ELIOT TUTORIALS

Contents

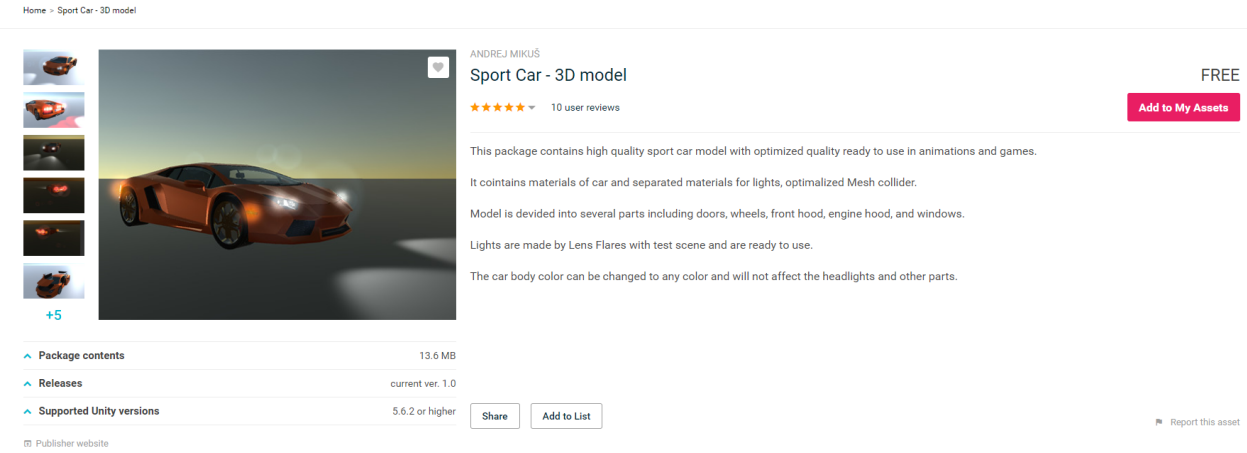
1 Basics

1.1 Car

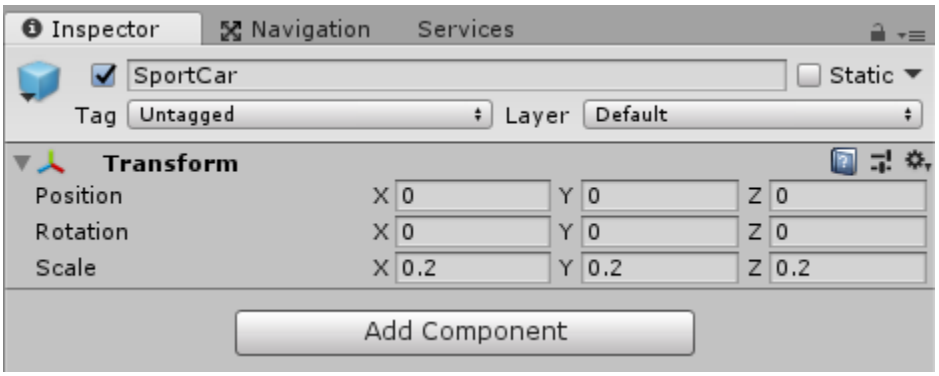
In this tutorial, you will learn how to create a new Agent, specify its path and create a simple behaviour for it.
Let's create a car that rides in circles using Eliot.
To follow this tutorial, open the scene "Tutorial 1", which is in "../*Eliot Tutorials/1"* folder.

1.1.1 Get prefabs ready

For this tutorial, we will use a car model that you can download from Asset Store for free here: https://assetstore.unity.com/packages/3d/characters/sport-car-3d-model-88076



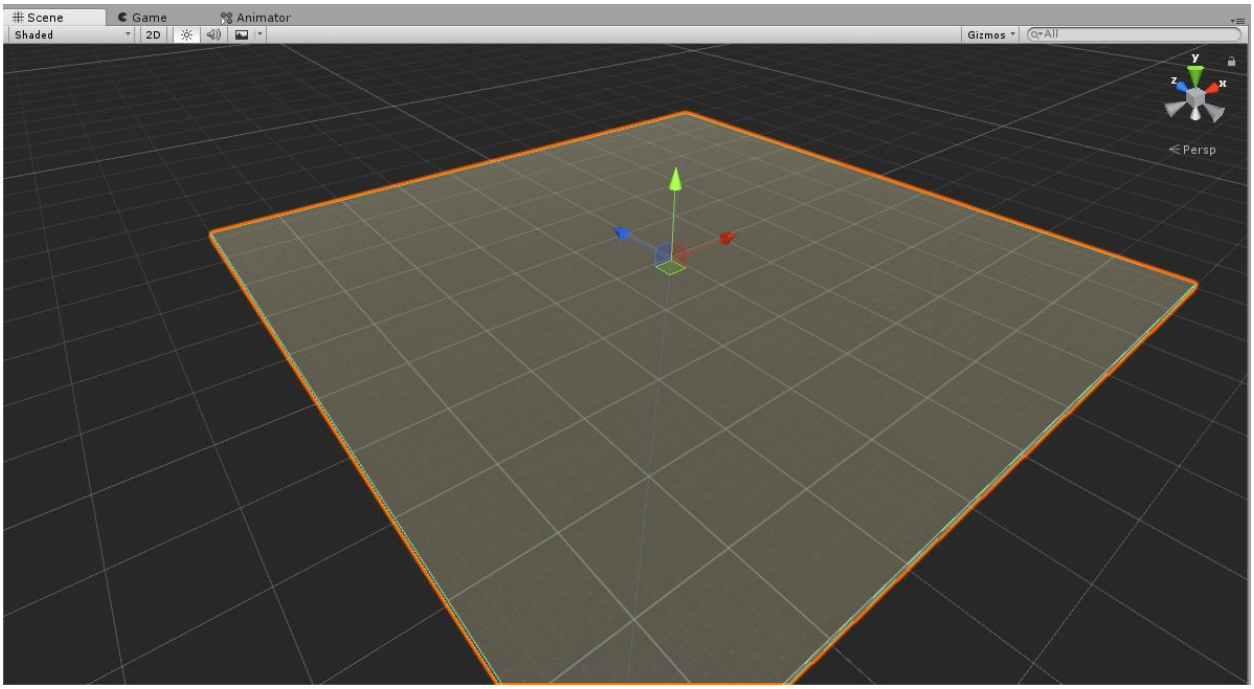
Download and import this asset. You can find all you need about importing assets in Unity Manual: https://docs.unity3d.com/560/Documentation/Manual/AssetStore.html
Go to folder "*../Sport Car – 3D model/Prefabs"*. Here you will find a prefab named "SportCar". Select it and remove all Components in the Inspector except Transform.
Set x, y and z of its scale to *0.2*. Now the prefab's Inspector should look like this:



Move this prefab to the folder ../*Eliot Tutorials/1/Prefabs* (create the folder if it is not already there).

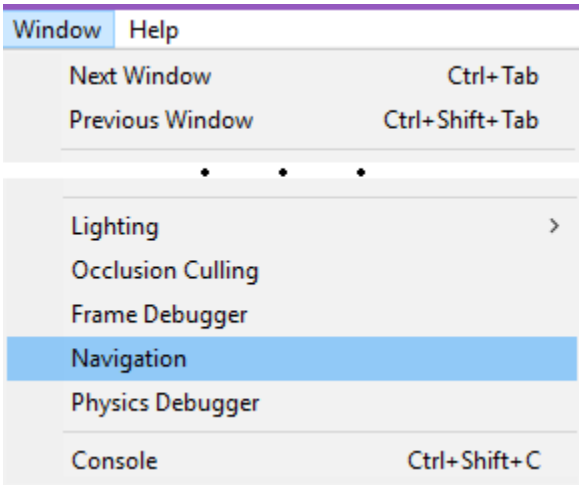
1.1.2 Bake the scene

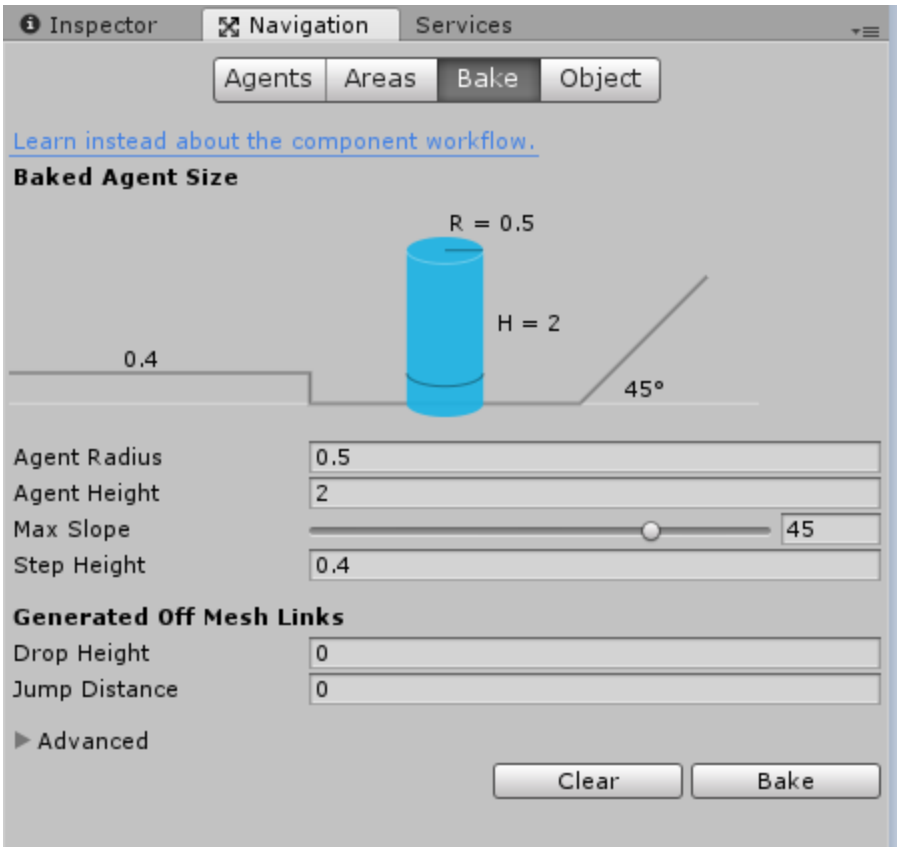
You should see a static cube with a grid texture on it.



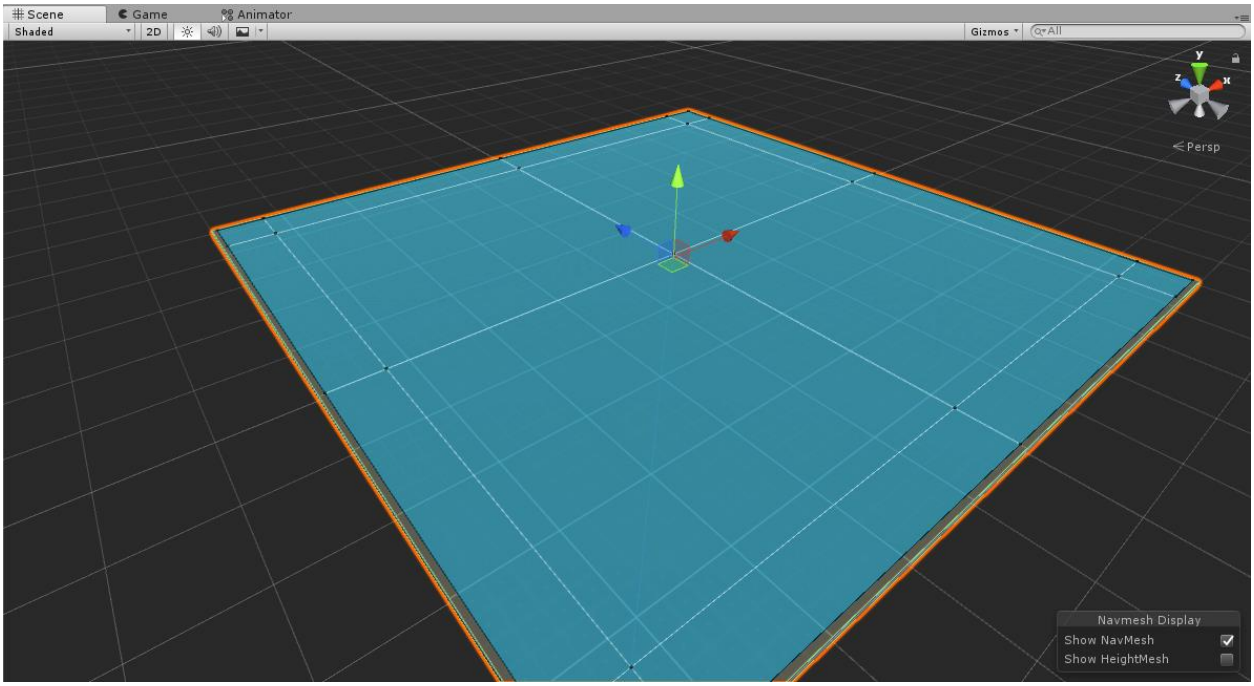
Since Eliot Agents move with the help of Unity's NavMeshAgent component, we need to make sure that we have a NavMesh in the scene for our Agents to be able to find paths and do all kinds of things. You can learn more about Unity's NavMesh here: https://docs.unity3d.com/Manual/nav-BuildingNavMesh.html

So, to build the NavMesh you need to go to Bake tab in Navigation window and click **Bake** button. To open the Navigation window choose *Window/AI/Navigation* (or *Window/Navigation*, it depends on your version of Unity) menu item.



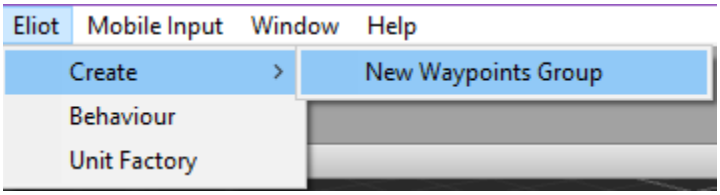


This will build a surface (NavMesh) on top of our cube. This surface is a walkable area for our Agents.



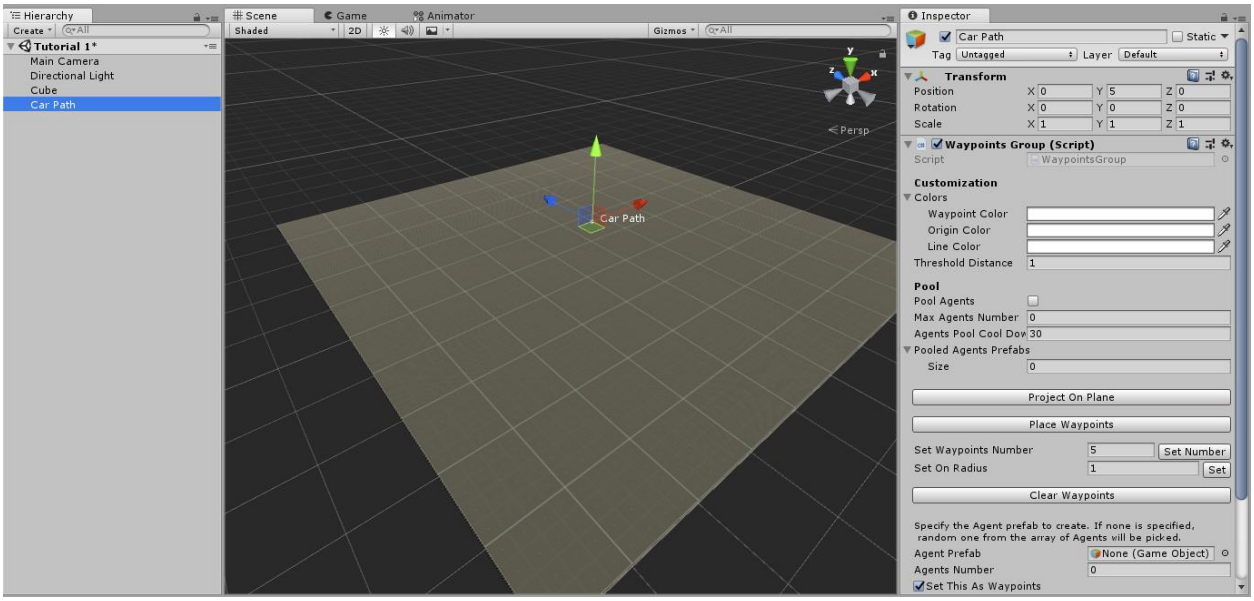
1.1.3 Create a path

Now we are going to specify the path that our car is going to follow. To do that, create a new Waypoints Group. The easiest way to do it is to choose *Eliot/Create/New Waypoints Group* menu item.

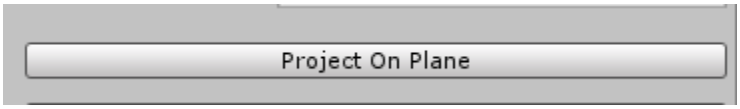


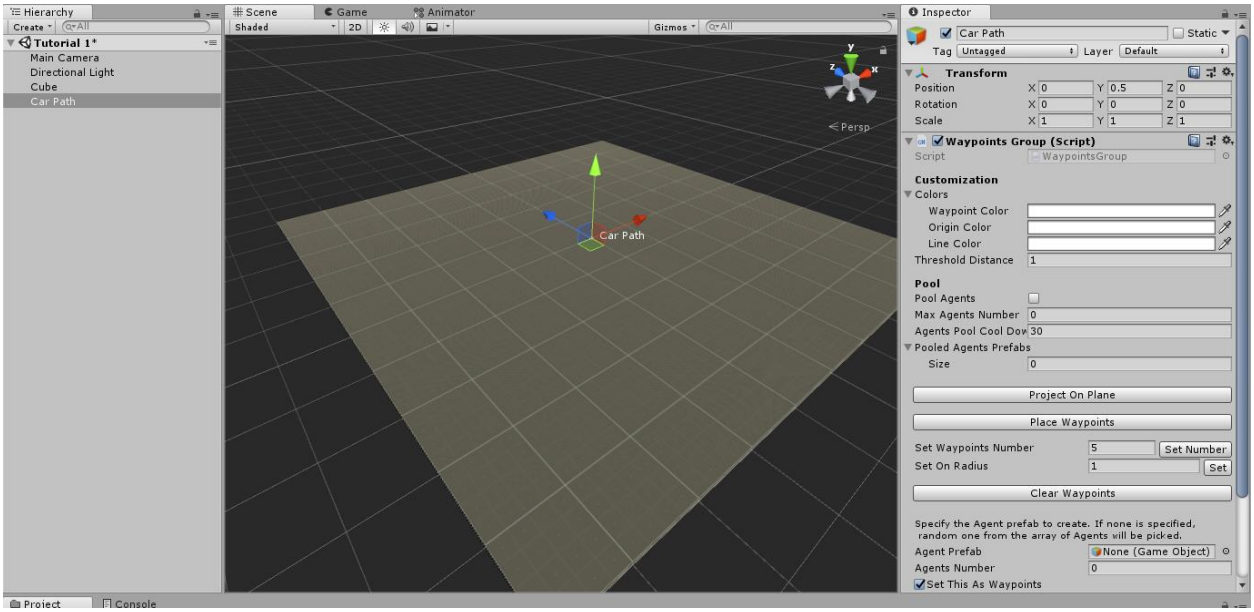
This will create a new object that has a WaypointsGroup component on it. Let's call it Car Path for convenience.

Now, to place it right on top of the cube, firstly drag Car Path anywhere above the cube. I'll just input its coordinates in the Transform component (0, 5, 0) so that it is centered relatively to the cube.



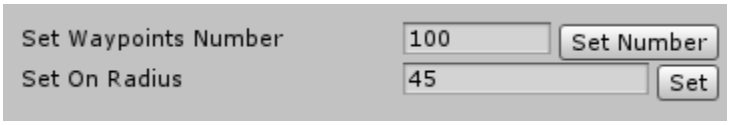
And now, to place it perfectly on top of cube's surface, just click **Project On Plane** button in Waypoints Group Inspector.



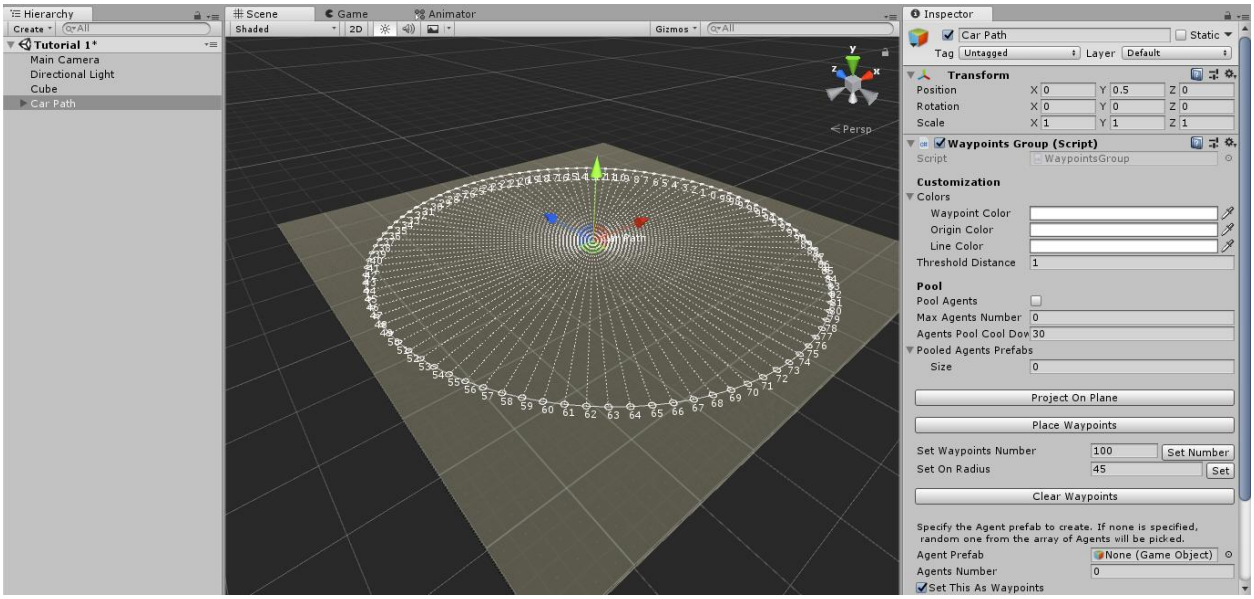


As you see, the Y coordinate of our path has changed.

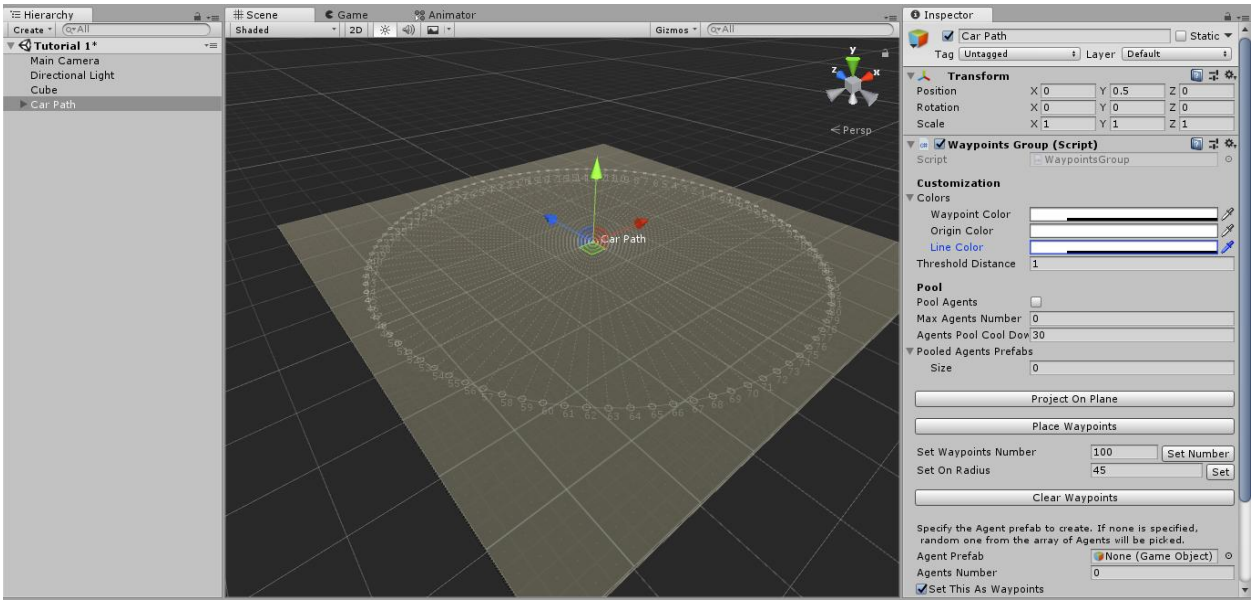
The next thing we want to do is to create Waypoints. Since our intent is to make a car that rides in circles, we need to place our Waypoints on circle's edge. The easiest way to do this is to input desired values in **Set Waypoints Number** and **Set On Radius** fields. They are used as a number of Waypoints to instantiate and radius of the circle on which they will be evenly placed respectively.
To create a smooth path, it would be a good idea to create a large number of Waypoints, like 100. And a reasonable radius for given surface would be 45.



Now, after clicking the **Set Number** button, all of previously created Waypoints in this group will be destroyed (if there are any) and **Set Waypoints Number** of new ones will be evenly distributed on the same distance from the center, which is equal to **Set On Radius**.



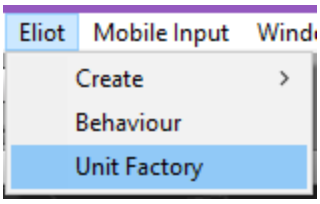
In order to be able to see the car, which we will create shortly, better let's reduce alpha of **Waypoints Color** and **Line Color**, which can be found under **Customization** header of Waypoints Group's Inspector.



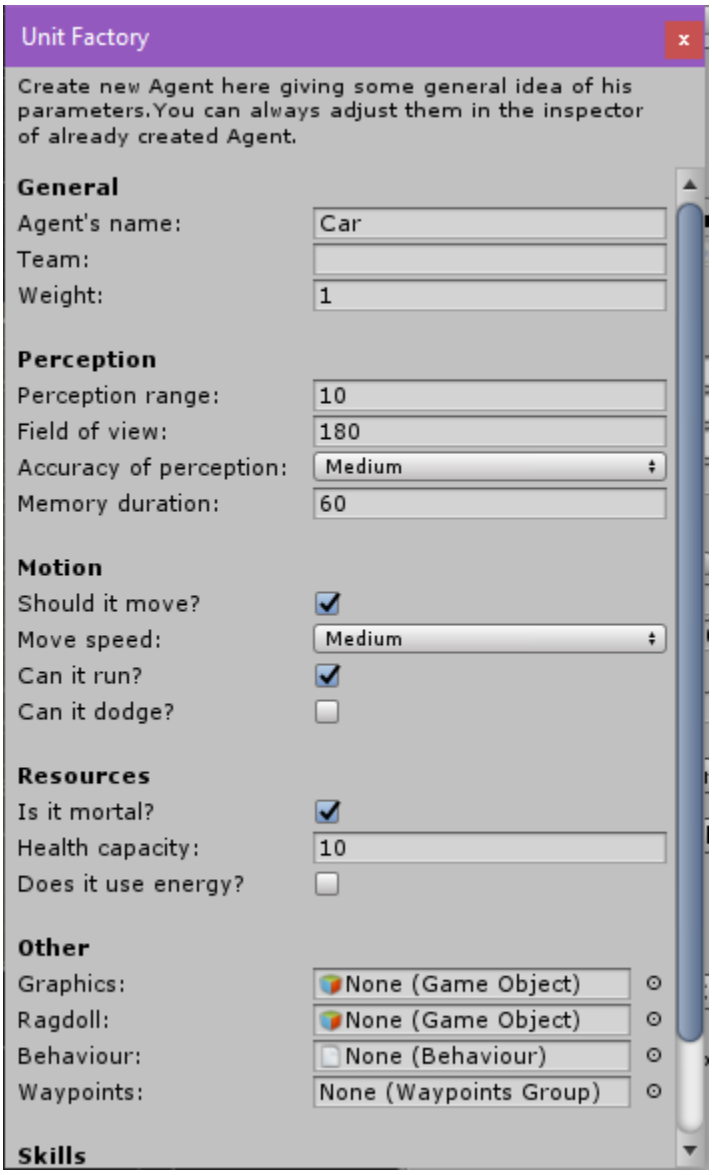
Great! Now it is time to create the Agent, which is going to represent a car.

1.1.4 Create Agent using Unit Factory

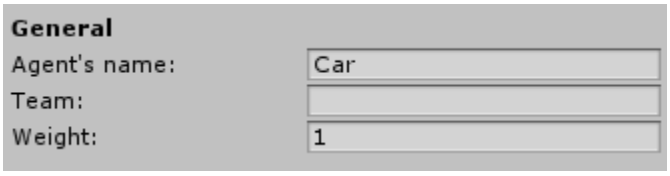
The most convenient way to create new Agents is to use Unit Factory. So let's do just this. Firstly, you need to open Unit Factory window. Do this by choosing *Eliot/Unit Factory* menu item.



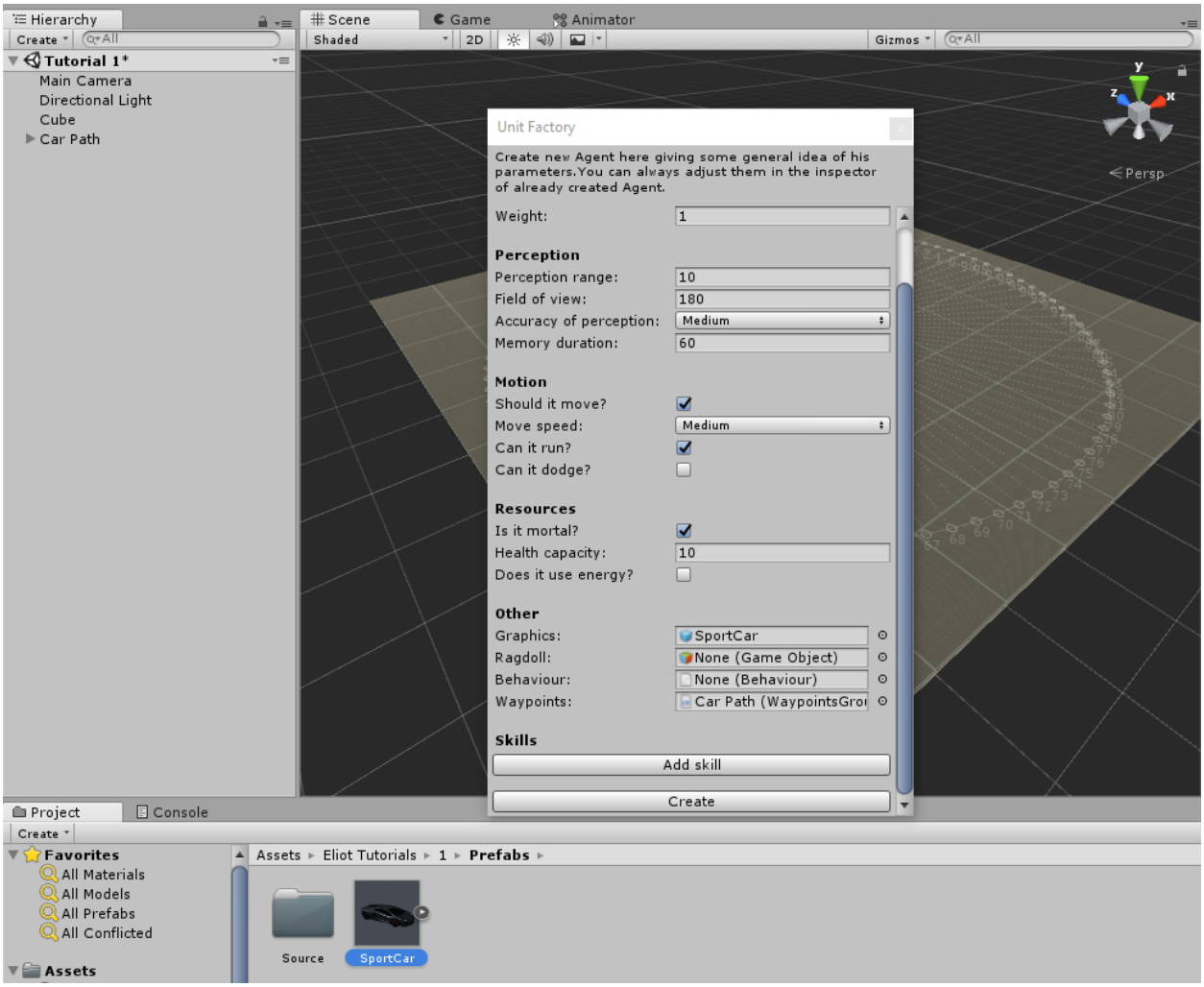
A new window will open up.



