



Introduction

This document goes in depth about all the scripts one by one, and is intended for educators or for developers who want to know more about the Unity Playground.

For basic information about the Playground and how to get started with it, please check the Getting Started guide (on the [Learn website](#), included in the Playground as a PDF, or online [here](#)).

Table of Contents

| | |
|--------------------------|----------|
| Introduction | 1 |
| Table of Contents | 2 |
| Movement scripts | 4 |
| Auto Move | 4 |
| Auto Rotate | 6 |
| Camera Follow | 7 |
| Follow Target | 9 |
| J | |

| | |
|--------------------|----|
| ConditionArea | 34 |
| ConditionCollision | 35 |
| ConditionKe | 3 |

Movement scripts

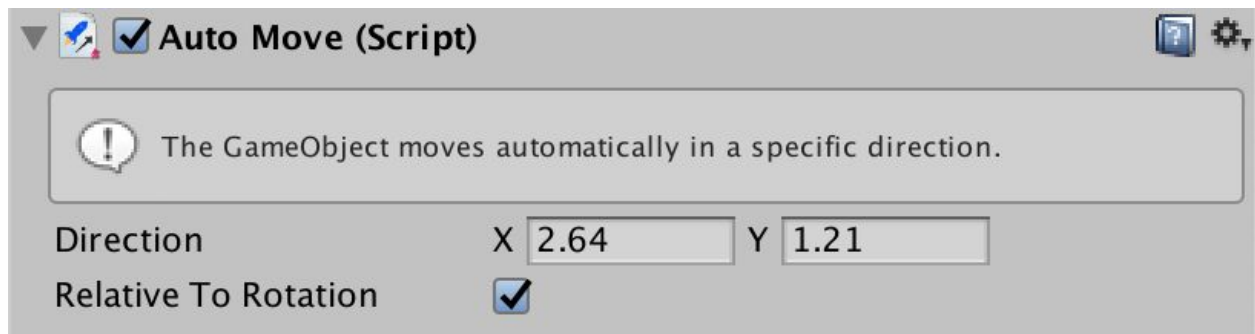
This category of scripts is all about moving GameObjects around, whether it is the player, hazards, or non-playing characters.

Being the Playground all based on physics, they almost all require a Rigidbody2D to produce movement, and potentially some type of Collider2D if you want the object to be able to interact with others.



Auto Move

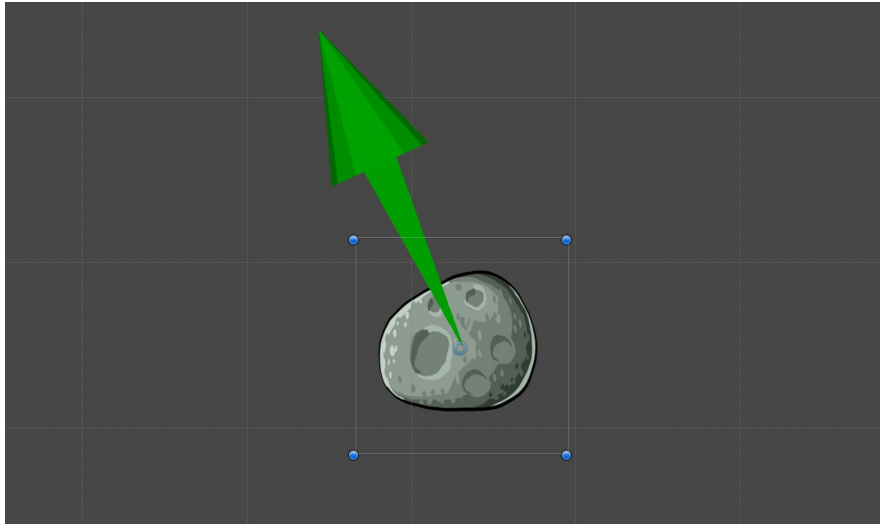
Requires: Rigidbody2D



AutoMove applies a continuous force to GameObject. Useful for things such as rockets, arrows, and other self propelled objects.

The direction is expressed through a Vector2 and includes the strength, and it can be absolute or relative to the object's rotation.

In the Scene view, a green arrow gizmo represents the direction of the push, while its size represents the strength.

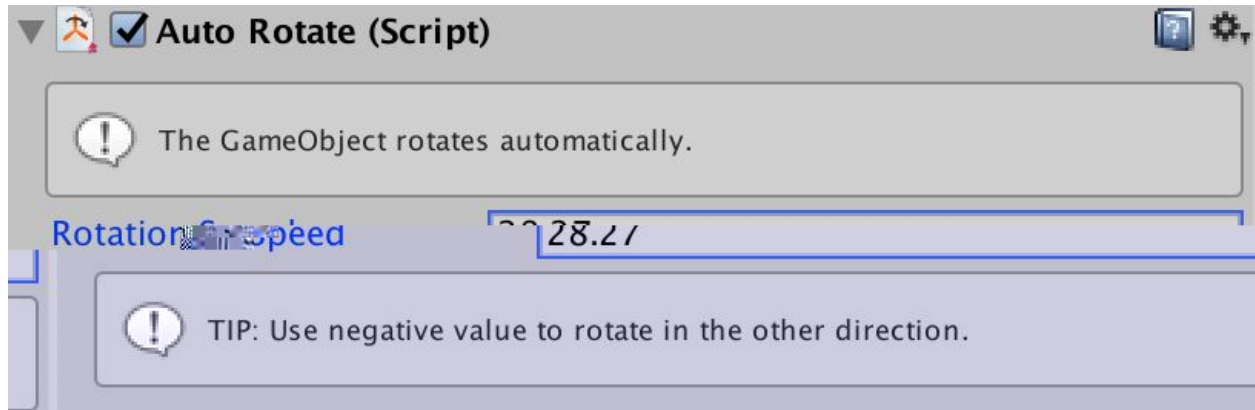


Note: If you are considering using this script on an prefab created with the help of the Object Shooter script, remember that Object Shooter already applies a force to objects when shooting them. In case of a non self-propelled object (like a catapult rock), you don't need Auto Move on the projectile.



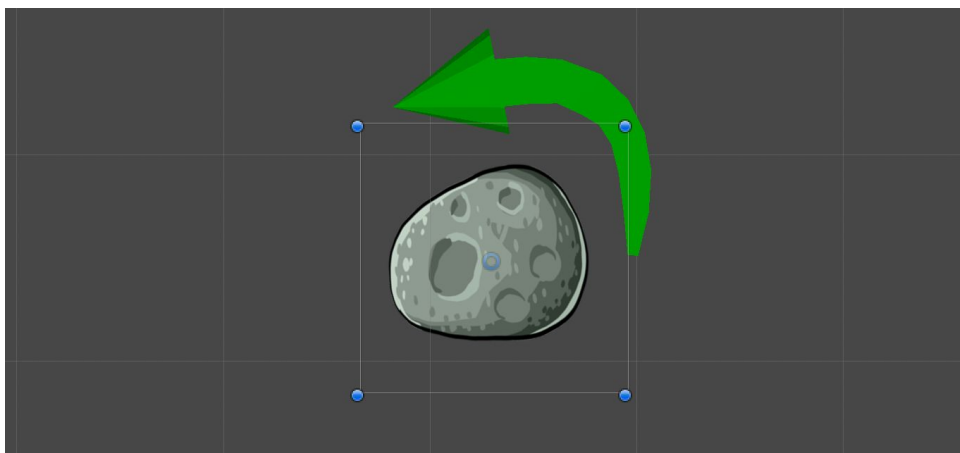
Auto Rotate

Requires: Rigidbody 2D



AutoRotation applies a continuous rotation to a GameObject on the Z axis. It can be used to add movement to a decorative object, but also to create rotating obstacles in conjunction with a [ConditionCollision](#) script. You can specify the speed, and setting a negative one makes the object rotate counter-clockwise.

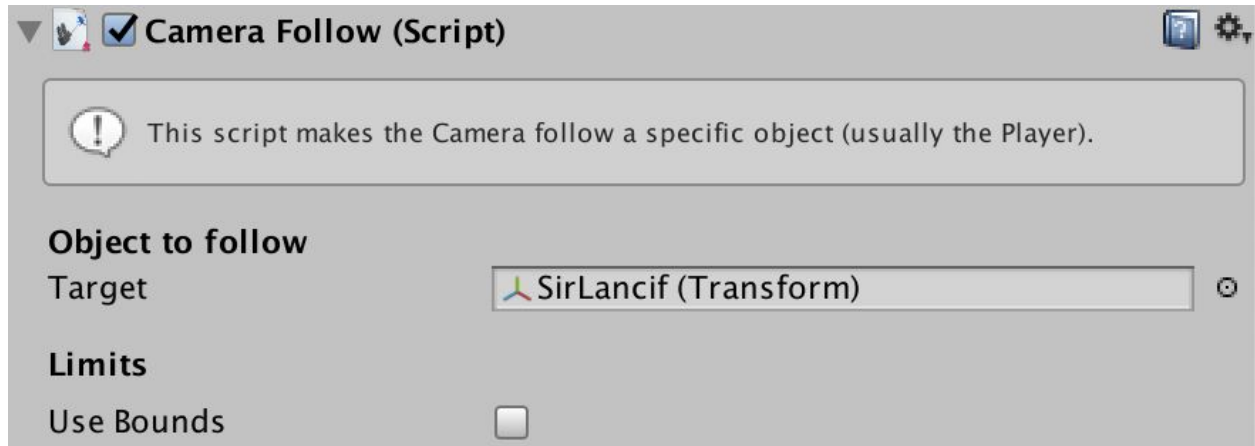
In the Scene view, a green arrow gizmo represents the direction of the rotation.





