radius



Set Waypoints Number 5 Set Number

Set On Radius 1 Set

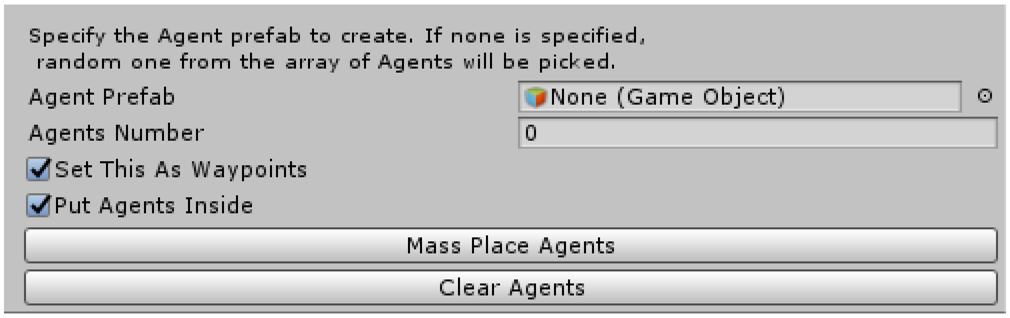
Pressing the **Set Number** button clears currently existing Waypoints in the group, creates specified by the field **Set Waypoints Number** amount of new Waypoints and places them evenly on circle that has the selected WaypointsGroup as a center and a radius specified by **Set On Radius** field.

Pressing the **Set** button places currently existing Waypoints in the group evenly on circle that has the selected WaypointsGroup as a center and a radius specified by **Set On Radius** field.

2.7 Clear Waypoints Button

The **Clear Waypoints** button removes all currently existing Waypoints in the group.

2.8 Mass Place Functionality



Specify the Agent prefab to create. If none is specified,
random one from the array of Agents will be picked.

Agent Prefab None (Game Object)

Agents Number 0

☑ Set This As Waypoints

☑ Put Agents Inside

Mass Place Agents

Clear Agents

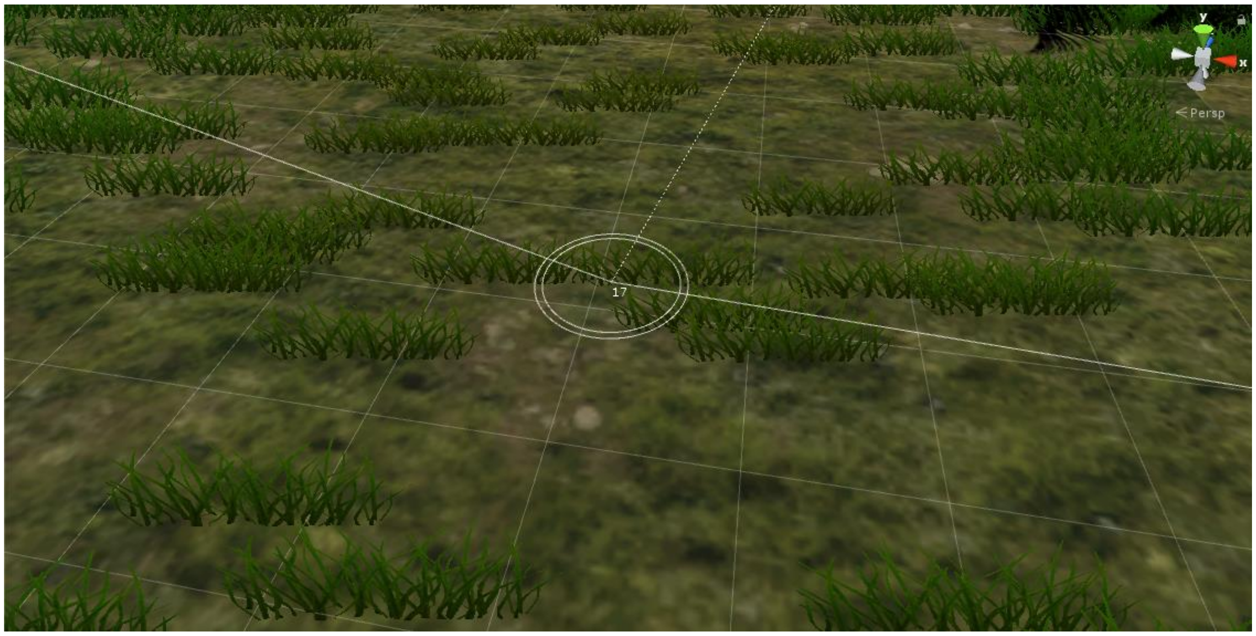
Pressing the **Mass Place Agents** button triggers the following sequence of actions:

1. Pick a random point inside the specified area (lines between Waypoints are the borders);
2. If **Agent Prefab** is specified, pick it as a prefab to spawn, otherwise pick a random one from the **Pooled Agents Prefabs** list;

3. Spawn picked prefab at picked position;
4. If **Set This As Waypoints** is checked, instantiated Agent’s **Waypoints** property is set to the selected WaypointsGroup;
5. If **Put Agents Inside** is checked, instantiated Agent becomes a child of a GameObject, named “__agents__”, which itself is a child of the selected WaypointsGroup;
6. Repeat the procedure **Agents Number** times.

Pressing the **Clear Agents** button deletes all children of the currently selected WaypointsGroup’s Agents container (which is its child that is named “__agents__”).

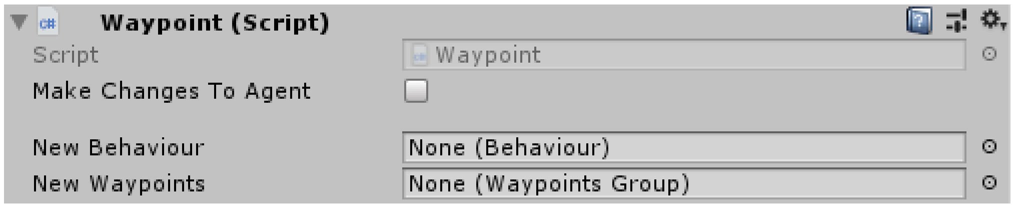
2.9 Waypoint



2.9.1 Description

One of the waypoints in the group. Can define one of the border points of the group or a target position for Agent. Can Interact with Agent changing his parameters if both Agent and Waypoint agree on that.

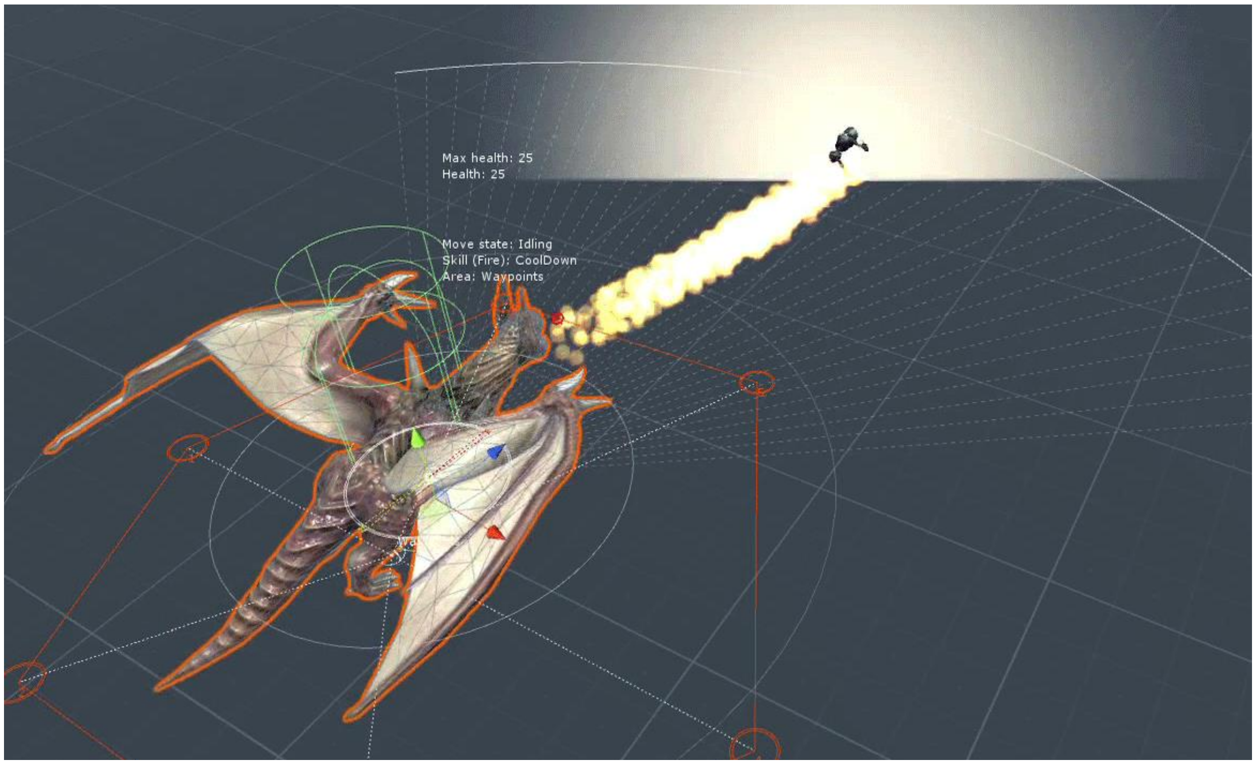
2.9.2 Make Changes To Agent


Waypoint (Script)

Script	Waypoint
Make Changes To Agent	☐
New Behaviour	None (Behaviour)
New Waypoints	None (Waypoints Group)

If **Make Changes To Agent** is true, the Waypoint will try to change Agent’s properties, setting his **Behaviour** and **Waypoints** to those, defined for the Waypoint. The operation of altering properties is triggered by Agent that has the Waypoint as a target and being on a distance from it that is equal or less than the **Threshold Distance** of the WaypointsGroup to which the Waypoint belongs. The change will occur only if Agent agrees to apply changes by setting his property **Apply Changes On Waypoints** to true.