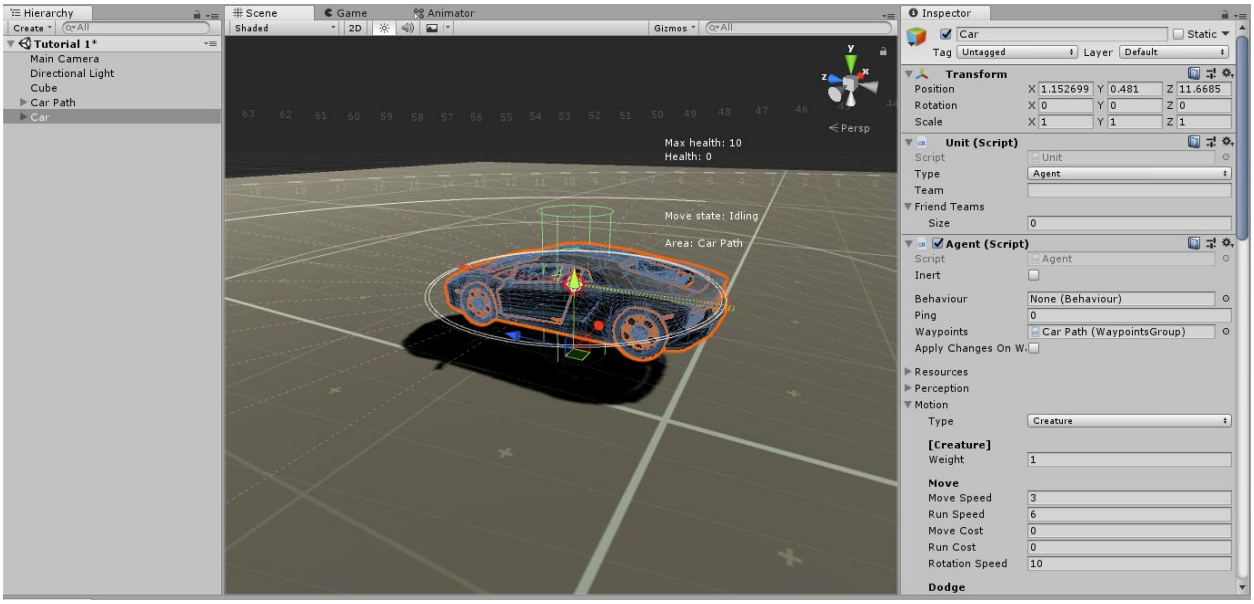
Let's call this Agent "Car", **Team** is irrelevant at this point, and we will leave **Weight** at the value of 1.



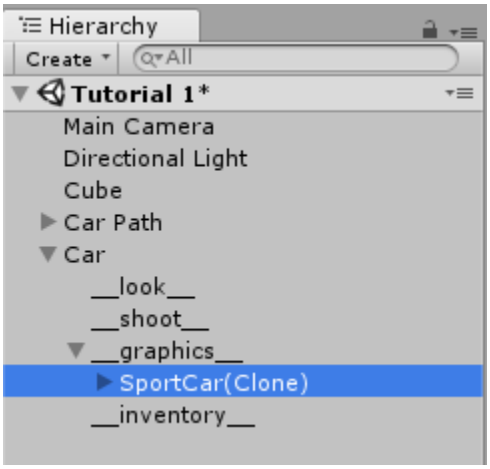
Perception, Motion and Resources are not going to be used for this particular Agent, so we can just skip them for now.
However, it is convenient to specify **Graphics** and **Waypoints** here. Go to "*../Eliot Tutorials/1/Prefabs"* folder in the Project window. Drag and drop SportCar prefab to **Graphics** field in Unit Factory. Drag and drop Car Path from our scene to Waypoints field in Unit Factory.



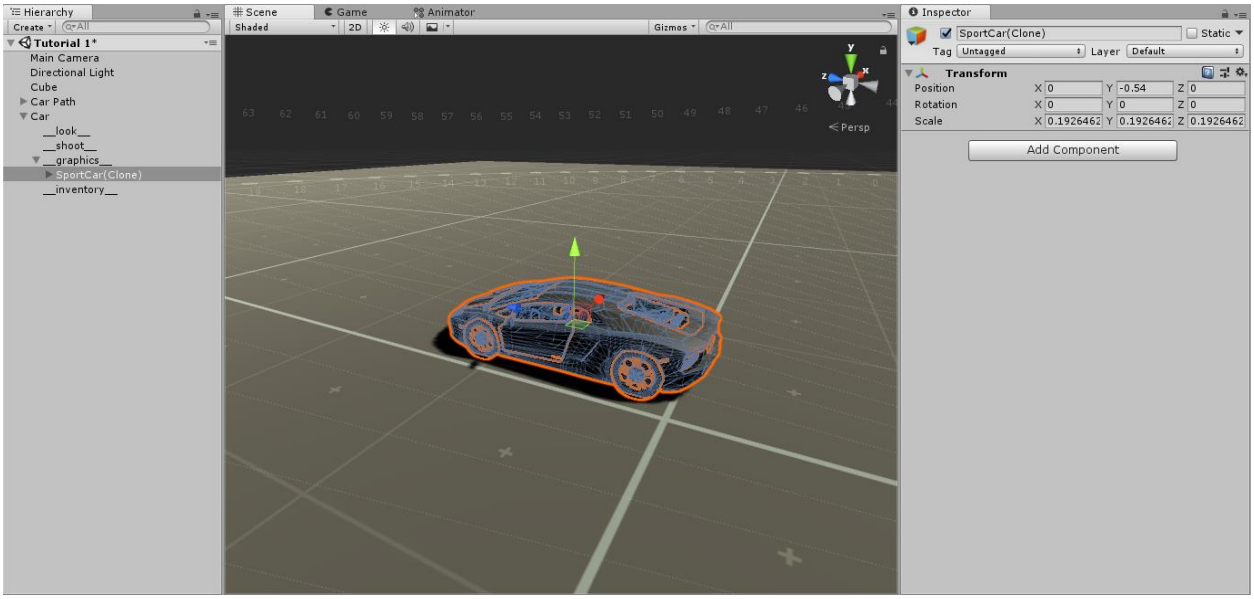
We are ready to push **Create** button and close the Unit Factory window. If everything done right, we should be able to see our new Agent. Put it somewhere on the cube's surface (as it is shown in the next picture) so that it is more comfortable to finalize tweaking its parameters.



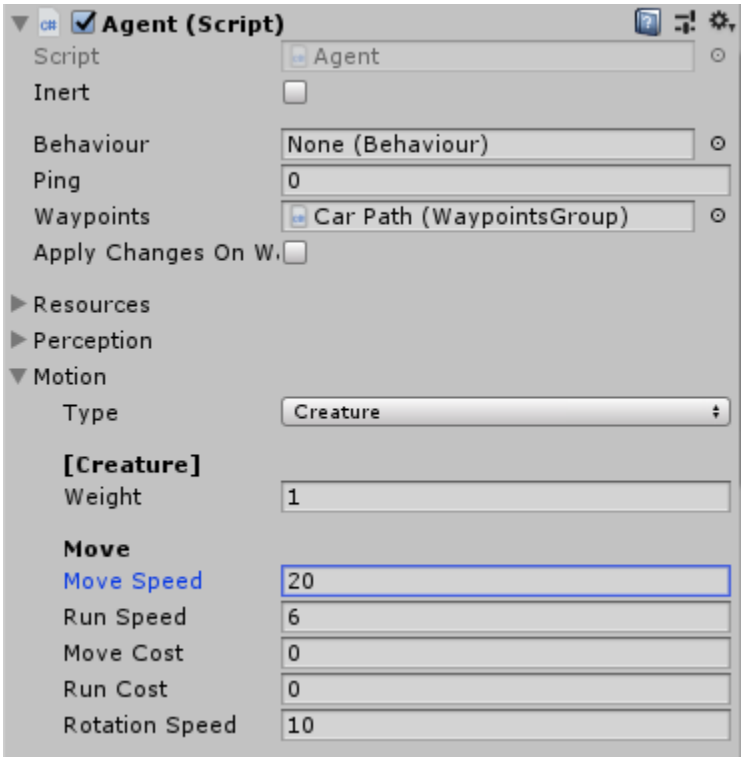
As you can see, although the Agent is right on top of the surface, the Car model is not. So, let's fix it. Select SportCar(Clone) GameObject, which is a child of Agent's __graphics__ container.



Drag it down in the scene to put it right on the floor.



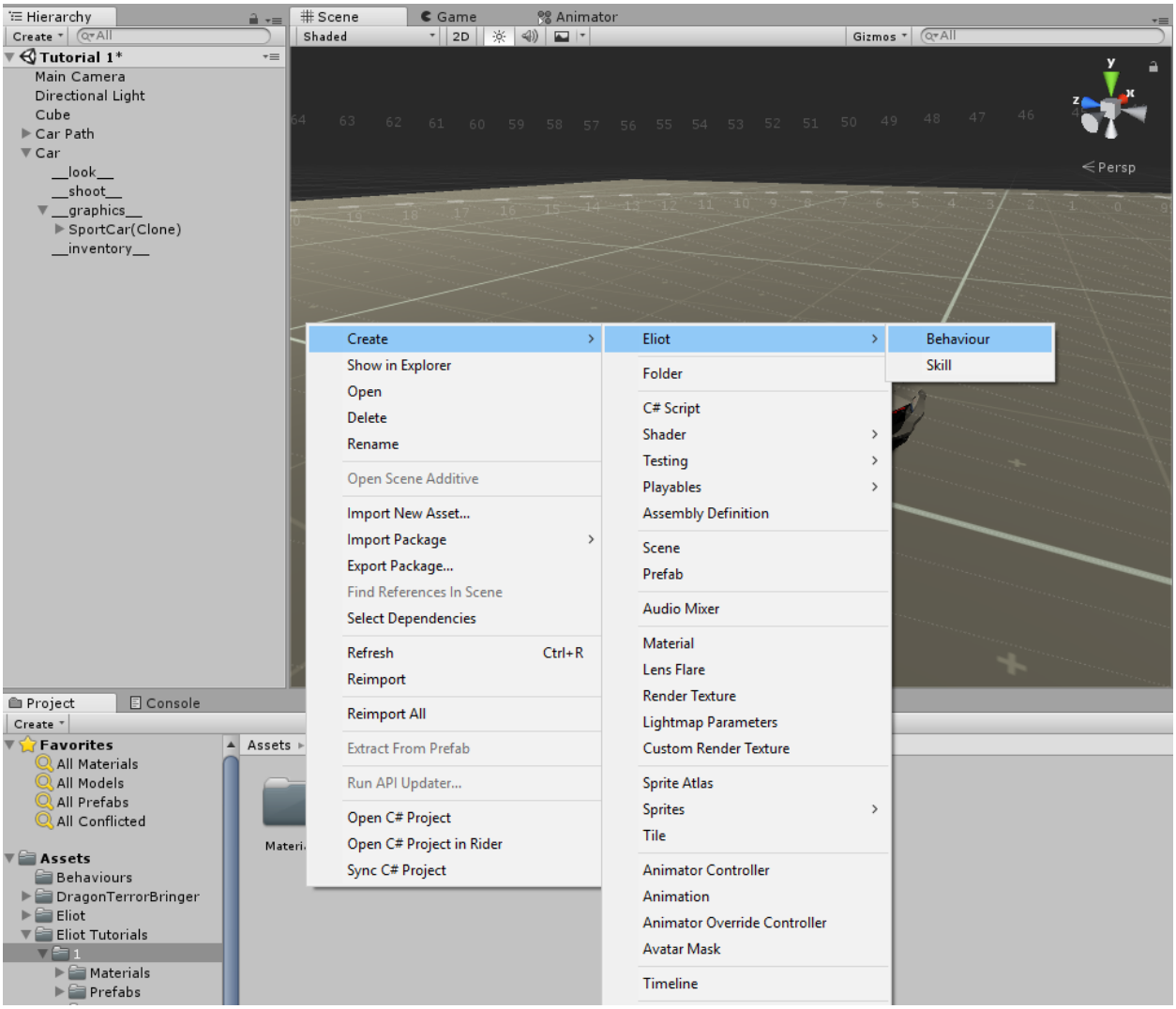
Perfect. Now, for aesthetic purposes, let's tweak some Agent's settings. Select the Agent and make sure you see the settings of its Motion component in the Inspector. Set **Move Speed** to 20. This way we will make sure that the car is not too boring to stare at.



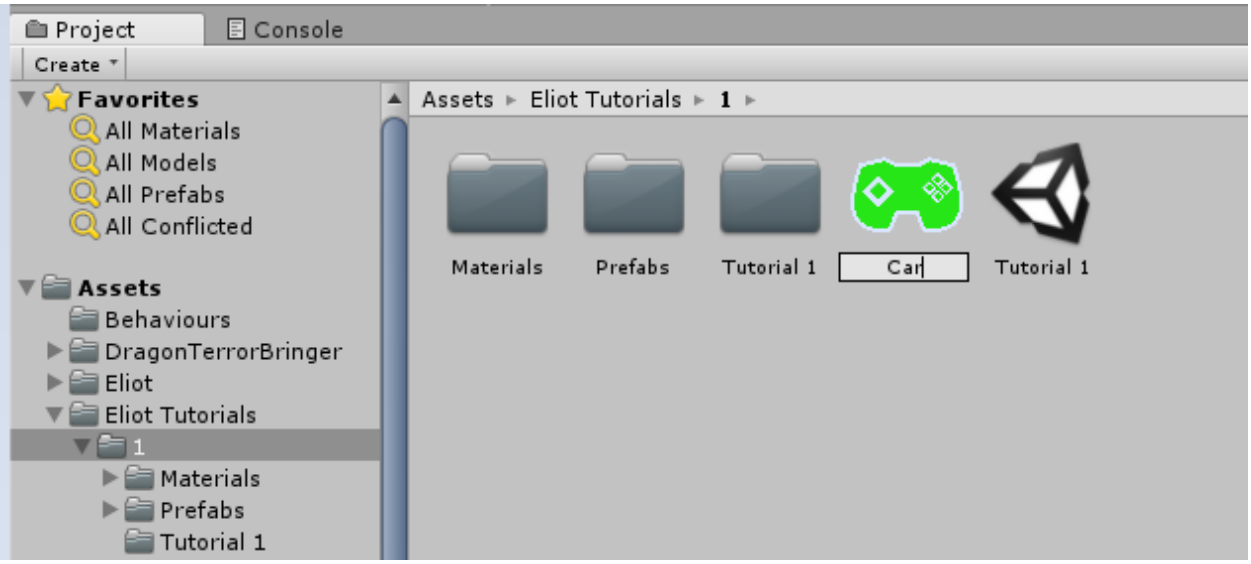
Alright, we are now ready to do a final step - create car's behaviour.

1.1.5 Create Behaviour

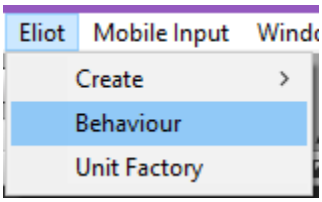
Firstly, we need to create a file that is going to keep the behaviour. Create a new Eliot Behaviour anywhere in your Project.



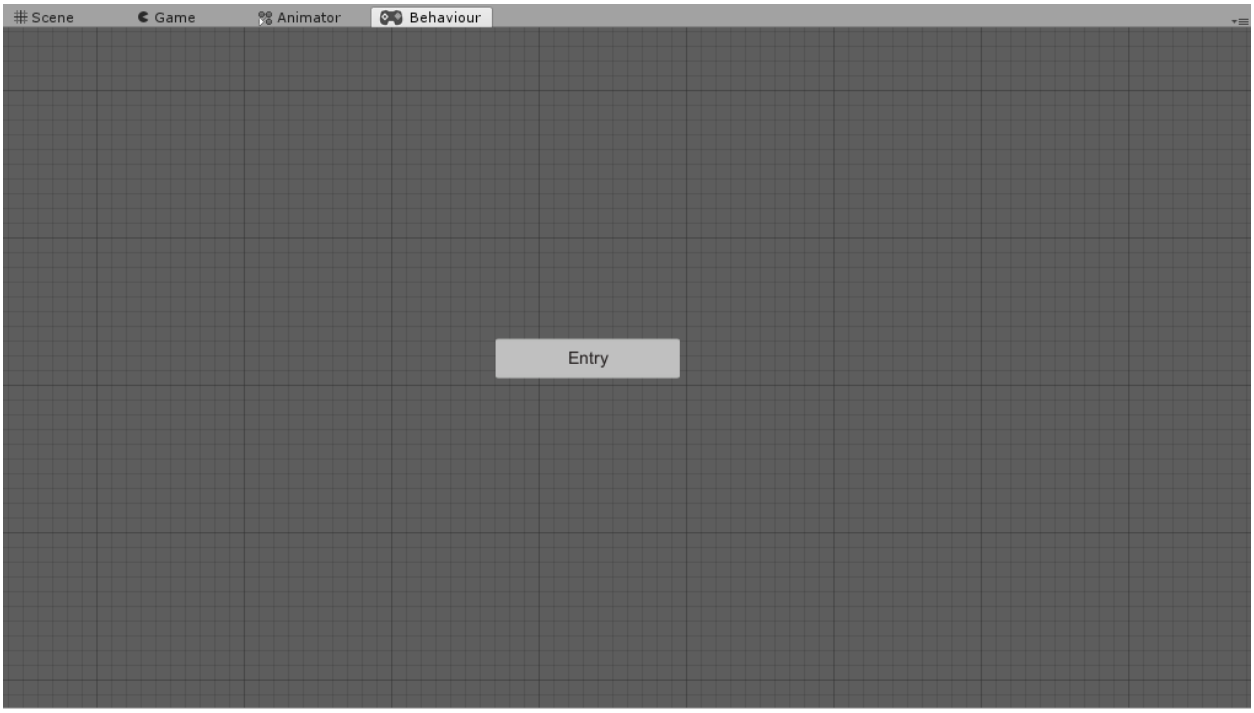
Let's name this one a Car.



Now open the Behaviour Editor window. You can do that by choosing *Eliot/Behaviour* menu item.

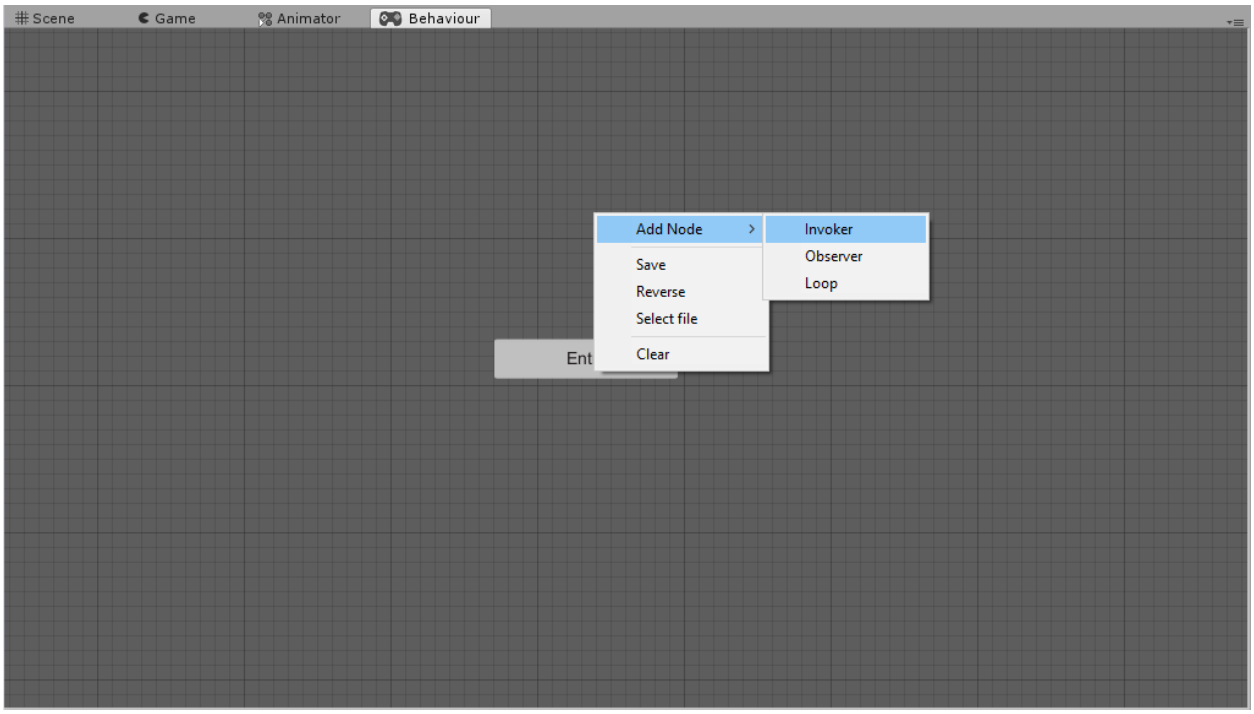


Drag and drop the Car Behaviour file anywhere inside the Behaviour window. Now we are ready to edit this file.

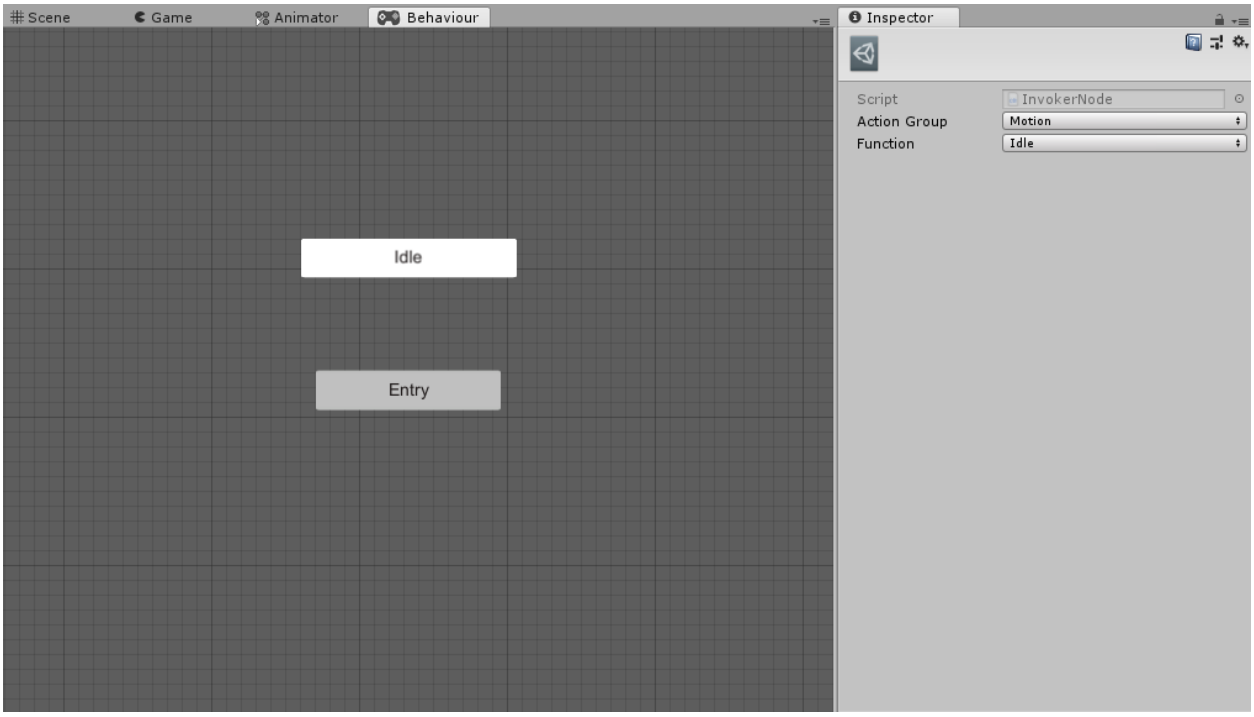


As you have probably noticed, there already is one element in our behaviour model, which is Entry. This element is the entry point of our algorithm, which means that all other elements to which Entry is directly connected, will be activated first.

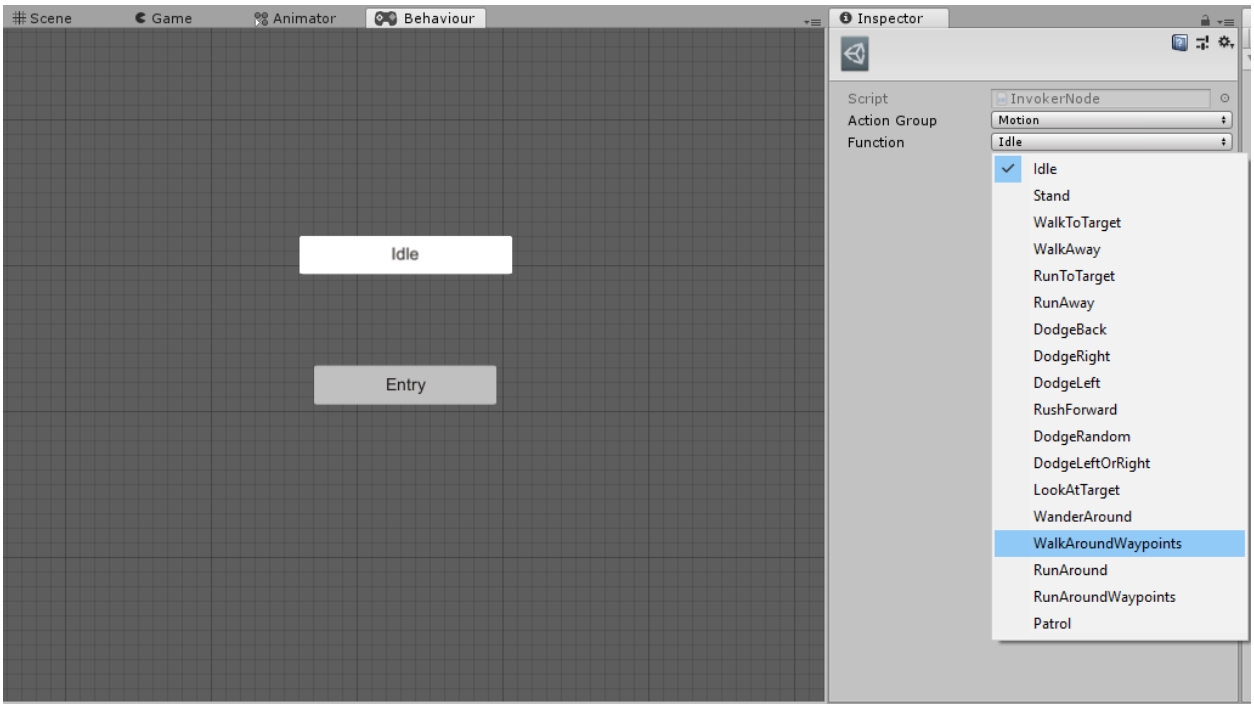
Now we need to make sure that the car rides in circles. Fortunately, there is already a method in Eliot's library that makes agents pass each Waypoint in the group in order. This method is called **WalkAroundWaypoints** and it can be found in MotionActionInterface (you can find more details about it in the Manual: http://www.eliot-ai.com/learn/).
To invoke this method, we need to use Invoker (surprise, surprise). To create one, choose *Add Node/Invoker* menu item or just click **I** while cursor is anywhere inside the window.



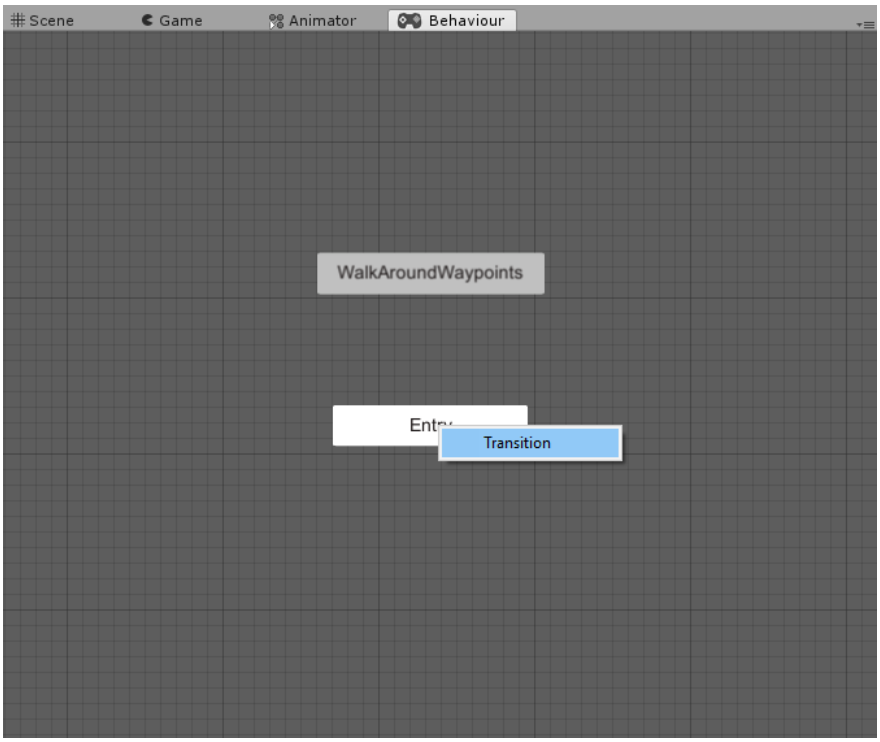
If you have done it right, you should be able to see a new element in the window. By looking at the Inspector while the new node is selected, we can see that it is setup to perform **Idle** function from Motion group of Agent's actions by default.



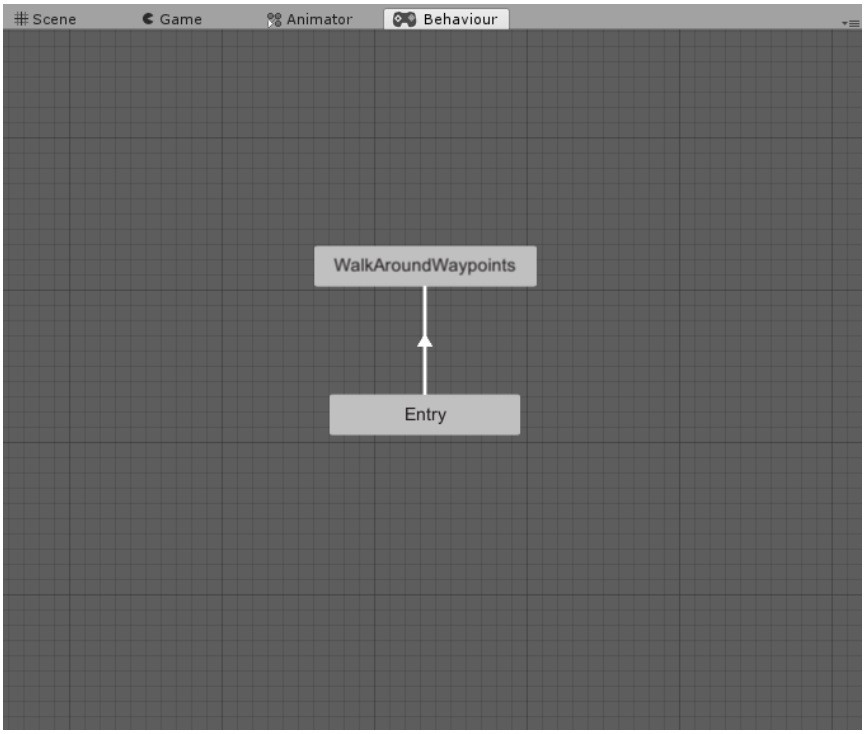
We need to change it to the **Walk Around Waypoints** function to make Agent do what we have intended.



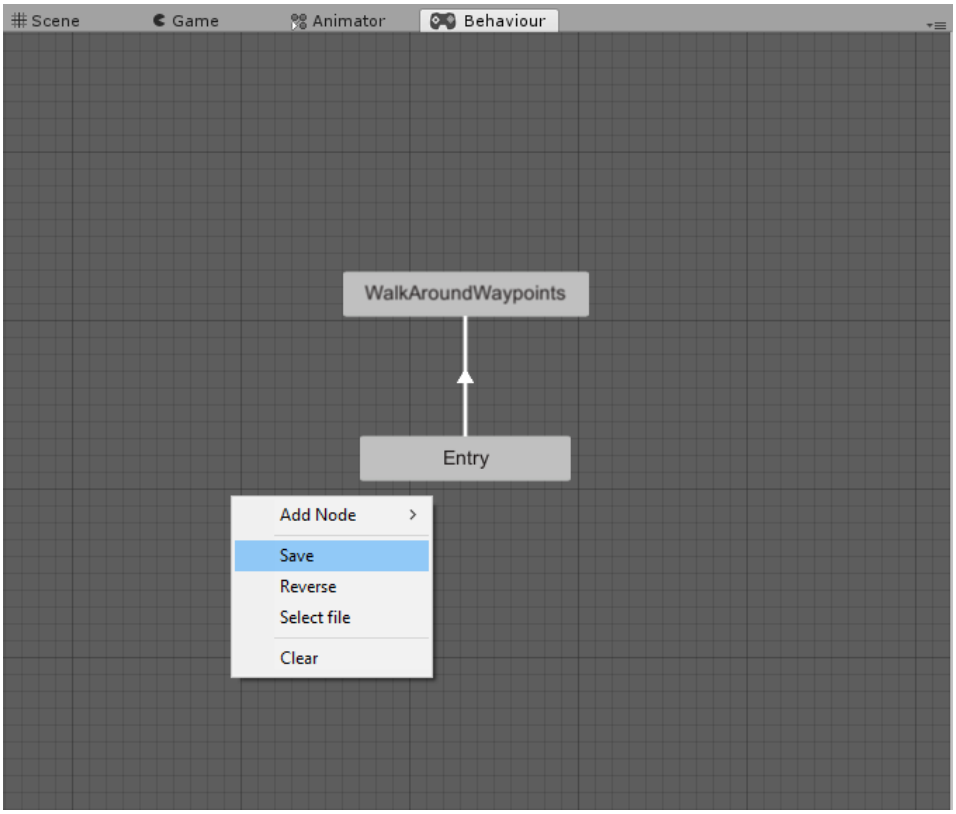
After we done this successfully, the only thing that's left is to connect it to Entry. To do this, choose Transition menu item from Entry's context menu or click Y while the mouse cursor is over the Entry.