Camera Follow

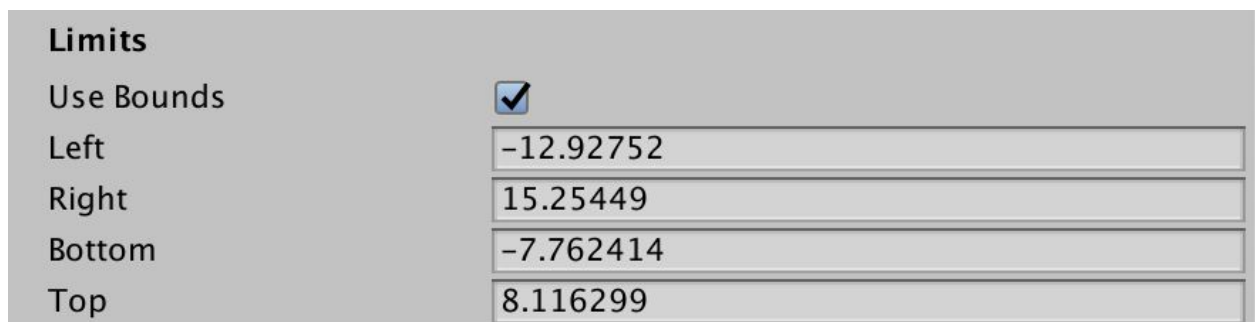
Requires: Camera



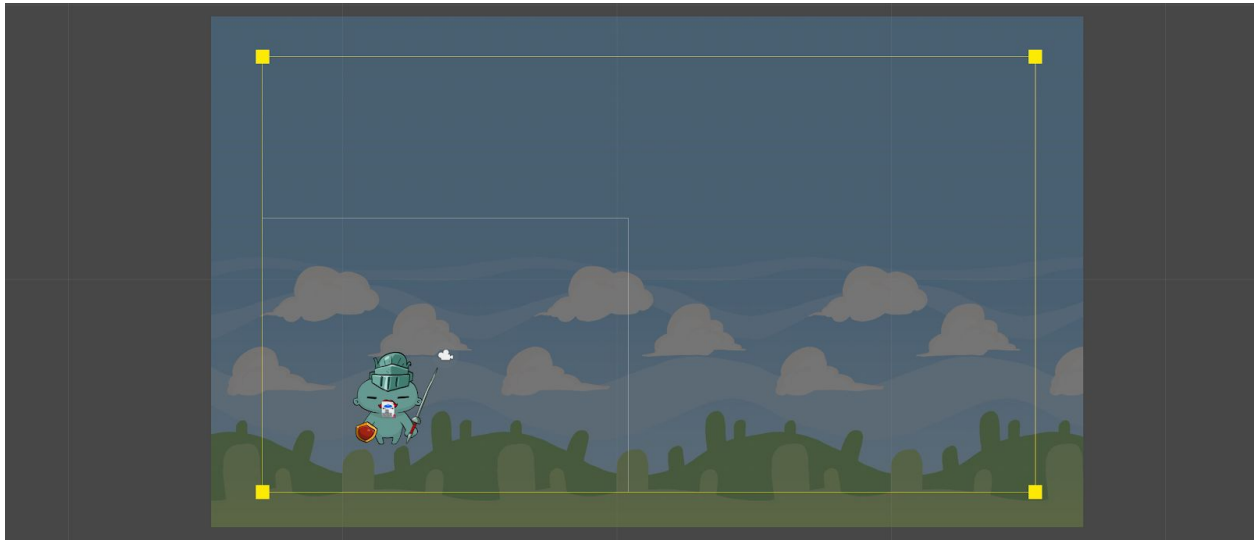
Use CameraFollow on a GameObject that has a Camera component. This is useful for action adventure games where the camera is centred on the Player. Assign a moving GameObject in the scene as the **Target**.

Note: Add this script to the Camera, not to the object being followed!

If you tick the property **Use Bounds**, you will be able to constrain the movement of the Camera to a rectangle.



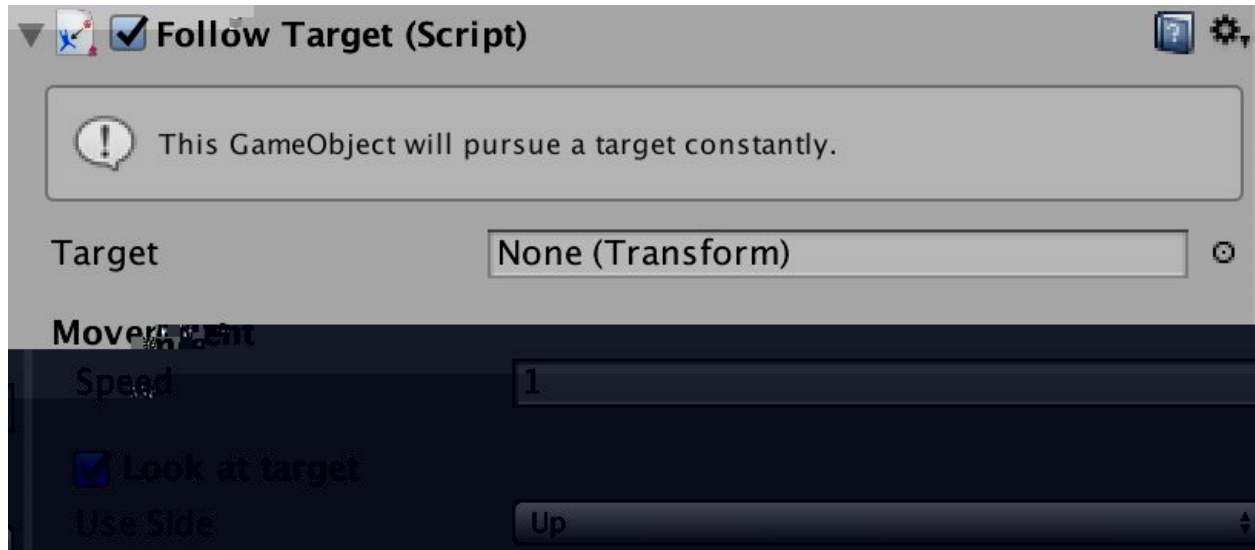
You can adjust the values of the bounds through the Inspector or in the Scene View with the yellow rectangle gizmo:





Follow Target

Requires: Rigidbody2D



FollowTarget forces the GameObject to follow a specified target indefinitely.

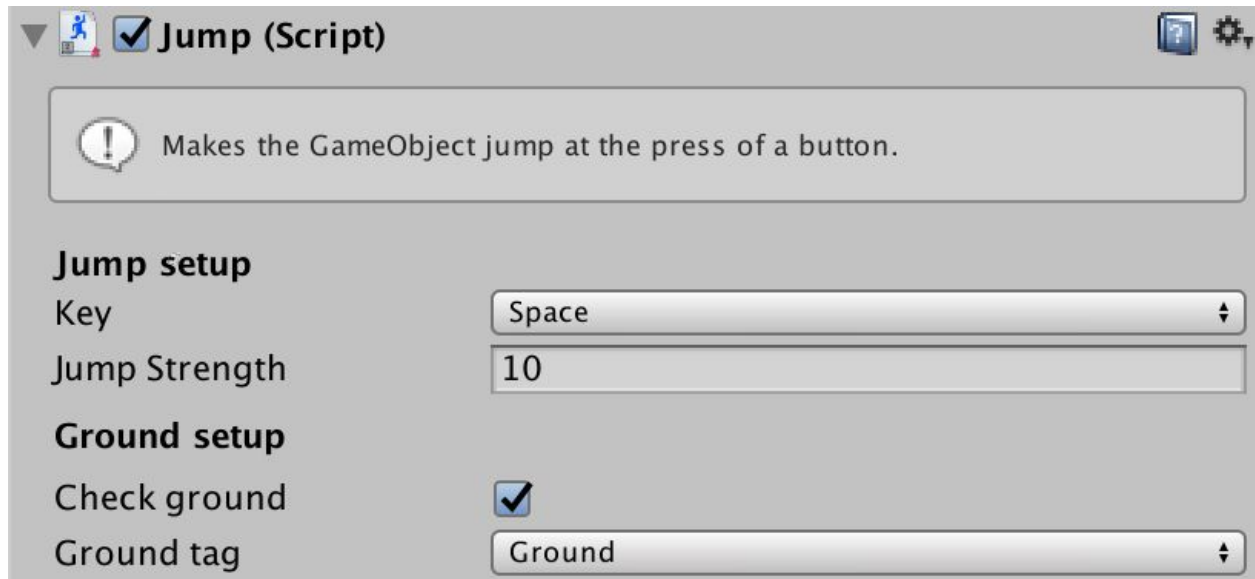
The **Look at target** option allows to select if the object orients itself to look at the target. See the [Move](#) script for more information on it.

Tip: You can assign this script to an enemy and use the player as a target to create a constant threat, or you can create a queue of characters by chaining them to each other with a series of Follow Target scripts.



Jump

Requires: Rigidbody2D (and a Collider to be able to land!)

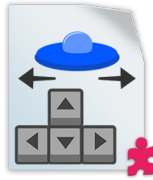


A simple script that propels an object upwards when a specific key is pressed, useful to create a jump behaviour. The **Key** property is the keyboard key that is used to jump.

To stop the player from jumping in the air, check **Check ground** and select a tag. Then, you need to tag anything you want to consider ground with that tag. As soon as the GameObject collides with the “ground”, it is able to jump again.

If **Check ground** is not checked, the character is able to jump multiple times in the air. That might be useful to create a wing flap more than a jump.

Tip: You might want to tune the **Jump Strength** property together with the **Friction** property of the Rigidbody2D, to obtain exactly the jump dynamics that you want.



Move

Requires: Rigidbody2D

The screenshot shows the Unity Inspector for the 'Move (Script)' component. At the top, there's a dropdown menu with a blue icon and a checkmark, followed by the text 'Move (Script)'. To the right of this are a question mark icon and a gear icon. Below this is a message box with an exclamation mark icon and the text: 'The GameObject moves when pressing specific keys. Choose between Arrows or WASD.' The main area is divided into three sections: 'Input keys' with a 'Type Of Control' dropdown set to 'Arrow Keys'; 'Movement' with a 'Speed' input field set to '5' and a 'Movement Type' dropdown set to 'All Directions'; and 'Orientation' with an 'Orient to direction' checkbox that is currently unchecked.

This script applies a constant force to the GameObject on two axes, while pressing either the Arrow keys or WASD.

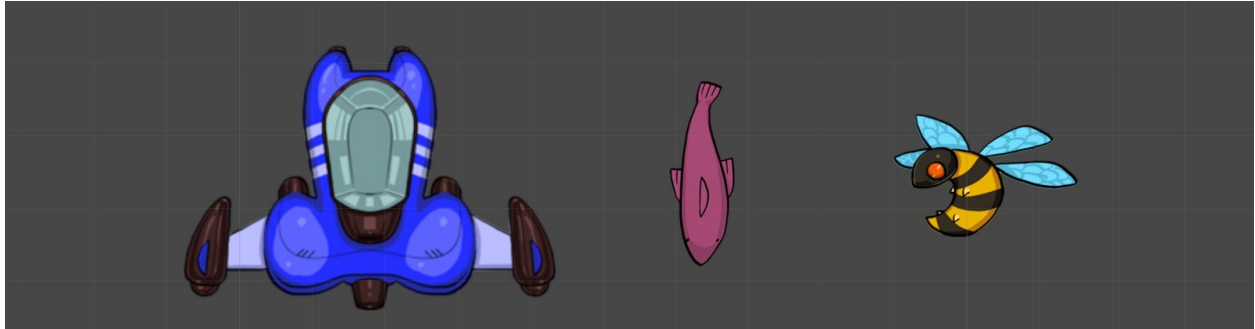
The **Type of Control** property assigns which control scheme to use. You can have two of these scripts in the scene and assign one to each player, to create multiplayer games played on the same keyboard.

The **Movement Type** property allows to restrict the movement on only one axis. You can think of it as if it's moving on a railing, but you can also combine it with other movement scripts to create more refined movement. For instance, you can create a platformer controller by using this script and forcing it to the horizontal axis, in conjunction with a Jump script.

Note: Keep in mind that even if the force you apply is on one axis, nothing is stopping the object from moving on the other axis as a result of a collision, so if this GameObject is hit by another one that might disrupt the gameplay. To account for it, check the **Freeze Position** option for the appropriate axis on the Rigidbody2D.