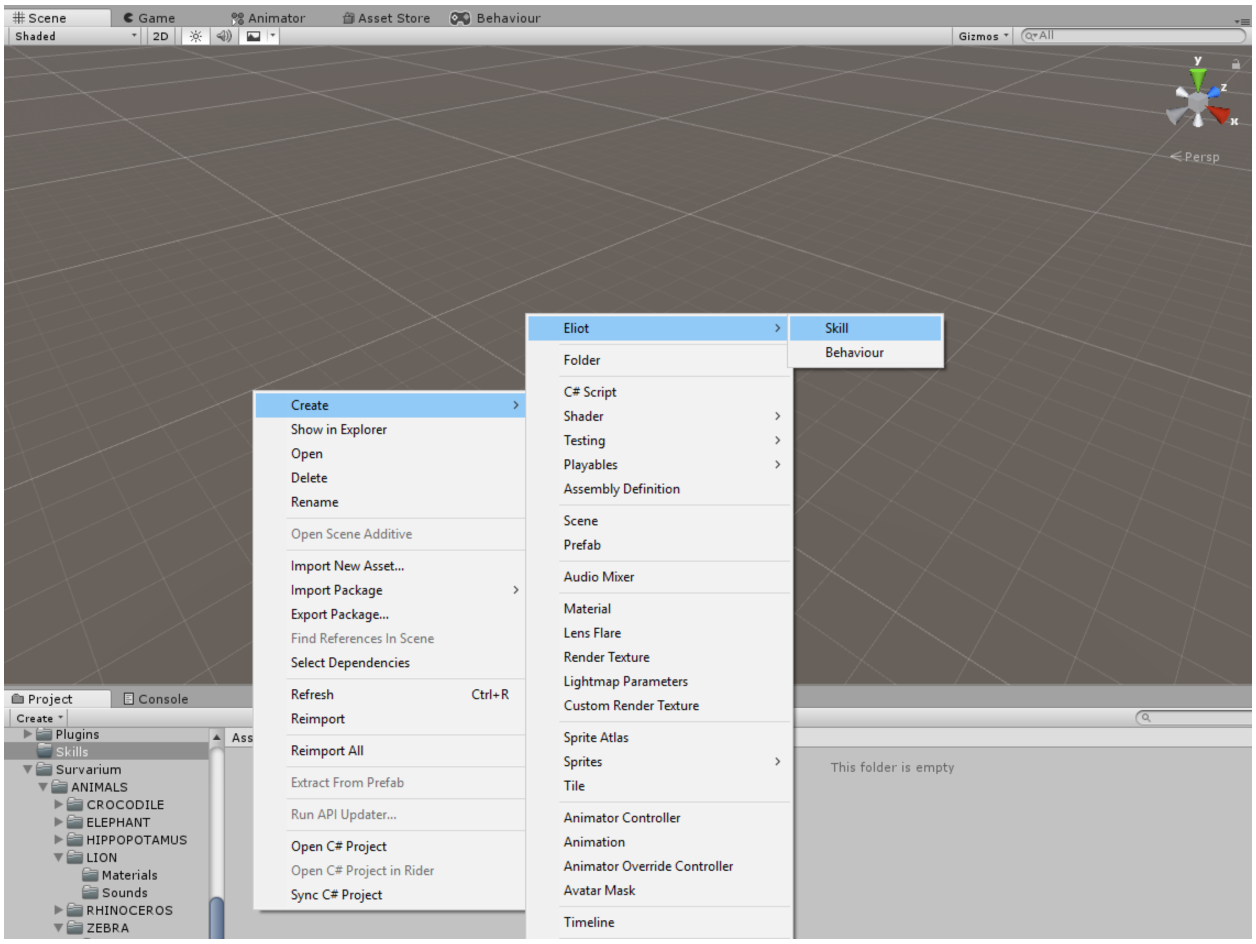
3 Skills



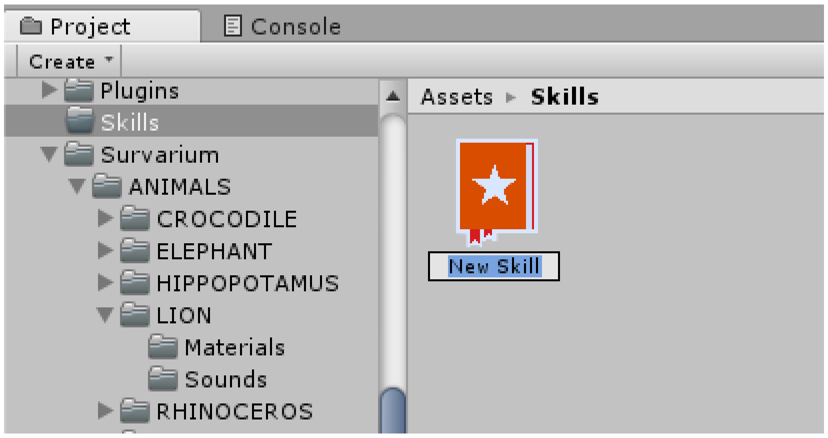
3.1 Description

Skills allow agents to interact with other objects in almost any imaginable way, as they even allow agents to directly invoke methods of arbitrary classes on any instance of any object. It means that the ways for agents of interaction with the world are limited only by the abilities of C#. However, most kinds of interactions like melee and ranged attacks, healing, sending messages to surrounding Agents, buffs, DoTs - all can be configured without even thinking about code. Each skill is saved as a separate file.

3.2 Creation of new Skills



You can create a new Skills from the Create menu at the top left of the Project panel or by selecting **Assets > Eliot > Skill** from the main menu.

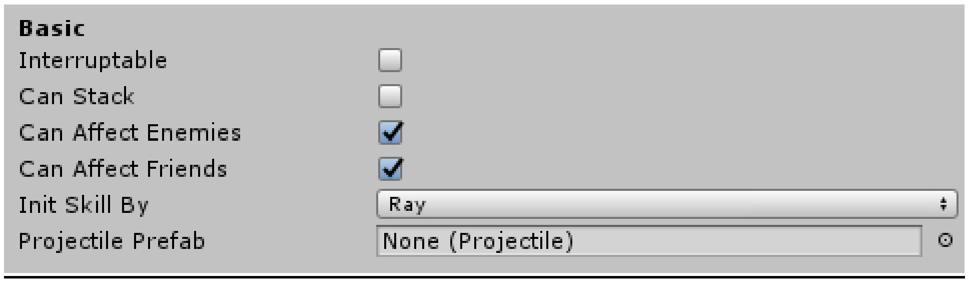


The new Skill will be created in whichever folder you have selected in the Project panel. The new Skill file’s name will be selected, prompting you to enter a new name.

3.3 Settings and mechanism of action

When a Skill is applied to Agent, it starts affecting his parameters accordingly to the Skill's settings. It can be both a momentary influence as well as something that affects Agent over time (in this case the Skill is displayed in Agent's Active Effects list).

3.3.1 Basic

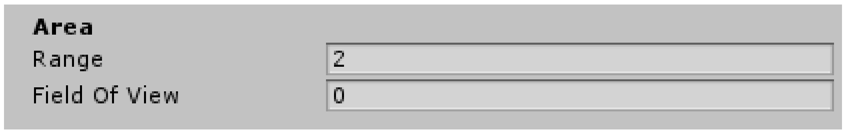


Basic

Interruptable	☐
Can Stack	☐
Can Affect Enemies	✓
Can Affect Friends	✓
Init Skill By	Ray
Projectile Prefab	None (Projectile)

Interruptible defines whether it is possible to prevent the Skill from being invoked once it started loading.
Can Stack defines whether the Skill can be applied to an Agent who already has this Skill applied.
Can Affect Enemies defines whether the Skill can be applied to enemy units of an Agent who casts the Skill.
Can Affect Friends defines whether the Skill can be applied to friendly units of an Agent who casts the Skill.
Init Skill By property defines the way Skills choose targets and get invoked.
Ray option makes Agent find a target within specified **Range** and **Field Of View** and cast a ray towards it. The Skill is applied if the ray succeeds in touching the target.
Projectile option instantiates a Projectile Prefab at the __shoot__ position (which can be configured in Agent's **General Settings**). Projectile then moves accordingly to its configuration.
Self option applies the Skill to its caster.
Direct option makes an Agent find a target within specified **Range** and **Field Of View** and the Skill gets applied to that target.
Radius option applies the Skill to all Agents in specified **Range**. Targets can be filtered by setting **Can Affect Enemies** and **Can Affect Friends** to preferred value.