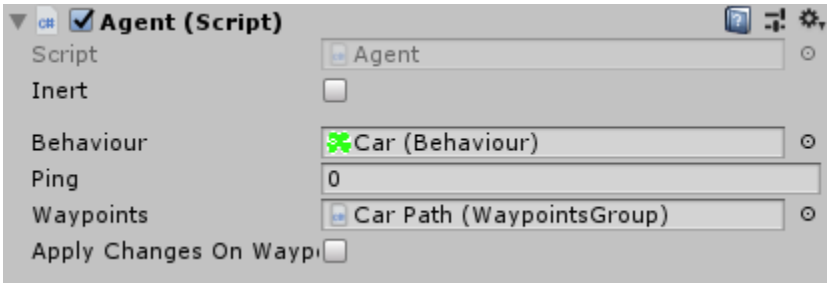
You should see a line which has its beginning at Entry's center and its end should be following the cursor. While this is the case, click on a node which you want to connect Entry to. In our case it is the WalkAroundWaypoints Invoker.



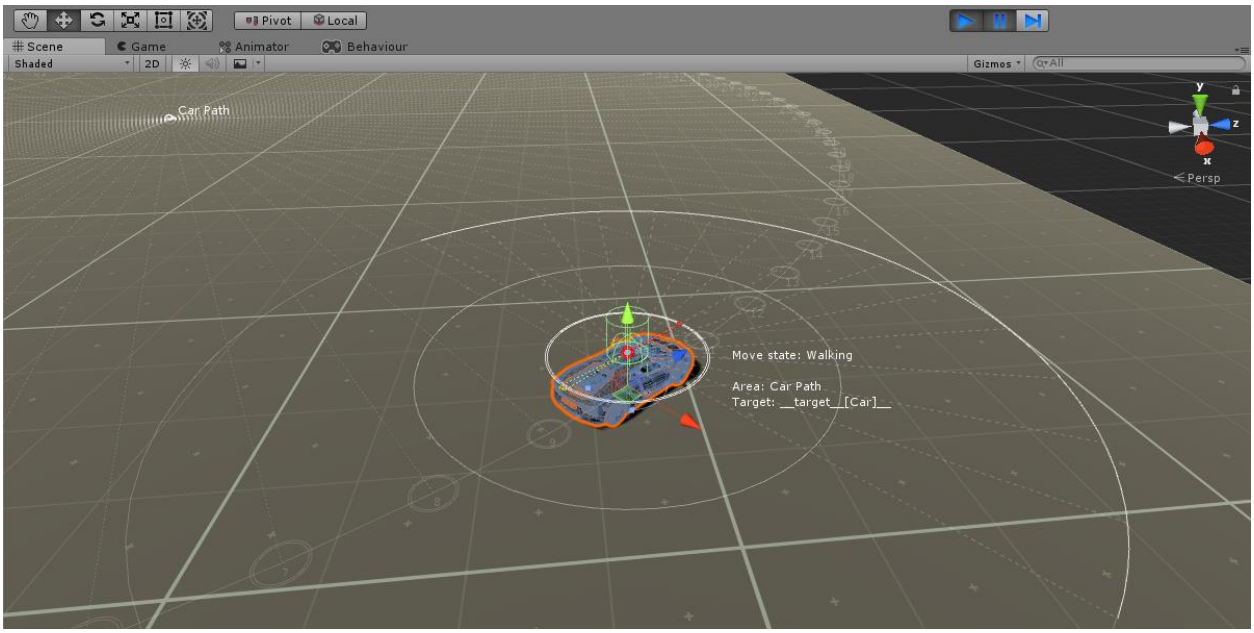
Now we should make sure our work is saved (either by using **ctrl+s** shortcut, or by choosing *Save* from window context menu)



Now it is time to assign our algorithm to the Agent. To do this, drag and drop the newly created Behaviour file to the Agent's **Behaviour** field.



Congratulations! You have successfully created and setup your first Eliot Agent!



2 Battle of magic creatures

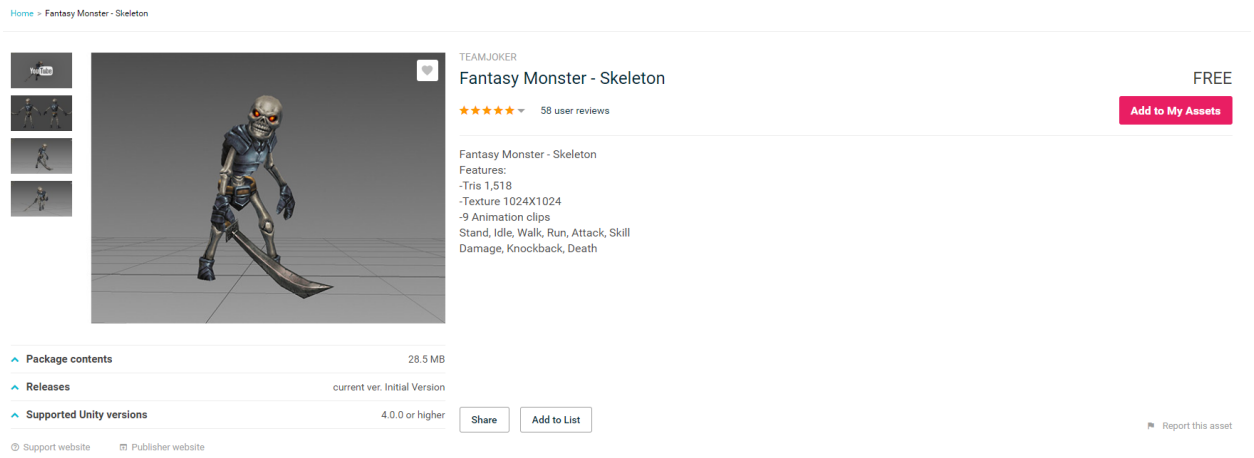
In this series of tutorials you will learn how to create more complex behaviours, how to create Skills, and how to make Agents cooperate or fight against each other.
We will create two skeletons, which will be in one team, one of them will be a melee damage dealer, and the second one will be a healer. Also we will create a dragon, who will fight against skeletons.

2.1 Melee attacking Skeleton

In this tutorial, we will create a Skeleton that can attack enemies with his sword.

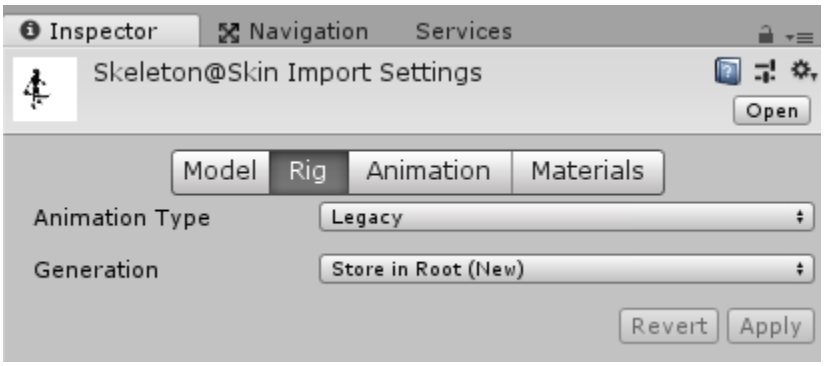
2.1.1 Get prefabs ready

For this tutorial, we will use a Skeleton model that has some animations on it and uses Legacy animations. You can get this asset for free from Asset Store here: https://assetstore.unity.com/packages/3d/characters/humanoids/fantasy-monster-skeleton-35635

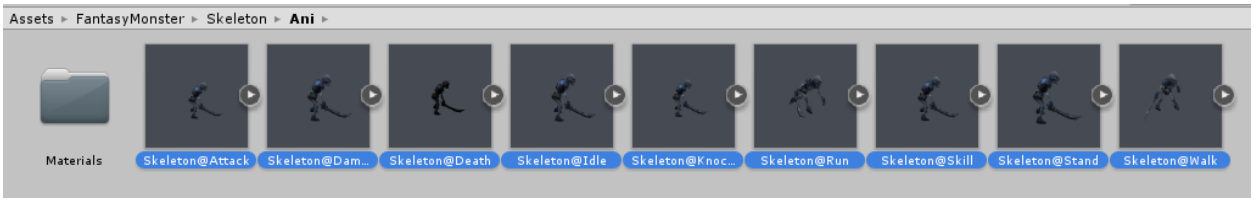


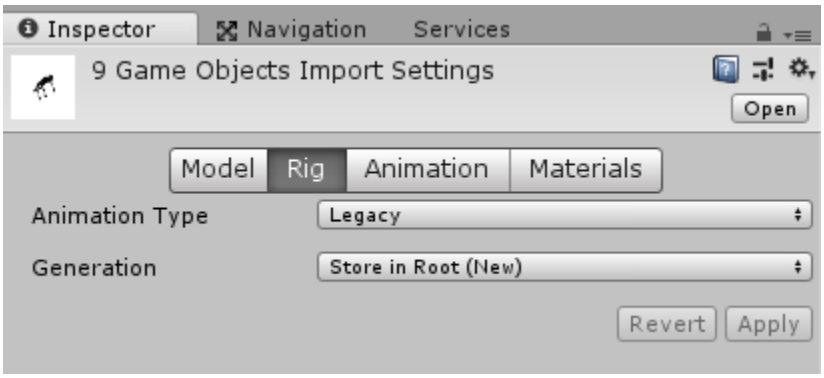
Download and import the package.

By default, this asset uses Mecanim for animations. But we are going to reconfigure it so that it uses Legacy animations instead (for the sake of exercise). Select “Skeleton@Skin” in “*../FantasyMonster/Skeleton/Character*” folder. Set its Animation Type under Rig tab to Legacy in the Inspector.

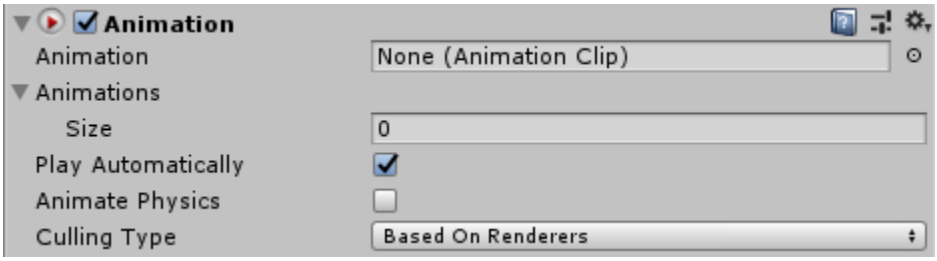


Do the same for all of the files in “*../FantasyMonster/Skeleton/Ani*”.

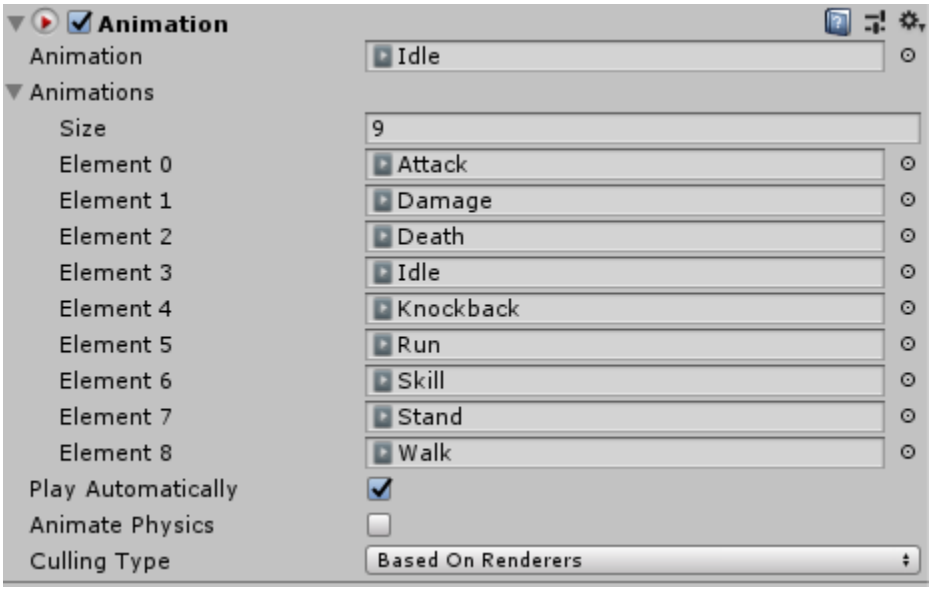




Go to folder "*../FantasyMonster/Skeleton/Character"*. There you are going to find a file named "Skeleton@Skin". We are going to use it to prepare graphics for some of our future Agents. Drag and drop this file anywhere in the scene. It does not matter what the scene is. We are just going to create a prefab, so this object is going to exist in a scene temporarily.
You are going to see that it automatically has an Animation component attached to the newly created object in the scene.

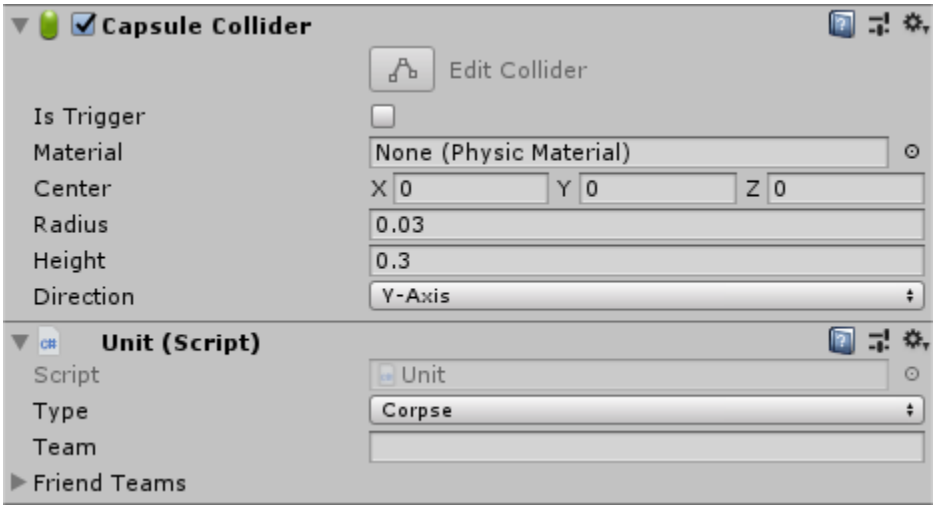


It needs some further configuration. Move folder "*../FantasyMonster/Skeleton"* (folder "*Skeleton"* which is inside folder "*FantasyMonster"*) to "*../Eliot Tutorials/2"* for convenience.
In folder "*../Eliot Tutorials/2/Skeleton/Ani"* you are going to find files that keep different Skeleton's animations. Use clips from these files to configure Animation component of our Skeleton that is currently in the scene so that its Animation component looks like this in the Inspector:



Name this GameObject “Skin” and make prefab out of it by dragging it from Hierarchy window into “../*Eliot Tutorials/2/Prefabs*” in the Project (create the folder if it is not already there).
Now after we have Skeleton saved as a prefab, we can delete it from the scene.

Now we are going to prepare Agent’s ragdoll, which means that this prefab is spawned at Agent’s position upon its death. Go to the folder “*../Eliot Tutorials/2/Skeleton/Ani/*”. Drag and drop *Skeleton@Death* anywhere in the scene. Again, it does not matter which scene it is as we are just preparing prefabs. Add **Capsule Collider** and **Unit** components to the newly created GameObject and configure them as shown in the screenshot:

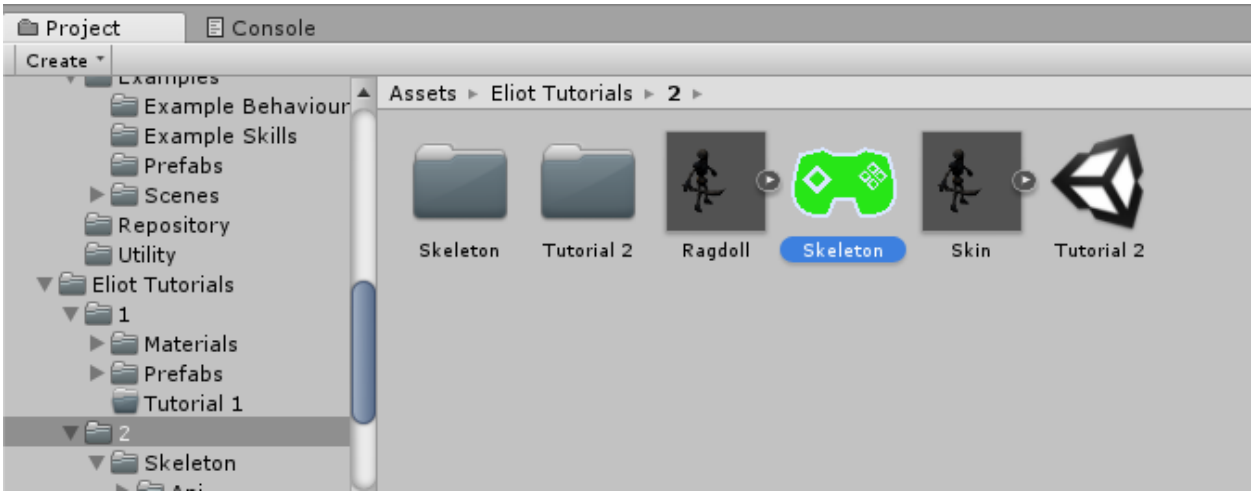


Name this GameObject “Ragdoll” and make prefab out of it by dragging it from Hierarchy window into “../*Eliot Tutorials/2/Prefabs*” in the Project.
Now after we have Skeleton Ragdoll saved as a prefab, we can delete it from the scene.

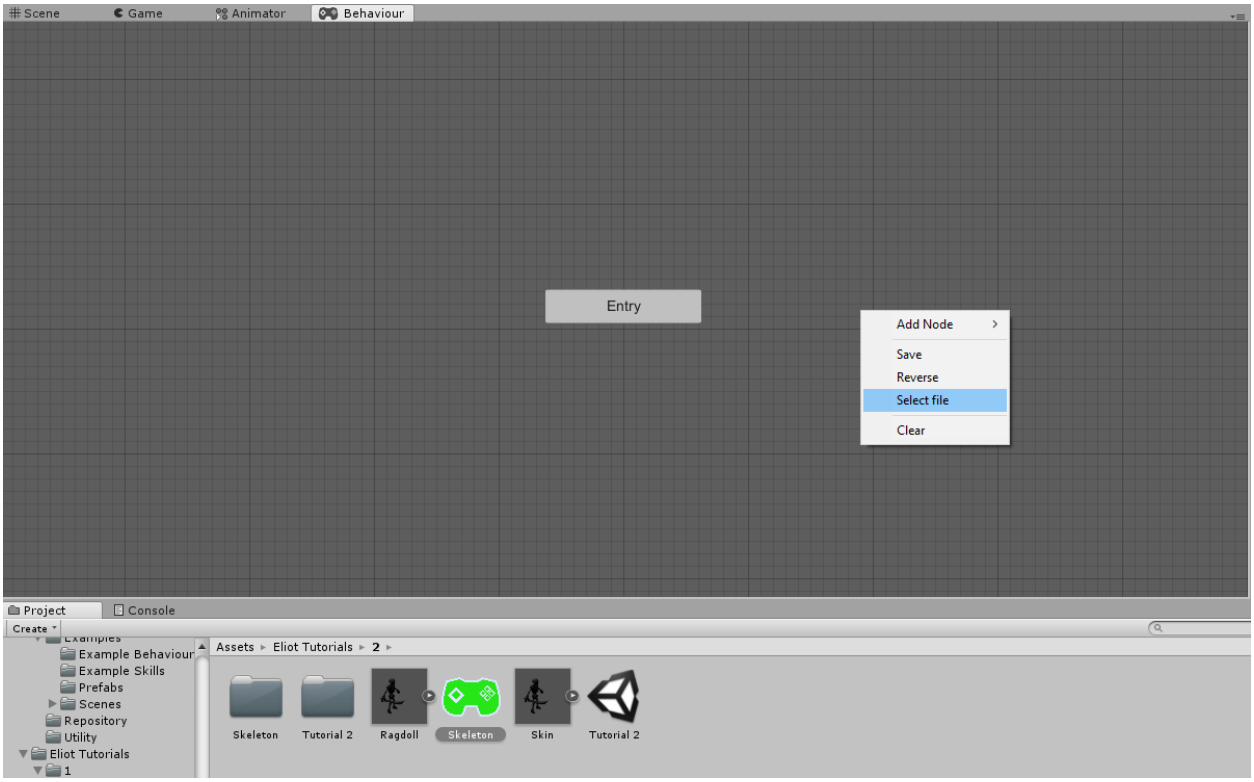
Our initial intent for this agent is to make him able to fight. We are going to need a Skill for this. Therefore, to be able to complete the creation of our Agent with Unit Factory, we need to have Behaviour and Skill ready. So, let’s start with a Behaviour.

2.1.2 Create Behaviour

Create a new Behaviour anywhere in your project. Let’s call this one “Skeleton”.



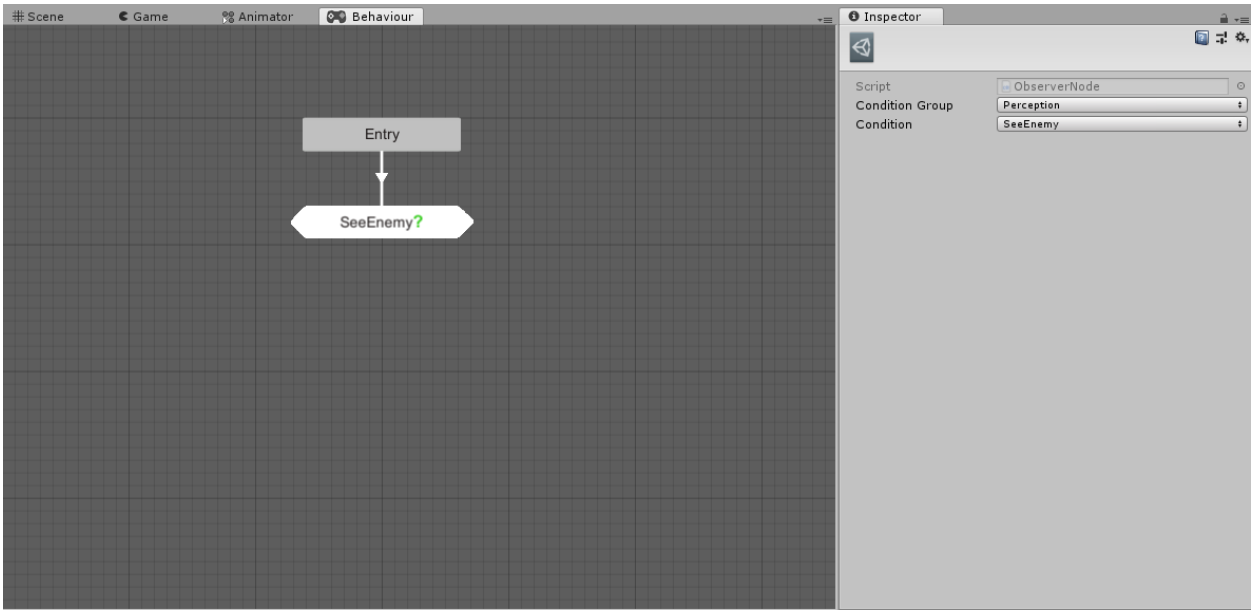
Now drag and drop it into Behaviour Editor window to edit it.



You can check what behaviour file is currently being edited by choosing Select file option in Behaviour Editor context menu.

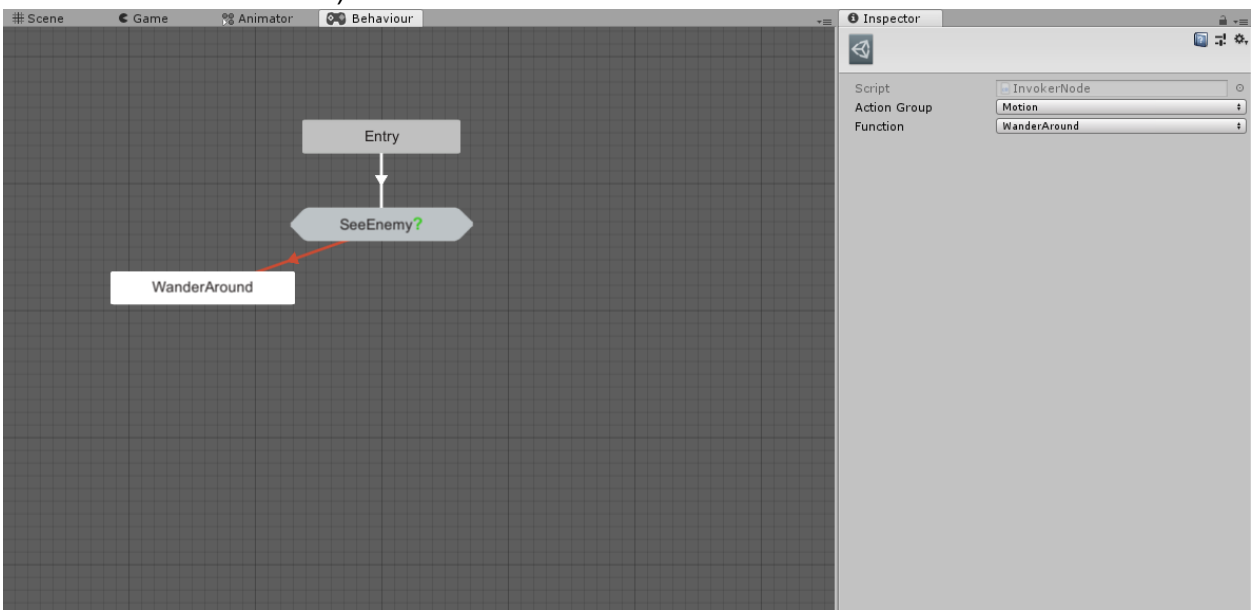
Now let's build the behaviour algorithm. Since our Agent will interact with other Agents, trying to attack enemies (Agents, who are in a different team), he is going to need to check whether he can see the enemy.
To do that, we need to create a new Observer, what is going to check that condition. Choose *Add Node/Observer* or use shortcut "**O**". A new Observer should appear. Set its **Condition Group** to *Perception* and its **Condition** to *SeeEnemy* in the Inspector. To make it the first

condition to be checked in our algorithm, create a Transition that goes from Entry to the Observer (use context menu of Entry or a shortcut “**Y**” while the cursor is over Entry).

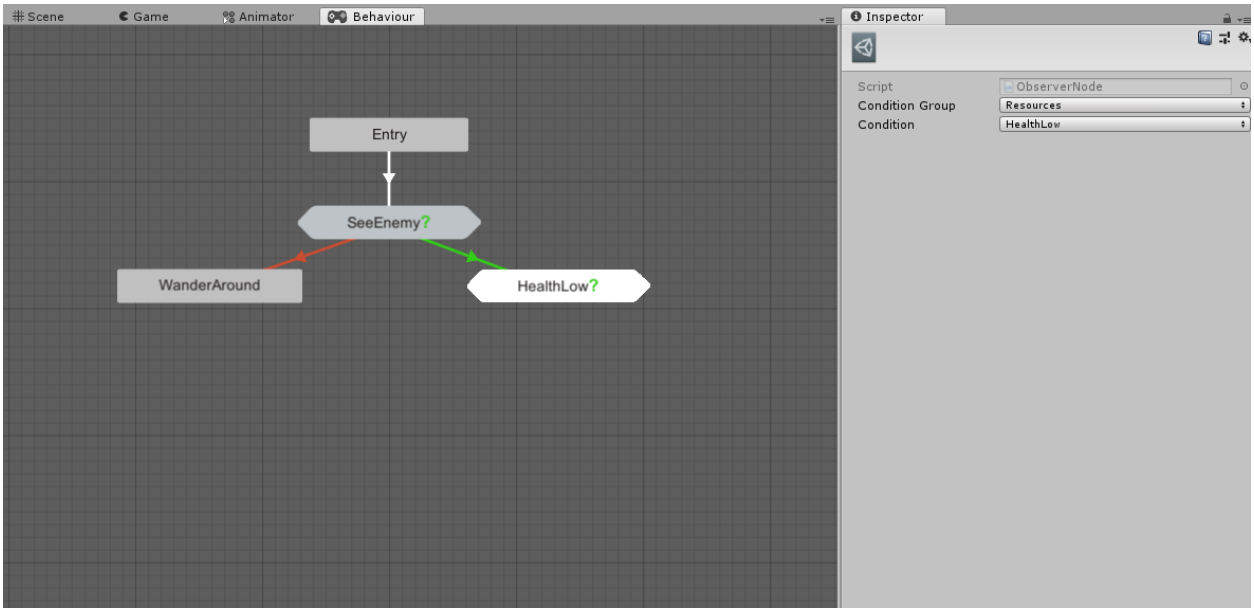


Now we need to define what is going to happen when Agent sees the enemy and when he does not.

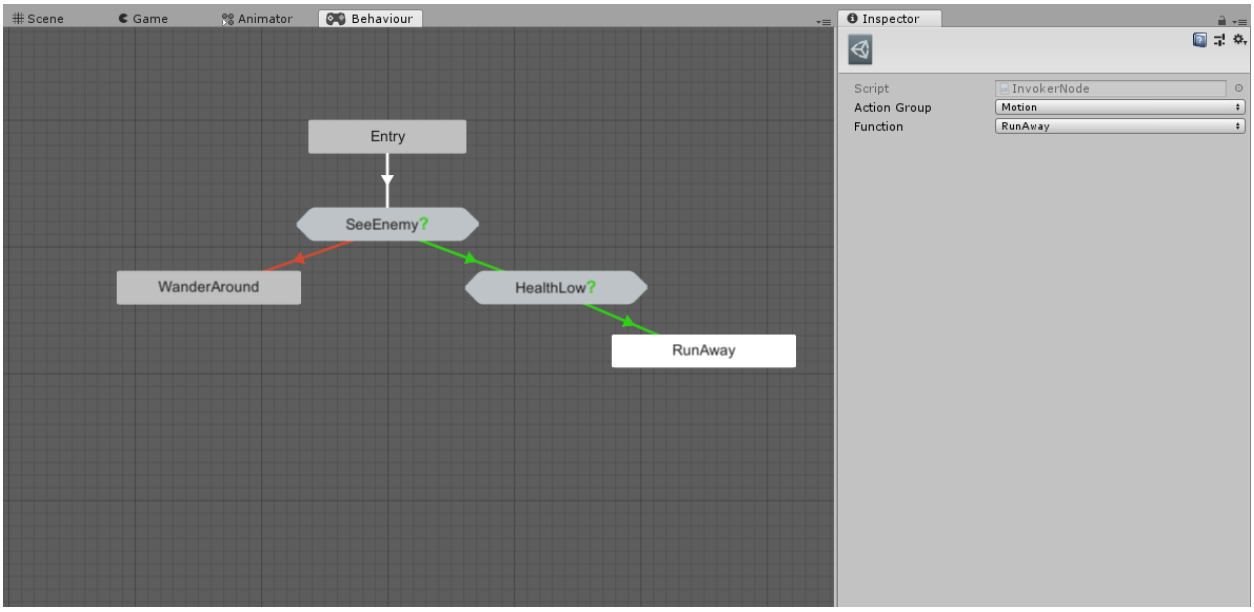
The second case is easier for this particular Agent, so we will define it first. Let’s think for a moment, what we want him to do when nothing is happening around? It might be Idling, but I think it will be more interesting to make him walk around. To do this, we need to create a new Invoker (use context menu or shortcut “**I**”) and set its **Action Group** to *Motion* and its **Function** to *WanderAround*. And, in order for it to get activated when **SeeEnemy?** is false, we need to create a “No” (red) Transition from Observer to Invoker (use Observer’s *Transition/No* context menu item or shortcut “**N**”).