Orientation gives you control over whether the object should rotate to face the direction of travel. This is useful for vehicles (spaceships, cars, boats, etc) and in general Sprites that are seen from the top.

If it is enabled, you can choose which side is used as the forward direction with the **Use Side** property. This depends on how the Sprite has been drawn.







In the image above, for instance, you would set it to Up for the spaceship, to Down for the fish, while you would leave **Orientation** off for the bee because it's framed from the side and not from the top, so rotating the Sprite would look strange.



Patrol

Requires: Rigidbody2D

☒ **Patrol (Script)**

 The object moves through a series of positions. This can be used for patrolling characters.

Movement

Speed

Stops

= X

Y

= X

Y

+

-

Reset Waypoints

Orientation

Orient to direction ☒

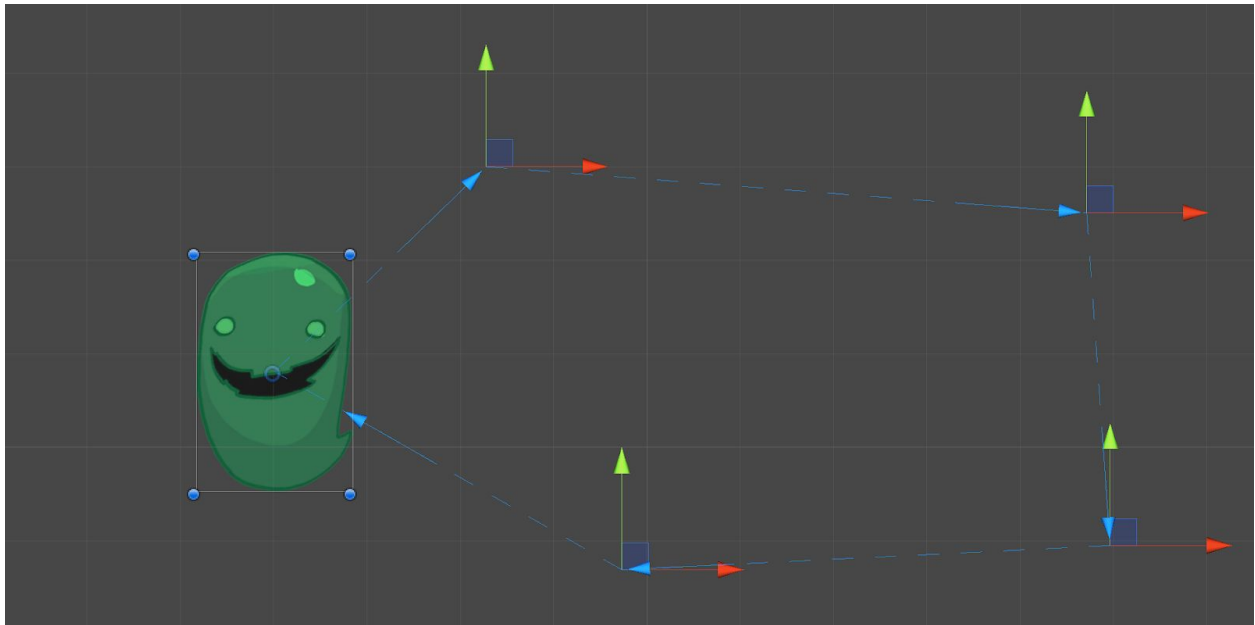
Look Axis

The Patrol script allows you to move an object along a path made of waypoints. The waypoints are organised in a list (**Stops**), so you can add, remove, or reorder them easily. The **Reset Waypoints** button clears the list and then adds only one stop.

The GameObject returns to the starting point once all the waypoints have been walked, and then restart on the path.

As with other Movement scripts, **Orientation** gives you control on how to orient your Sprite while moving. See its description in [Move](#) for more information.

As you create waypoints, they are visualised in the Scene View as Translate handles. You can move them around in here, or in the Inspector by changing the position values.

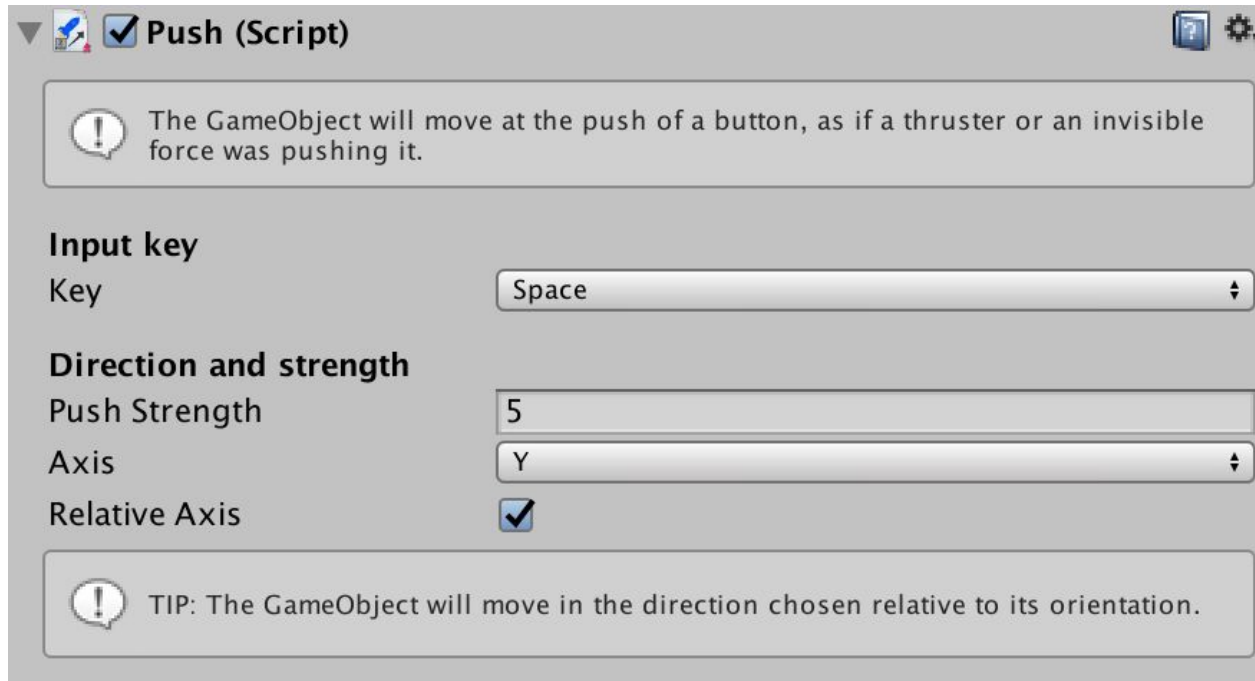


A little blue arrow marks the direction of movement.



Push

Requires: Rigidbody2D



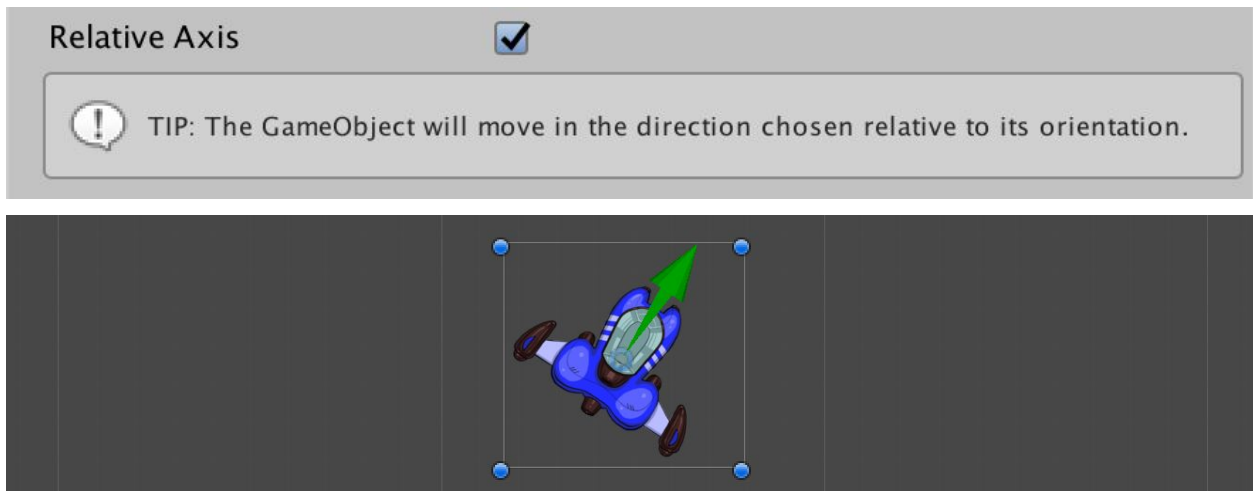
Push applies a continuous force in one direction when holding a specific key on the keyboard. It's useful to create a controller for vehicles, rockets, etc. and you can use it in combination with [Rotate](#) to allow steering.

In the Scene View, you will see a green arrow gizmo that shows you the direction and strength of the force (see below).

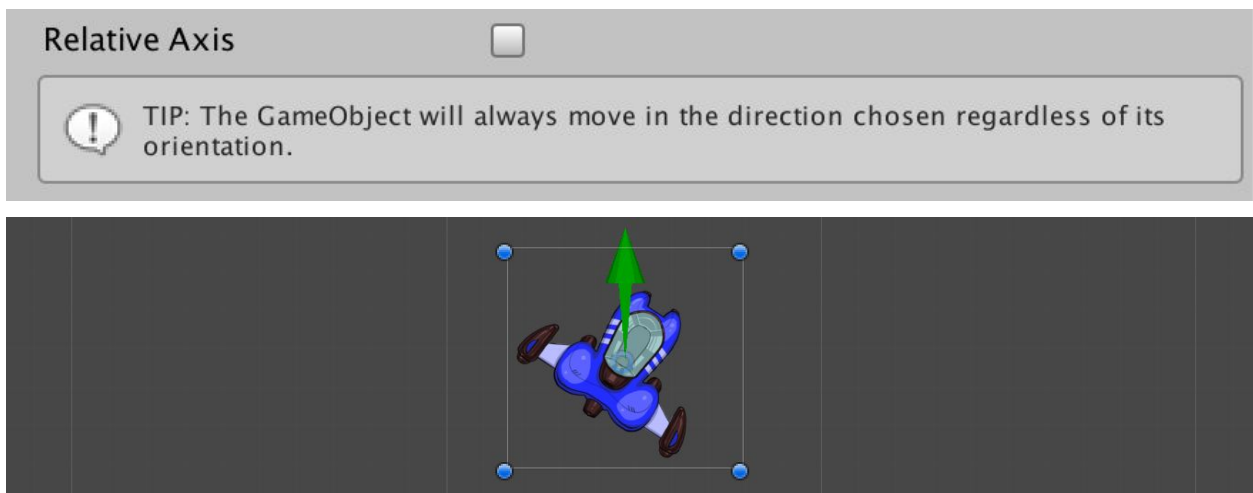
By setting the **Axis** property, you can control in which direction to push (Y means up, X means to the right). To achieve a push in the opposite direction, simply set **Push Strength** to a negative value.