3.3.2 Area

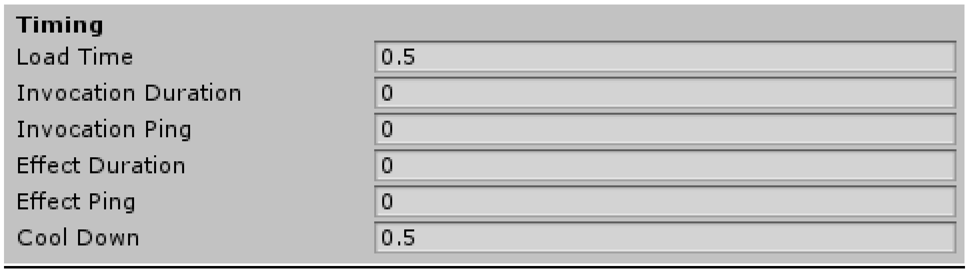


Area

Range	2
Field Of View	0

Range defines maximum distance at which the Skill can be applied to a target. It can be used in the Behaviour to check if a target is close enough for Skill to take effect.
Field of view is the width of the Agent's view angle, measured in degrees along the local Y axis, within which the Skill can take effect. If **Field Of View** equals to 0, the value of **Aim FOV** from Agent's **General Settings** is used instead.

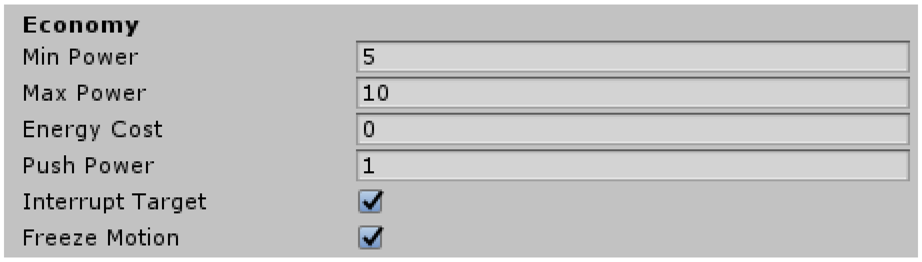
3.3.3 Timing



Timing	
Load Time	0.5
Invocation Duration	0
Invocation Ping	0
Effect Duration	0
Effect Ping	0
Cool Down	0.5

Load Time is the amount of time that should pass from the beginning of attempt to use the Skill to its actual invocation. The **Loading Animation** is played while the skill is being loaded.
Invocation Duration is the amount of time the Skill will be casted by an Agent. Set this to 0 if it should be invoked once, like a punch, or a shot. If it's more than 0, the Skill will be casted for the specified duration, being invoked every **Invocation Ping** seconds. The **Executing Animation** is played when the Skill is invoked or while it is being casted.
Effect Duration is the amount of time the Skill will be affecting the target Agent, being invoked every **Effect Ping** seconds. Set **Effect Duration** to more than 0 if the Skill should be a buff or a DoT.
Cool Down is the amount of time an Agent should wait before being able to use the Skill again.

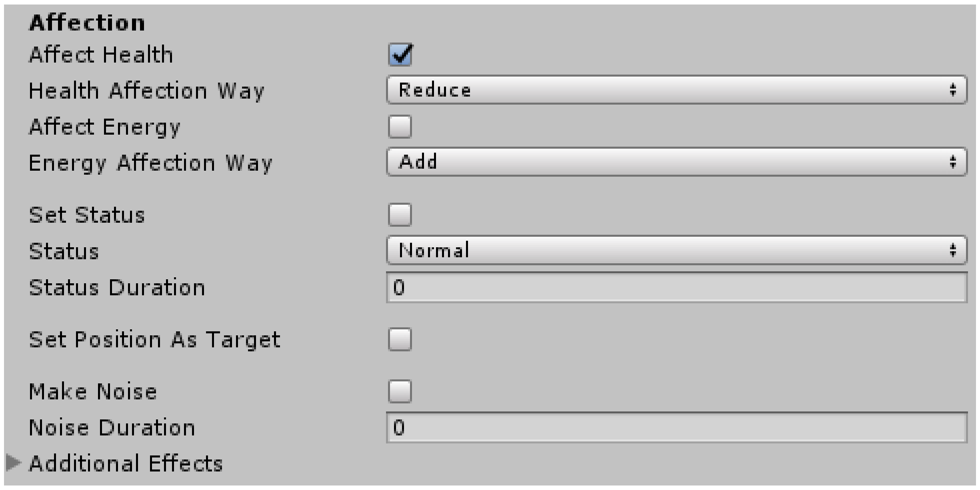
3.3.4 Economy



Economy	
Min Power	5
Max Power	10
Energy Cost	0
Push Power	1
Interrupt Target	✓
Freeze Motion	✓

A random value between **Min Power** and **Max Power** is added or subtracted from specified Agent's economy values (health and energy) when the Skill gets invoked.
The Skill costs **Energy Cost** points of energy to invoke, so if an Agent has less of it than the specified amount, he won't be able to use the Skill.
Push Power defines how strongly the target will be pushed in the direction away from the caster.
If **Interrupt Target** is true, the target of the Skill will be frozen in movement for **Agent.Resources.Lock Time** seconds and prevented from finishing loading a Skill if such event is happening when the Skill gets invoked.
If **Freeze Motion** is true, the caster of the Skill will be frozen in movement for **Agent.Resources.Lock Time** seconds.

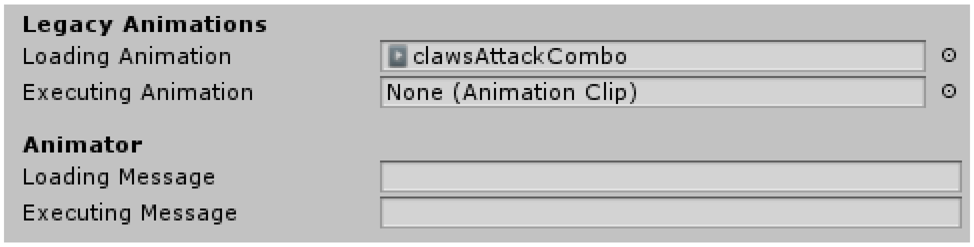
3.3.5 Affection



Affection	
Affect Health	✓
Health Affection Way	Reduce
Affect Energy	☐
Energy Affection Way	Add
Set Status	☐
Status	Normal
Status Duration	0
Set Position As Target	☐
Make Noise	☐
Noise Duration	0
▶ Additional Effects	

If **Affect Health** is true, target's health will be affected upon invoking the Skill. Will the target be healed or damaged, depends on the **Health Affection Way** value.
If **Affect Energy** is true, target's energy will be affected upon invoking the Skill. Will the target's energy be added or reduced, depends on the **Energy Affection Way** value.
If **Set Status** is true, target's **Agent.Status** will be set to **Status** for **Status Duration** seconds upon invoking the Skill.
If **Set Position As Target** is true, the target Agent's target position will be set to the position of invocation of the Skill until the target Agent gets distracted.
If **Make Noise** is true, target's **Agent.Perception.SuspiciousPosition** will be set to the position of invocation of the Skill for **Noise Duration** seconds upon invoking the Skill.
Additional Effects is a list of Skills that will also be applied to the target upon invoking the Skill.

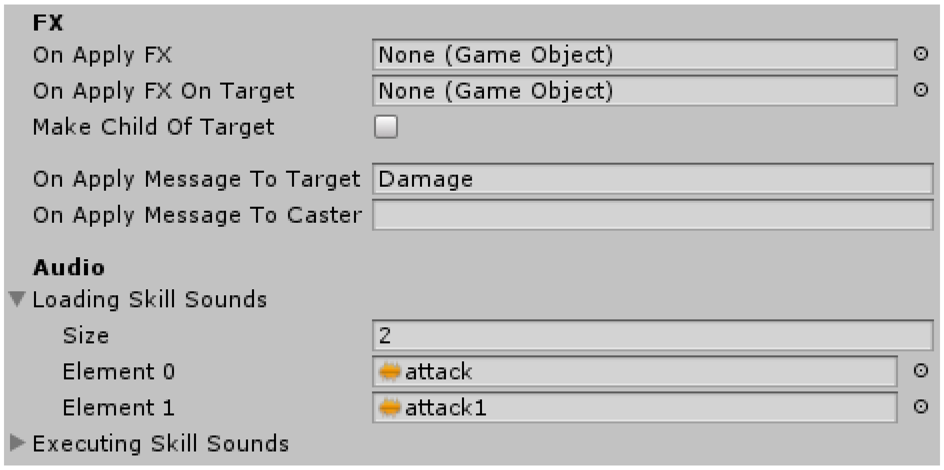
3.3.6 Animations



Legacy Animations	
Loading Animation	clawsAttackCombo
Executing Animation	None (Animation Clip)
Animator	
Loading Message	
Executing Message	

If **Agent.Animation.Animation Mode** is set to Legacy, the **Loading Animation** is played while the skill is being loaded and the **Executing Animation** is played when the Skill is invoked or while it is being casted.
If **Agent.Animation.Animation Mode** is set to Mecanim, the **Loading Message** is sent as an Animator trigger message to **Agent.Animation.Animator** while the skill is being loaded and the **Executing Message** is sent as an Animator trigger message when the Skill is invoked or while it is being casted.