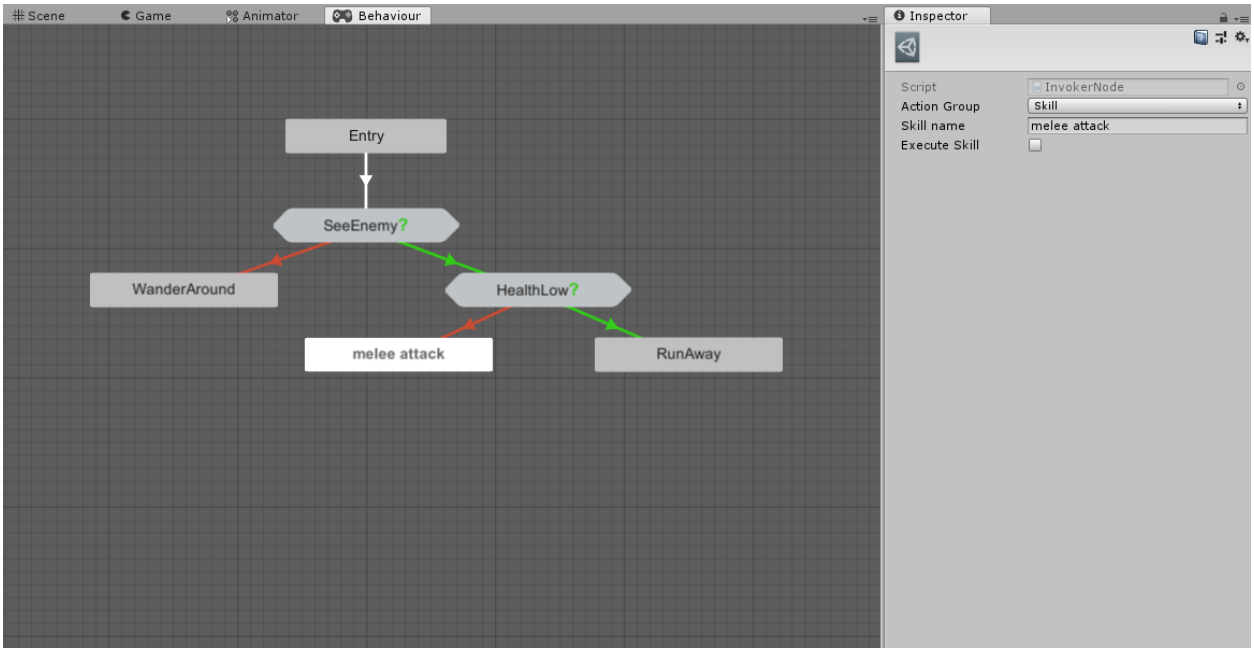
Now let's define what to do when he sees an enemy. We could just make him get close and attack, but I don't see any reasons not to make it a little more interesting. Let's make him also flee from the engagement when his health is "low" and engage and attack otherwise.
To do this we should check whether Agent's health is "low" and run away from target if it is so. This can be done by an Observer that has a **Condition Group** set to *Resources* and **Condition** - to *HealthLow*



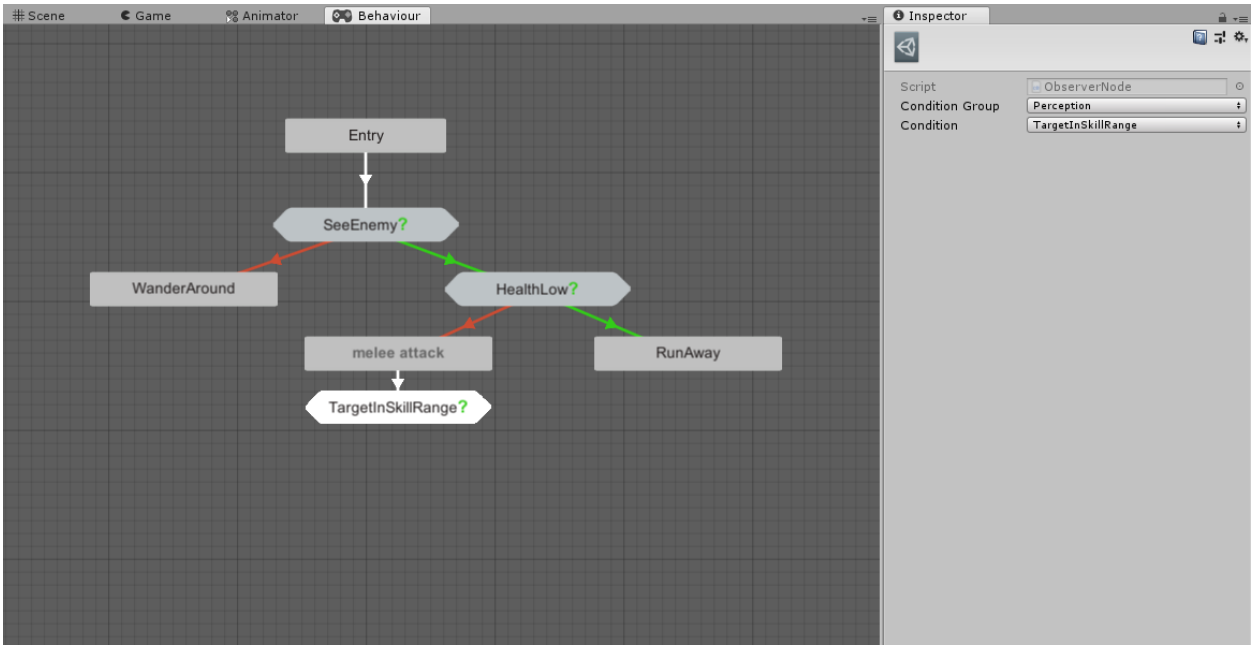
and an Invoker that has **Action Group** set to *Motion* and **Function** - to *RunAway*.



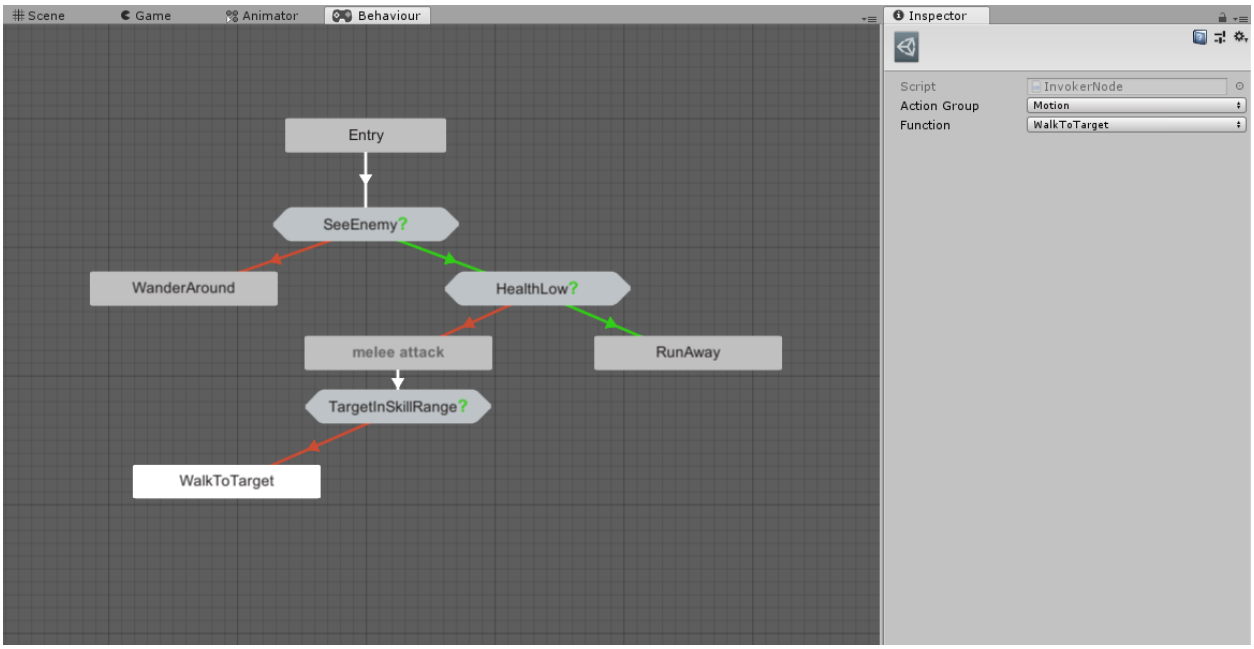
Now, in order to be able to check if target is close enough to an Agent for using a Skill, we will create Invoker, whose **Action Group** is *Skill*, but this Invoker will not **Execute Skill**. By unchecking the **Execute Skill** toggle, we'll make sure that a Skill with a **name** "melee attack" is set as a **CurrentSkill** of an Agent, so that we can retrieve different information about it, like the Skill's **Range**, which we are going to use next.



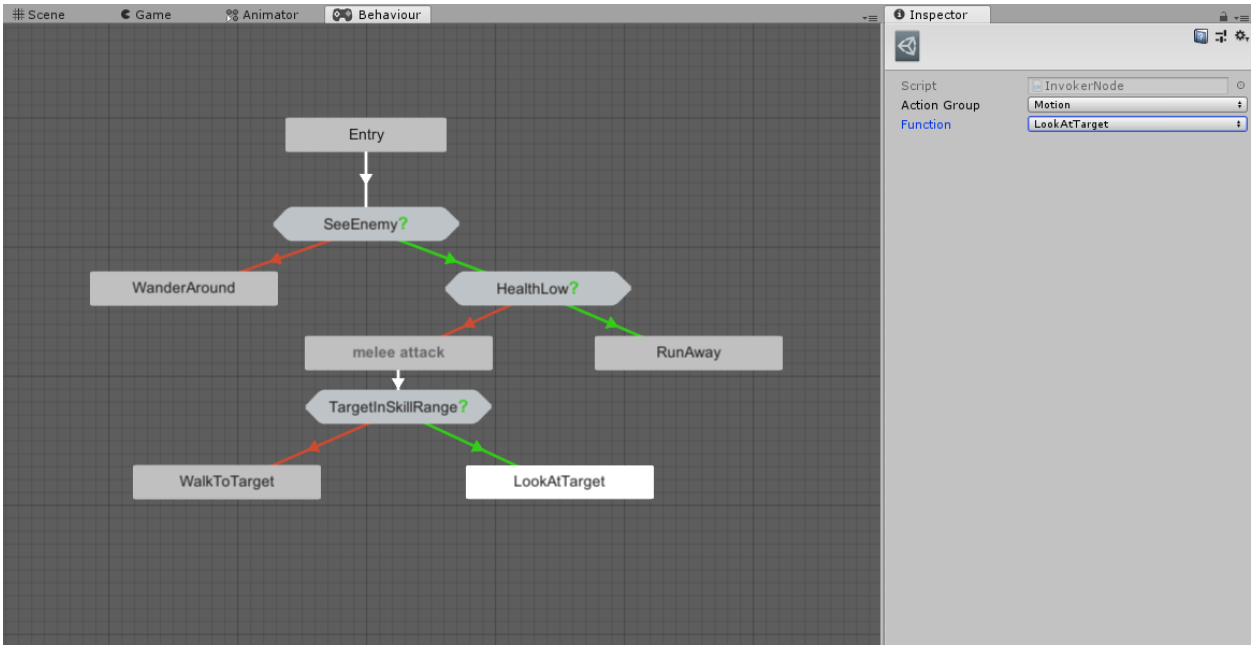
When Agent's health is not low and we have prepared Agent to use Skill that goes by the name "melee attack" (or just to check if we can use it), we want to check if Agent is close enough to its target. We can do so by checking a condition *Target In Skill Range* from *Perception* Condition Group. This will use a **Range** property defined in a Skill.



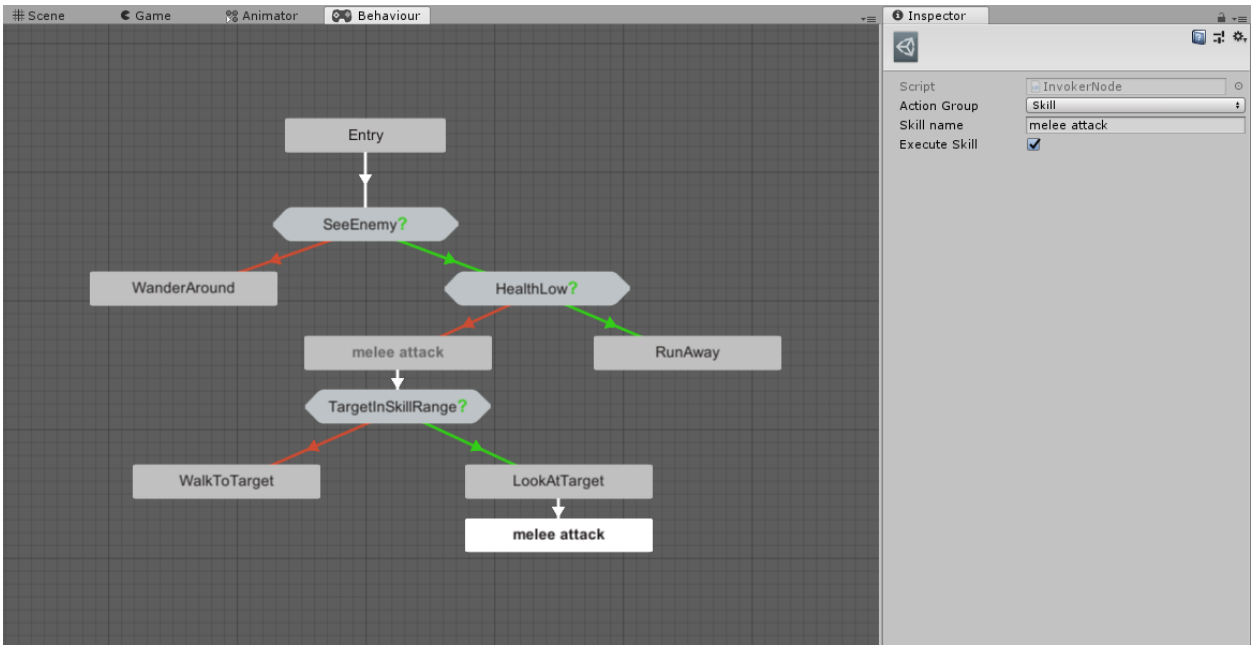
If target is not close enough, we want Agent to *WalkToTarget*.



When target is close enough, we want Agent to *LookAtTarget* (so that Agent doesn't try to kick air near the target instead of kicking an enemy, like a coward)...



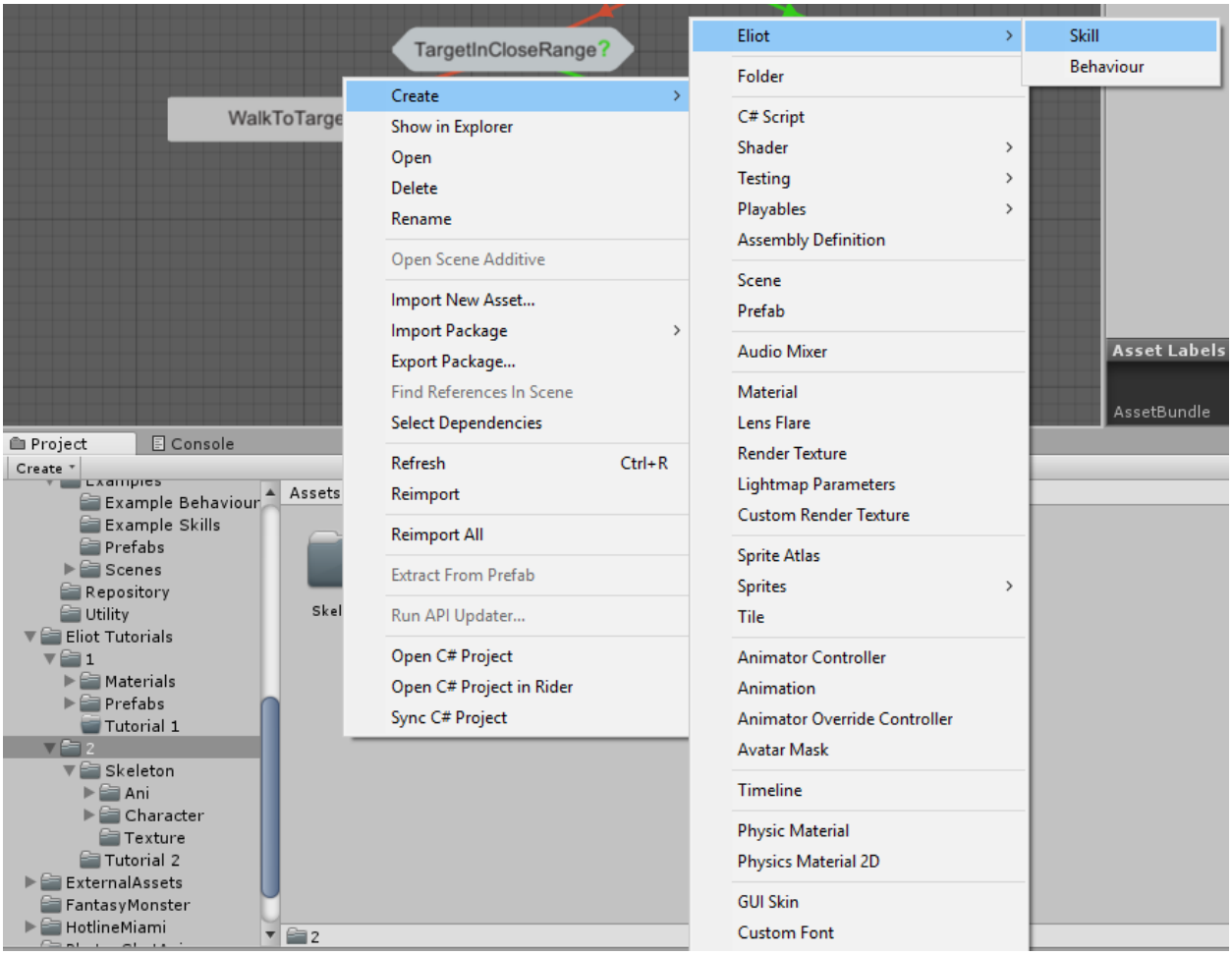
… and we want Agent to attack at the same time he faces his target. In order for Agent to attack, he needs to use a Skill. Let's refer to the Skill Agent will use for attack as "melee attack". We can invoke the Skill by its name using an Invoker that has its **Action Group** set to *Skill* and its **Skill name** to "*melee attack*" in this current case, because we will create a Skill with this name a little later.



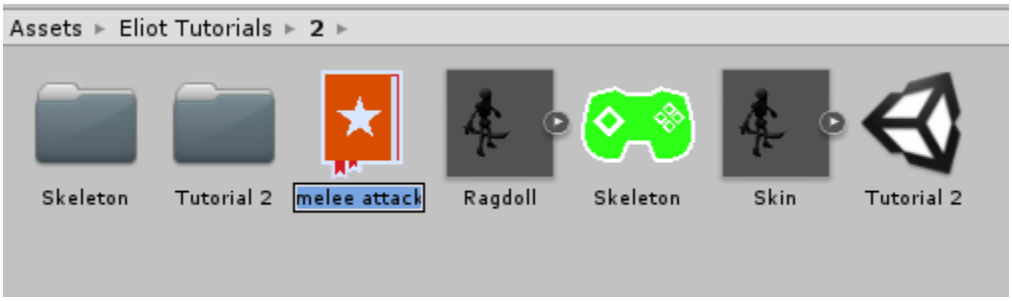
Alright, looks like this is what we need for the moment being. Time to create that **melee attack** Skill for our skeleton. Don't forget to save the progress before proceeding (**ctrl+s** or via context menu).

2.1.3 Create Skill

Skills, just like Behaviours, are encapsulated in files. So, we will start with creation of a new Eliot Skill file.

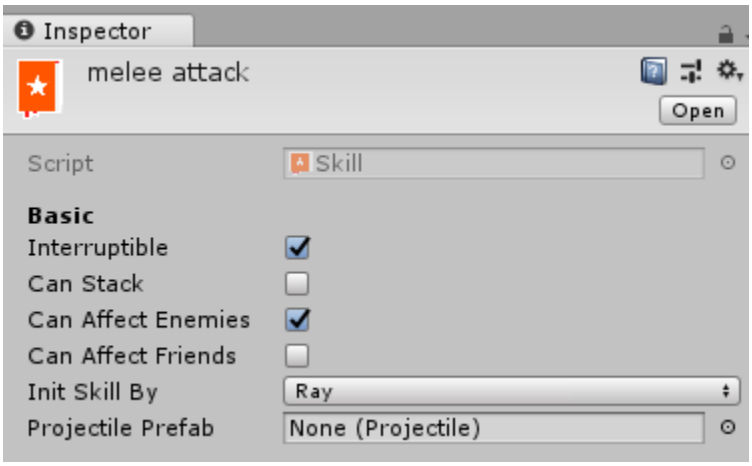


When we created the Behaviour, we decided that Agent is going to call a Skill that is named “melee attack”, so let’s use this as a name for our newly created Skill.



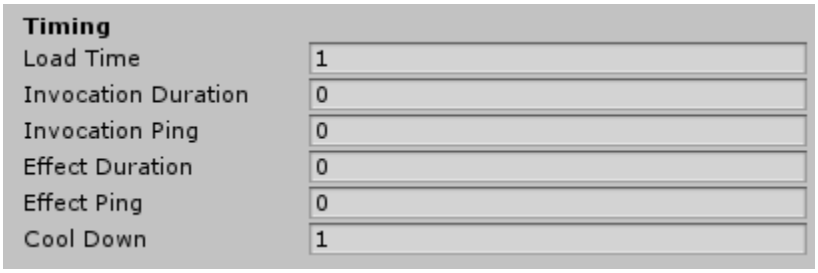
Now we want to set its settings up properly. If you look at Inspector while the file is selected, you’ll be able to see and change its properties. Let’s refresh what we want Agent to do with Skill. We want skeleton to be able attack on a short distance. It’s a one-time influence on target’s resources, with no duration.

In *Basic* group of properties there is one thing we might want to change in this particular case. If you don’t want Agents to be able to attack friendly units with this Skill, uncheck **Can Affect Friends**.

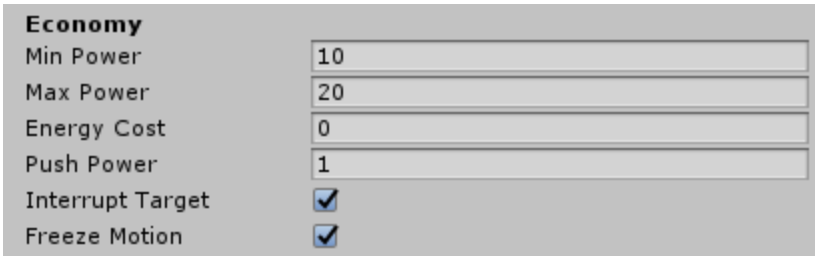


In *Area* section we want to set **Range** to something around 2-3. Behaviour that we just created uses this value for checking if the target is close enough.

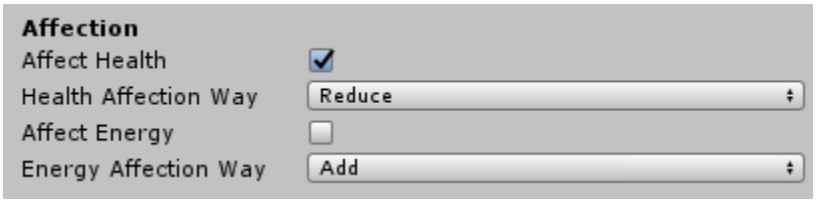
As there is no duration for this Skill, for this is neither a DoT nor a buff, we want only to delay the cast of a ray to align the moment of its instantiation with skeleton attack animation using **Load Time**, which works fine at the value of 1 for this case.
Also, skeleton should not be able to swing again immediately after the ray was cast, so let's set **Cool Down** to 1 as well.



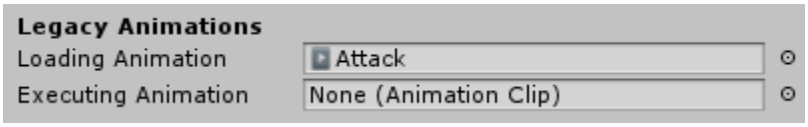
Before tweaking the values from *Economy* section, we want to decide the amount of health our Agents will possess. I think that skeletons should be able to kill each other with 5-10 hits. So, let's give them 100 health points for convenience (we'll do it at the stage of creation of the Agent), and make their hit power vary between 10 (**Min Power**) and 20 (**Max Power**).
It would also be cool for Agents to push each other with these hits, so, to do this, we can set **Push Power** to something more than 0. 1 works fine in this case. Remember, that the actual value that is used for pushing Agents around is this **Push Power** divided by Agent's weight.
A sword hit would definitely prevent its victim from finishing its swing, so let's set **Interrupt Target** to true.
Swords are heavy and our skeletons are not very physically strong (at least they seem so), which requires them to stand still and try to balance while they try to swing that sword, so, to facilitate this fact, we need to set **Freeze Motion** to true.



Of course, we want to affect target's health with this Skill, by reducing it. Being hit with a sword is no joke after all.



We want to be able to see the animation, unless this is a text game. You can find skeletons attack animation in "*../Eliot Tutorials/2/Skeleton/Ani"*. The *Attack* animation includes both loading (swinging) and execution (hit), so you can just specify **Loading Animation** clip.

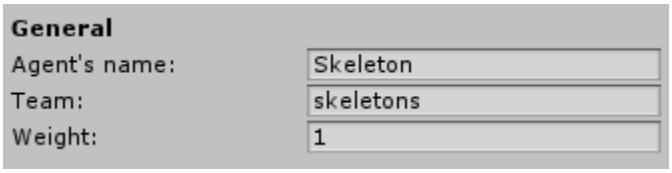


Now we are ready to make the final step - creation of the Agent itself.

2.1.4 Create Agent

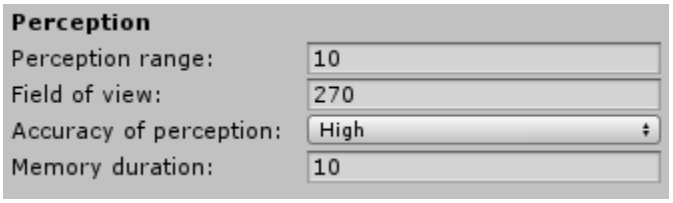
As always, we are going to use Unit Factory to create a new Agent (open it using *Eliot/Unit Factory* menu item).

I think, "Skeleton" is the perfect name for this Agent. Let's assign him to a team with some creative name, like "skeletons".

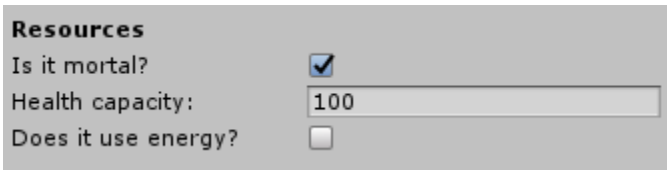


We want to widen his field of view a little, so that he can react to things that are not straight behind him, but a little further than just to the side. To do that, we need to set **Field of view** to bigger value, than 180. I think 270 degrees is good enough.
We're going to have just a handful of Agents here, so we can grant our Agents with a *High* **Accuracy of perception**, (or even higher, later, via Inspector).

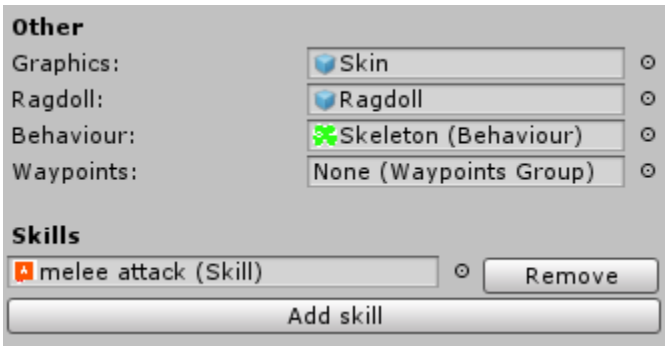
We probably don't want our Skeleton to think about his tough engagements for too long. This might be stressful for him (we probably don't want skeleton to look for his enemies, or run from them, for a minute straight). Having said that, we can set his **Memory duration** to 10 seconds.



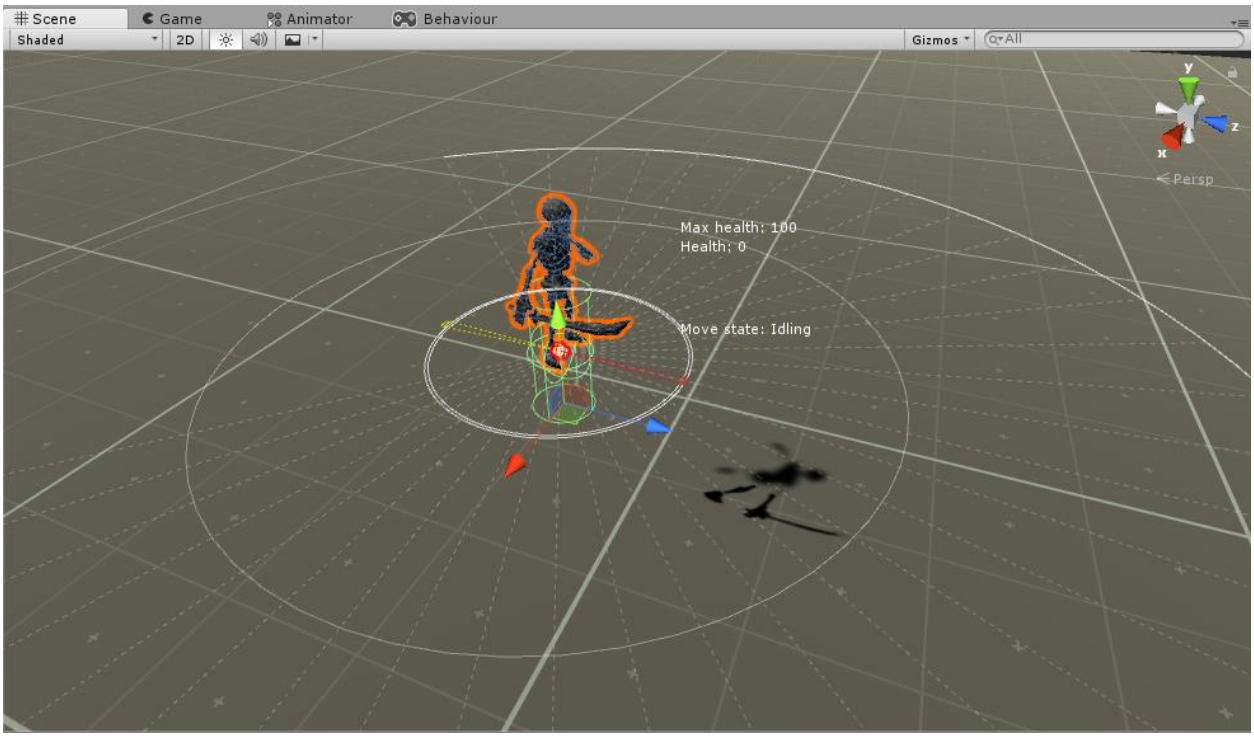
As we discussed earlier, health capacity of our skeleton is going to be 100. Moreover, it does not need to use energy.



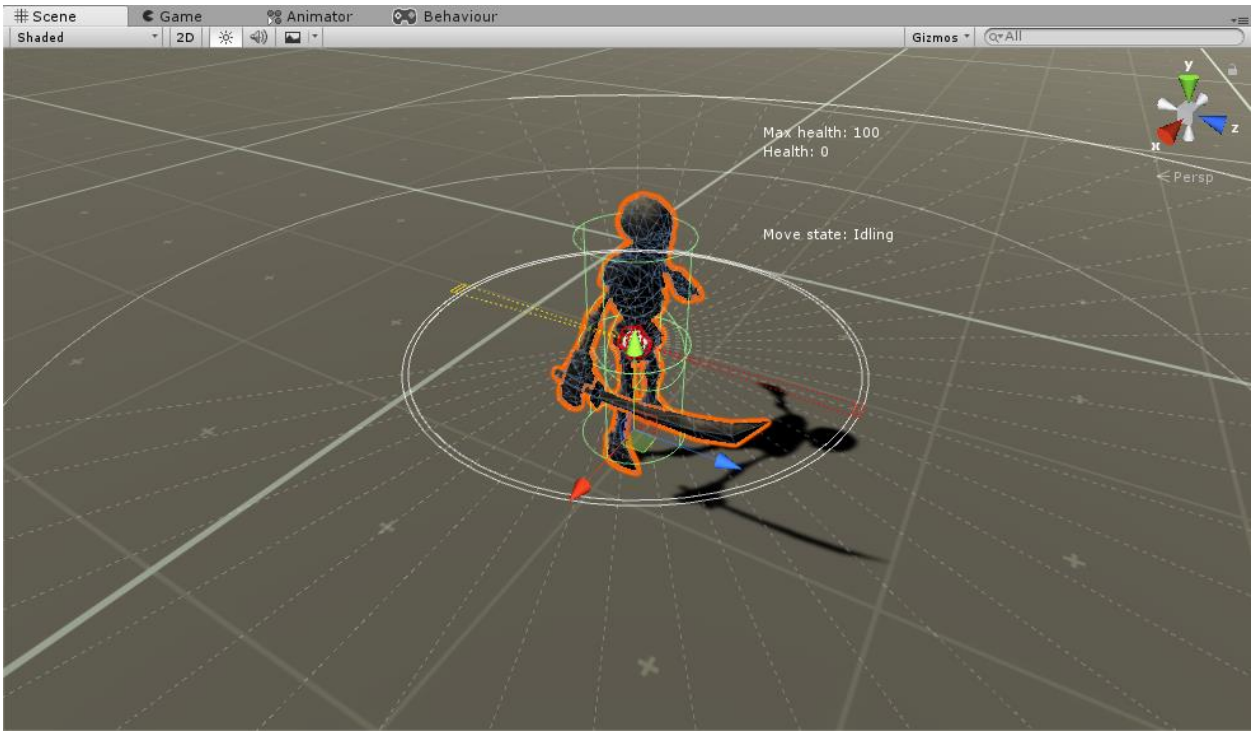
Now set Agent's **Graphics** and **Ragdoll** to prefabs that go under names of *Skin* and *Ragdoll* which you can find in "*../Eliot Tutorials/2/Prefabs"* folder.
Remember that Behaviour we created in 2.1.2? Drag and drop it into **Behaviour** field.
Considering our requirements, we don't need restriction of skeleton's "home" area or any kind of path, so this Agent doesn't require Waypoints, letting us leave this field empty.
Under "Skills" header there is a button "Add skill". Press it and you'll see a new field for a Skill file. Find the Skill that we created in 2.1.3 and drag and drop it into the field.



Now we are ready to press that "Create" button and close the Unit Factory.

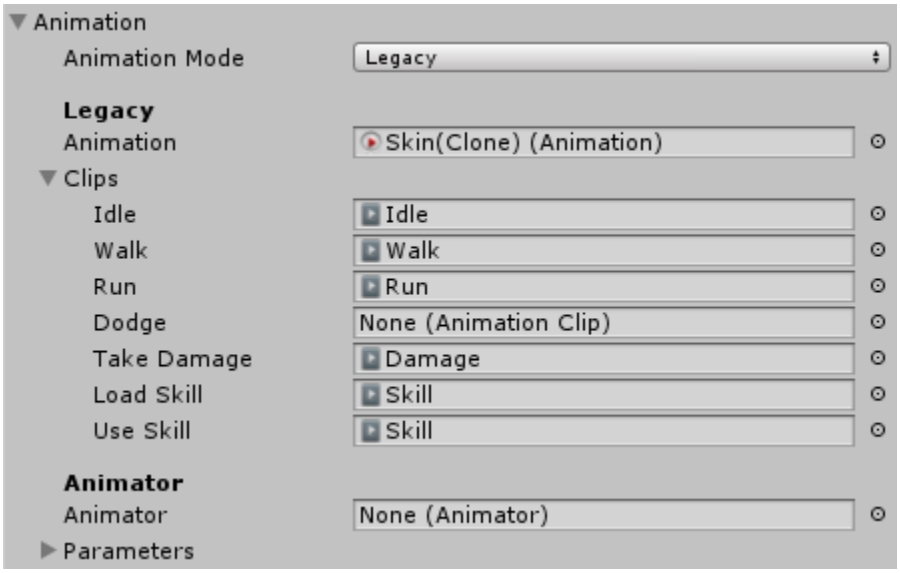


Just like the car graphics from previous tutorial, skeleton's graphics requires some alignment of its Y coordinate. Also you might want to scale the Skeleton graphics up so that it fits the collider size better if it does not by default.