In conjunction with **Axis**, the **Relative Axis** property controls whether the push "rotates" with the GameObject or whether it is applied in absolute terms. You will not see a change in the gizmo if the object has no rotation. To understand the difference, look at the images below.

With **Relative Axis** on, rotating the object means that the direction rotates with it (it's basically in Local Space):



When **Relative Axis** is off, the direction is absolute (basically in World Space):

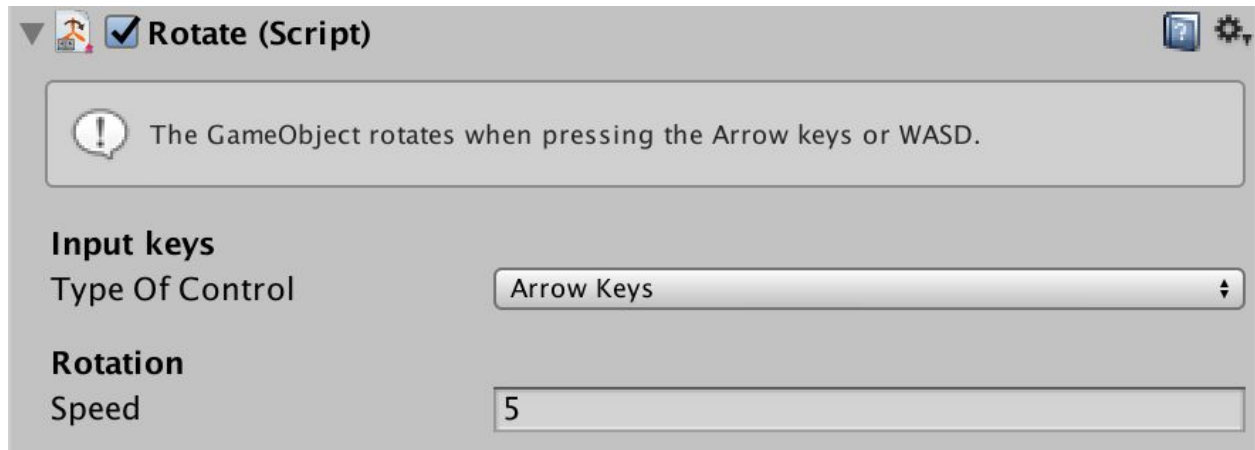


The tooltip below the option reflects the change. Most of the time you want to keep it on so that the vehicle moves on its forward direction.



Rotate

Requires: Rigidbody2D



Rotate is a script that applies a torque - that is, a rotation on the Z axis. Like [Move](#), this is controlled with the left/right Arrows, or AD keys. You can use it together with a [Push](#) script to create a vehicle-like controller, where you can steer and move forward in the direction the vehicle is pointing to.

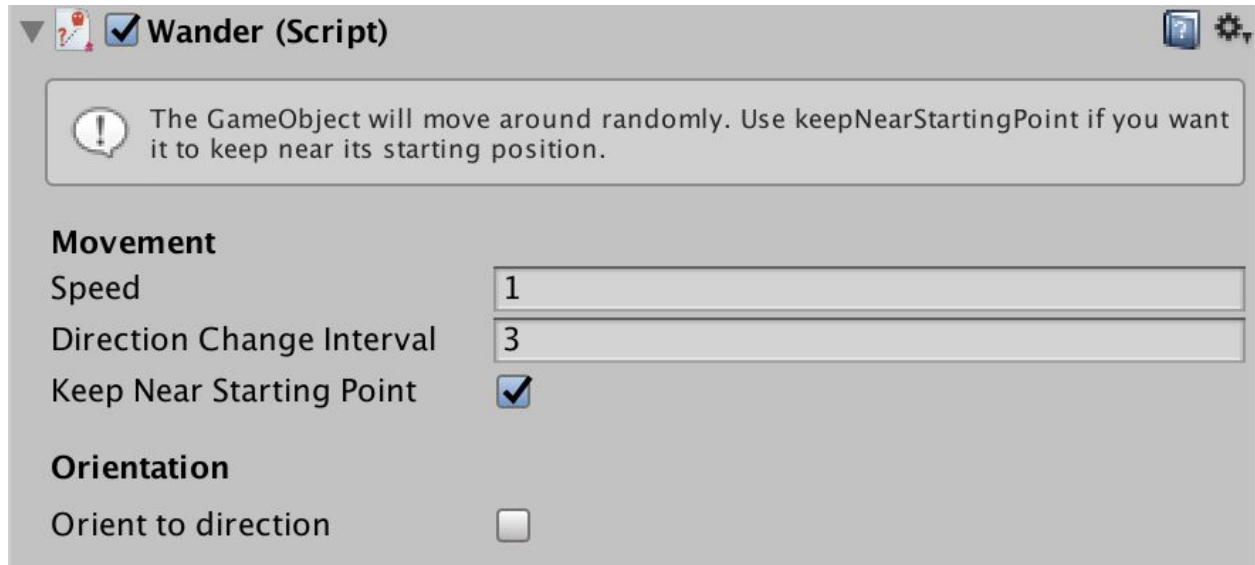
Tip: If you don't like the centre of rotation of your object, you can parent it to another GameObject and then apply the Rotate script to that one. For example, think of a bicycle, where the centre of rotation is on the back wheel.

This way you have much more control on the centre (gizmo), without needing to change it in the actual Sprite asset.



Wander

Requires: Rigidbody2D



With Wander, the GameObject randomly moves around in short bursts. The movement speed is controlled by **Speed**, which as usual goes together with the **Friction** property in the Rigidbody2D to make it feel right.

Direction Change Interval controls the timing (in seconds) after which the object goes in a new direction. Setting it very low means a lot of shorter, sudden movements.

Keep Near Starting Point means that the GameObject performs a check every now and then, and if it's straying too far, it heads towards the initial point in its next movement.

Note: If you set the **Speed** too high or there's not enough **Friction** on the Rigidbody2D, the object might still be able to wander off very far!

As with other Movement scripts, **Orientation** gives you control on how to orient your Sprite while moving. See its description in the [Move](#) script for more information.

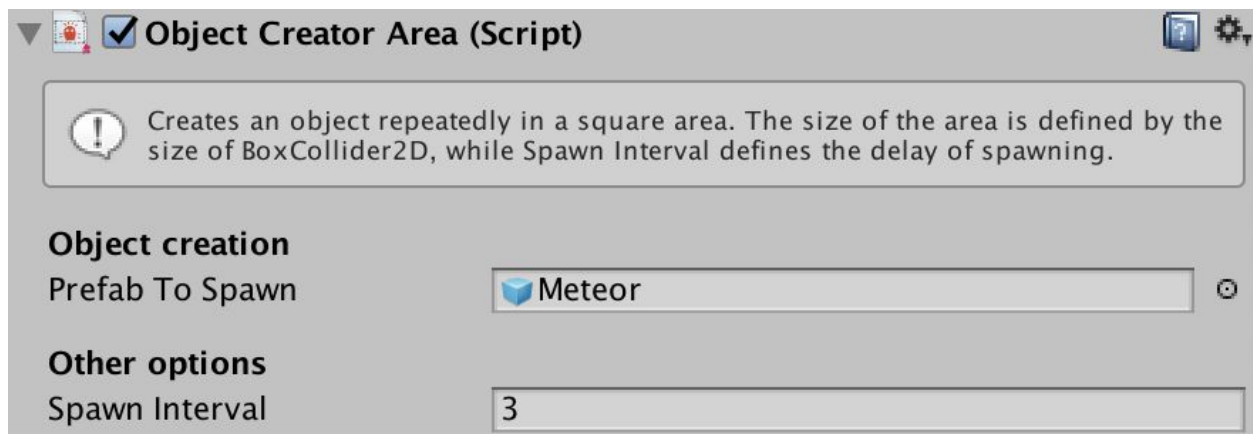
Gameplay scripts

Gameplay scripts are a miscellaneous category of scripts to produce gameplay effects. They act on their own, meaning they don't require [Conditions](#) to activate them.



ObjectCreatorArea

Requires: BoxCollider2D



ObjectCreatorArea is a script that generates new objects from a prefab, in a rectangular area. To work it requires a BoxCollider2D (applied automatically), which defines the area where the instances appear.




The **Prefab to Spawn** property can accept an object from the Scene, but it's good practice to create a prefab and assign that instead. If you assign a normal GameObject from the Scene, the component displays a warning.


The **Spawn Interval** property determines the interval between generated objects, and it's expressed in seconds.



ObjectShooter

Requires: nothing

 **Object Shooter (Script)**  

 Spawns an object at the press of a button and it applies a force to it in the direction chosen.

Object creation
Prefab To Spawn
Key To Press

Other options
Creation Rate
Shoot Speed
Shoot Direction X Y
Relative To Rotation ☒

ObjectShooter is a script that allows to propel or shoot a prefab when pressing a key (by default Space). You can use it to create weapons shooting projectiles or lasers, a tennis ball machine, or any kind of item that propels something out repeatedly. It works in conjunction with [BulletAttribute](#) (see below).

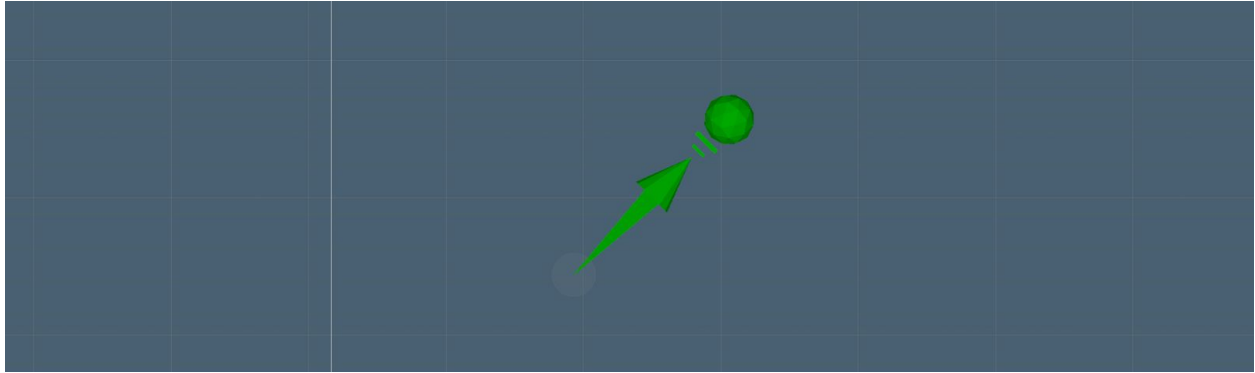
Often you want to assign this script to an empty object, parent it to another object that has some kind of graphics, and use it as a spawning point. This way you can control exactly where the projectile is emitted from.

The **Prefab to Spawn** property can accept an object from the Scene, but it's good practice to create a prefab and assign that instead. If you assign a normal GameObject from the Scene, the component displays a warning.