3.3.7 FX and Audio



FX

On Apply FX	None (Game Object)
On Apply FX On Target	None (Game Object)
Make Child Of Target	☐
On Apply Message To Target	Damage
On Apply Message To Caster	

Audio

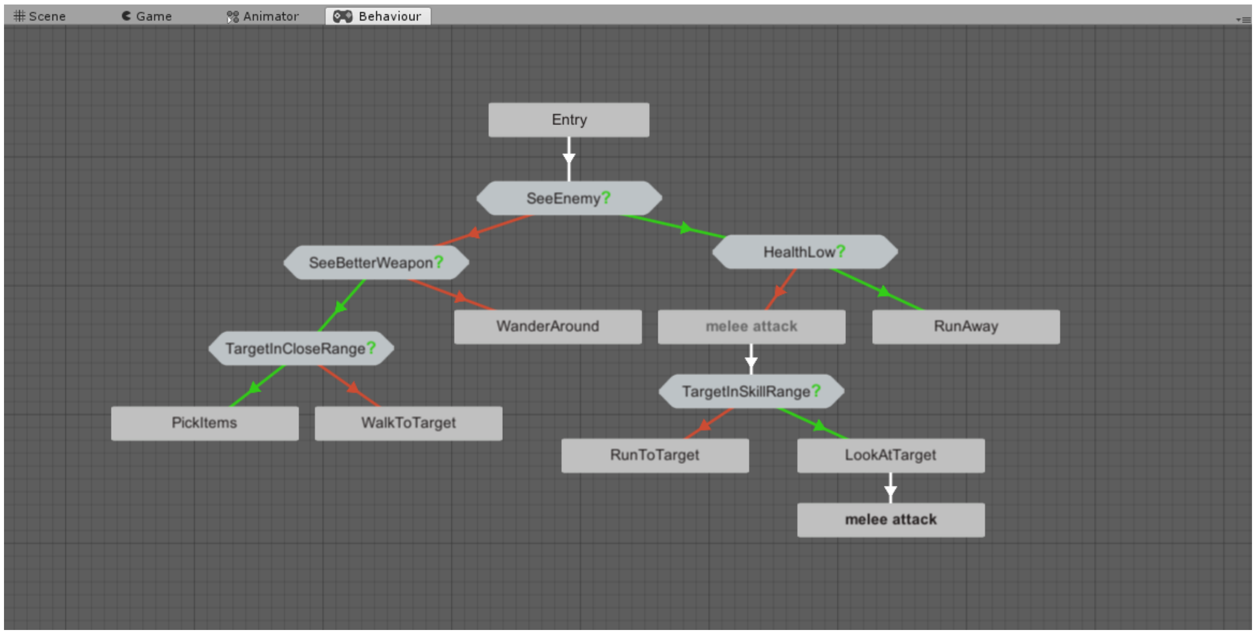
▼ Loading Skill Sounds

Size	2
Element 0	attack
Element 1	attack1

▶ Executing Skill Sounds

When the Skill is invoked, **On Apply FX** gets instantiated at the position of a caster and **On Apply FX On Target** gets instantiated at the position of the target of the Skill. If **Make Child Of Target** is true, the instantiated prefab's transform.parent is set to the caster of the Skill (**On Apply Effect FX**) or to its target (**On Apply FX On Target**).
If **On Apply Message To Target** is not empty, the string is used to send a message to the target of the Skill using Unity's SendMessage(message) method.
If **On Apply Message To Caster** is not empty, the string is used to send a message to the caster of the Skill using Unity's SendMessage(message) method.
The **Loading Skill Sounds** is a collection of AudioClips, a random one of which is played while the skill is being loaded and a random AudioClip of the **Executing Skill Sounds** collection is played when the Skill is invoked or while it is being casted.

4 Behaviour

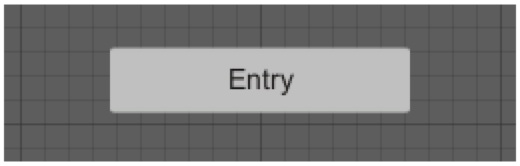


4.1 Description

Behaviour is a visually represented algorithm of Agent's realtime decision making. It consists of components that can execute specified Agent's functionality to, for example, check something about the surroundings using Agent's perception, and to behave accordingly to the result of the check.

4.2 Behaviour Components

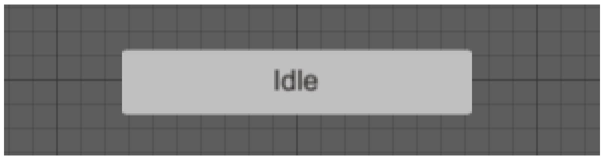
4.2.1 Entry



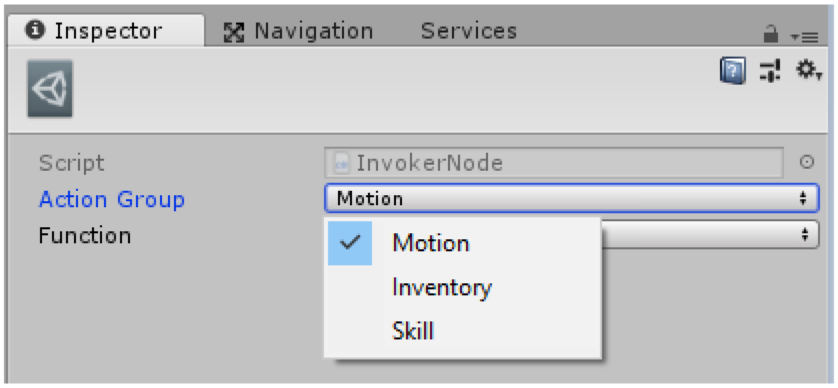
Entry is the beginning of an algorithm. There can be only one Entry in a model. This component does not do anything on itself, except from activating connected elements. The model gets updated every frame (or once every X seconds, if Agent's Ping is equal to X), starting from the entry point of a program (which is usually the Entry, but it also can temporarily be a Loop) and propagating itself depending on what elements get activated each update.

Entry does not need to be explicitly created, and it cannot be. There is always a single Entry in each Behaviour.

4.2.2 Invoker

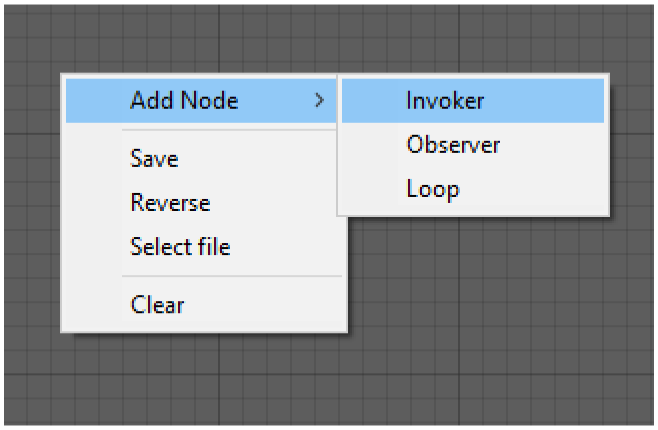


Invokers hold the information about Agent's functionality that should be executed. There can be any number of Invokers in the model. Agent's functionality that can be executed with Invoker is the same kind of functionality that a user would invoke using an input, such as gamepad. To be more precise, the functionality of Invoker is divided into three groups: Motion, Inventory and Skill.

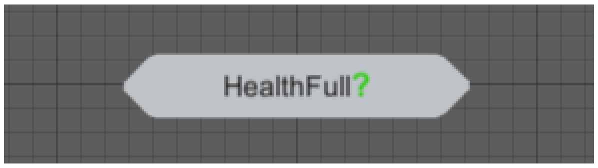


Motion methods let Agent move around in the world, Inventory lets agent pick up, use, wield and throw away items, and Skill executes an Agent's Skill if it happens to be in Agent's Skills list or just sets it as a CurrentSkill without execution for retrieving information, like is there enough energy for using the Skill, or is the target in the range of Skill (this depends on **Execute Skill** value in an InvokerNode's Inspector).
After an Invoker executes an assigned method from available Agent's API, it immediately propagates the signal further, activating connected elements.

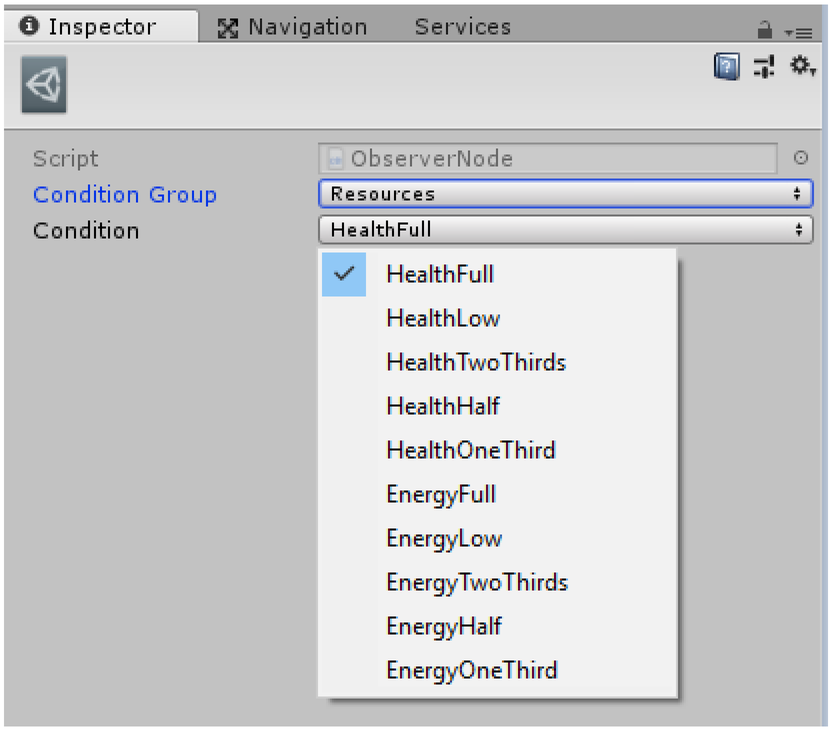
To create a new Invoker, either choose *Add Node/Invoker* from the Behaviour Editor Window context menu, or use a shortcut "**I**" while working inside the window.