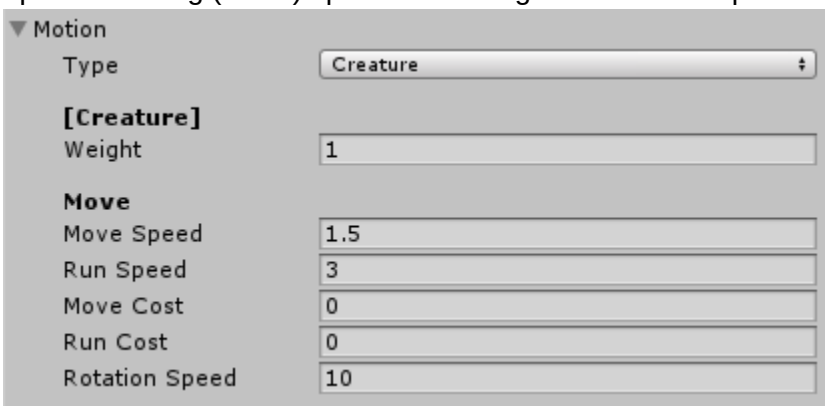


Yep, this looks much better.

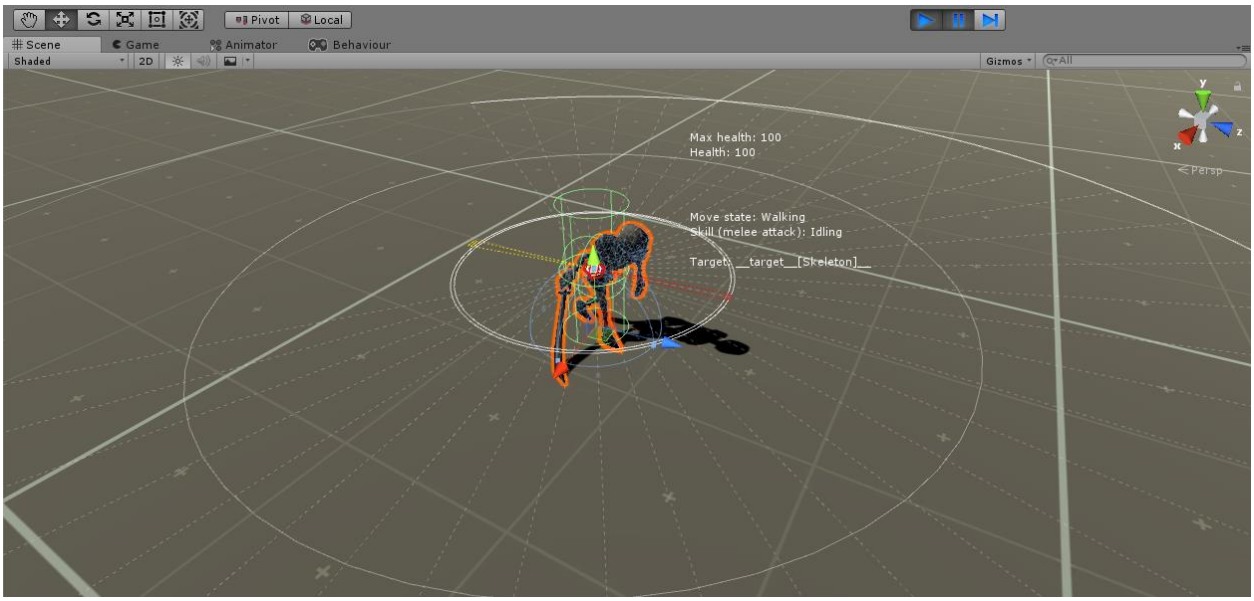
Now, if you enter the Game Mode, you are going to see that only the default animation is being played and that the Agent probably walks way too fast.
Let's fix it one at a time. Set **Agent.Animation.Animation** (Animation property of Animation component of Agent component. You can see it on the next picture) to a Skin(Clone) GameObject that is a child of Agent's __graphics__ container (which is Agent's child itself).
Now we need to set animation clips so that Agent knows what clip to play in certain states. You can find all the clips in "*../Eliot Tutorials/2/Skeleton/Ani"* folder. Drag and drop them so that your Agent's Inspector looks like the one shown in the screenshot.



We also want to somewhat reduce Agent's movement speed. Experiments have shown that optimal walking (Move) speed for this Agent is 1.5 and optimal running speed is 3.

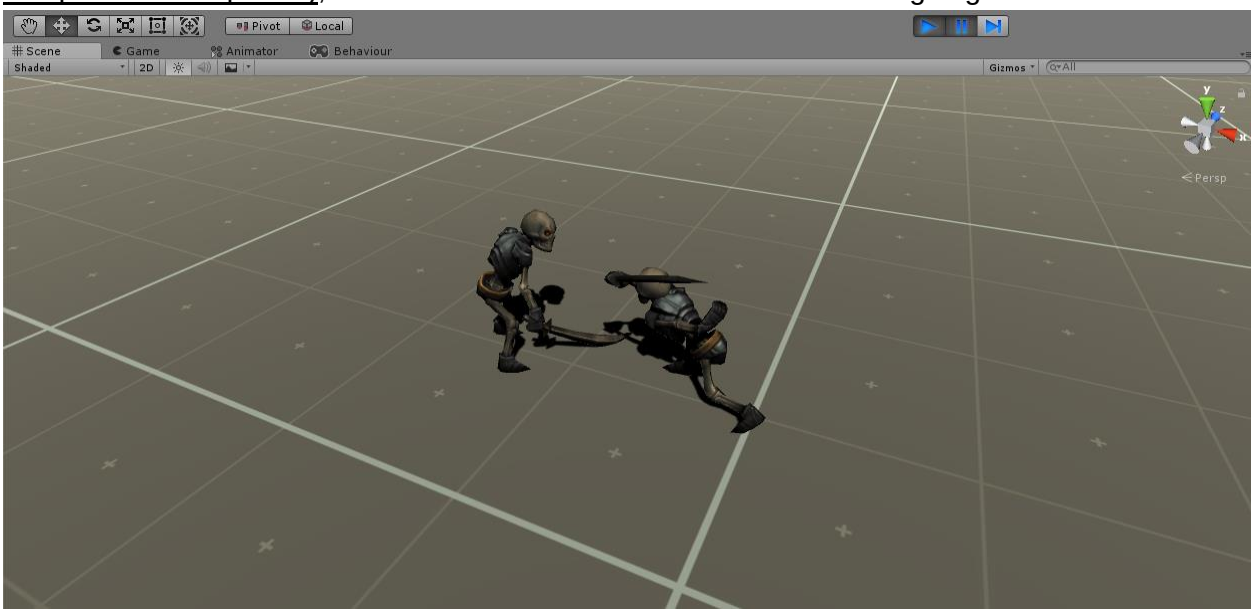


If you have done everything right, you should be able to enjoy skeleton's pointless attempts to change his position in space.



Make a prefab out of our Skeleton Agent by dragging and dropping it from Hierarchy window into "../Eliot Tutorials/2/Prefabs/" in the Project window. We are going to need it later.

If you create one more skeleton (by duplicating an original one) and change his team (in Unit component in Inspector), the whole hard work done in this exercise is going to reveal itself.

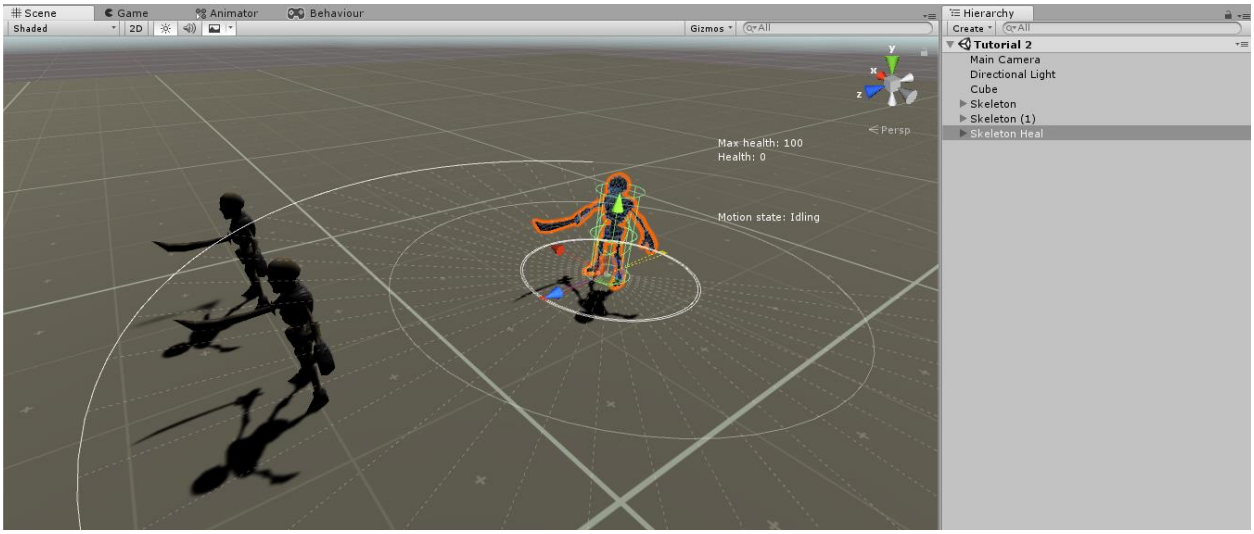


2.2 Healer Skeleton

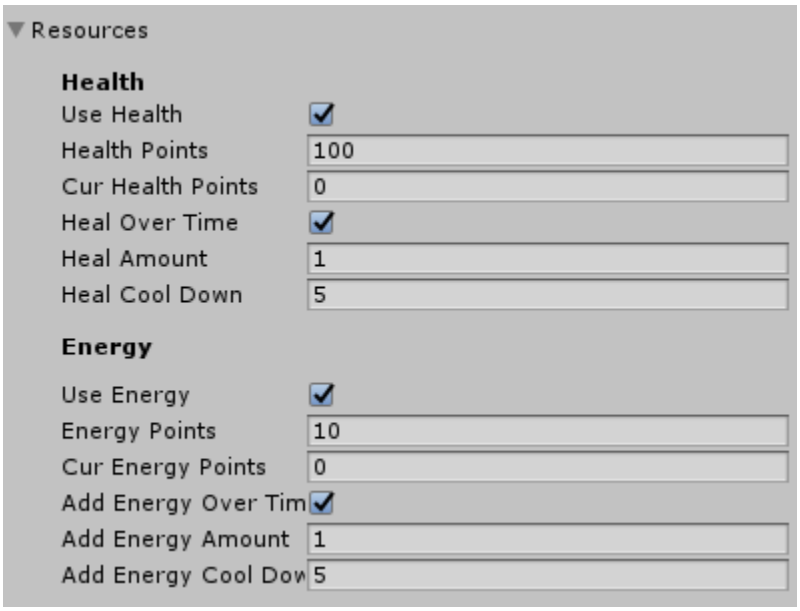
In this tutorial, we will create a Skeleton whose role is to support his friends by healing them. We will continue from what we have achieved in the previous tutorial.

2.2.1 Create Agent

In this case, since only Behaviour and Skills of our healer skeleton will be different from his more aggressive partner, we can just make a copy of the Skeleton that is already created and proceed from there. Let's call this copy "Skeleton Heal".

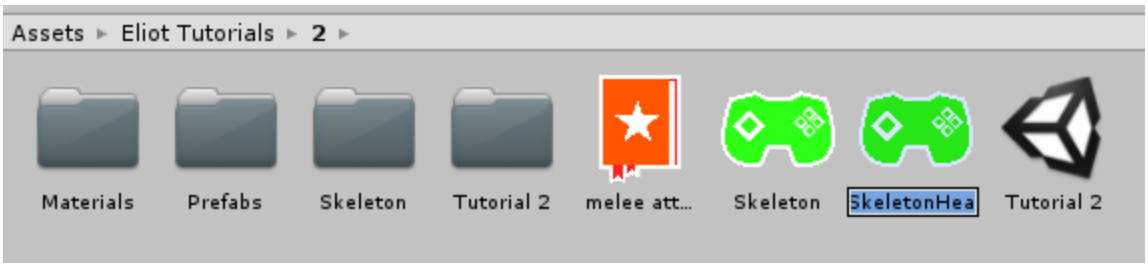


Since this Agent is going to use magic spell to heal his friends, it would be reasonable to make him require usage of energy. To enable it, go to Skeleton Heal's Inspector and in Resources section of Agent component check **Use Energy** toggle.

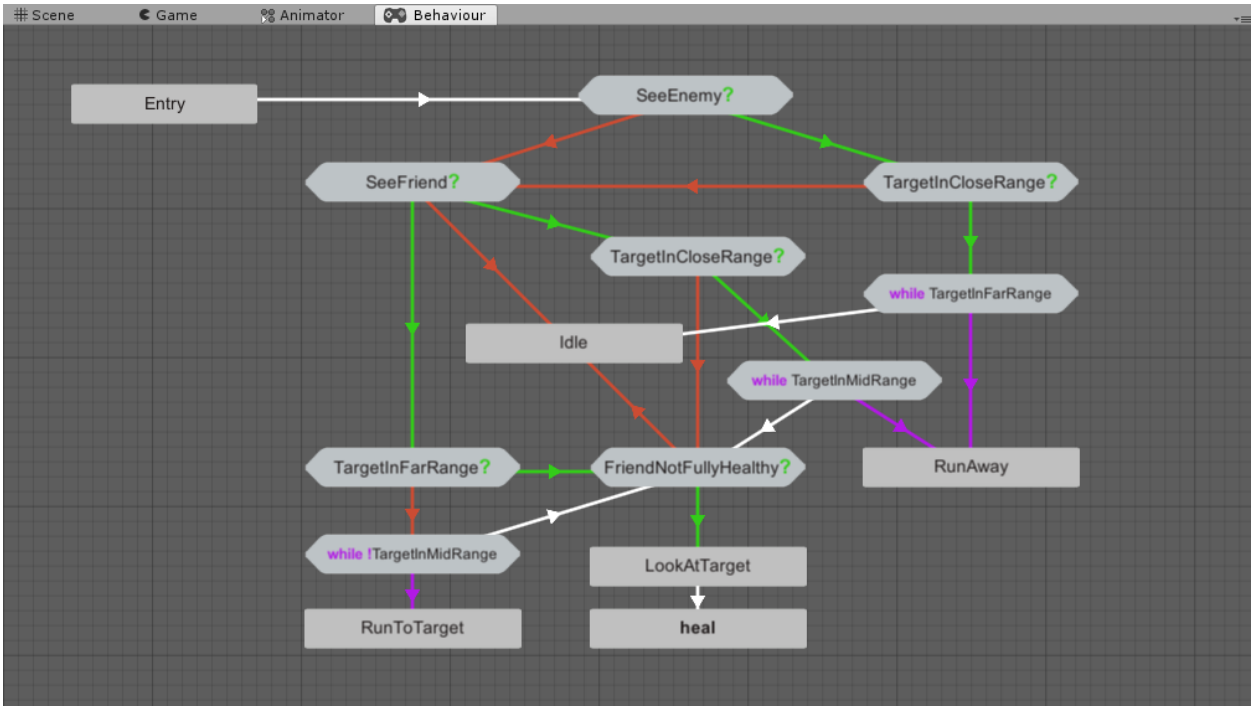


2.2.2 Create Behaviour

Create a new Eliot Behaviour file anywhere you feel comfortable in the Project. Since this will be a Behaviour model for our "Skeleton Heal" Agent, that is a perfect name for Behaviour as well.

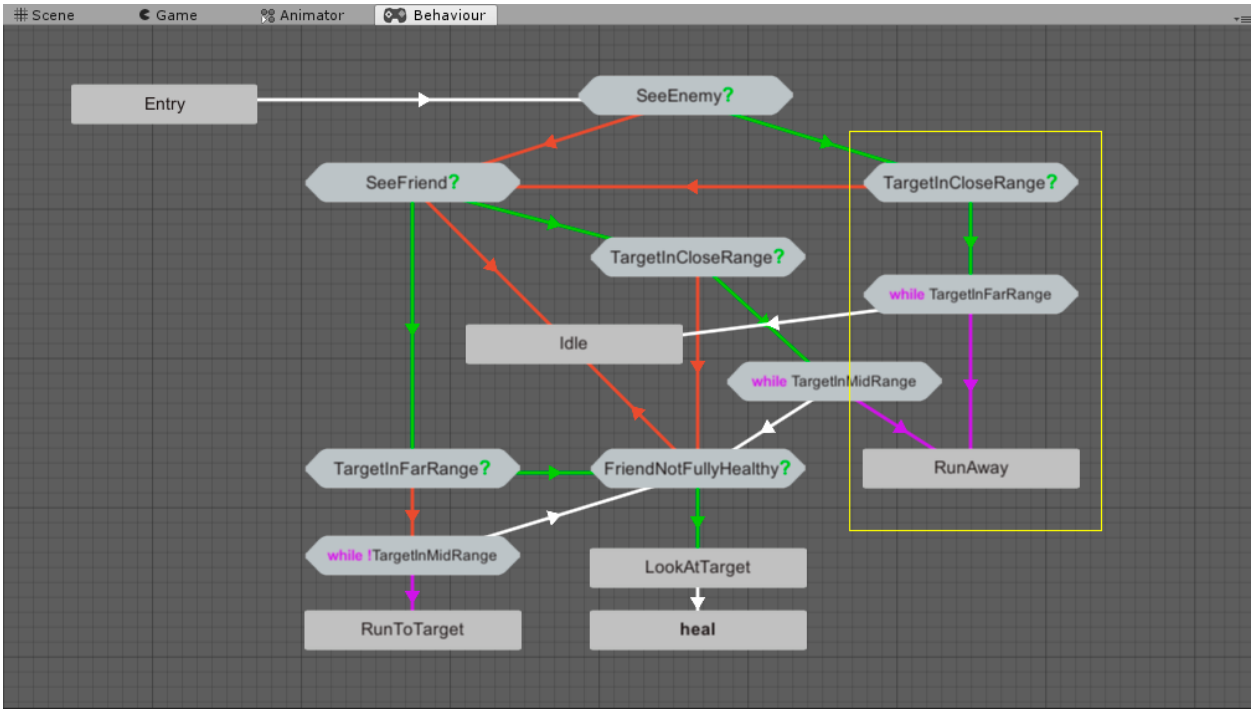


Drag and drop it onto Behavior Editor window to start editing it. Here I will show you a complete Behaviour and explain that parts of it, which you might not yet understand. Hopefully, at this point you are ready to create a new Behaviour by copying it from the picture.
The part that might be not clear yet that's related to editing a Behaviour is creation of Loops (rhombuses with purple **while** in them). You can either do this by choosing *Add Node/Loop* context menu item, or just use shortcut "**L**".
Loops work is a similar way to Observers, but while the condition that they check is true, Agents don't get any instructions except from those, which are connected to the Loop with a purple arrow. They replace Entry in the model for that time. When the condition, checked by a Loop is false, Entry becomes the beginning of an algorithm again and the nodes that are connected to a Loop with a white arrow are activated.

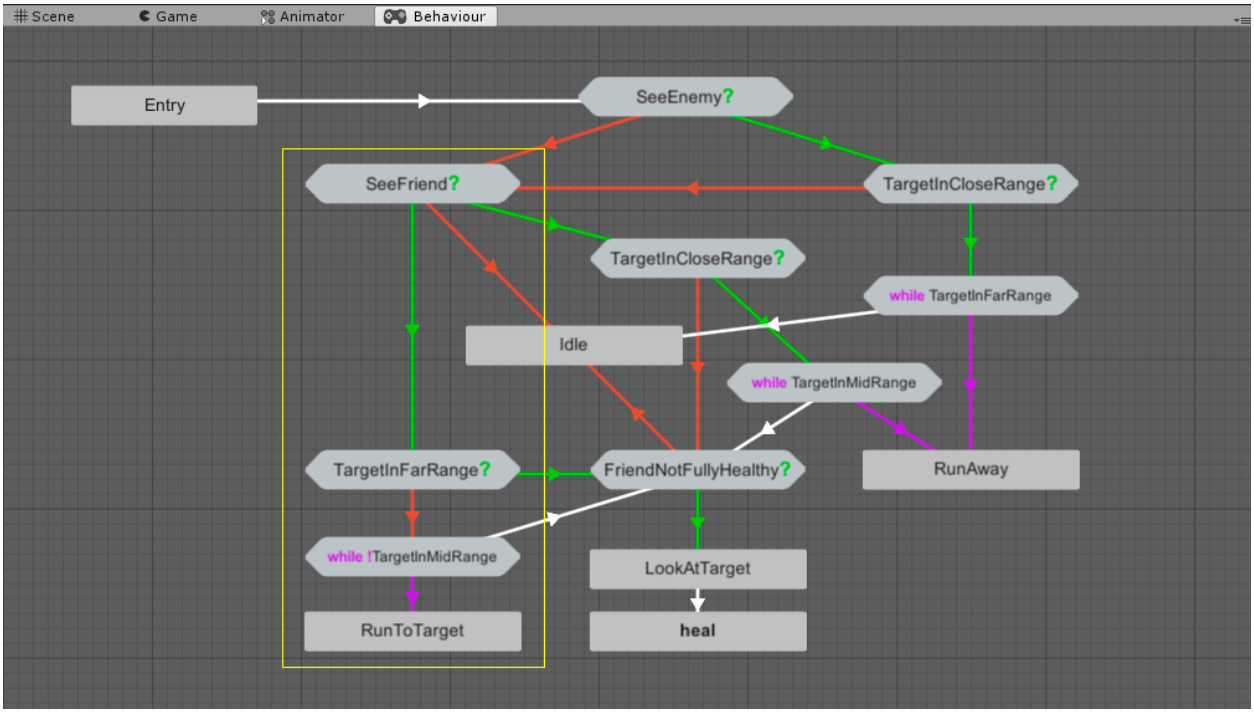


Here I highlighted three nodes, which start to get activated when an Agent can see an enemy unit. In this case, Enemy unit becomes Agent's target. We check if this target is close to an Agent. In case it is indeed close, we want to make sure that our healer is safe, so we make him run away from that threat until it is far enough. Note that while an enemy is closer than the distance that we defined as a "Far Distance" in Agent's GeneralSettings, it will do nothing

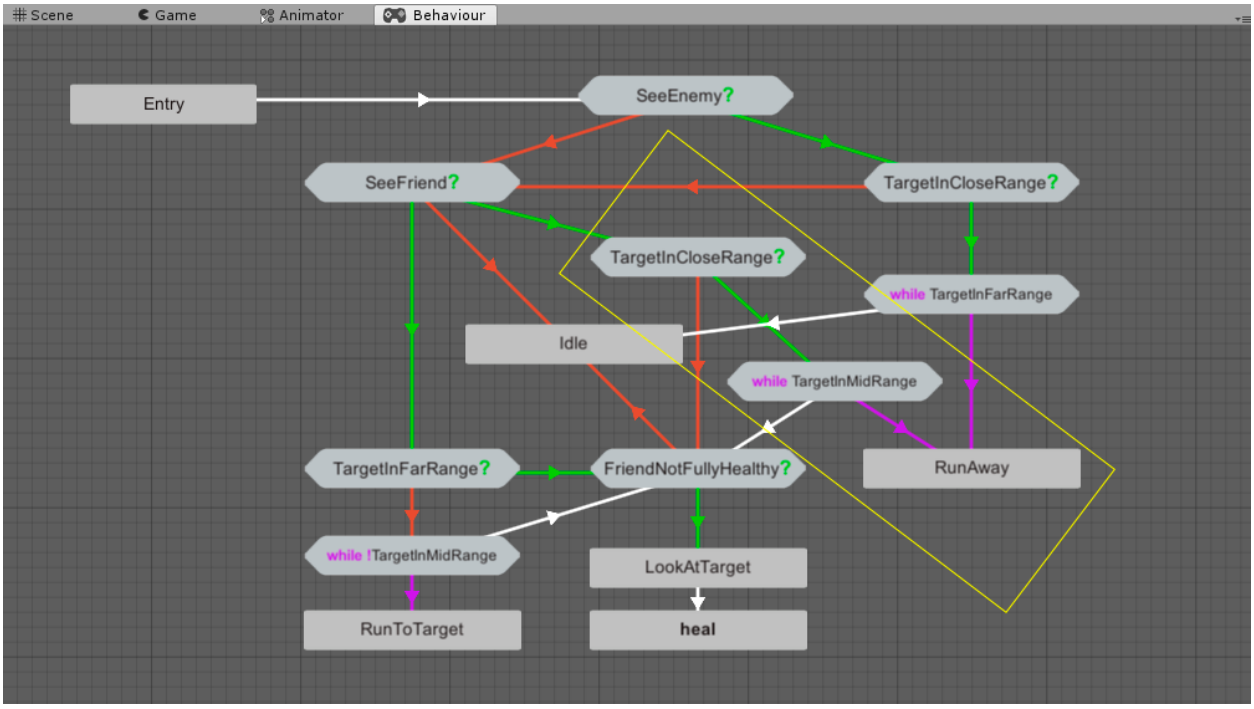
except from running from its target. Once it’s far enough, Agent Idles and starts the algorithm from Entry again.



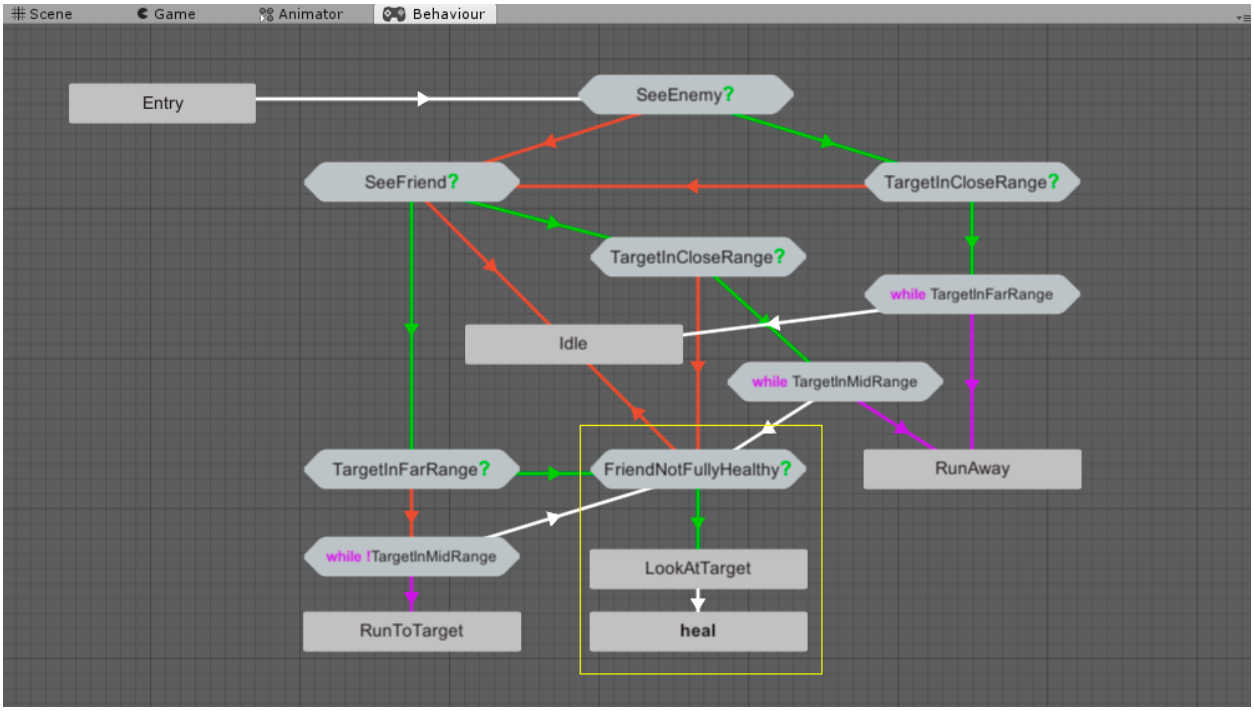
When an Agent can see a friendly unit, the actions he takes are similar, but almost reversed. Agent tries to come close (on the distance that is defined as “Mid Distance”) to his friend in case that friend leaves a “Far Distance” zone.



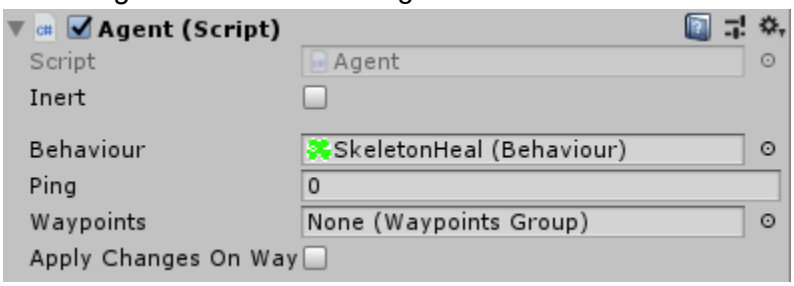
But when that friend invades personal space of an Agent, he makes sure to get his oxygen back by running away just a little.



When a friend is at a respectful distance and needs help, Agent makes sure to use a Skill on his friend that is called “**heal**”. We will define the content of that Skill in the next part of the tutorial.

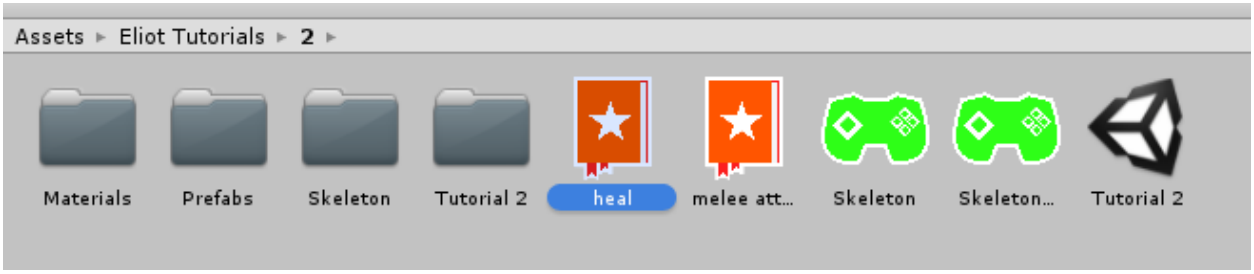


Don’t forget to save it and assign it as a Behaviour of Skeleton Heal Agent.

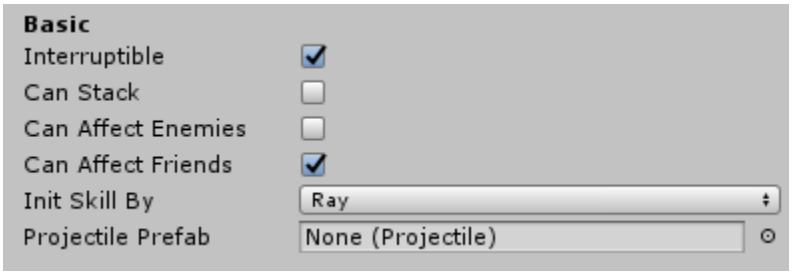


2.2.3 Create Skill

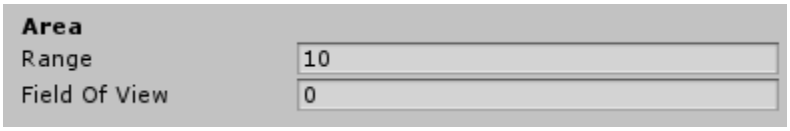
If you paid enough attention to the Behaviour we created in this tutorial, the Skill that will be used by our healer skeleton should be named “heal”. So let’s create a new Skill in our Project and start editing it.



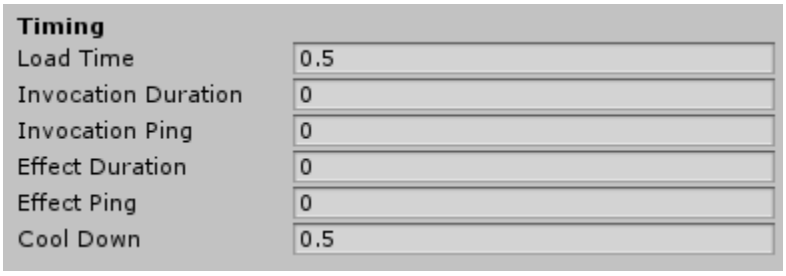
It should be possible to prevent an Agent from finishing this Skill by hitting him with something hard enough, so **Interruptible** is set to true.
We don’t want to accidentally heal our enemy, so the Skill can only affect friends (the Skill still can be cast on enemies, but it won’t do anything to them).
Also we want to initialize skill by Ray so that an Agent needs to see who he is about to heal before he actually wastes energy for that.



Since this Skill is pretty much a magic spell, a **Range** of 10 meters should do its job. We’ll leave **Field Of View** at 0, as this will make Skill use a value, defined in Agent’s GeneralSettings (**Aim Field Of View**).

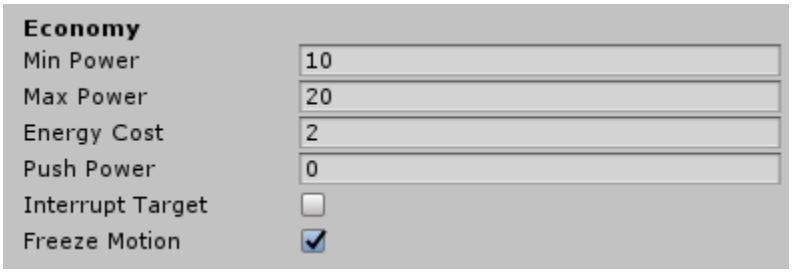


With the configuration of timing, shown in the next screenshot, the Skill will be applied once, Agent will have to stand still and not attack for 0.5 seconds to load the Skill and will have to wait for another 0.5 seconds to be able to use the Skill again. I encourage you to have a small experiment and play with all of the values in a timing group to see what each of them does.