Creation Rate controls the interval between shots, and it's in seconds. **Shot Speed** determines the speed, and **Shot Direction** the direction as a Vector2.

Relative to Rotation allows the direction mentioned above to be rotated with the GameObject, so if (for instance) you use this script on an object like a spaceship that rotates, the shooting direction is realistic. Leaving it off means the shooting direction is in World Space.

When you apply this script, a green arrow gizmo will be visualised in the Scene View.



The size of the arrow is not connected to the strength of the shot.

Note: You can also set the shooting speed to zero. This way the Player can leave behind objects as it moves. Don't get fooled by the name of the script which mentions "shooting": the generated object doesn't need to be a projectile!

Projectile IDs

Finally, ObjectShooter also has the ability to assign a player ID to the projectiles. This works in conjunction with the [BulletAttribute](#) script, which needs to be assigned to the projectile prefab that you want to shoot out. Doing so means that if the projectile hits another object that has the [DestroyForPointsAttribute](#) script, points are assigned to the correct player.

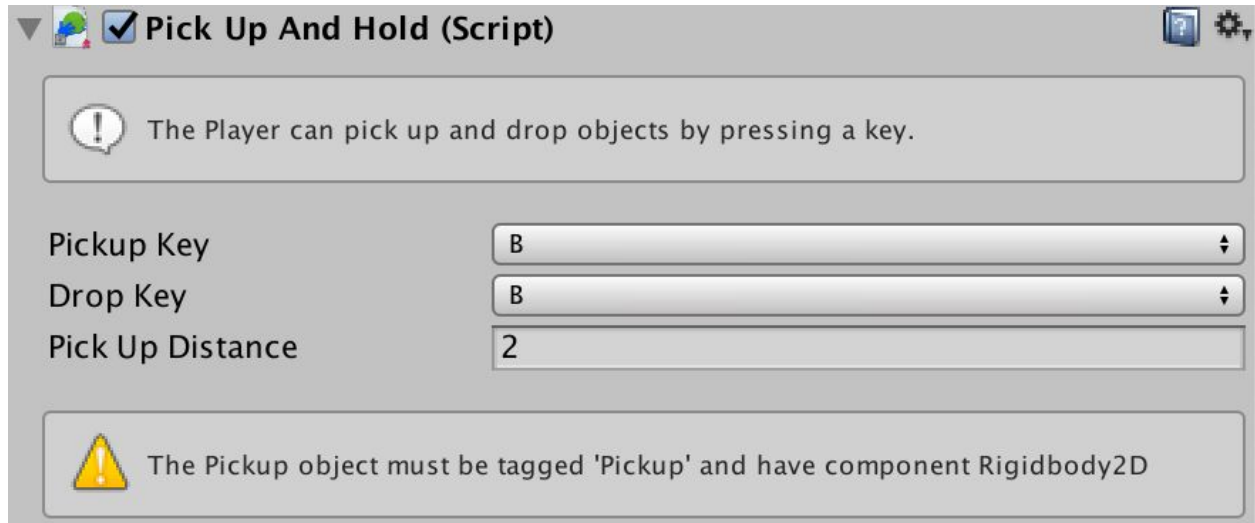
To make sure ObjectShooter assigns the correct ID, you need to tag the GameObject as Player or Player2.

Note: If you don't tag the GameObject, the bullet is considered as coming from Player1, so for single-player games you don't need to worry about tagging.



PickUpAndHold

Requires: nothing



This script is used to give a character the ability to pick up (and drop) something, like an item or a ball in a sports game. Coupled with clever use of a [ConditionArea](#) script, you can create gameplay that revolves around picking up an object and delivering it to a specific area.

To make an object “pickable”, you need to assign it the Pickup tag and give it a Collider2D of any kind. You probably want to make the Collider a trigger, otherwise the object collides with the player and is harder to pick up. If the object has a Rigidbody2D, it is made kinematic and stopped before being parented to the character.

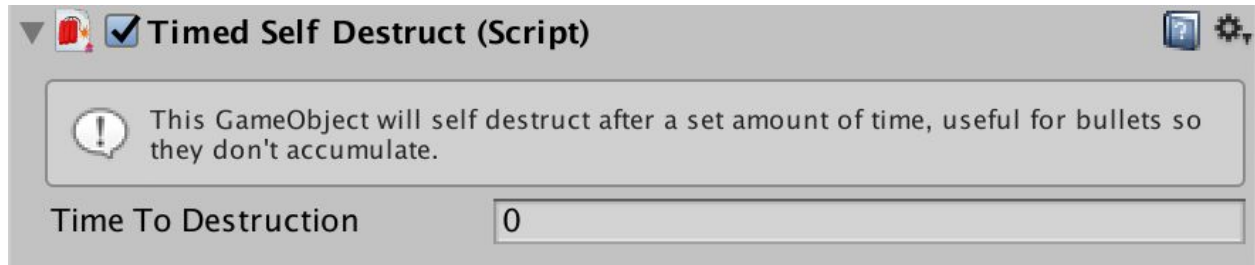
Pickup Key and **Drop Key** define which key needs to be pressed to pickup and drop the item. They match by default, but they don’t need to.

The **Pick Up Distance** defines the maximum distance that the object can have from the player to be picked up. If multiple pickup objects are present, the closest is picked up.



TimedSelfDestruct

Requires: nothing



TimedSelfDestruct is a utility script to get rid of objects that are no longer needed in the scene after a certain time. **Time to Destruction** is the time in seconds after which the object disappears.

Use it on projectiles, and any generated object to make sure the scene doesn't get cluttered. Also, if your game is generating a lot of objects, in time it might slow down. Putting TimedSelfDestruct on those objects - even if the timer is high - helps the game run better.

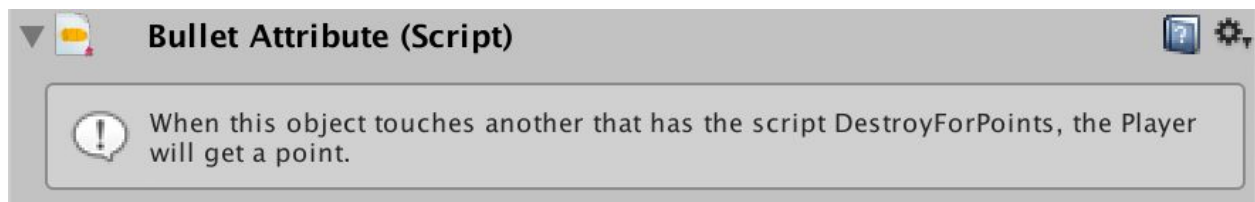
Attributes

Attributes are a category of scripts that usually don't do much on their own: they just define qualities that object have, and then some other script is going to act based on those Attributes. They have a role similar to Tags, but being scripts they can come with extra data.



BulletAttribute

Requires: Collider2D of any shape



The Bullet script has no functionality of its own, but it holds a reference to which Player has shot the projectile. This number (**playerID** in the code) can be 0 (player 1) or 1 (player 2).

The ID is automatically set by the [ObjectShooter](#) script when the projectile is launched.

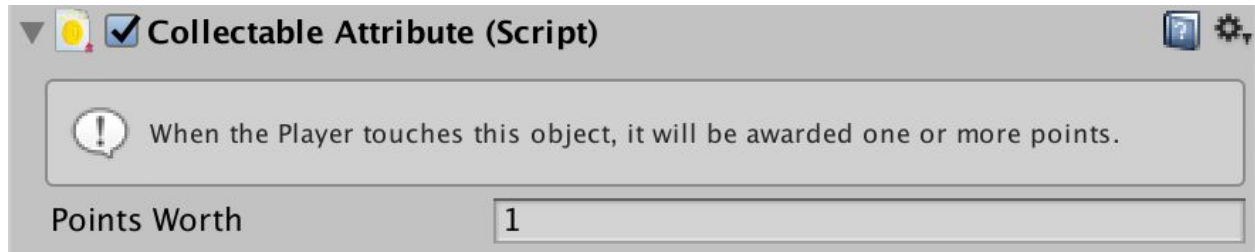
Note: If for any reason you want to set the **playerID** property in the Inspector, open the BulletAttribute script and remove the attribute [HideInInspector] from the property.

This could be useful in case the projectiles are created by a script that is not ObjectShooter.



CollectableAttribute

Requires: Collider2D of any shape



CollectableAttribute awards a point to any player who touches the object. As such, it requires a Collider2D and potentially that this is marked as a Trigger to avoid collisions.

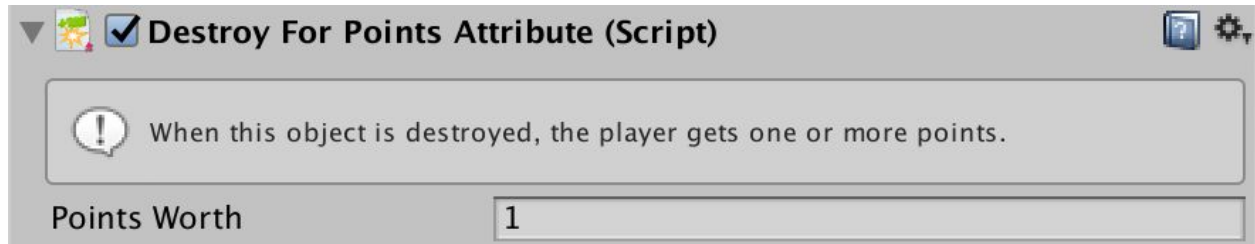
Its only property, **Points Worth**, allows to assign a different value to each object.

Note: To see the total score on screen, a UI prefab needs to be present in the Scene. Refer to the Getting Started manual for more information on the UI prefab.



DestroyForPointsAttribute

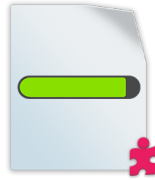
Requires: nothing



DestroyForPointsAttribute is good for targets and enemies in shooting games. It destroys the GameObject on collision with another object, only if this has the [BulletAttribute](#) script. Also, it awards points to the Player who originally shot the bullet.

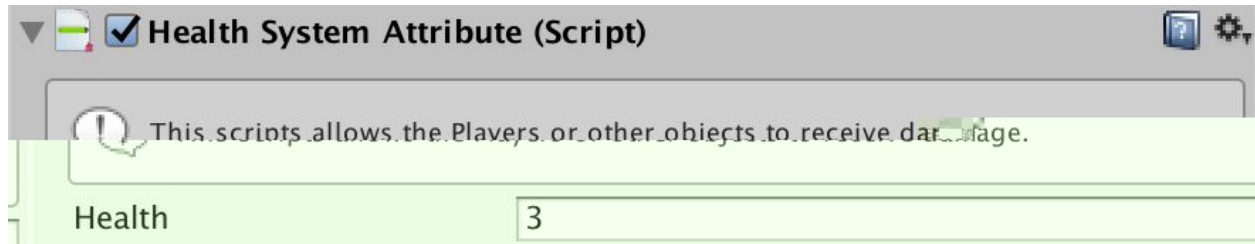
See [BulletAttribute](#) and [ObjectShooter](#) scripts for more information on how IDs are assigned to bullets.