4.2.3 Observer



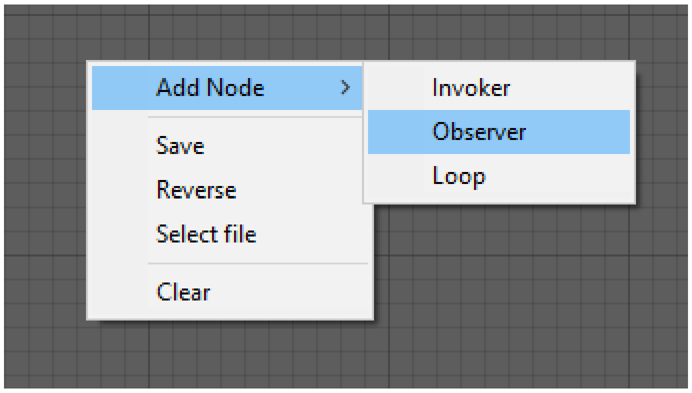
Observers check certain conditions using Agent's API. An Observer's check returns either true or false, and, depending on the result, activates elements that are connected with either green Transitions (if the result of the condition check is true) or red ones (if the result of the condition check is false). There can be any number of Observers in the model. To see what functionality is currently available to Observers, just select one and look at Inspector.



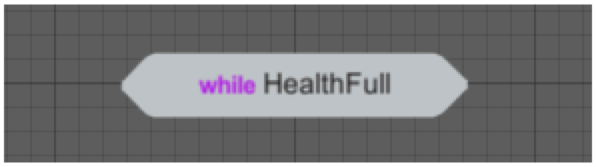
Script: ObserverNode
Condition Group: Resources
Condition: HealthFull

- ✓ HealthFull
- HealthLow
- HealthTwoThirds
- HealthHalf
- HealthOneThird
- EnergyFull
- EnergyLow
- EnergyTwoThirds
- EnergyHalf
- EnergyOneThird

To create a new Observer, either choose *Add Node/Observer* from the Behaviour Editor Window context menu, or use a shortcut “**O**” while working inside the window.



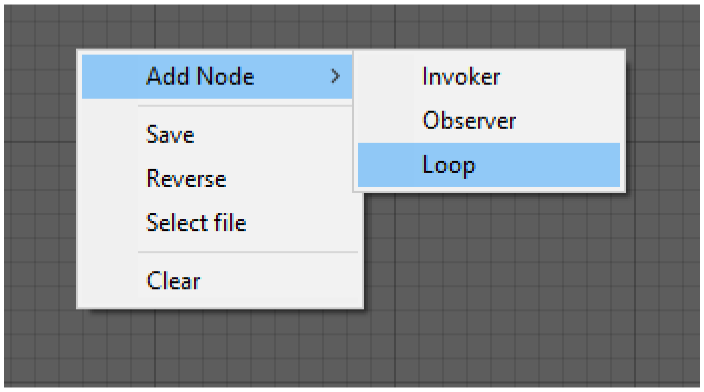
4.2.4 Loop



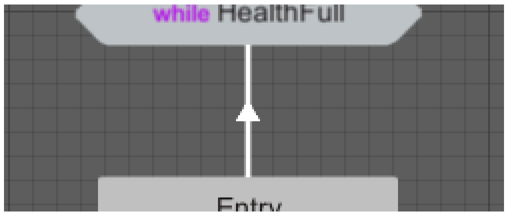
Loops work similarly to Observers, but while the condition that is being checked by a Loop is true, the Loop takes a role of an entry point of an algorithm, preventing all the elements that go before this one from being activated. When the condition is false, the role of an entry point of an algorithm turns back to Entry. Loops have purple Transitions instead of green ones, and white Transitions instead of red ones. There can be any number of Loops in the model.

Loops have access to the same functionality that Observers do.

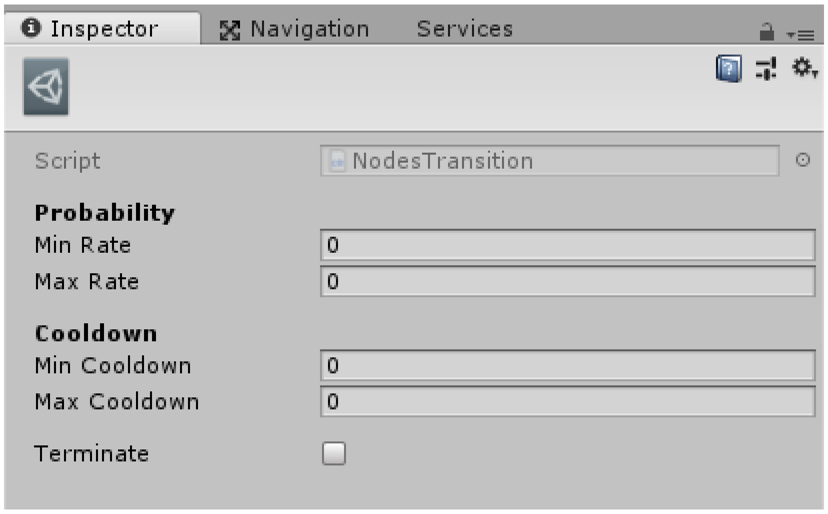
To create a new Loop, either choose *Add Node/Loop* from the Behaviour Editor Window context menu, or use a shortcut "**L**" while working inside the window.



4.2.5 Transition



Transitions connect other elements in the model, letting them propagate the signal while the algorithm is being executed.



Script: NodesTransition

Probability
Min Rate: 0
Max Rate: 0

Cooldown
Min Cooldown: 0
Max Cooldown: 0

Terminate: ☐

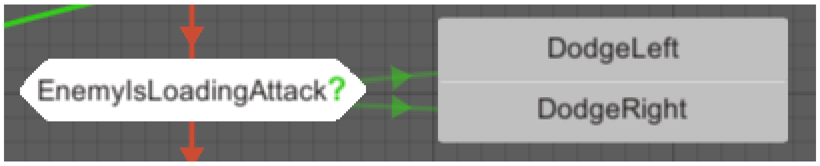
Transitions have some adjustable parameters, which can add stochastic properties to the algorithm. This is useful if, for example, the side to which an Agent should dodge after he detected that he is going to be attacked. Stochastic properties can let him dodge left or right randomly.

Min Rate is the minimum times the Transition will skip the Update before activating its target Component.

Max Rate is the maximum times the Transition will skip the Update before activating its target Component.

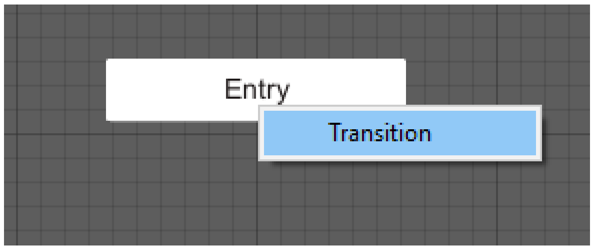
Min Cooldown is a minimum duration of time the Transition should skip the Update.
Max Cooldown is a maximum duration of time the Transition should skip the Update.

If **Terminate** is true, activation of this element prevents all other elements from the group from being activated in this Update.

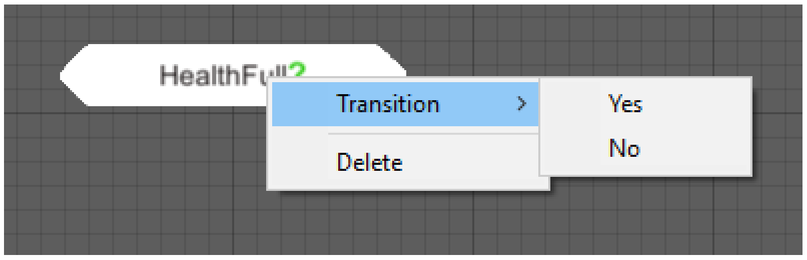


If the truthiness of a condition EnemyIsLoadingAttack gives some chance for several Invokers (e.g. DodgeLeft and DodgeRight) to be activated, like in the example above, setting the green Transitions' **Terminate** property to *true* would prevent the other one from being activated if one of them already was.