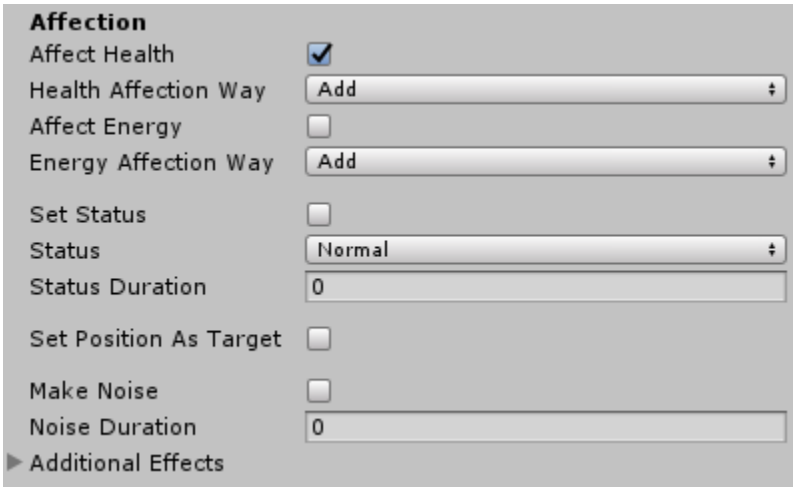


The power of the Skill is up to you, really, just see what values are working best at any given situation.

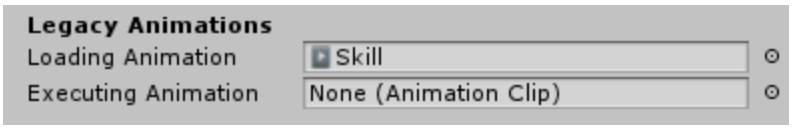
Energy Cost is the amount of energy points it takes Agent to use the Skill. If our Agent has 10 energy and **Energy Cost** is 2, then he will be able to use the Skill 5 times in a row, but that also depends on the rate of Agent's energy regeneration.



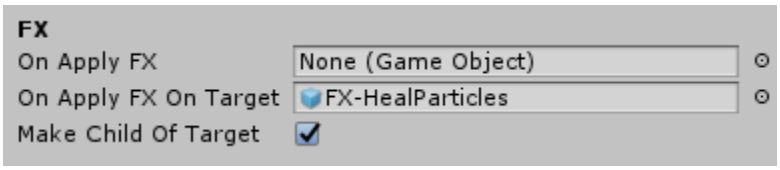
We don't want the Skill to affect its targets in any way except from healing, so we'll just make sure **Affect Health** is checked and the **Health Affection Way** is *Add*.



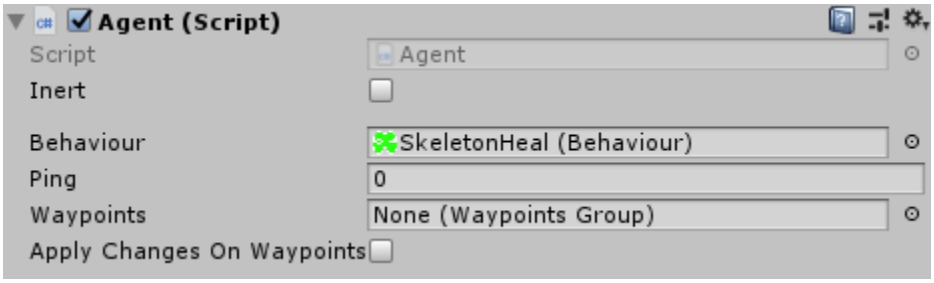
The most appropriate Animation Clip from available ones is the one that is names "Skill". It can be found in "*../Eliot Tutorials/2/Skeleton/Ani"* folder.



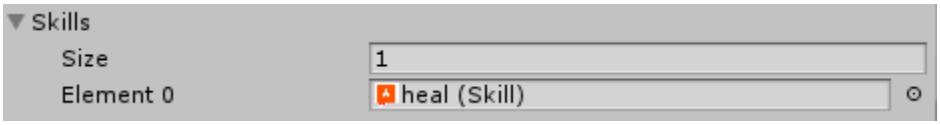
If we assign a prefab to **On Apply FX On Target**, it will be instantiated at target's position when the effect of the Skill takes place.



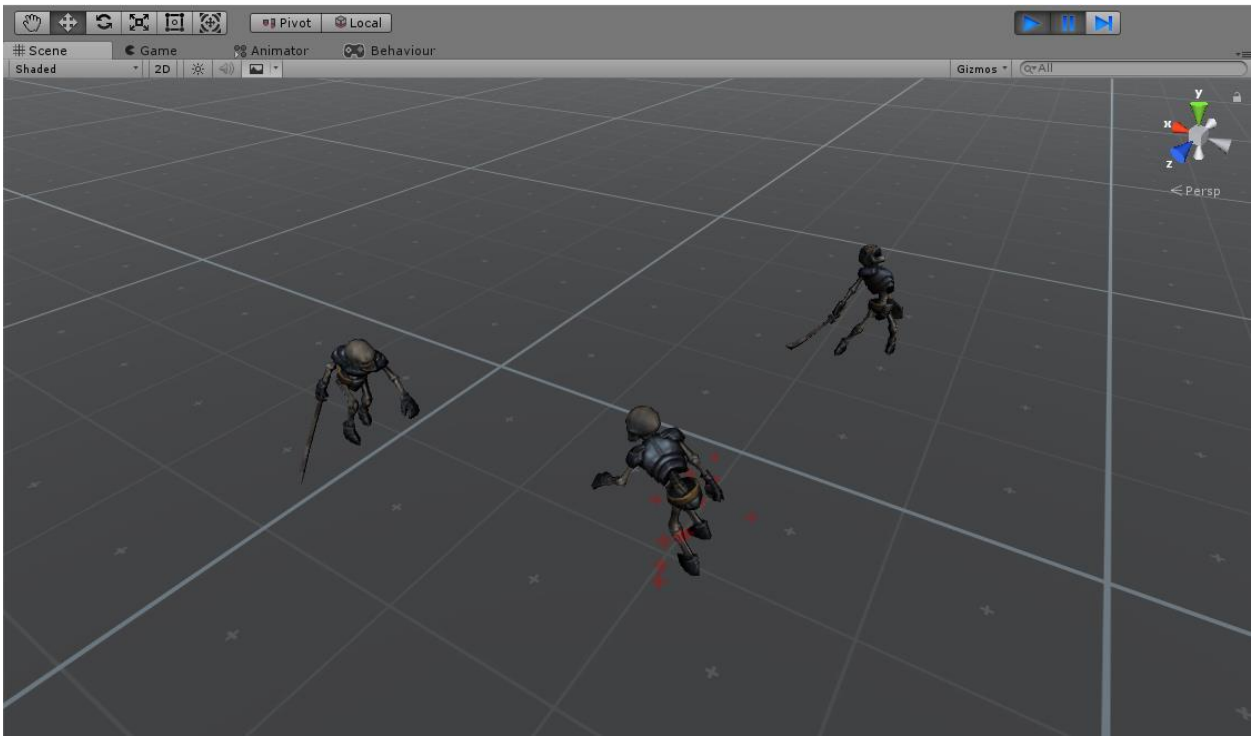
That's it. Once we assign created behaviour to Skeleton Heal's Behaviour field in Agent component...



... and add created Skill to the list of Skills of the Agent...



..., we are finally ready to test and admire our creation.

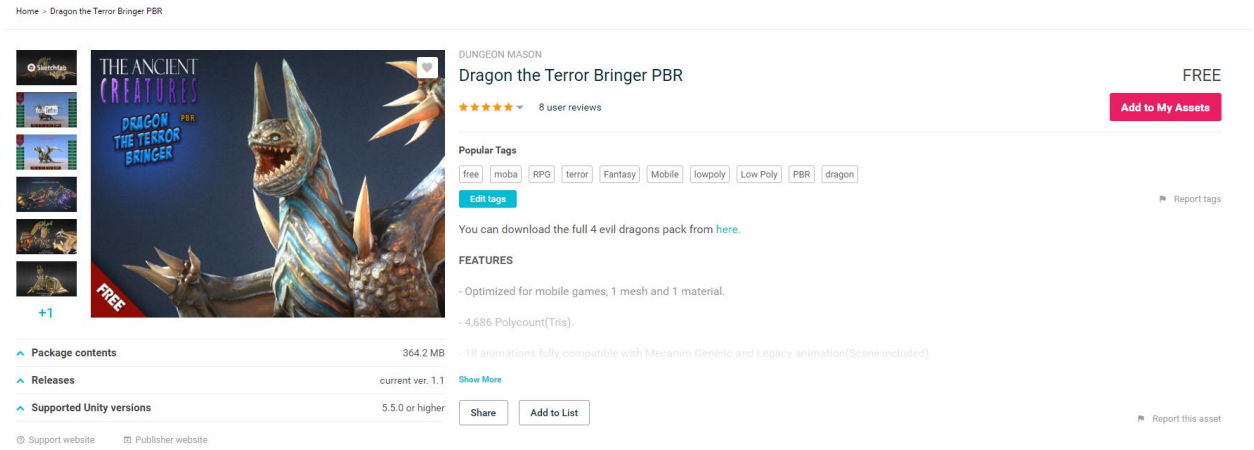


2.3 Dragon

In this tutorial, we are going to create a huge fire-breathing Dragon, whom Skeletons will try to kill for loot.

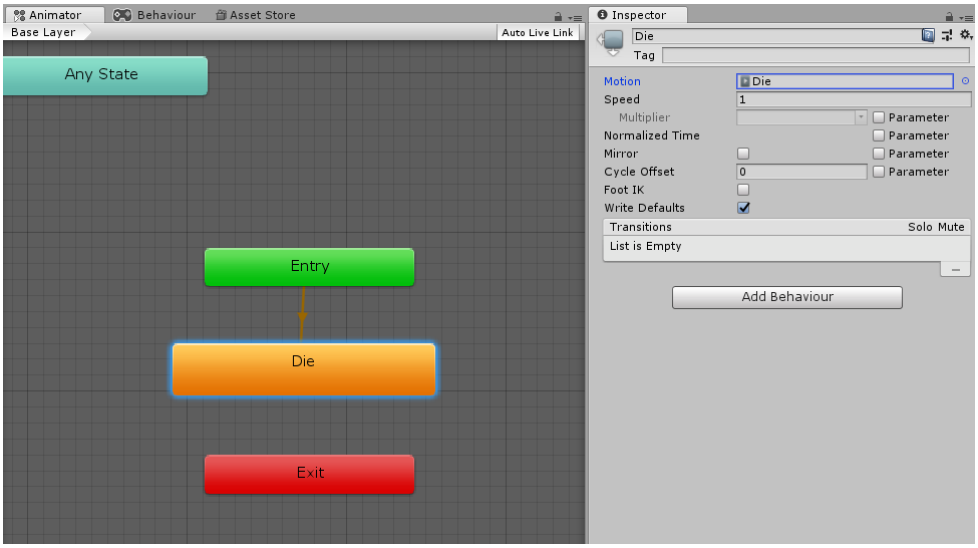
2.3.1 Get prefabs ready

For this tutorial, we will use animated model of a Dragon, which uses Unity Mecanim to control animations. You can get this asset for free from Asset Store here: https://assetstore.unity.com/packages/3d/characters/creatures/dragon-the-terror-bringer-pbr-77121



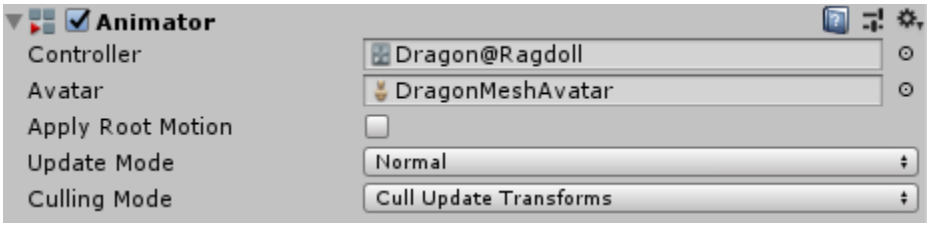
Download and import the package. Rename the root folder of the package to “Dragon” and move it all to “*../Eliot Tutorials/2*”.

Now we are going to create a dragon ragdoll. Let's start with configuring Animator Controllers. In folder "*../Eliot Tutorials/2/Animator Controllers*" you can find Animator Controller named "*Dragon@Ragdoll"*. It is configured so that it plays animation of death by default. You just need to select *Die* node and assign proper animation clip to it in the Inspector. You can find animation clips in "*../Eliot Tutorials/2/Dragon/Animations*".



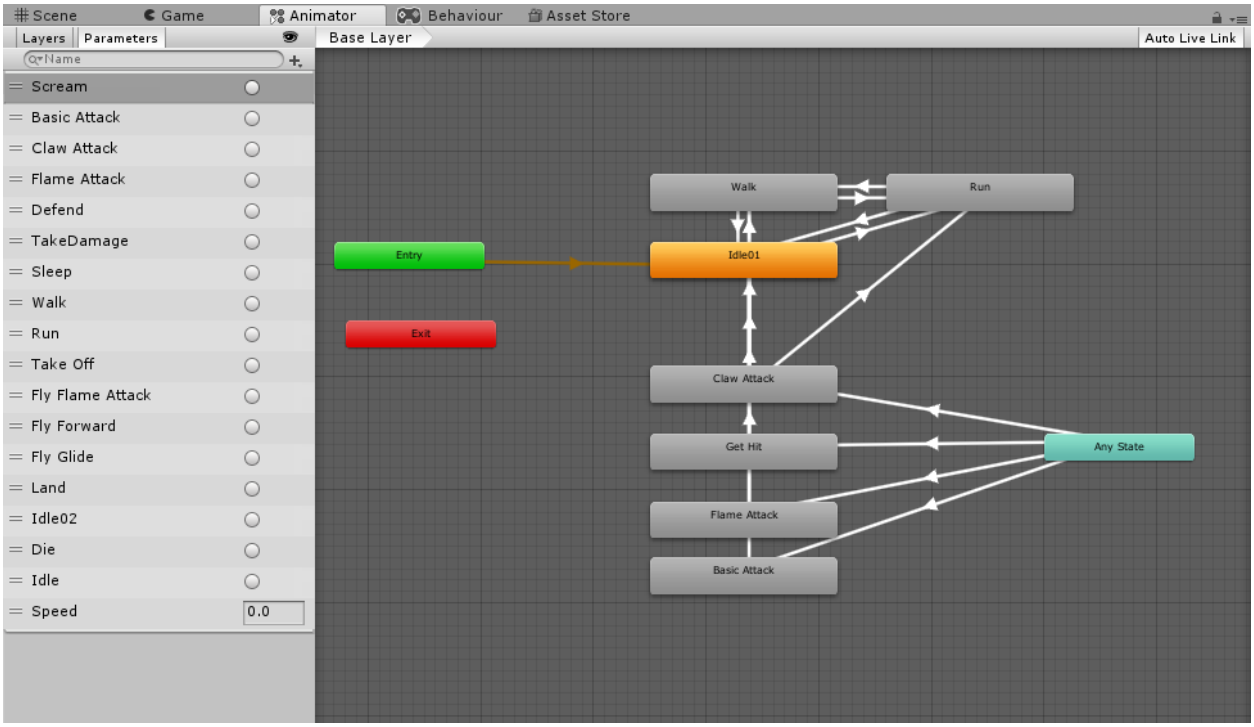
You can find more about Animator Controllers in Unity Manual: https://docs.unity3d.com/Manual/Animator.html

Now go to folder "*../Eliot Tutorials/2/Dragon/Prefabs*". Drag and drop any of existing prefabs onto the scene and name the newly created GameObject "Dragon@Ragdoll". Set the Animator Controller that we just created as a Controller of Animator component in GameObject's Inspector.



Drag and drop it back to "*../Eliot Tutorials/2/Dragon/Prefabs*" to create a prefab that is going to be used as dragon's ragdoll. As soon as Dragon is saved as a prefab, it is safe to delete it from the scene.

In the folder "*../Eliot Tutorials/2/Animator Controllers*" there is an Animator Controller named "Dragon" that is going to be used to control Dragon's animations.



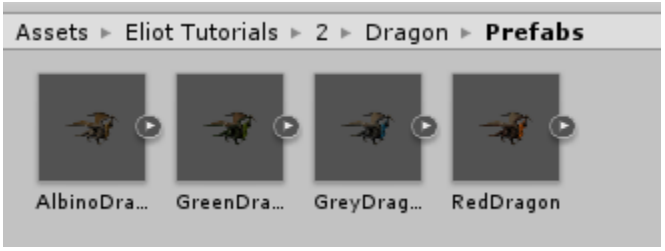
Note that it uses float Speed to make transitions between idling, walking and running and it uses Trigger messages to trigger different Animator States. These trigger messages will be used by Eliot Skills.

Assign this file as Controller of Animator Components of all dragons in "*../Eliot Tutorials/2/Dragon/Prefabs*" except our Ragdoll.

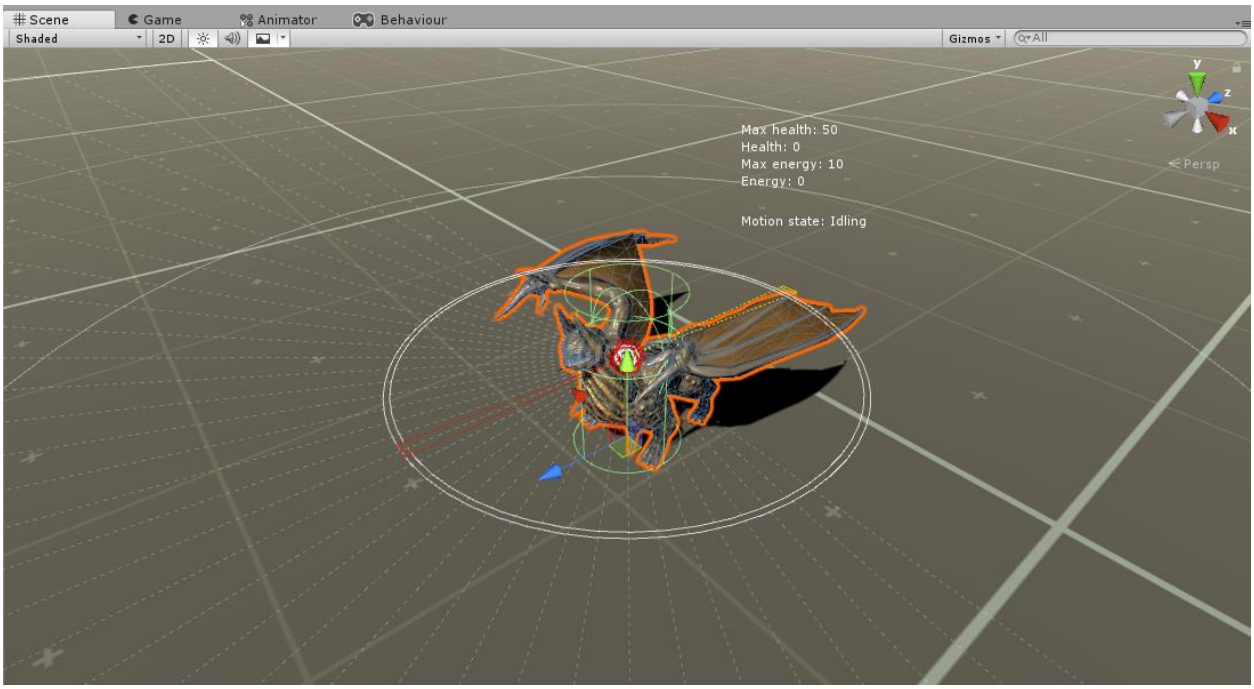
Use Animation Clips from "*../Eliot Tutorials/2/Dragon/Animations*" to configure nodes in the Dragon Animator Controller. It is necessary because Dragons are newly imported so we will have to link Animator nodes to proper clips by hand.

2.3.2 Prepare Agent

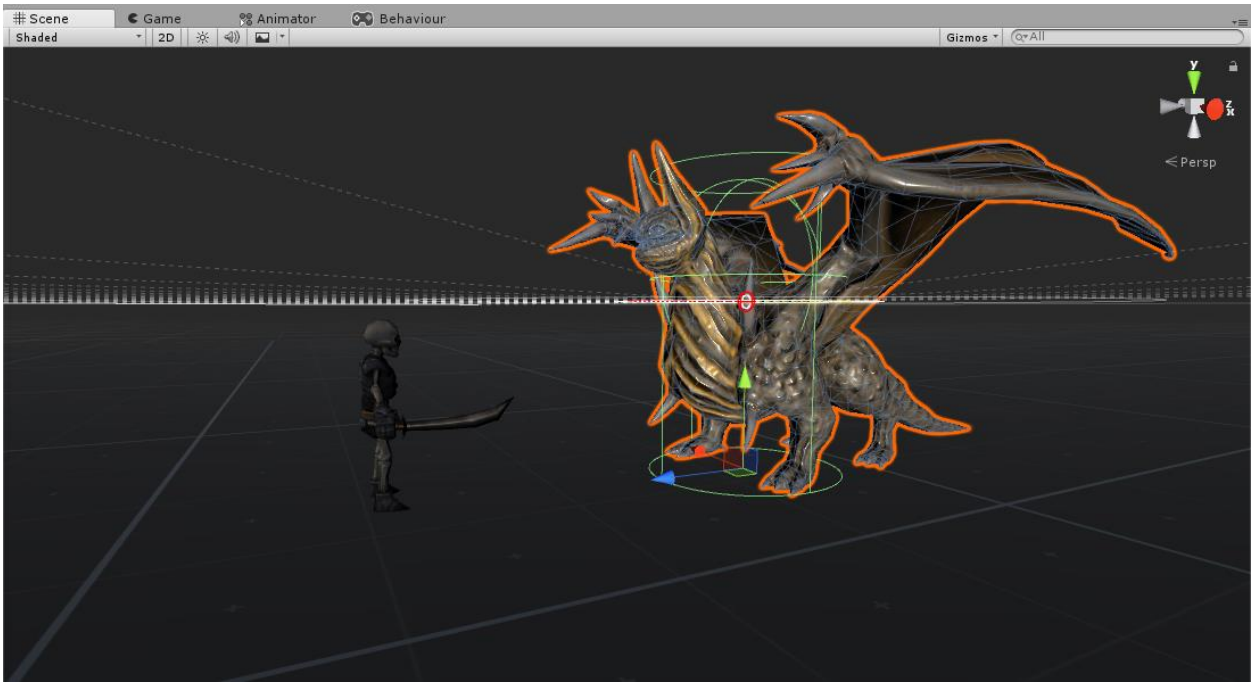
In "*../Eliot Tutorials/2/Dragon/Prefabs"* folder you are going to find prefabs that will be used as our Agent's graphics. Note that these ones use Mecanim for animations and not Legacy system as previously used graphics prefabs did.



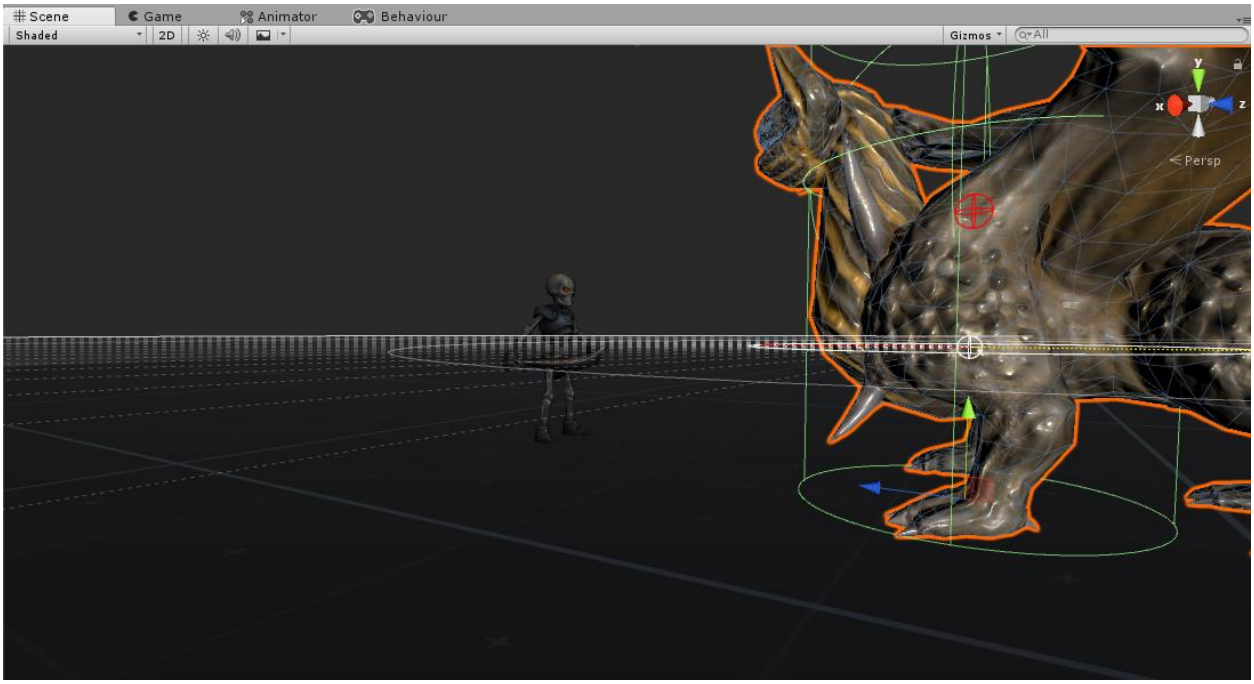
Create an Agent (using Unit Factory) and use one of the prefabs mentioned above as this Agent's graphics and "Dragon@Ragdoll", which was prepared a bit earlier as Agent's ragdoll. Adjust the size of the graphics GameObject so that it fits Agent just fine.



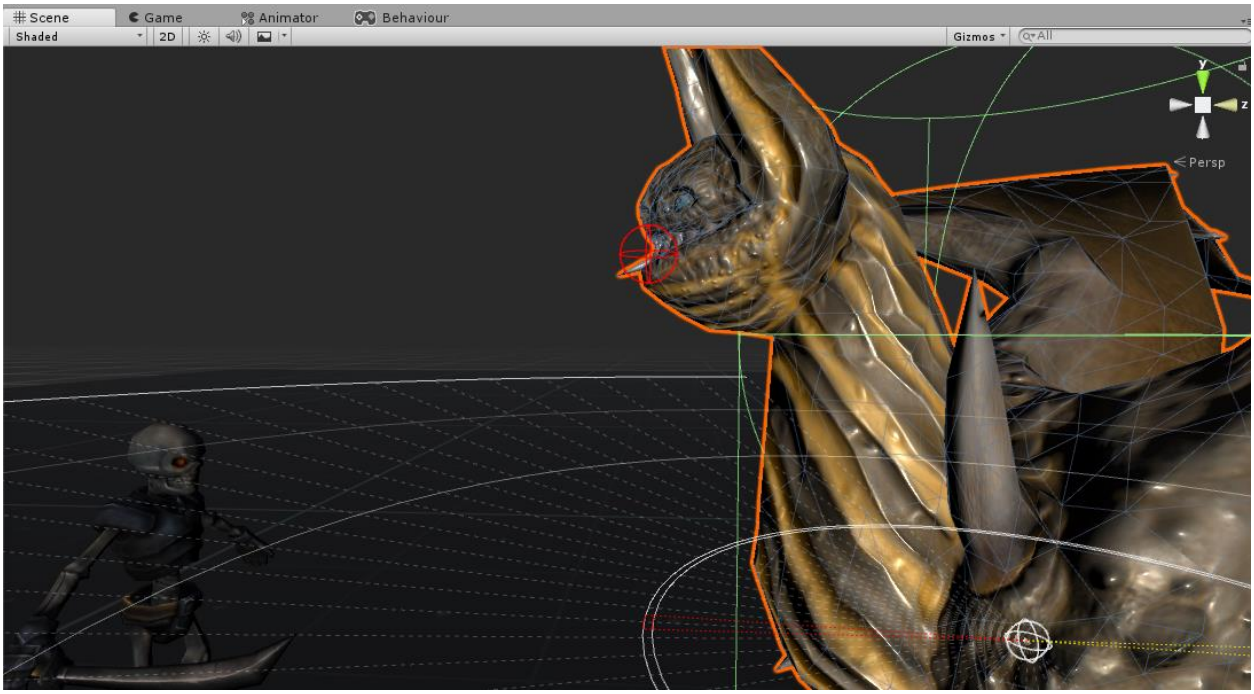
At this point, Dragon is the same size as Skeletons, so let's make him a little more scary. Adjust his scale so that Dragon is big enough for your taste. I set it to 3 on x, y and z. If you make it too big, it might get harder to follow the rest of the tutorial.

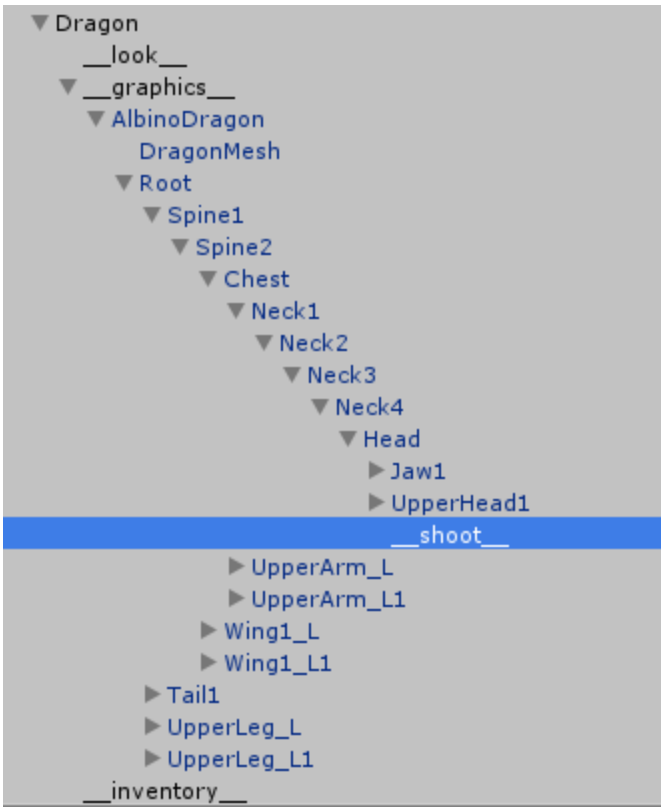


As you can see, if we put a Skeleton right next to the Dragon, we can see that Dragon's __look__ and __shoot__ are above the Skeleton, which would prevent Dragon from seeing and attacking Skeletons. We can fix it by simply dragging Dragon's __look__ down so that it would have no problem casting rays straight onto Skeleton.

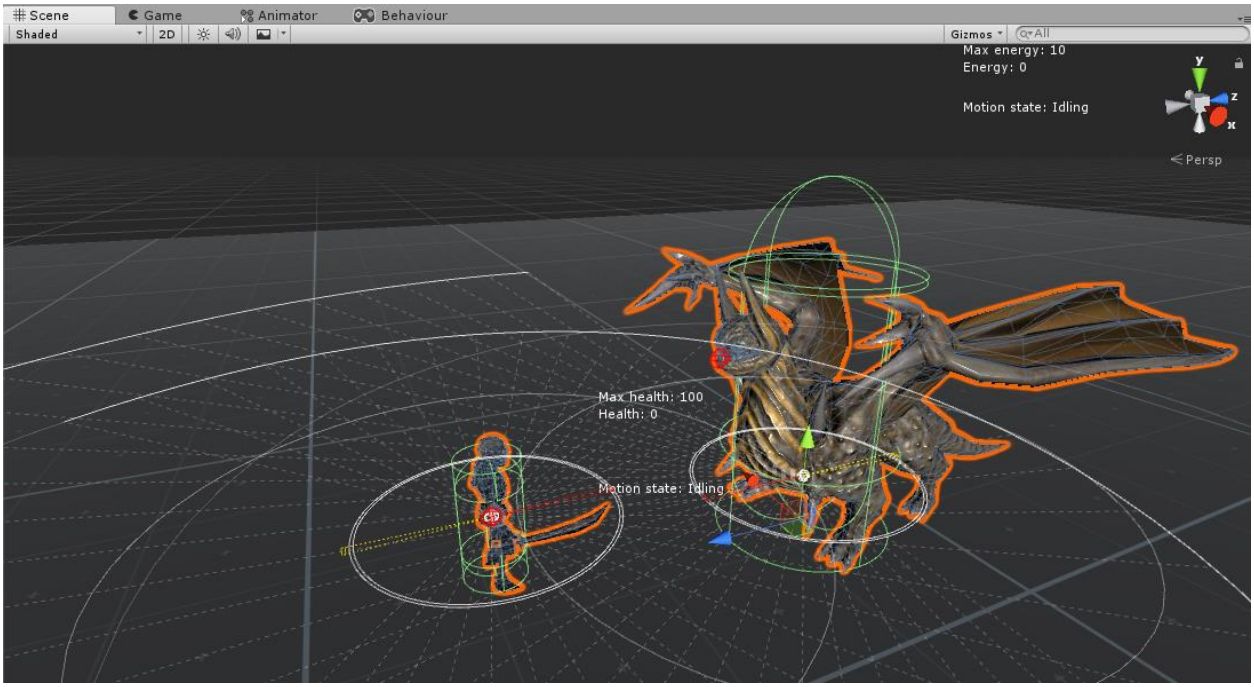


We are going to use __shoot__ as an origin for Dragon's fire, so let's put it near Dragon's mouth and also make it a child of Dragon's head, so that __shoot__ follows Dragon's animation.