Note: To see the total score on screen, a UI prefab needs to be present in the Scene. Refer to the Getting Started manual for more information on the UI prefab.



HealthSystemAttribute

Requires: nothing



The HealthSystemAttribute can be added to characters, enemies or objects. It allows them to take damage, and potentially be removed from the game if this goes to 0 - which in the case of the Player it might mean Game Over.

The script works in conjunction with the [ModifyHealthAttribute](#), which is necessary even on bullets. Similarly, an object with ModifyHealthAttribute doesn't affect an object that doesn't have HealthSystemAttribute.

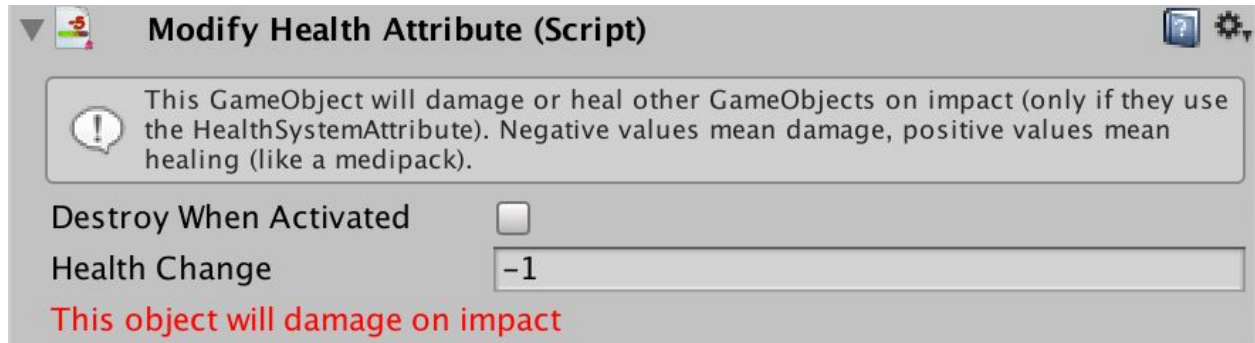
HealthSystemAttribute doesn't enforce the presence of a Collider2D, but it's recommended to have one if you are relying on collisions to subtract health from your player.

Note: To see the player's health on screen, a UI prefab needs to be present in the Scene. Refer to the Getting Started manual for more information on the UI prefab.



ModifyHealthAttribute

Requires: a Collider2D of any shape



This attribute makes any physical object capable of subtracting or adding health to any object that has the [HealthSystemAttribute](#). It is useful for bullets, hazard zones, etc. but also for things that can heal, like medipacks, food, and more.

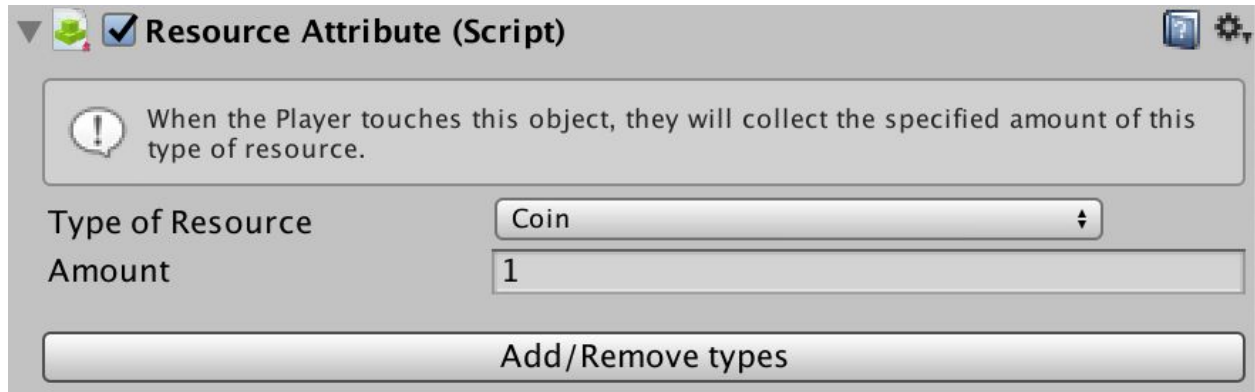
Destroy When Activated is a property that, when checked, removes this object the first time it produces its effects. Use it for bullets, consumables, and anything that has to act only once.

The **Health Change** property indicates the change in health that this object produces. If it's negative, it's damaging. If positive, it is healing. The last line in the Inspector can turn red or blue depending on which "mode" you are on.



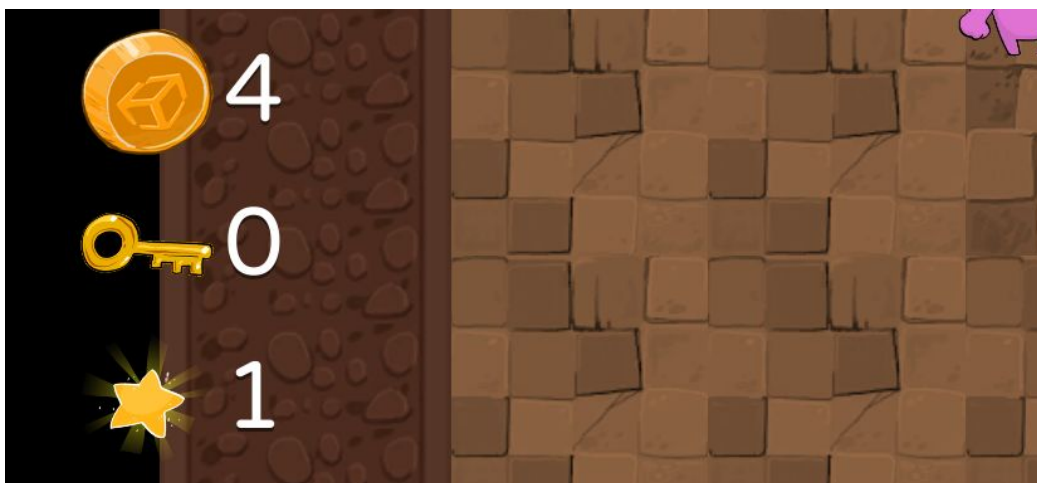
ResourceAttribute

Requires: a Collider2D of any shape, and a SpriteRenderer



ResourceAttributes, together with [ConsumeResourceAction](#), opens the ability to include resources and an inventory to games made with Playground. Touching an object that has this script picks it up, and adds it to an inventory displayed by the UI.

The script requires a SpriteRenderer since the Sprite is used to display an icon in the lower-left corner, alongside a number representing the amount of that resource the player has:



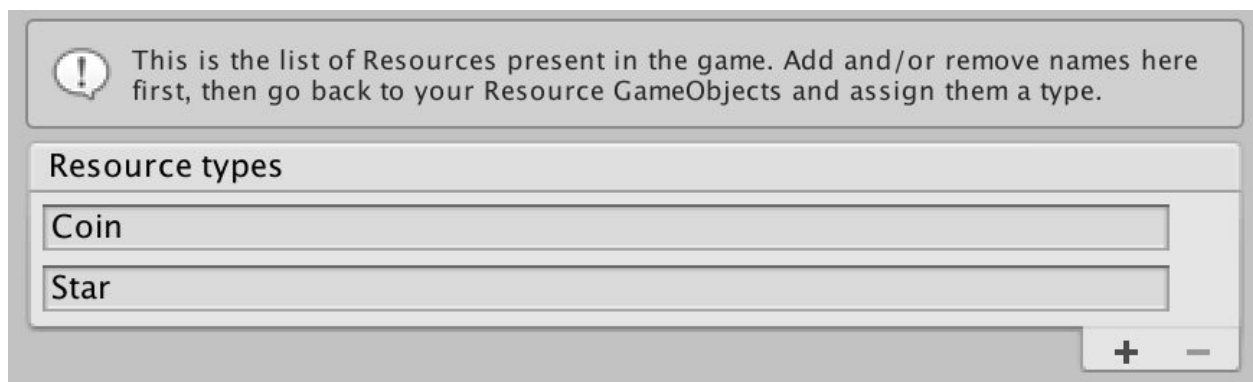
Picking up more resources of the same type adds up to the number. A [ConsumeResourceAction](#) script can then request and consume them. See its description for more information.

With resources, you can create a crafting system (collect resources, then use them to “pay” and get an item in return) or simply a door/key system, where each door needs the right key to open.

You define which type this resource belongs to with the **Resource of Type** property (see below), and optionally you can specify an **Amount** (useful for coins and money).

Defining resource types

Resource types can be defined by clicking on the **Add/Remove types** button. This focuses the editor on a ScriptableObject called “InventoryResources”. On this object, a list of strings defines all the available types of resources in the game.



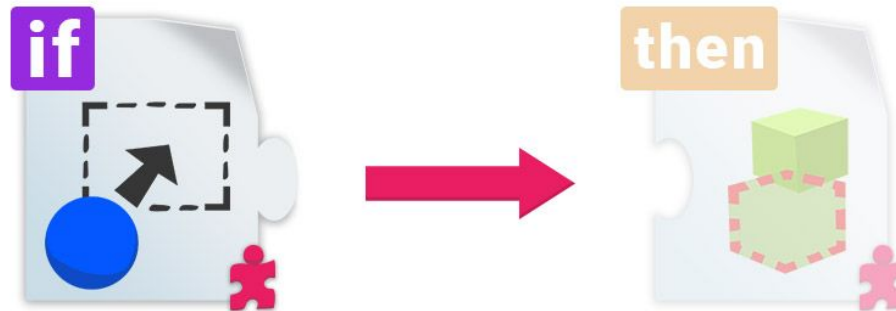
Several have already been added for your convenience, but you can add and remove more to create more custom gameplay. Once you add a resource type, be sure to go back to the object that has the ResourceAttribute script and assign it.

Note: Resource types are shared between all scenes, so if you remove the base types (Coin, Star, etc.) some Example scenes like Roguelike will stop working.

Moving the location of the “InventoryResources” object will also break the system.

Conditions

Conditions are very similar to If Statements in programming, meaning that they act as a gateway to other behaviours. If the condition is verified, then the attached Actions are executed. See [Action](#) scripts for more details on what Conditions can execute. Condition scripts have a little purple “if” tag at the top-left corner of their icon.



All Condition scripts have some common properties which are described below.

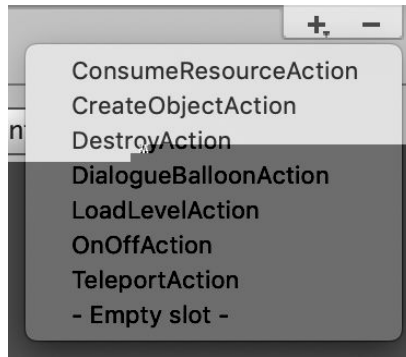
Gameplay Actions

The **Gameplay Actions** is a list of [Action](#) scripts that are executed when the requirement of this Condition script is met. Once that happens, Unity executes the Actions in this list until one of them fails (only a few Action can fail). If the Action doesn't fail, then the next one is executed. At the end of the list, **Custom Actions** (if present, see below) are carried out.