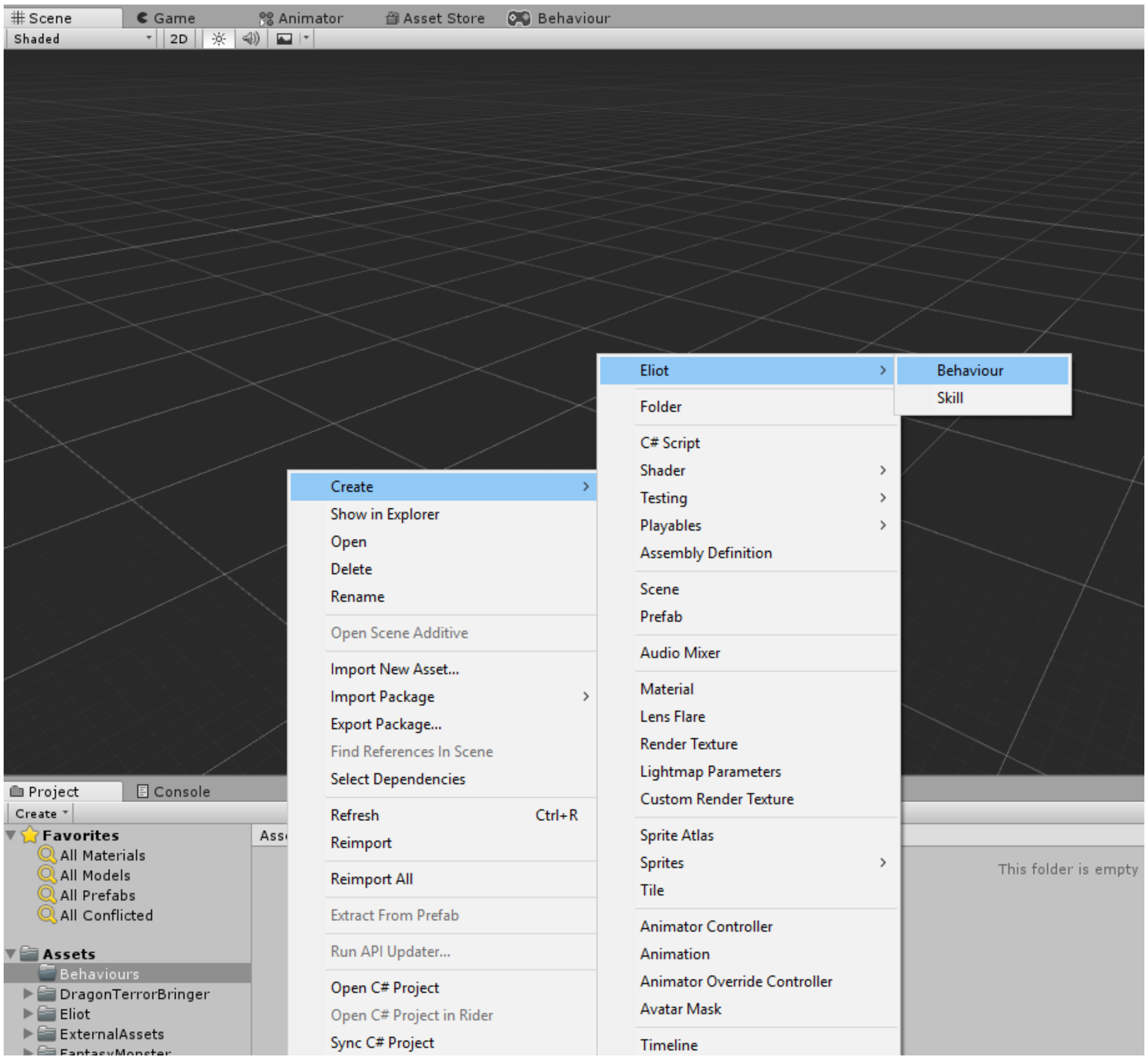
To create a new Transition that goes from Entry or Invoker, either choose *Transition* from the context menu of the element, for which you called the context menu (by clicking right mouse button while hovering over the element), or use a shortcut "**Y**" while hovering over the element.



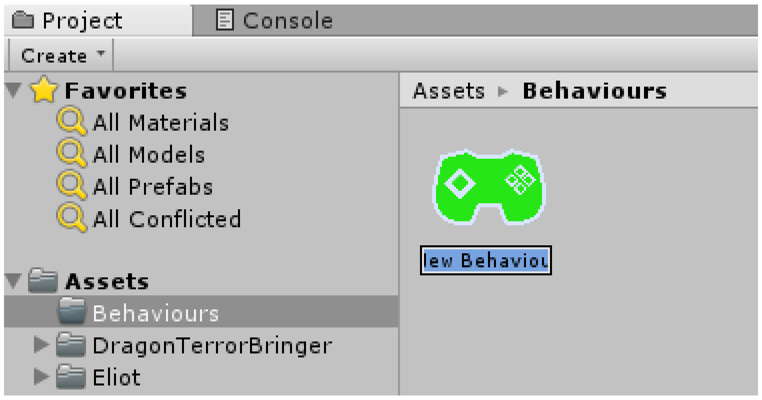
Transitions that go from Observers and Loops can be "positive" (that get activated if a checked condition is true. Green for Observers and purple for Loops) or "negative" (that get activated if a checked condition is false. Red for Observers and white for Loops).
To create a new Transition that goes from Observer or Loop, either choose *Transition/Yes* (*Transition/While* for Loops) or *Transition/No* (*Transition/End* for Loops) from the context menu of the element, for which you called the context menu (by clicking right mouse button while hovering over the element), or use a shortcut "**Y**" (for positive Transitions) or "**N**" (for negative ones) while hovering over the element.



4.3 Creation of new Behaviours



You can create a new Behaviour from the Create menu at the top left of the Project panel or by selecting **Assets > Eliot > Behaviour** from the main menu.

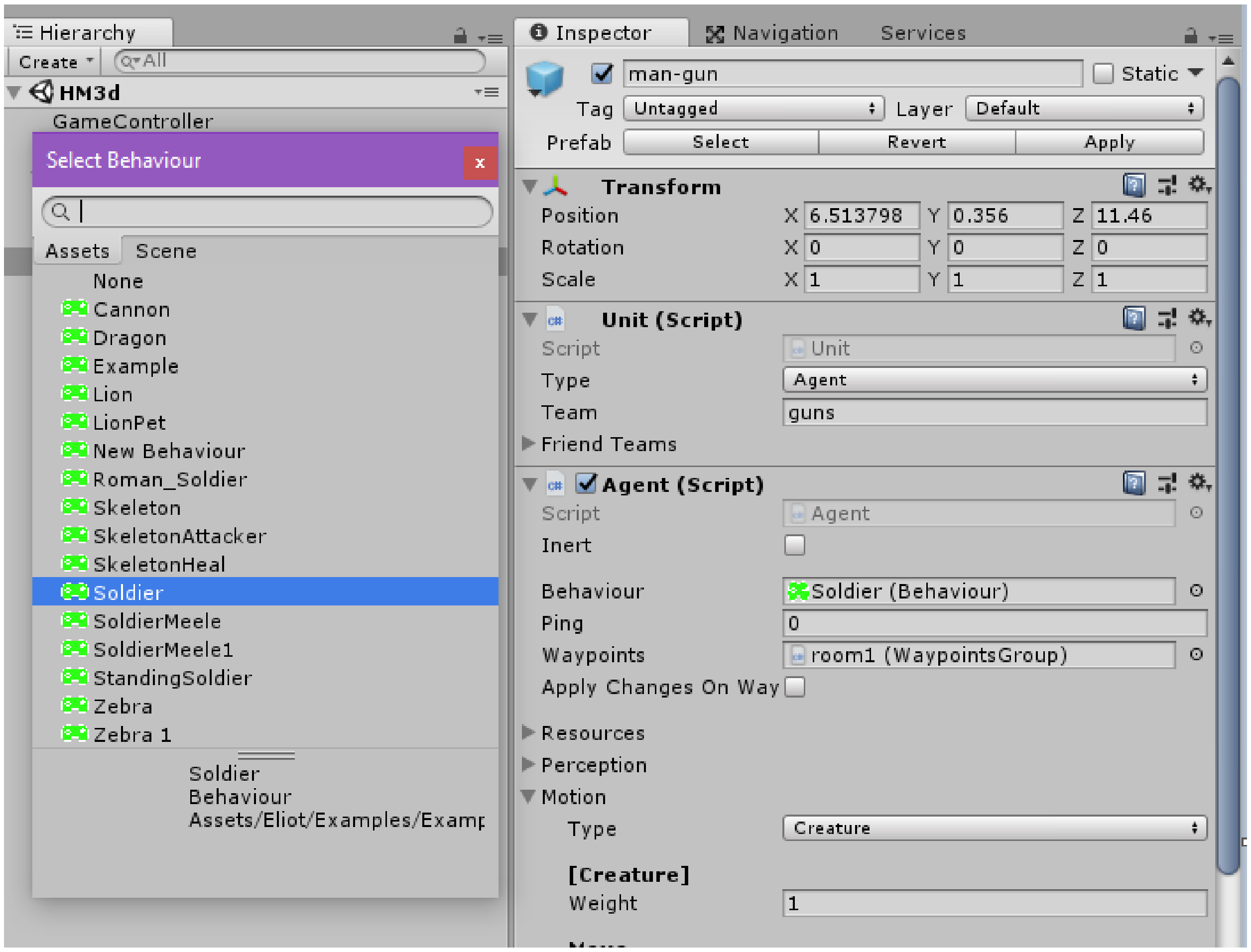


The new Behaviour will be created in whichever folder you have selected in the Project panel. The new Behaviour file's name will be selected, prompting you to enter a new name.

To edit Behaviour in Behaviour Editor Window simply drag and drop the file anywhere inside the open window.
Remember that you can have as many Behaviours as you want. And don't forget to save your changes by choosing *Save* from window's context menu or by using a shortcut "**ctrl+s**"!

4.4 Assign Behaviour to an Agent

To assign a Behaviour to an Agent you can either choose it from the list of created ones in Agent's Inspector, or drag and drop it into the corresponding field.



Select Behaviour

Assets | Scene

None
Cannon
Dragon
Example
Lion
LionPet
New Behaviour
Roman_Soldier
Skeleton
SkeletonAttacker
SkeletonHeal
Soldier
SoldierMeele
SoldierMeele1
StandingSoldier
Zebra
Zebra 1

Soldier
Behaviour
Assets/Eliot/Examples/Examp

man-gun ☐ Static
Tag: Untagged Layer: Default
Prefab: Select | Revert | Apply

Transform

	X	Y	Z
Position	6.513798	0.356	11.46
Rotation	0	0	0
Scale	1	1	1

Unit (Script)

Script	Unit
Type	Agent
Team	guns
Friend Teams	

Agent (Script)

Script	Agent
Inert	☐
Behaviour	Soldier (Behaviour)
Ping	0
Waypoints	room1 (WaypointsGroup)
Apply Changes On Way	☐

Resources
Perception
Motion

Type	Creature

[Creature]

Weight	1

5 Interfaces

5.1 Description

Eliot’s Interfaces are a bridge between an Agent and the Behaviour engine. They are the way these two systems communicate. Interfaces use Agent’s API to execute methods, like walking, or using skills and to check Agent’s stats (like Health level) and Agent’s representation of the world (which he gets via Perception).
Behaviour Editor also uses Eliot’s Interfaces to group Agent functionality and to let user choose from available public methods, which are marked with **IncludeInBehaviour** attribute from these Interfaces.
You are free to add new methods to Interfaces, or to remove ones that you don’t need.