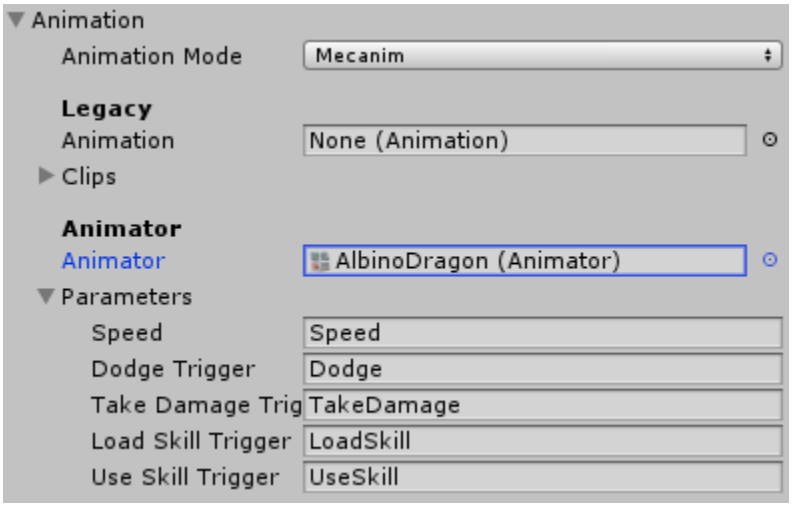




[!] There is an important thing to mention here. Agent's Perception casts rays that collide with Colliders. Agents, created with Unit Factory come along with Capsule Colliders on them. When Agents are different sizes, their Colliders might be at different altitudes, which might prevent them from seeing each other. To make sure it's not the case in your project, simply adjust the height of Colliders on your Agents so that they are around the height of your Agents themselves, like it is shown on the next screenshot.
You can find out more about Colliders in Unity Manual:
https://docs.unity3d.com/Manual/CollidersOverview.html
And specifically about Capsule Collider: https://docs.unity3d.com/Manual/class-CapsuleCollider.html



Since Dragon uses Mecanim for animations instead of Legacy animations, we need to configure that in Animation subcomponent of Dragon's Agent component in the Inspector. Set **Animation Mode** to *Mecanim* and drag and drop AlbinoDragon, which is a child of Agent's graphics container (named "__graphics__"), onto **Animator** field of Agent's Animation subcomponent.



Great! Now we are ready to do the next step.

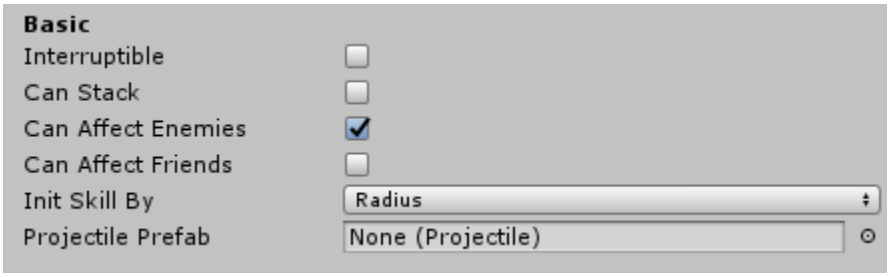
2.3.3 Create Skills

We are going to make Dragon protective. He will not try to go and look for trouble. He is going to simply stay where he is from the beginning of level and try to counterattack any harassment done to him.

In order to not get overwhelmed with attacks, Dragon needs a way to kick his opponents back. Also, since it is a Dragon, we'll make sure he can breathe with fire, attacking at a distance.

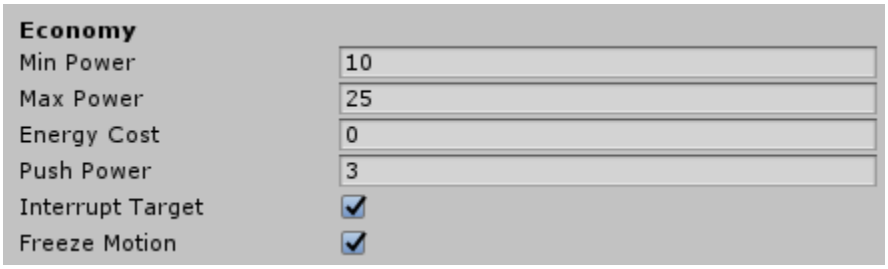
Dragon Claws Attack

The first Skill we are going to make is going to be the one that lets Dragon keep some distance between him and his opponents, however the order in which Skills are created is not important at all.
Create a new Skill anywhere in the project. Let the name of this Skill be "dragon-claws". The purpose of this Skill is to push opponents away from the Dragon. We'll make it so that it affects all enemies in certain radius. In order to do that, we should **Initialize Skill By** *Radius*.

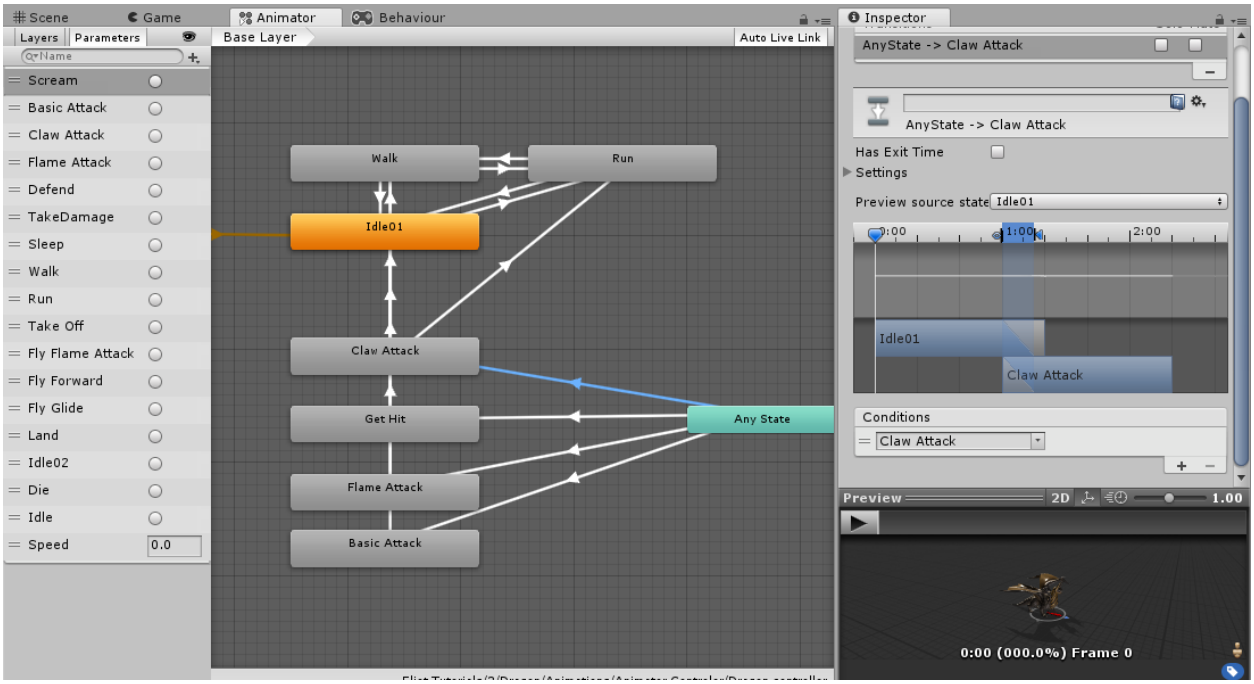


Since this Skill should affect only those enemies, which are close to the Dragon, a **Radius** of 3 should work fine.

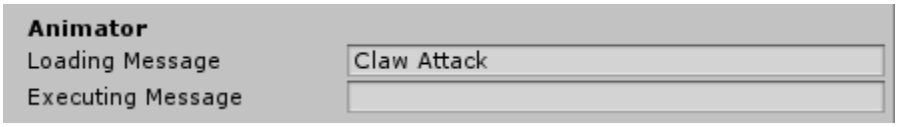
The only option in Economy group of settings, that plays a critical role for this Skill, is the **Push Power**. It should be enough to free some space for the Dragon.



In order to make Dragon play a corresponding animation, we should look at the way our Animator controller is set up.
Here we can see that the animation that is of interest for us is the one that is called Claw Attack. This animation is invoked by a trigger that is called **Claw Attack**.



So we set our **Loading Message** (in Skill configuration) to the name of that trigger ("Fly Flame Attack"). Since there is no distinction between loading skill and executing it in the Animator controller, we can leave **Executing Message** field blank.

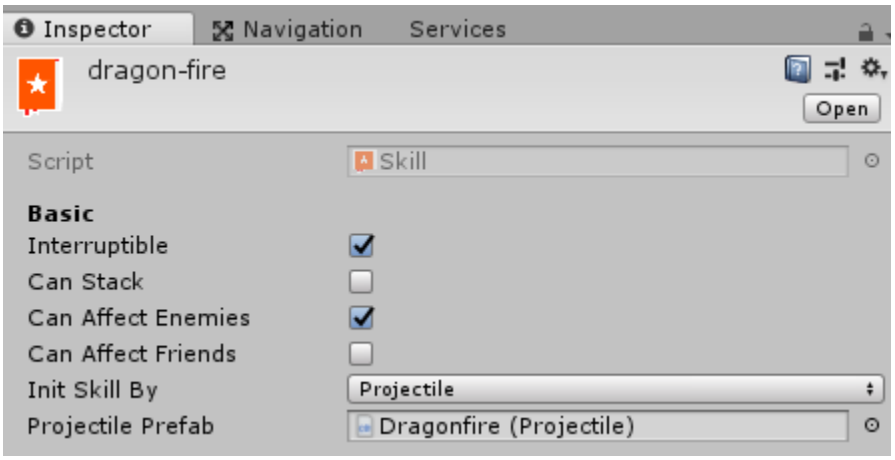


All the other settings of this Skill are completely up to you. Feel free to experiment with them.

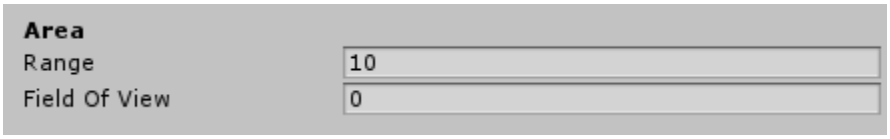
Dragon Fire Attack

The second Skill is going to let Dragon attack his opponents at a distance. As you should have already guessed, it is time to create a new Skill in the project and name it. Let's call it "dragon-fire".

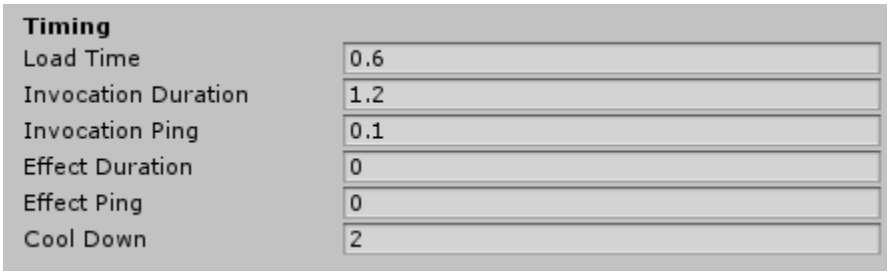
This Skill is going to be invoked by a Projectile. In fact, to create something similar to fire channeling, this Skill is going to shoot many projectiles during short period of time.
In "*../Eliot Tutorials/2/Prefabs"* folder you can find a prefab that is named "*Dragonfire"*. It has a Projectile component attached and fire particle system as a child to create effect of fire trail.
Set that Dragonfire prefab as a **Projectile Prefab** in the Skill's settings.



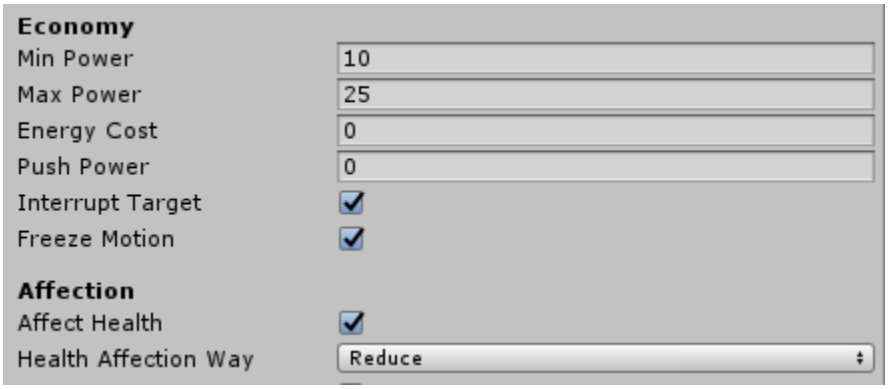
This is a range attack, so **Range** of 10 should be just fine.



Here is one of interesting parts of the Skill's settings. Take a look at **Invocation Duration** and **Invocation Ping**. The values of these variables that you can see in the next screenshot mean that Agent will Invoke the Skill (create new Projectile in this case) every 0.1 seconds for 1.2 seconds after waiting for 0.6 seconds for Skill to load.



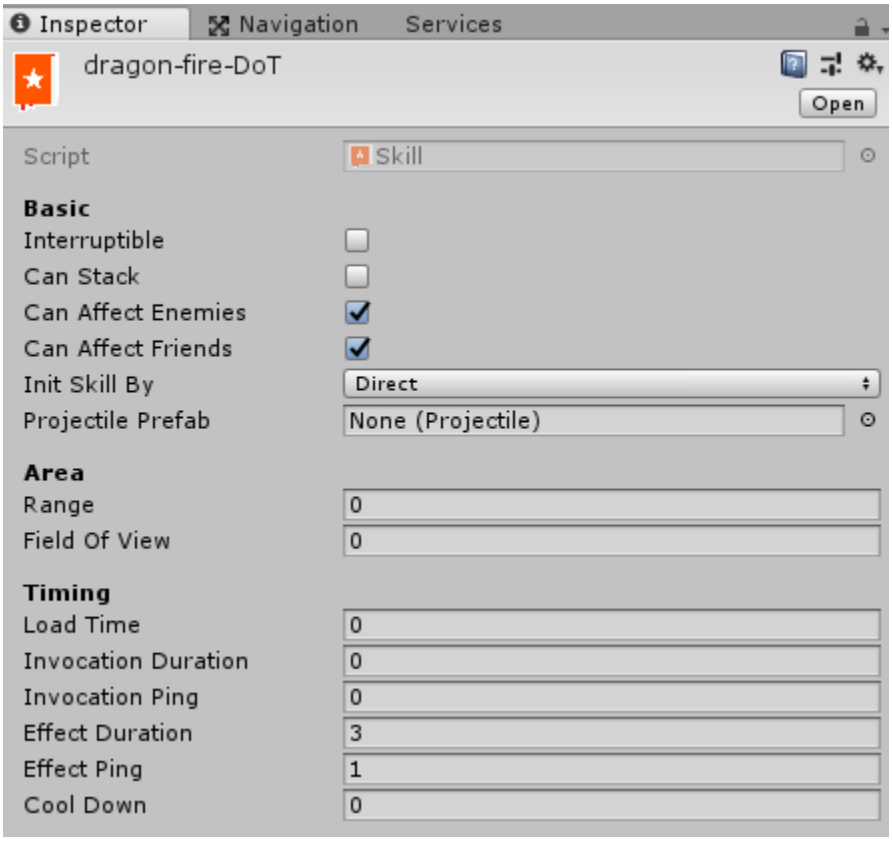
Now, sure enough, we want fire to deal some damage to the target.



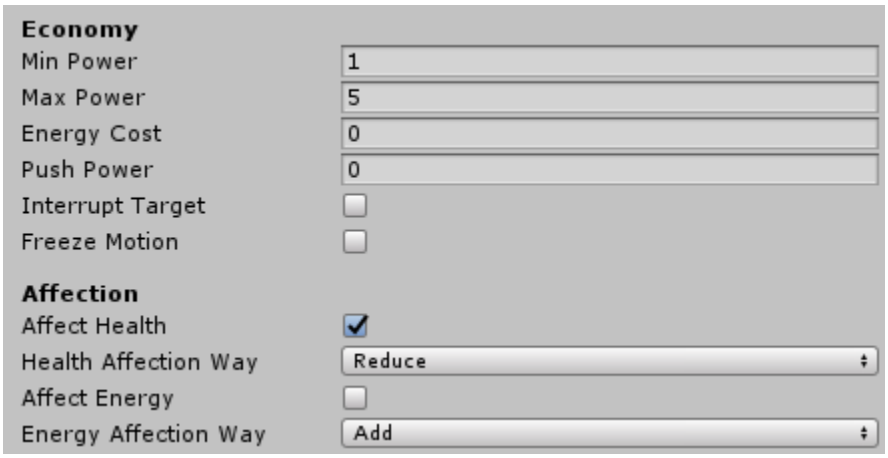
The Animator trigger parameter that invokes an animation that we want is called "Fly Flame Attack".

And here goes the second interesting thing about this Skill. We can put targets on fire and deal damage to them over time after they got into contact with Dragon's breath.
To do that we should create another Skill that will be applied right after the main dragon-fire takes its effect. So, create a new Skill and name it, for example, "dragon-fire-DoT".

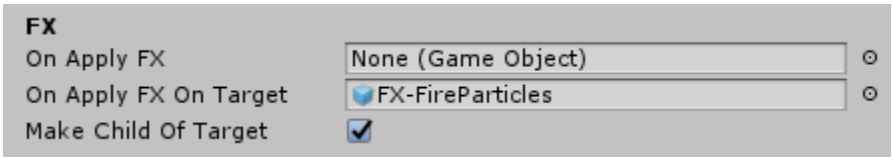
We do not need to cast any rays, as we already know the target of the Skill, so Skill can be initialized directly. It is going to be invoked by target itself, so we do not even need to configure its range.
To make it affect its target over time, we need to configure **Effect Duration** and **Effect Ping**. In this case the Skill will take its effect every 1 second for 3 seconds.



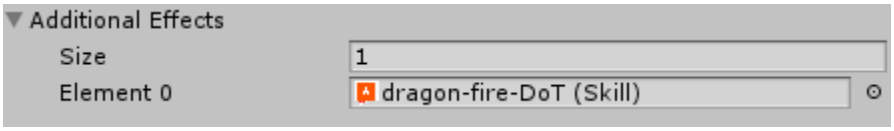
The Economy and Affection parts of the Skill should already be transparent to you. If not, take time to do previous exercises.



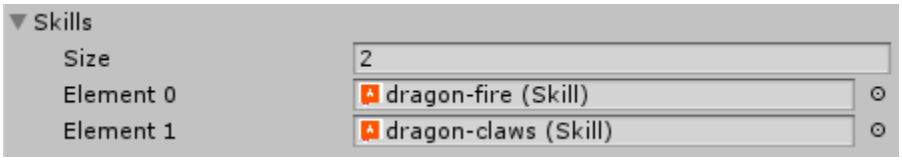
In order to create an effect of a target being on fire, we can set a prefab that represents special effects of fire as a value of **On Apply FX On Target** field.



Alright, our helper Skill is ready and we can add it to the list of Additional Effects of our main **dragon-fire** Skill.

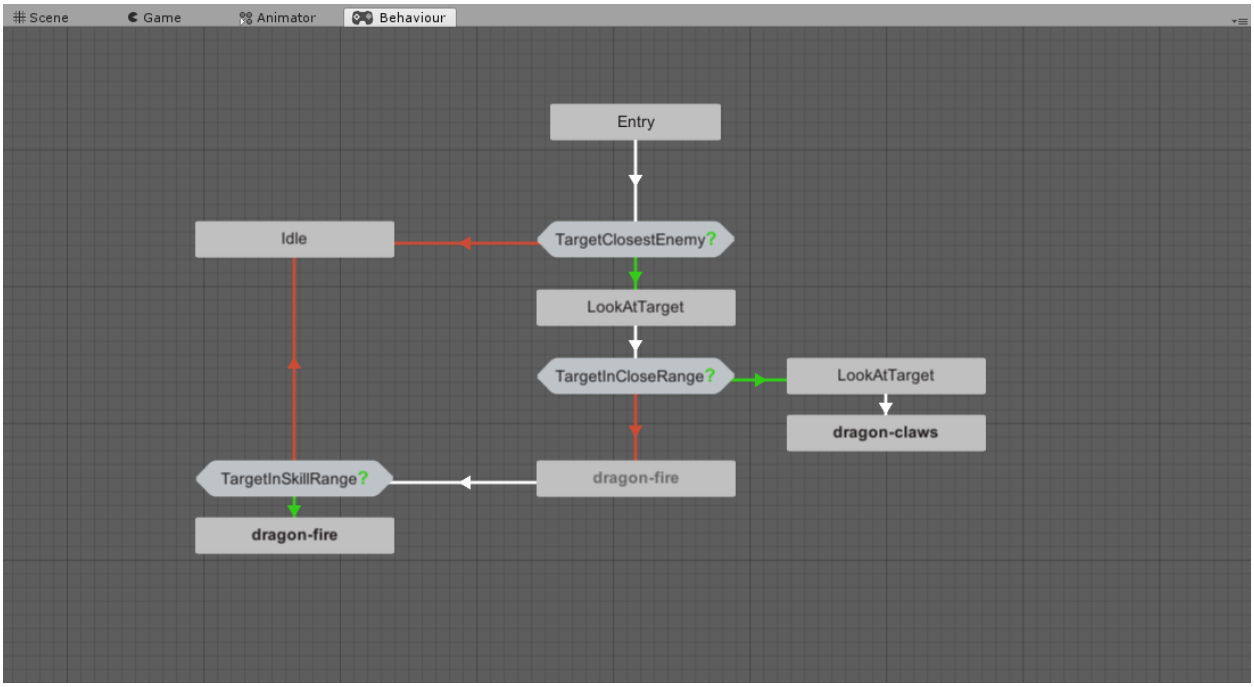


Of course, we do not forget to add newly created Skills that will be used by our Dragon to his **Skills** list.

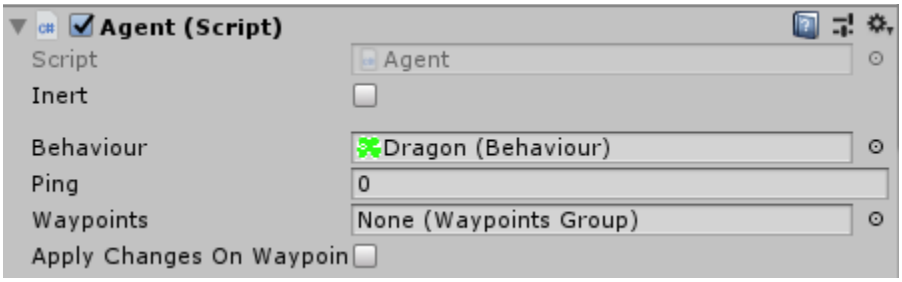


2.3.4 Create Behaviour

In this particular case we need a rather simple Behaviour. We just want our Dragon to Idle when nothing bothers him, push opponents away with **dragon-claws** when they are too close and attack them with **dragon-fire** when they are within Skill's range.



Sure enough, we don't forget to set this Behaviour to the corresponding Agent's field.

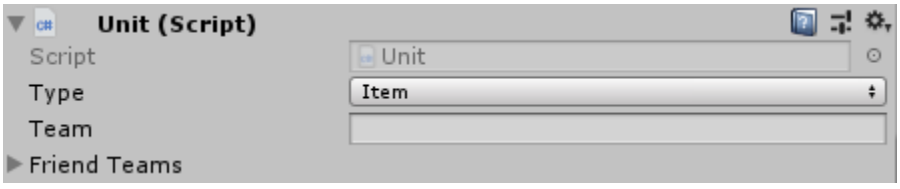


2.3.5 Put coins into Dragon's Inventory

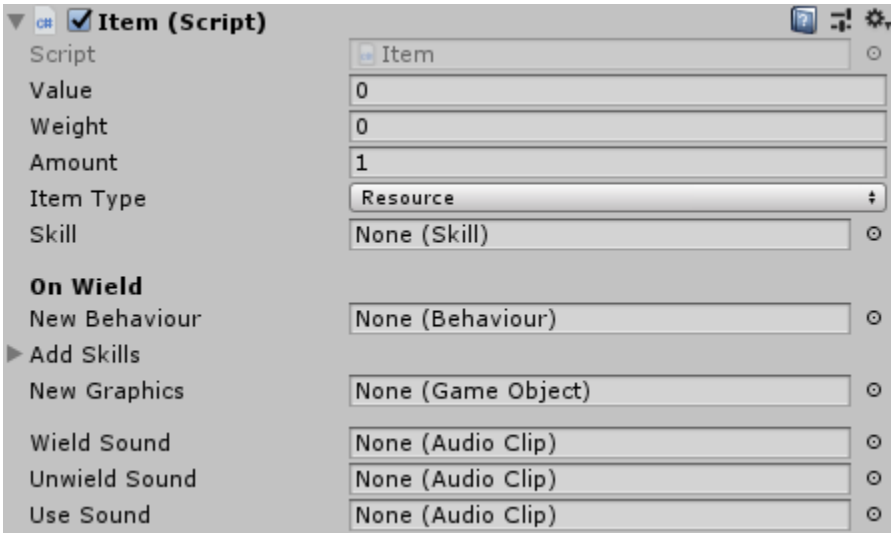
We can easily justify killing an animal, like Dragon, by putting some loot inside him. In this case, we'll use coins that Skeletons will be able to collect if they succeed in defeating the beast.

First of all, we should have those coins ready. In "*../Eliot Tutorials/2/Prefabs"* folder you will be able to find a prefab called "coin". Let's take a closer look at it.

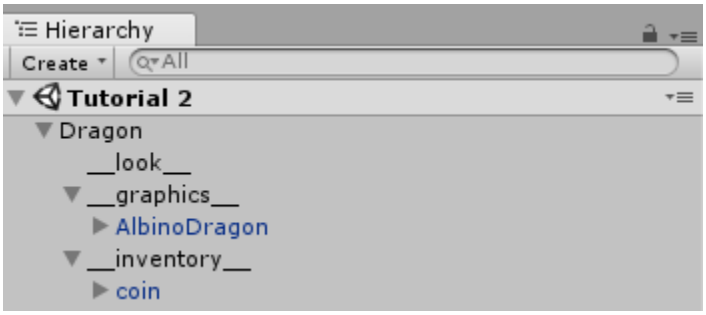
In coin's Inspector we can see that it has a Unit component attached, which helps Agents identify the object as an Item.



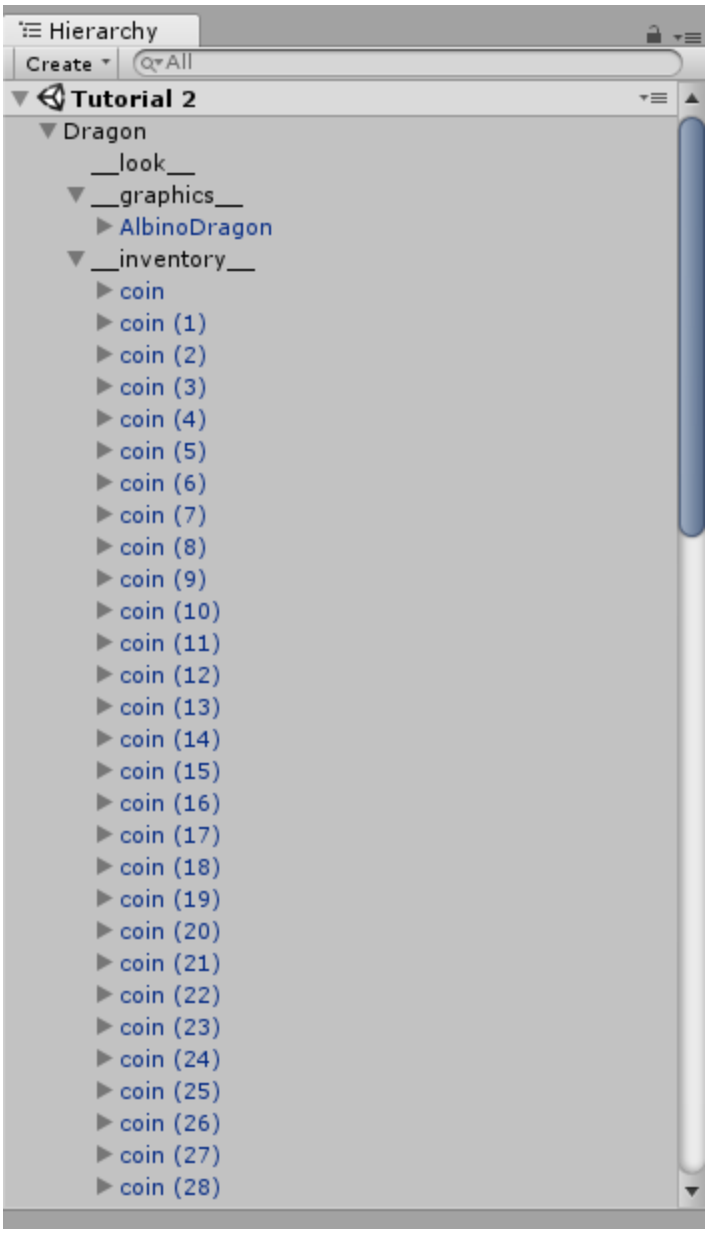
We are also able to see that there is an Item component attached to the coin. All of its fields are left untouched, except from its **Item Type**, because we do not need to use any of its functionality right now.



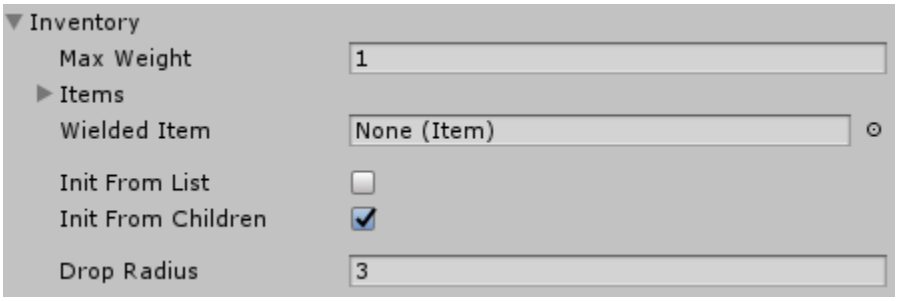
To put a coin into Dragon's Inventory simply drag and drop the coin prefab somewhere on the scene and put it inside Dragon's __inventory__ (make it a child of the GameObject, named "__inventory__", which you can find among Dragon's children in Hierarchy window).



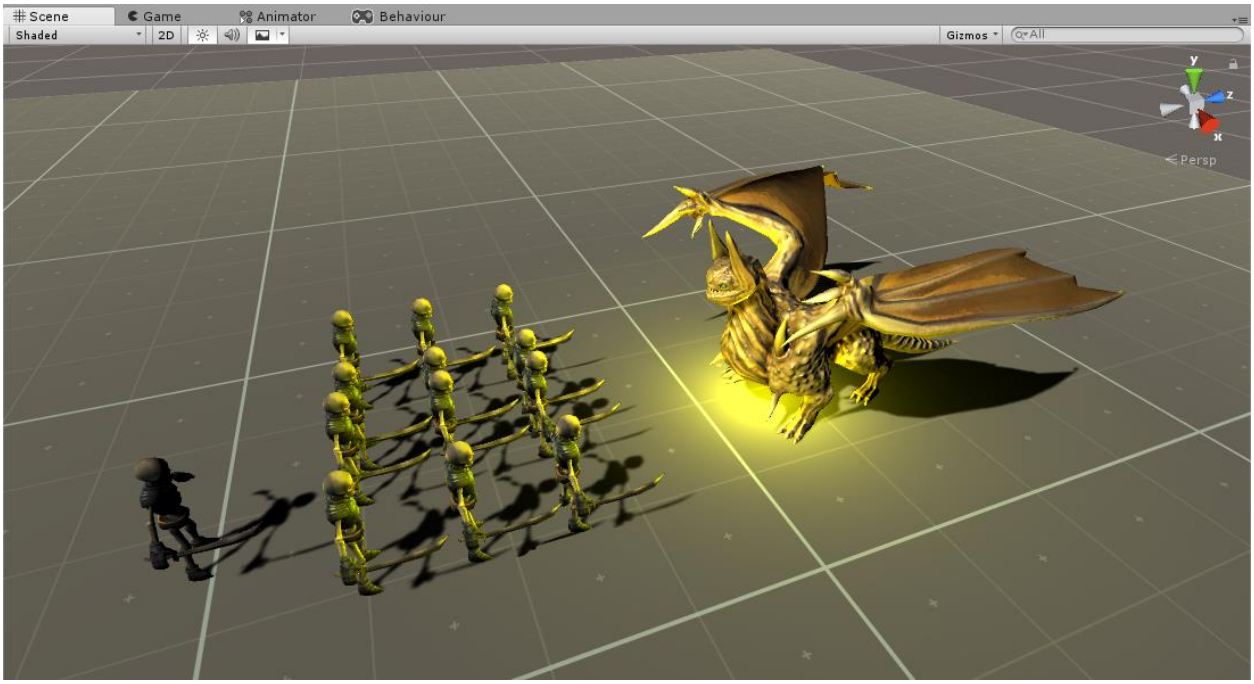
A dragon with a single coin is not a very good justification for murder, so clone those coins until you feel like Skeletons have a serious reason to mess around with the Dragon.



Now go to Dragon's Inventory settings and make sure **Init From Children** toggle is checked.



One more step needs to be done before we can admire what we have accomplished here. Take those Skeletons that we created in previous tutorials and put them somewhere near the Dragon. Make sure that Skeletons are all in one team and the Dragon in the other.