A list of empty Actions would look like this:



Pressing the plus icon reveals a list of the Actions available to add:



They correspond exactly to the scripts in the Action category. If you select any of them, the script is added as a component and automatically connected to the list.

Note: Action scripts need to be connected in the **Gameplay Actions** list to work. Simply adding an Action script to a GameObject will not execute it.

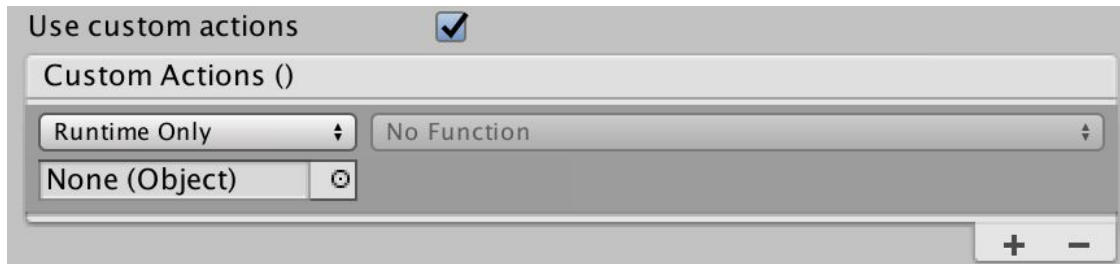
Tip: At the end of the dropdown, you also have an option to add an empty slot. That's really useful to be able to connect an Action present on another GameObject. This way you can break your logic into parts, making it easier to manage.

To see this in practice, open the Roguelike example game and select the object named InvisibleTrigger. Notice how 2 of the Actions are on separate objects parented to this one.

Custom Actions

Similarly to Gameplay Actions, **Custom Actions** are executed when the requirement of the Condition script is met. It's not necessary to use them, so if **Use Custom Actions** is unchecked, no custom action is executed.

Checking the option reveals a list of Unity Events, in which you can connect here anything you would normally use an UnityEvent for.



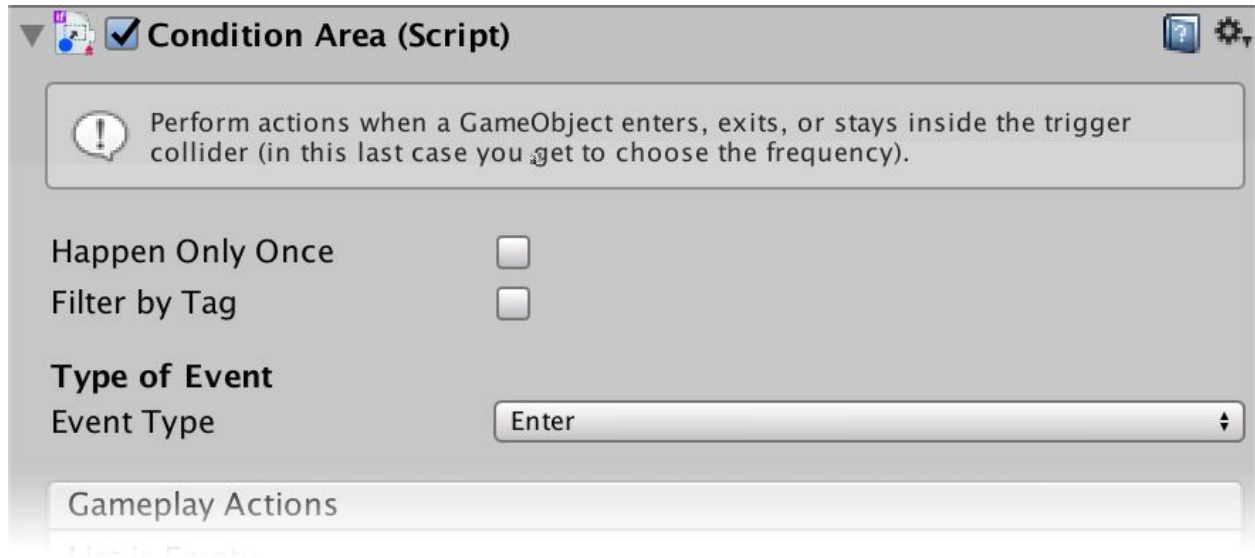
Because of this, they really allow you to expand the gameplay aspect of your game: if you know programming, you can write a simple script and expose a public function, and then connect it in here. This way your script can be called by one of the default Condition scripts of the Playground. This is a great way for teachers to quickly extend the functionality of the Playground during a workshop.

Another use would be to connect in here standard Playground public functions. A good example is the UI Script and its GameWon and GameOver functions. By wiring one of them into the Custom Actions list of a Condition, you can for instance create a winning condition when an object collides with something else, or enters an area.



ConditionArea

Requires: a Collider2D set as Trigger



ConditionArea requires a Collider2D set as trigger. **Event Type** determines when the event occurs: on entering the area, on leaving it, or while staying inside (if you select this last one, a **Frequency** parameter is revealed to determine how often it happens).

Because you probably don't want any object to trigger the event, **Filter by Tag** allows to restrict the event happening only if a specific category of objects enters the area. It is common to use Player, but you can use whatever you prefer.

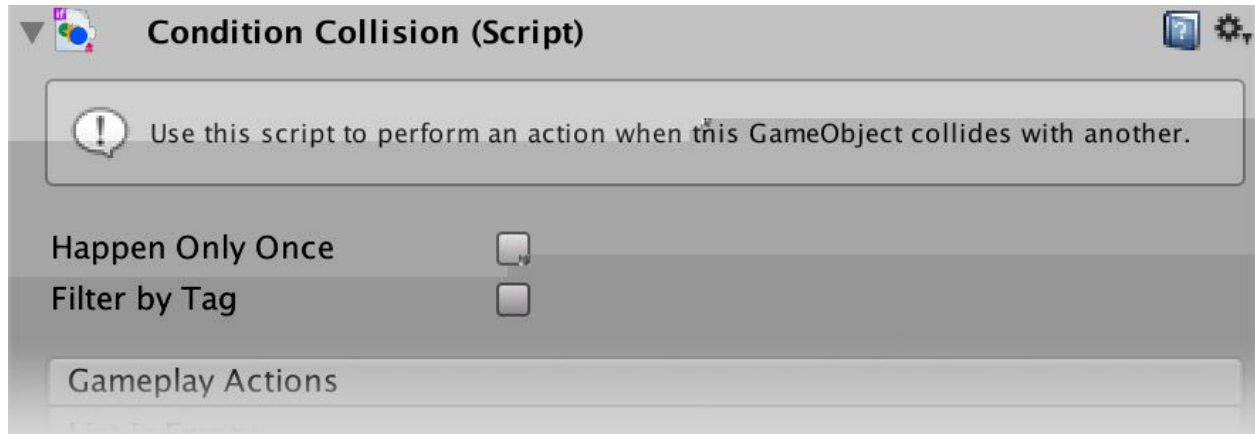
Happen Only Once allows the condition to be ignored after the event has happened once.

Gameplay Actions and **Custom Actions** are common to all Conditions, so you can find more info in the general sections: [Gameplay](#) / [Custom](#).



ConditionCollision

Requires: nothing



ConditionCollision is a simple condition to make something happen when an object collides with the one that has this script. As with other conditions, you can filter which type of object actually produces the event by setting a tag in **Filter by Tag**.

Happen Only Once allows the condition to be ignored after the event has happened once.

Gameplay Actions and **Custom Actions** are common to all Conditions, so you can find more info in the general sections: [Gameplay](#) / [Custom](#).



ConditionKeyPress

Requires: nothing

The screenshot shows the 'Condition Key Press (Script)' configuration window. It has a title bar with a dropdown arrow, a checkmark icon, and a settings icon. Below the title bar is a message box with an exclamation mark icon and the text: 'Use this script to perform an action when a button is pressed, released, or as long as it's kept pressed (in this case you get to choose the frequency)'. The main area contains two settings: 'Happen Only Once' with an unchecked checkbox, and 'Key To Press' with a dropdown menu showing 'Space'. Below these is the 'Type of Event' section, with 'Event Type' set to 'Just Pressed' in a dropdown menu. At the bottom, there is a 'Gameplay Actions' section with a list box that is currently empty.

ConditionKeyPress is a generic way of binding an Action to a key press. In addition to choosing which key with **Key to Press**, you can choose which type of key event to listen to with **Event Type**: Just Pressed (similar to GetKeyDown), Released (GetKeyUp) or Kept Pressed (GetKey). As with other continuous actions, Kept Pressed mode has a **Frequency** property.

Happen Only Once allows the condition to be ignored after the event has happened once.

Gameplay Actions and **Custom Actions** are common to all Conditions, so you can find more info in the general sections: [Gameplay](#) / [Custom](#).



ConditionRepeat

Requires: nothing

▼ ☒ Condition Repeat (Script) ? ⚙

! Use this script to perform an action repeatedly.

Initial Delay 5

Frequency 1

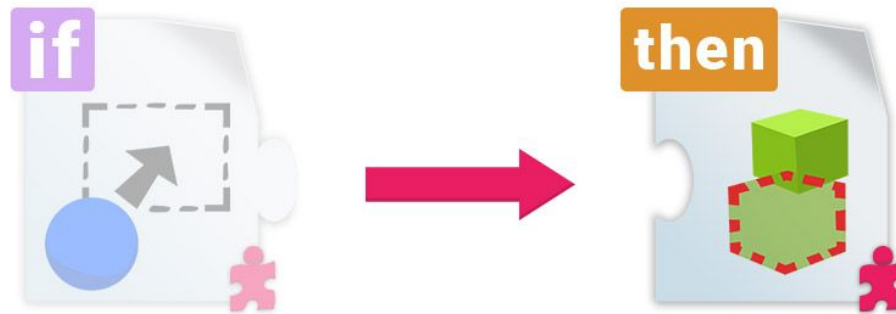
Gameplay Actions

To make a programming comparison, ConditionRepeat is more like a WHILE than an IF. It executes the list of Actions repeatedly without user input or interaction. It supports a **Frequency**, and an **Initial Delay**.

Gameplay Actions and **Custom Actions** are common to all Conditions, so you can find more info in the general sections: [Gameplay](#) / [Custom](#).

Actions

Actions are scripts that don't work on their own, but need to be executed by [Conditions](#). Only if the Condition is verified, the Action is executed. You can recognise Action scripts because they have a little yellow "then" tag at the top-left corner of their icon.



Action scripts all feature the concept of "success". It means that some actions can fail, and if they do, the Condition that is playing them will interrupt the chain of Actions and stop.

Adding and removing Actions

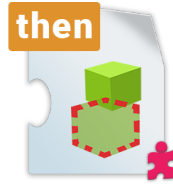
Actions can be added like any other regular component, but on their own they won't do anything - they always need to be connected to a Condition.