5.2 Condition Interfaces

Condition Interfaces are used to check specific conditions, related to Agent’s internal economy or to his representation of the world.
For system to treat methods from these interfaces as usable for Behaviour building and execution, methods should be marked with **IncludeInBehaviour** attribute, should be public, nonstatic and return a boolean value. For example:

```
[IncludeInBehaviour] public bool ExampleMethodName()
{
  return ...;
}
```

You can find these interfaces in the directory *Eliot/AgentComponents/ConditionInterfaces*.

5.2.1 Resources Condition Interface

Resources Condition Interface encapsulates methods for checking conditions related to Agent’s Resources component, e.g. health level, energy level etc.

5.2.2 Perception Condition Interface

Perception Condition Interface encapsulates methods for checking conditions related to Agent’s Perception component. This mostly relates to scanning observable by agent Units, or checking whether Agent can see specific Units at the moment, like checking if Agent can see a Unit, whose Team is different from the Agent’s (a.k.a. enemy).
When adding new methods to this Interface, you’ll probably need to check for some custom conditions on Units that Agent is able to see. You can do this easily with a **SeeUnit(UnitQuery)** method on Agent’s Perception component. UnitQuery is an expression that all the perceived (and remembered) objects are compared against. This method returns true if one of Agent’s perception rays touches a Unit that matches an expression and <u>sets that matched Unit as Agent’s target</u>.
Let’s say, we want to check if Agent can see (or has recently seen) a Unit, whose **transform.localScale** is more than 1 (You probably won’t need to check for such an irrelevant

feature, but I do it to show that you can check whatever you want with just a single expression). Then this is the way to do this:

```
[IncludeInBehaviour] public bool SeeTargetWithLargeScale()
{
  return _perception.SeeUnit(unit => unit.transform.localScale > 1f);
}
```

This method will be automatically added to a Perception Group of conditions and will be instantly accessible in Behaviour Editor.

5.2.3 Motion Condition Interface

Motion Condition Interface encapsulates methods for checking conditions related to Agent's Motion component, like current Motion state of the Agent.

5.2.4 Inventory Condition Interface

Inventory Condition Interface encapsulates methods for checking conditions related to Agent's Inventory component, like looking for a particular item in an Inventory or comparing currently wielded Item with others that are available.

5.2.5 General Condition Interface

General Condition Interface encapsulates methods for checking conditions related to Agent's Settings component, which means checking for any stuff that is not related to any of previously discussed ones. This can be checking for Agent's status (which is useful since status can be affected by Skills, which means that it is a convenient way for Agents to communicate) or checking for a value of a User-created variable that is project-specific.

5.3 Action Interfaces

Action Interfaces are used to execute Agent's functionality using his API. It's kind of like a gamepad input. There are two Interfaces (one for Motion and another for Inventory), but there are three groups of actions that are available through Behaviour Engine (Motion, Inventory and Skill).
For system to treat methods from these interfaces as usable for Behaviour building and execution, methods should be marked with **IncludeInBehaviour** attribute, should be public, nonstatic and return type of void would make most sense for them. For example:

```
[IncludeInBehaviour] public void ExampleMethodName()
{
   ...
}
```

You can find these interfaces in the directory *Eliot/AgentComponents/ActionInterfaces*.

5.3.1 Motion Action Interface

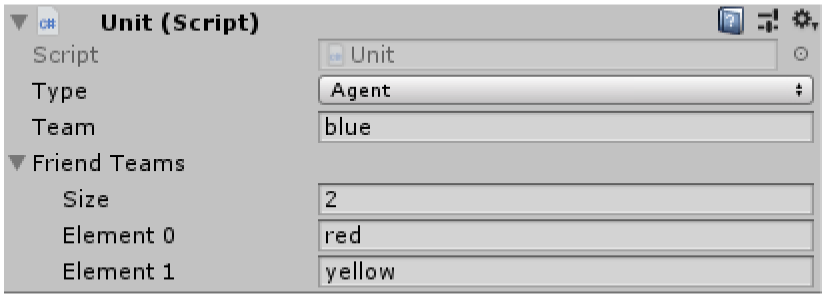
Motion Action Interface encapsulates methods for execution of functionality related to Agent’s Motion component, like idling, walking, running, dodging, you name it.

5.3.2 Inventory Action Interface

Inventory Action Interface encapsulates methods for execution of functionality related to Agent’s Inventory component, like using potions, wielding weapons, throwing items away etc.

6 Utility components

6.1 Unit



Unit (Script)	
Script	Unit
Type	Agent
Team	blue
Friend Teams	
Size	2
Element 0	red
Element 1	yellow

6.1.1 Description

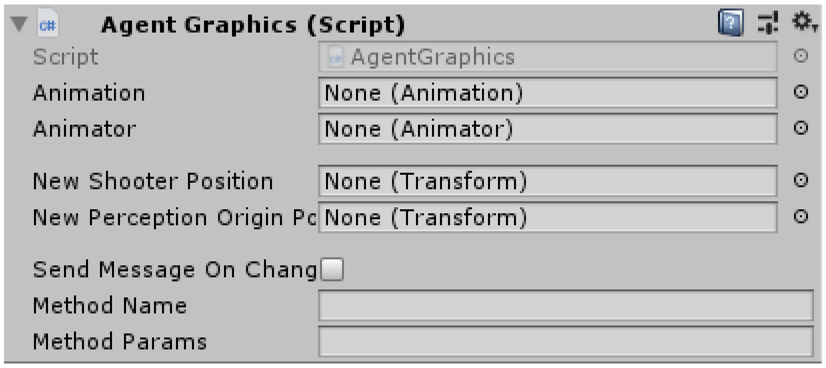
Although Unit is not a part of Eliot’s Utility namespace (it is part of Environment), it will be mentioned here, because it is more of a helper class.
Unit is a component that helps Eliot Agents orient themselves in the world. Agents can see only Units, which is a way of enforcing the explicitness of a decision to include or exclude game objects from field of view of Eliot Agents. Unit is also a convenient way to organize those properties of objects that are going to be frequently looked up by Agents, like team of a Unit and its type (which are default ones).
Since Agents can see only Units, use this component to mark objects that Agents need to be aware of. These objects can be anything, not just another Agents. The way Agent reacts to a certain Unit is totally determined by User via Behaviours.
Feel free to customize properties of Unit any way you like to facilitate the needs of your project.

6.1.2 Properties

Type represents a group of objects to which the Unit belongs. It is just a convenient and explicit way for Agents to know if they see another Agents, corpses, Items, flying projectiles, or anything else that’s declared inside **UnitType** enum.

Team is a string that Agents use to know how to treat Units they see. Agents don't have to be in one team to be friends. You can add friendly teams to **Friend Teams** list and Units that belong to friendly teams will pass the 'friends' check.

6.2 Agent Graphics


Agent Graphics (Script)

Script: AgentGraphics
Animation: None (Animation)
Animator: None (Animator)
New Shooter Position: None (Transform)
New Perception Origin Pc: None (Transform)
Send Message On Chang: ☐
Method Name:
Method Params:

6.2.1 Description

Agent Graphics helps change the appearance of an Agent when it is needed, for example, when Agent wields a new weapon or tool. It is Important to not miss the link to a new Animation/Animator component of a new graphics object. It also might be important to change the position of the origin of Skills and/or perception rays. Agent Graphics lets you do all of this easily. It also allows to Invoke methods on Agent gameObject by name using Unity's SendMessage(message) method.
Add this component to a top-level object of a target prefab that is going to be used as a new graphics for Agents. Specify one of the animation components, Animation or Animator, if it is present and used on a new graphics object. Specify the objects that hold information about new transform settings of Agents perception and shooter origins (__look__ and __shoot__). Specify method name to be invoked on any of MonoBehaviours that are present on Agent's gameObject. Any of these configurations are optional. It's going to work with all of these fields staying empty. But in this case there is no point in using this component.