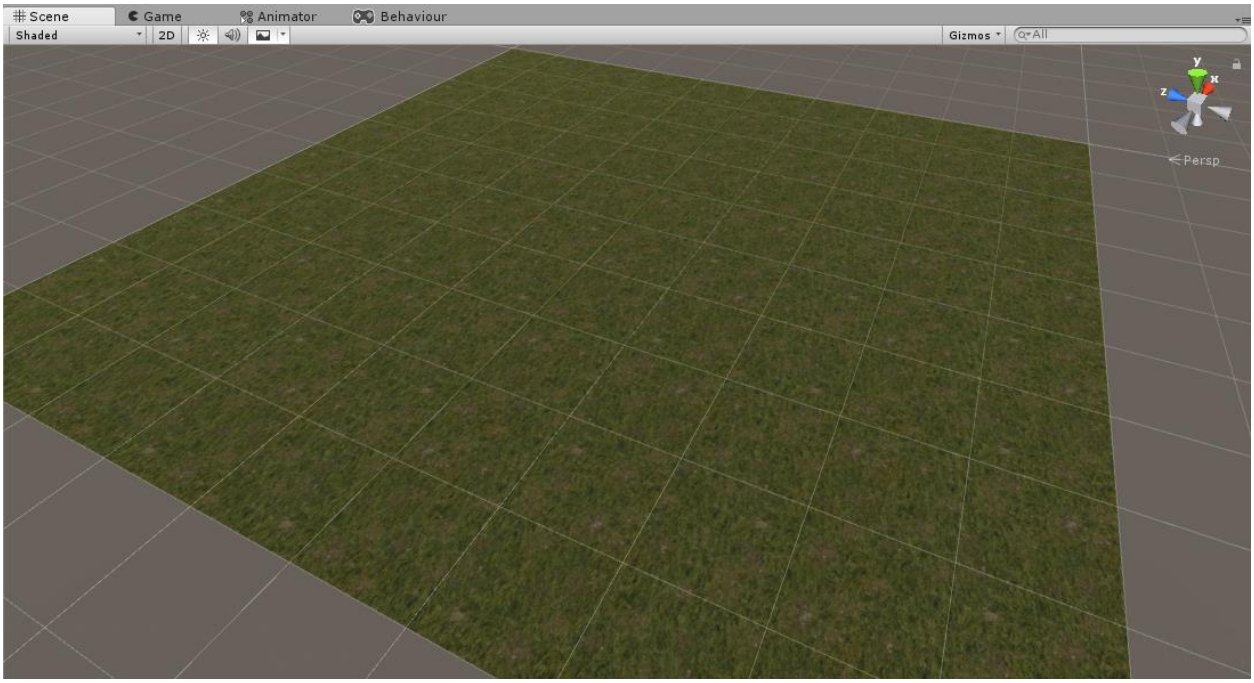
(This shining is produced by coins)

Hit the Play button.

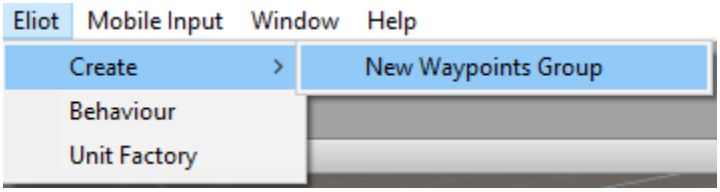
3 Advanced use of Waypoints

3.1 Spawning skeletons in specified area with WaypointsGroup

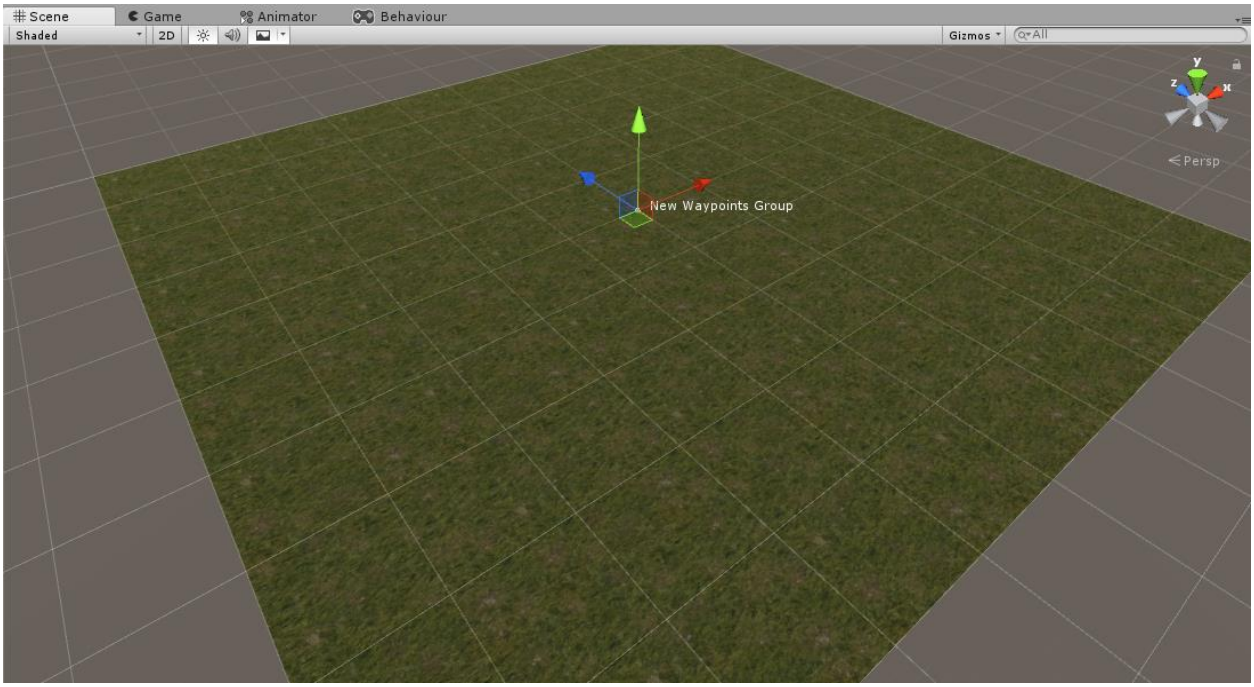
In this tutorial you will learn how to use Eliot Waypoints to spawn Agents in Unity Editor and how to use Agents Pool to spawn Agents at runtime. Spawned Agents will be represented by skeletons in this example.
To follow this tutorial, open scene “Tutorial 3.1” in “*../Eliot Tutorials/3”* folder.



Create new Waypoints Group using *Eliot/Create/New Waypoints Group* menu item.



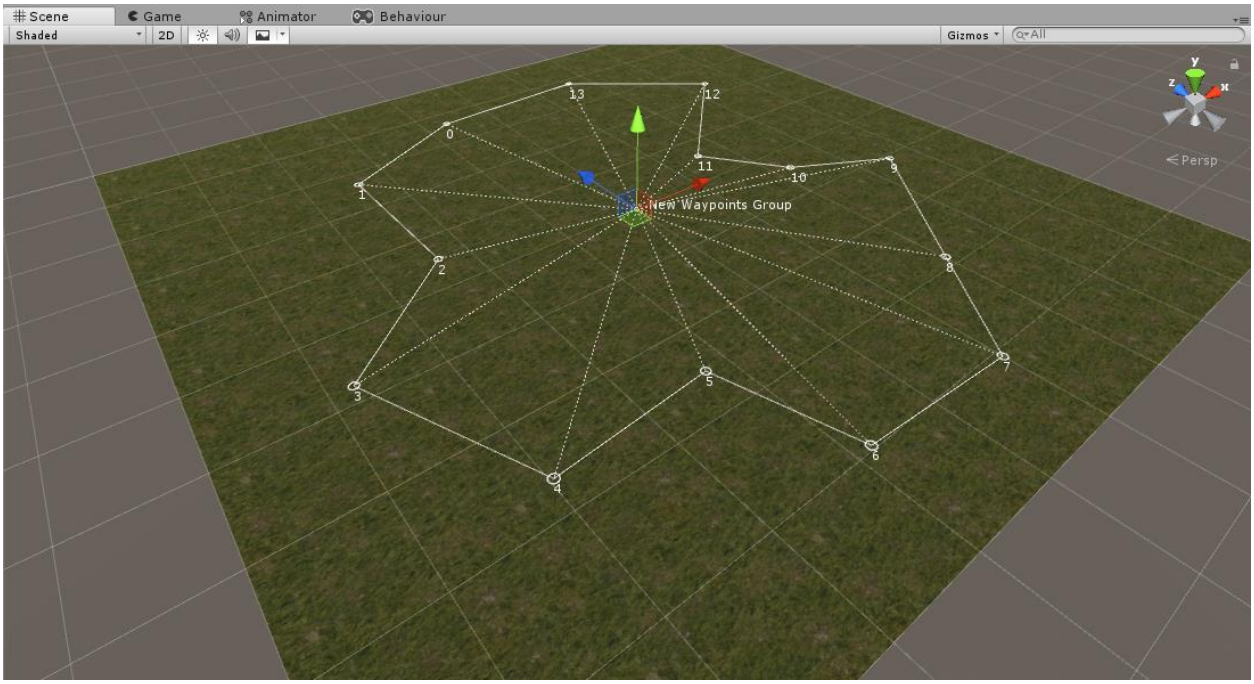
Place it on the terrain with the help of **Project On Plane** button (which can be found in WaypointsGroup’s Inspector). This will align object’s Y coordinate.



Now we want to define an area in which skeletons will be spawned. To do that, activate Waypoints placement mode by clicking Place Waypoints button in WaypointsGroup's Inspector.

After you click it, the button should change its looks letting user know if the Waypoints placement mode is on.

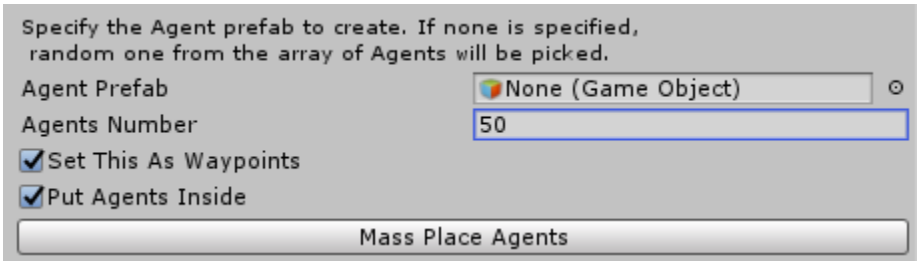
Now just click on the terrain in the Scene where you want to place Waypoints, collection of which is going to define our area's borders.

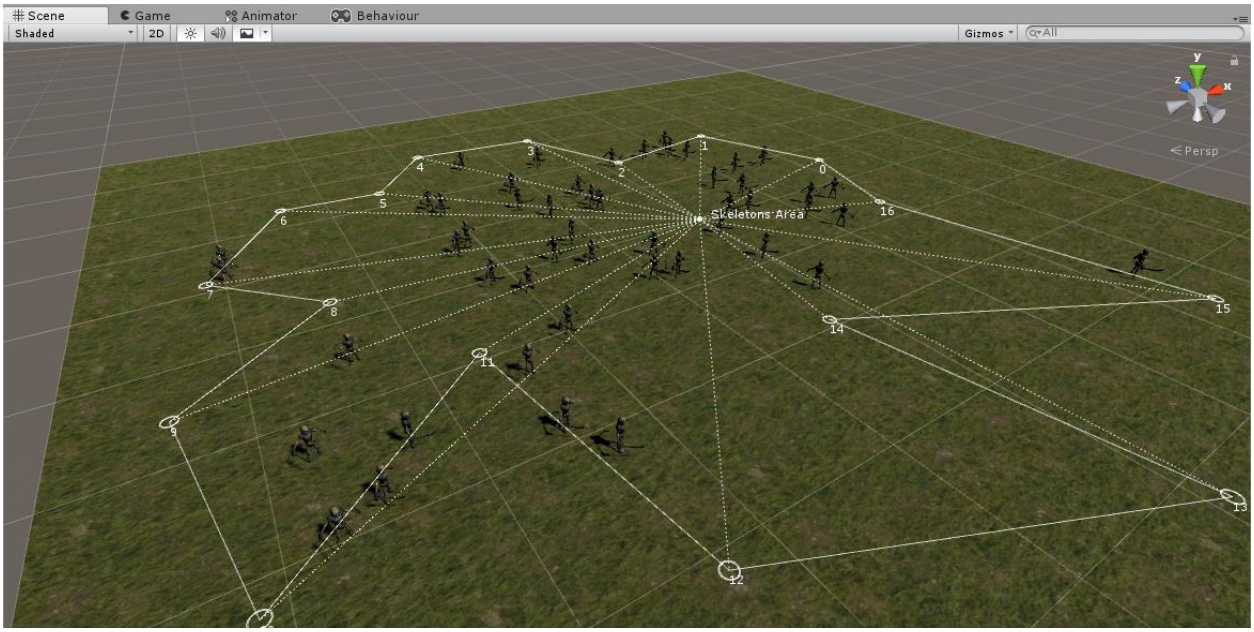


When you are done placing Waypoints, don't forget to turn Waypoints placement mode off by clicking the Place Waypoints button.

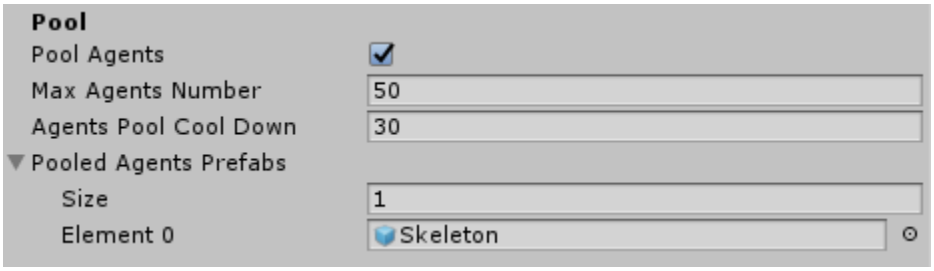
Now, in order to spawn skeletons, we need to define the prefab, which needs to be spawned. In "*../Eliot Tutorials/2/Prefabs"* folder you can find a prefab, called Skeleton. It is an Eliot Agent, ready for interactions.
Drag and drop that prefab into **Pooled Agents Prefabs**.

As soon as we specify **Agents Number** and press **Mass Place Agents** button, we should be able to see that our predefined area is now populated with skeletons.





If you want skeletons to be respawned inside the area in specified time after their death, you just need to make sure that **Pool Agents** toggle is checked and choose appropriate values for variables in Pool section.



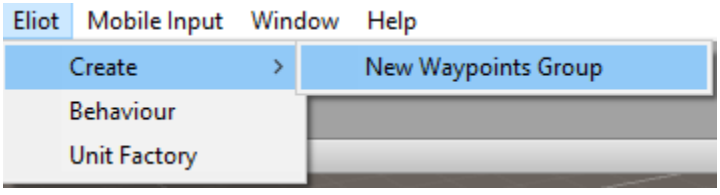
The values, shown in the screenshot mean that one Skeleton prefab will be spawned every 30 seconds at a random point inside the area if current number of skeletons is less than 50.

3.2 Creating sequence of tasks with Waypoints

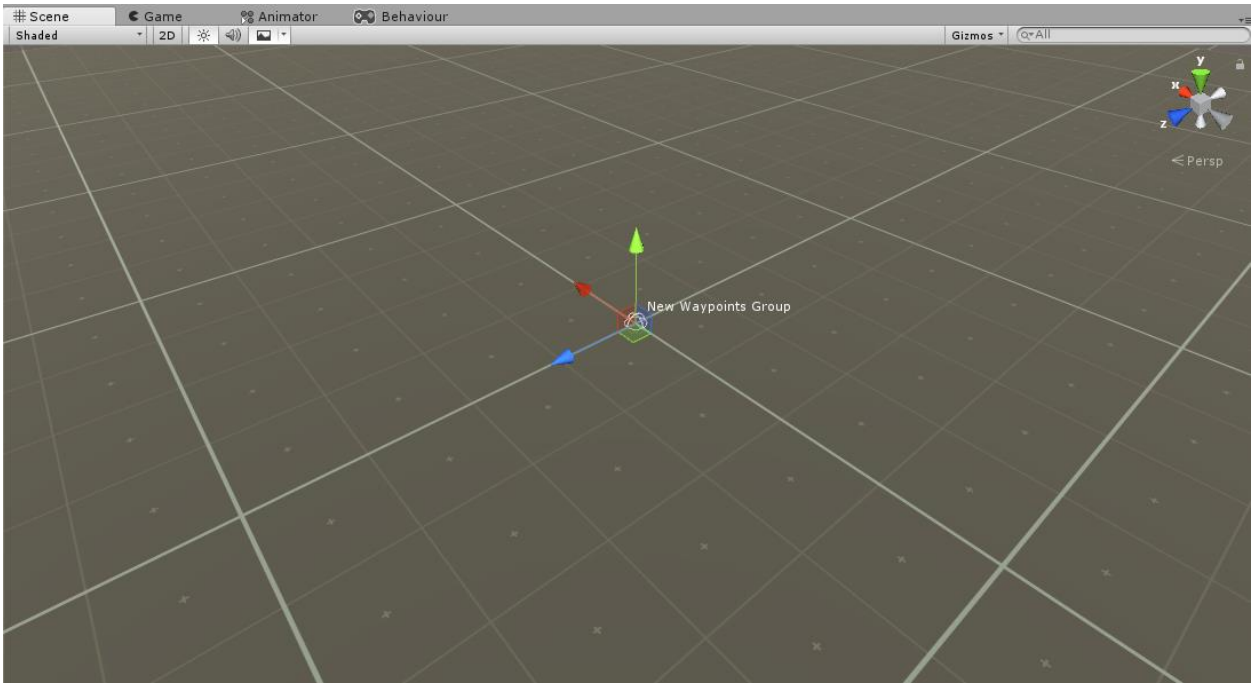
In this tutorial you will learn how to use Eliot Waypoints to assign different Behaviours to Agents at runtime to create controllable sequence of “tasks”. We will make skeleton alternate walking and running while it gets round Waypoints.

To follow this tutorial, open scene “Tutorial 3.2” in “*../Eliot Tutorials/3”* folder.

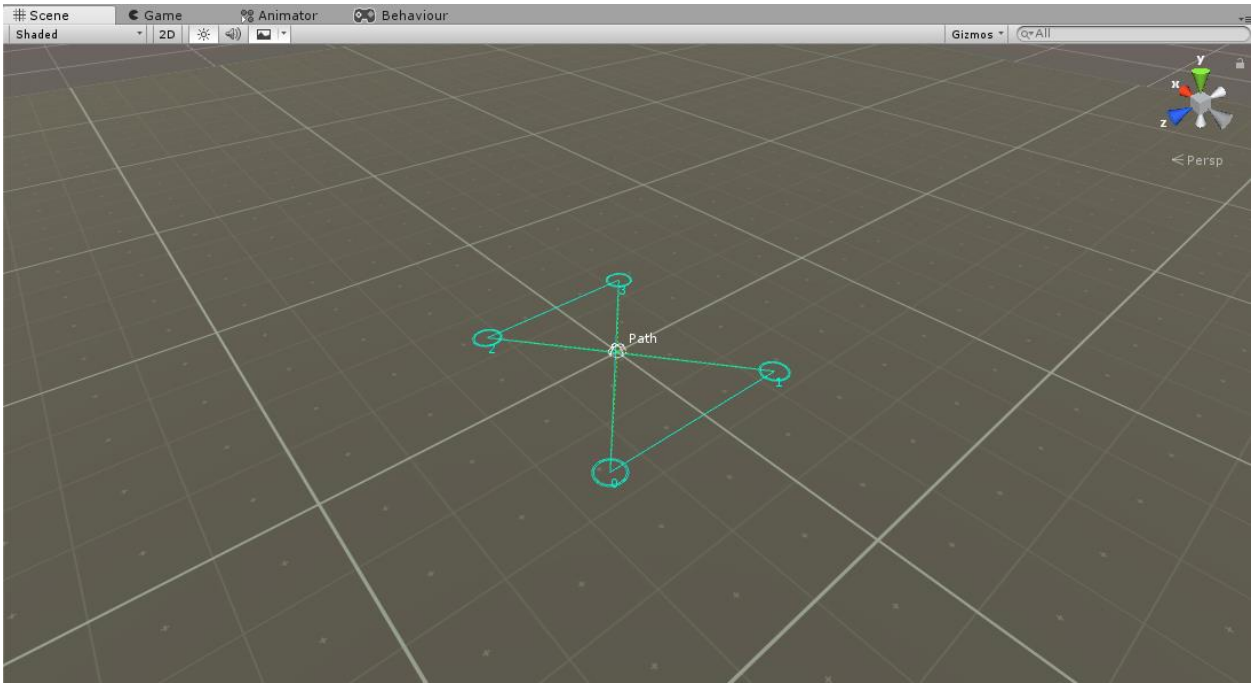
Create new Waypoints Group using *Eliot/Create/New Waypoints Group* menu item...



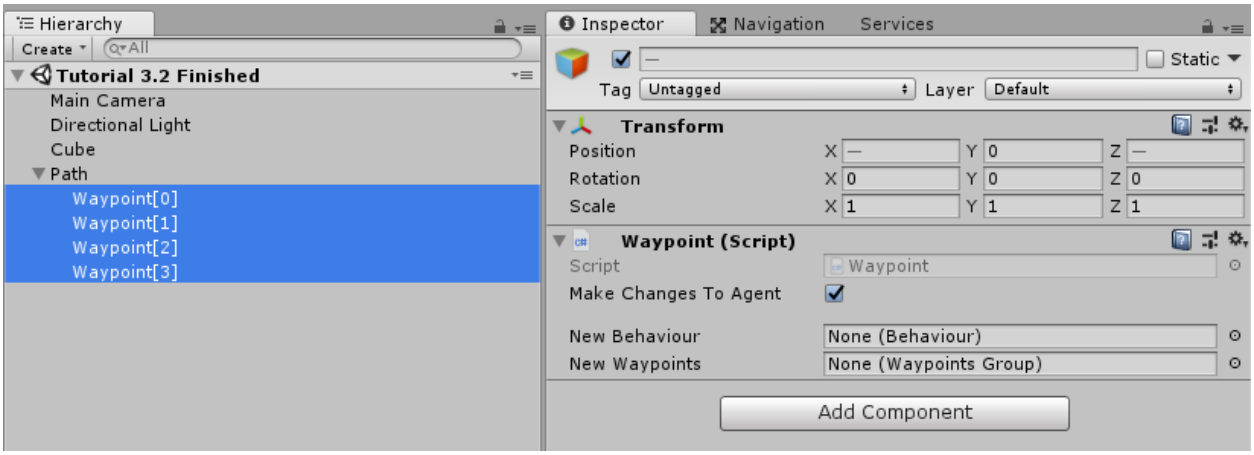
… and place it where you feel comfortable.



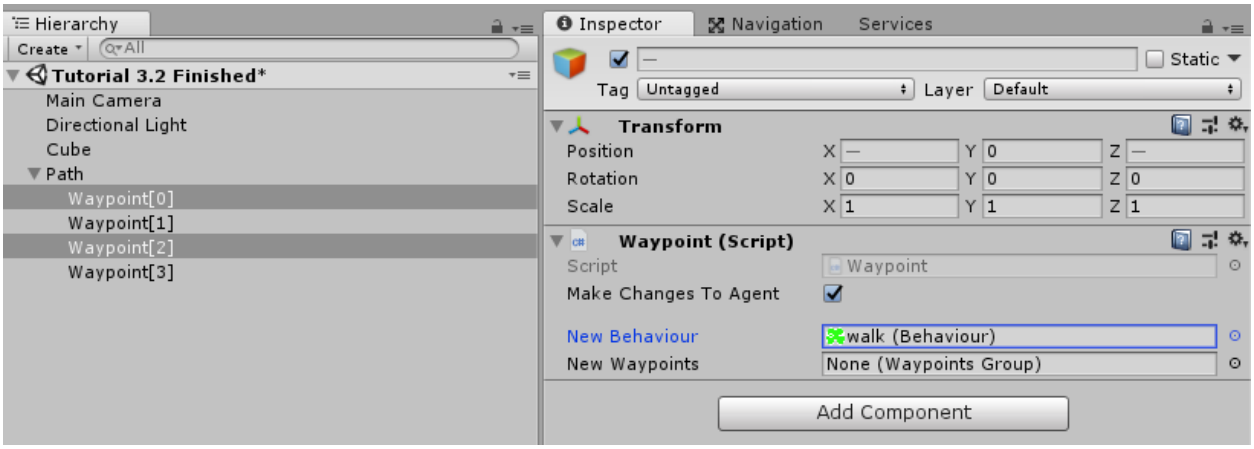
Now place Waypoints not too far from each other so that we don’t need to wait too long to see the result.



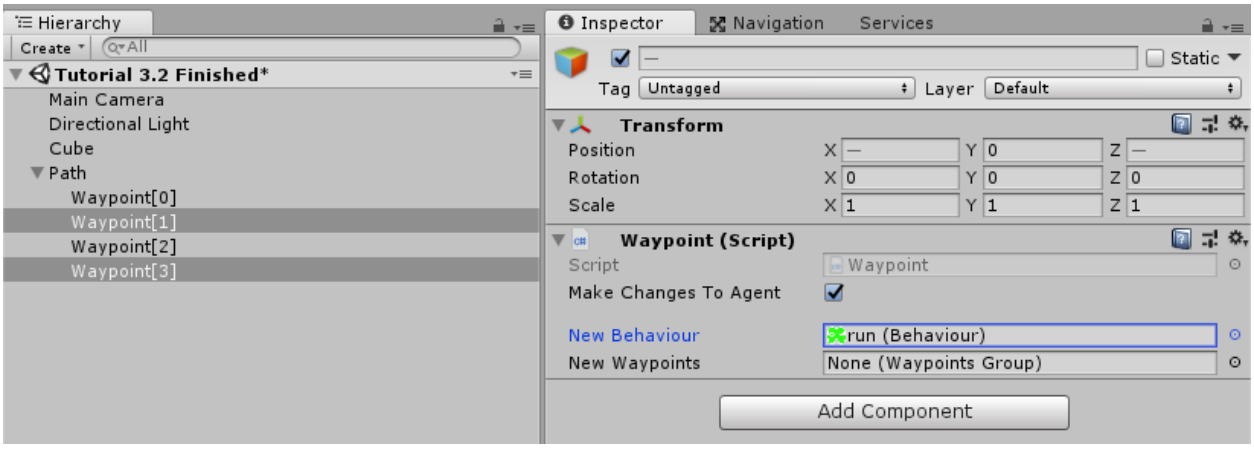
Select all the Waypoints in the group and make sure they have **Make Changes To Agent** toggle checked. This will make an attempt to apply its values of **New Behaviour** and **New Waypoints** to an Agent who’s over a Waypoint if that Agent also agrees to make changes.



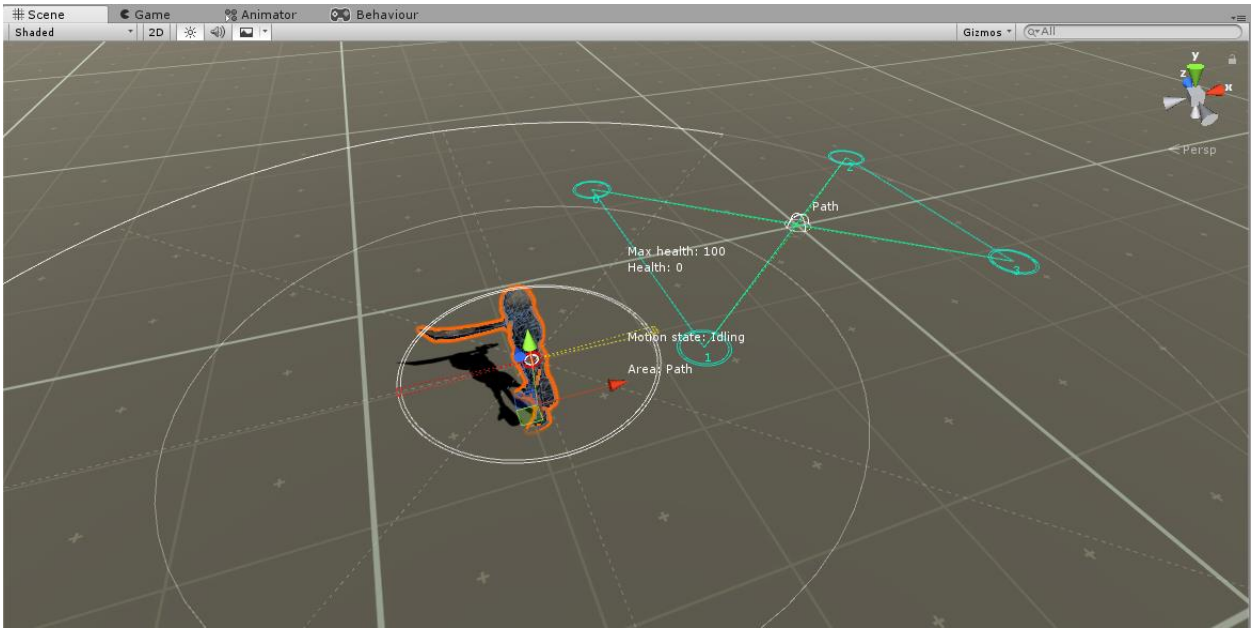
Now let's define Behaviours for each Waypoint that are going to be applied to an Agent when he steps over. Select first and third Waypoints. Drag and drop a Behaviour named "walk" (which can be found in "*../Eliot Tutorials/3"* folder) into the corresponding field.



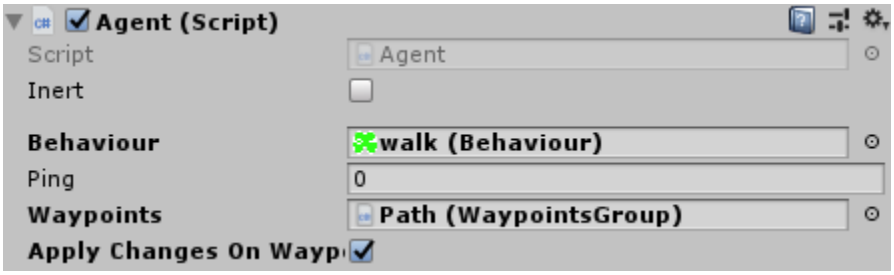
Select second and fourth Waypoints. Drag and drop a Behaviour named "run" (which can be found in "*../Eliot Tutorials/3"* folder) into the corresponding field.



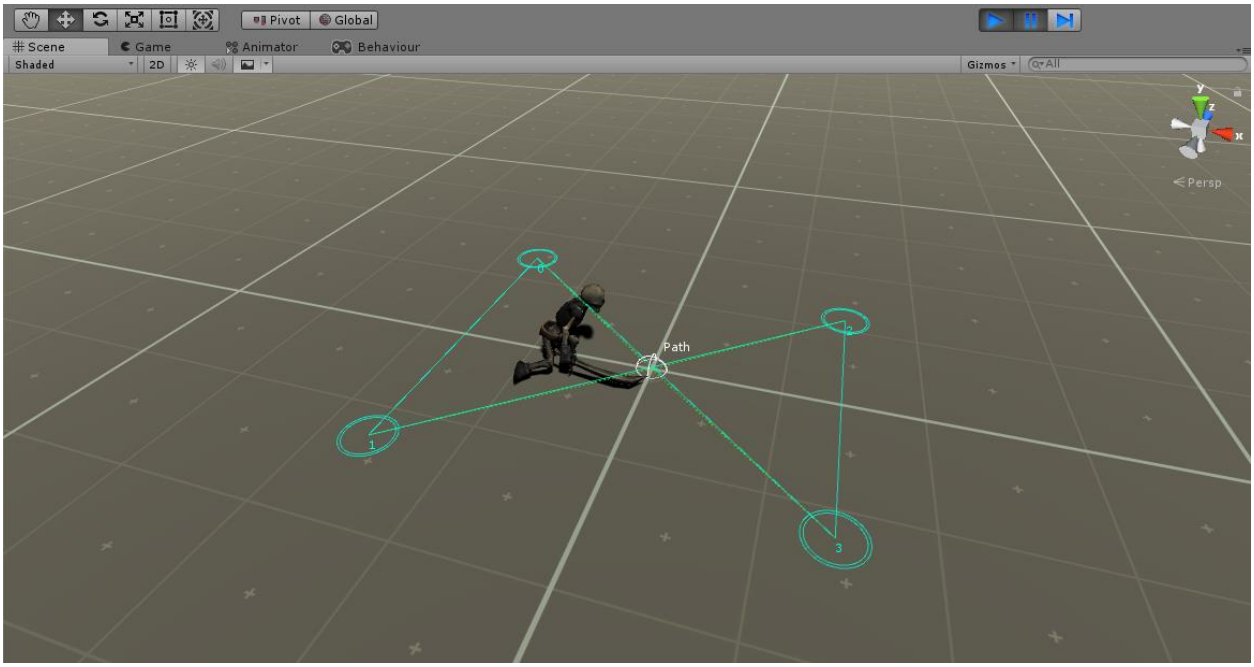
Now place a Skeleton from "*../Eliot Tutorials/2/Prefabs"* folder into the scene, ideally somewhere near the first Waypoint to minimize waiting.



Make sure its Behaviour is initially set to either “walk” or “run” so that it does not miss an interaction with Waypoints. Set our path (WaypointsGroup) that we created and configured as Agent’s **Waypoints** and make sure that **Apply Changes On Waypoints** toggle is checked so that Agent agrees to apply changes, suggested by a Waypoint.



That’s it. In real world applications, you would probably use more complicated Behaviours than those presented in this tutorial, but it should be enough to give general idea about the feature and how to use it.


Pivot
Global
Scene
Game
Animator
Behaviour
Shaded
2D
Gizmos
All
Persp
Path
1
2
3