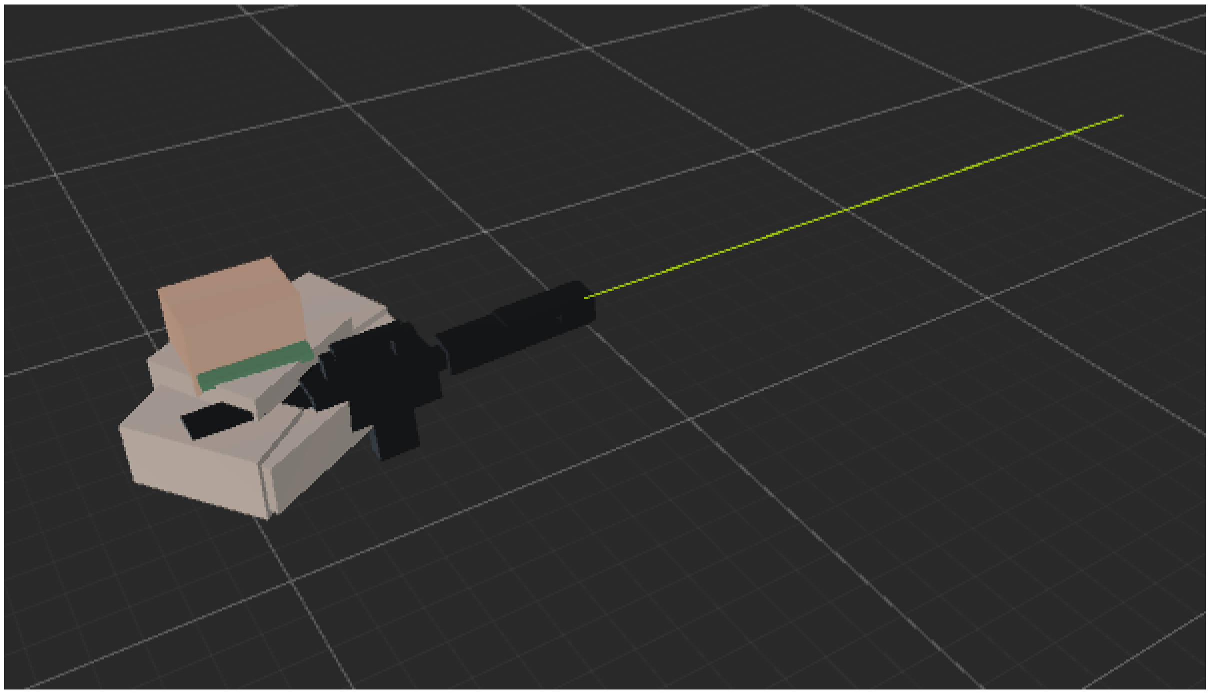
6.2.2 Properties

Animation is an object with an Animation component on a new graphics.
Animator is an object with an Animator component on a new graphics.

New Shooter Position - Agent's __shoot__ will copy position and rotation from this one.
New Perception Origin Position - Agent's __look__ will copy position and rotation from this one.

Send Message On Change defines whether or not to invoke any method on Agent when changing graphics.
Method Name is a name of the method to be invoked.
Method Params is a string parameter of the method. Can be left empty. Method will be invoked with no parameters in this case.

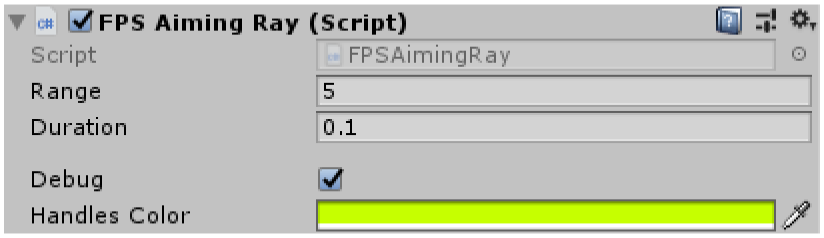
6.3 FPS Aiming Ray



6.3.1 Description

FPS Aiming Ray is a helper component that makes Agents aware of the fact that they are being aiming at, letting them react to it accordingly.
It works by setting status of any Agent that it crosses to *BeingAimedAt*, so that it can easily be checked with an Observer or a Loop in a Behaviour by an option in General Condition Interface.
You can add this component to any GameObject, it does not require any other classes on the same GameObject. It would probably make most sense to add FPS Aiming Ray to an empty GameObject that is a child of a transform that most accurately indicates the direction towards which a gun (or equivalent) is facing.

6.3.2 Properties

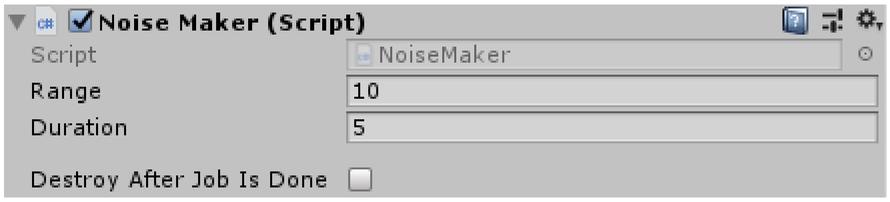


FPS Aiming Ray (Script)

Script	FPSAimingRay
Range	5
Duration	0.1
Debug	✓
Handles Color	🟩

Range is a maximum distance to which the ray is casted.
Duration is the amount time for which target Agent's status will be *BeingAimedAt*.
Debug determines whether to show the ray in Unity Editor. The ray's color is **Handles Color**.

6.4 Noise Maker


Noise Maker (Script)

Script	NoiseMaker
Range	10
Duration	5
Destroy After Job Is Done	☐

6.4.1 Description

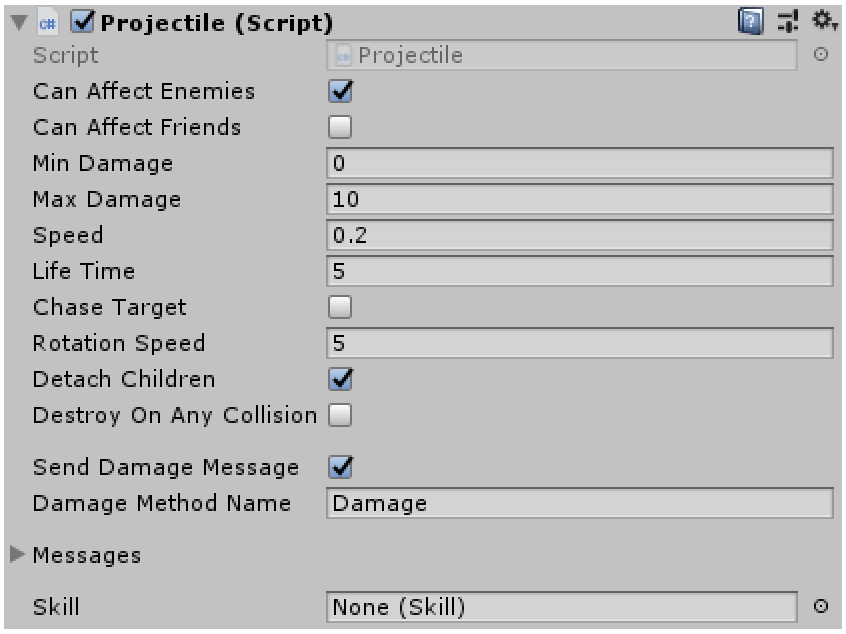
Noise Maker lets all the Eliot Agents in certain radius know that there is something going on at the position of the gameObject.
Agents can set the origin position of noise as their target by checking for **HearedSomething?** condition from Perception Condition Group in Behaviour Editor.
Makes noise in its Start() method, which means that it makes noise either at the start of a play mode or immediately after being instantiated. So it works if being added to, for example, bullets or any other projectiles prefabs.
In order to make Agents “hear noise”, this class uses a static method from Eliot Skill which is called MakeNoise(), which needs 3 parameters to work: range in which Agents will be affected, origin transform of the “sound”, and duration of the noise.
You are free to use **Skill.MakeNoise(float range, Transform origin, float duration)** anywhere in your code. Use case is certainly not restricted to NoiseMaker class.

6.4.2 Properties

Range is a radius at which the Agents will know about the event.
Duration is the amount of time for which Agents are going to be aware of the noise.

Destroy After Job Is Done indicates whether the gameObject should be destroyed right after making the noise.

6.5 Projectile



Projectile (Script)

Script	Projectile
Can Affect Enemies	✓
Can Affect Friends	
Min Damage	0
Max Damage	10
Speed	0.2
Life Time	5
Chase Target	
Rotation Speed	5
Detach Children	✓
Destroy On Any Collision	
Send Damage Message	✓
Damage Method Name	Damage
Messages	
Skill	None (Skill)

6.5.1 Description

Projectile is an object that an Agent instantiates on invoking a Skill by Projectile (defined in Skill settings). It might be a bullet, a fireball etc. Projectile is a helper class that carries settings of a Skill that initiated an instantiation of a projectile in itself.
Projectile also has its own settings that are taken into account when Projectile is instantiated not by a Skill.
This component should be added to a prefab of a projectile by hand.

6.5.2 Properties

Can Affect Enemies defines whether the Projectile can affect enemy units of an Agent who initiates an instantiation of the Projectile.
Can Affect Friends defines whether the Skill can affect friendly units of an Agent who initiates an instantiation of the Projectile.
A random value between **Min Damage** and **Max Damage** is used as a parameter of a method, named **Damage Method Name**, which is invoked on a collision object if **Send Damage Message** is true. Note that this is true only if **Skill** field is empty. Otherwise, Skill's settings are used to define the interaction with a collision object.
Speed defines the speed of projectile's movement.
Projectile gameObject is destroyed in **Life Time** seconds after an instantiation.
If **Chase Target** is true, Projectile rotates towards its target (defined by target of an Agent who invokes the Skill that instantiates the projectile) with **Rotation Speed**. Otherwise Projectile moves strictly forward (along local axis).
If **Detach Children** is true, all the children of Projectile's Transform are assigned null as a new parent right before projectile's destruction.

If **Destroy On Any Collision** is true, Projectile is going to attempt applying its effect to any collision object and destroyed immediately. Otherwise, this will happen only if a collision object has a Unit component on it and it is not another Projectile.
Messages is a list of strings that are used as names of methods that are invoked on a collision object using Unity's SendMessage(message) method.
Skill is an Eliot Skill that is used to define the way Projectile affects a collision object if it happens to be an Eliot Agent.