For this reason, it makes sense not to add Actions in the usual way but to use the dropdown menu at the bottom of Conditions' **Gameplay Actions** list. This both adds the Action component and connects it to the list. Similarly, the minus icon both removes the item in the list and the component from the GameObject, leaving it clean.



If you want to place an Action on another object that has the Condition, simply add the Action to it as a normal component, then go to the list and use the last option, "Empty Slot". Finally, you need to drag the GameObject that has the Action onto the list slot that you just added.

More info in the [Gameplay Actions](#) section of Conditions.



ConsumeResourceAction

Requires: nothing

Consume Resource Action (Script)

Use this script to check if the player has enough of a specific resource. If they have it, it will be removed from the Player's inventory.

Type of Resource: Coin

Amount Needed: 1

Add/Remove types

ConsumeResourceAction is an action that acts only if a certain condition is verified: the specified **Type of Resource** needs to be present in the player's inventory in the quantity specified in **Amount Needed**.

If this is true, that amount of resources is consumed and the following Actions are executed. If false, then no resource is consumed and the list of Actions is stopped (meaning any following Action is not executed).

Like in the [ResourceAttribute](#) script, the **Add/Remove types** button allows you to define resource types. See [Defining resource types](#) for more info and a rundown of what resources mean.



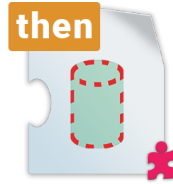
CreateObjectAction

Requires: nothing



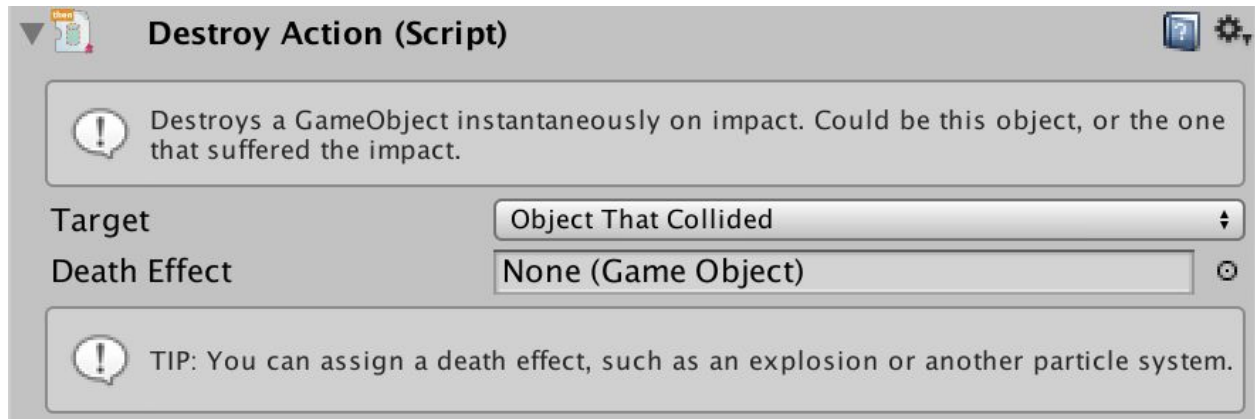
CreateObjectAction generates a new object from a prefab (**Prefab to Create**).

To decide where the new object is created, you can use **New Position**, which initially is in World Space (meaning 0,0 is the origin). When **Relative to this Object** is checked, **New Position** can be considered Local Space.



DestroyAction

Requires: nothing



DestroyAction can be used to remove objects from the game.

The **Target** property can have two values: This Object (pretty self-explanatory) and Object That Collided. When using the latter, this Action needs to be connected to either a [ConditionArea](#) or [ConditionCollision](#), or it will fail.

You have the option to specify a **Death Effect**, that is another object that gets generated when the **Target** object is destroyed. This could be a particle system, or other objects (like debris, a broken version of the object being destroyed, etc.).



DialogueBalloonAction

Requires: nothing

Dialogue Balloon Action (Script)

Use this script to create a dialogue balloon on a character's head.

Contents

| | |
|------------------|--|
| Text To Display | Hey! |
| Background Color | <div style="background-color: blue; width: 100px; height: 15px;"></div> |
| Text Color | <div style="background-color: white; width: 100px; height: 15px;"></div> |

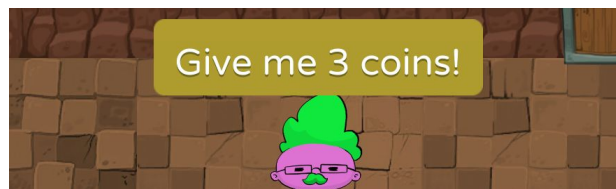
Options

| | |
|----------------|------------------|
| Target Object | None (Transform) |
| Disappear Mode | Button Press |
| Key To Press | Return |

Continue dialogue

| | |
|----------------|--------------------------------|
| Following Text | None (Dialogue Balloon Action) |
|----------------|--------------------------------|

The DialogueBalloon script allows to put simple dialogues in the game. You can see an example of it in the Roguelike example scene.



The first block of properties, **Text to Display**, **Background Color** and **Text Color** are pretty self-explanatory.

Target Object, if set, allows the text to appear above a character or an object. If it's not set, the text just appears in the middle of the screen.

Disappear Mode allows for two values: Button Press requires the user to press a key (**Key to Press**) to remove the dialogue, while with Time the dialogue disappears after the seconds specified in **Time to Disappear**.

Regardless of how the dialogue is removed, you can connect another DialogueBalloonAction in the last slot, **Following Text**, to create a continuous dialogue.

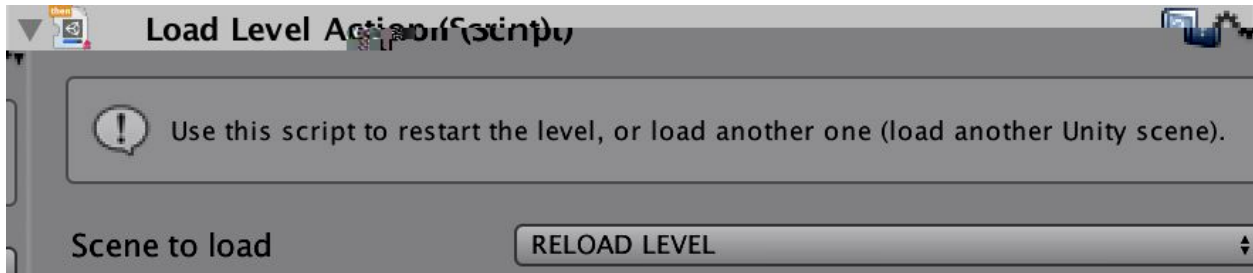
Tip: Use chained DialogueBalloonAction scripts with no target and in Button Press **Disappear Mode** to create small tutorials for your game.

Or, by chaining several DialogueBalloonActions and focusing them on different characters with the **Target Object** property, you can create conversations between two or more characters.



LoadLevelAction

Requires: nothing



LoadLevelAction adds the ability to load Unity scenes from Conditions.

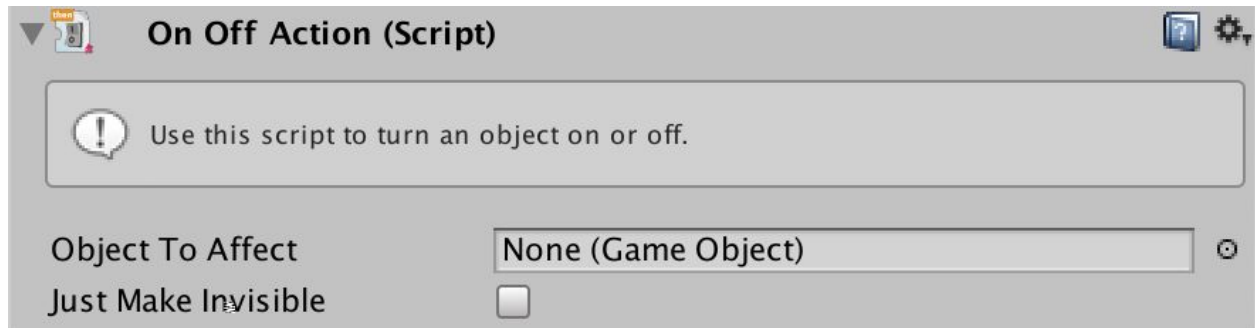
The **Scene to Load** property displays a dropdown menu which includes all of the scenes that have been added to the Build Settings menu (File > Build Settings...). To be loaded, a scene needs to be added to the list and also be enabled.

The first item, "RELOAD LEVEL", just reloads the current scene so it's useful to reset the state of the game after game over.



OnOffAction

Requires: nothing

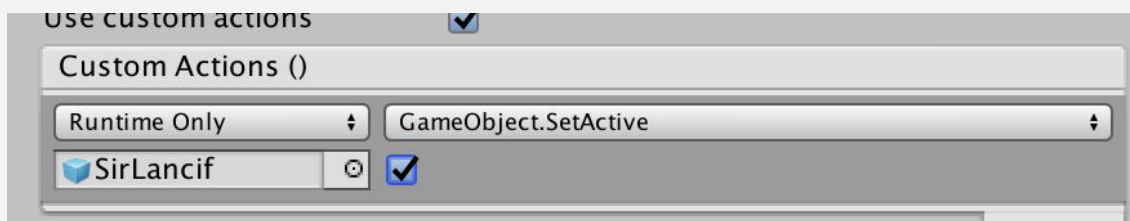


OnOffAction is a simple action to turn an object on and off, meaning setting its Active flag to true or false. You need to select the target in **Object to Affect** in order for this to work.

Just Make Invisible allows you to turn on/off a SpriteRenderer instead, meaning the object is still part of the gameplay, including any collision events it might have.

OnOffAction always “flips the switch”, setting the active flag to its opposite. This means the second time the Action is executed on the same object it restores its previous state, and so on.

Tip: If you want to set an object on/off in an absolute way (meaning the second time there will be no effect), you can use just a regular UnityEvent, enabling [Custom Actions](#) on the Condition and selecting SetActive on the target GameObject:





TeleportAction

Requires: nothing

The screenshot shows the 'Teleport Action (Script)' configuration window. At the top, there is a title bar with a small icon and a gear icon. Below the title bar, there is a warning box with an exclamation mark icon and the text: 'Use this script to teleport this or another object to a new location.' Below this, there is a label 'Object To Move' followed by a dropdown menu showing 'None (Game Object)'. Below the dropdown, there is another warning box with an exclamation mark icon and the text: 'WARNING: If you don't assign a GameObject, this GameObject will be teleported!'. Below this, there are two input fields for 'New Position', labeled 'X' and 'Y', both containing the value '0'. At the bottom, there is a label 'Stop Movements' followed by a checked checkbox.

TeleportAction moves an object instantly to a new location. If nothing is assigned in **Object to Move**, the same object that has the script is teleported. The **New Position** property is in World Space.

For objects that have a Rigidbody2D, enabling **Stop Movements** means that in addition to be teleported, they are also stopped, meaning their speed and torque are zeroed - which is good for resetting the game state (for instance, after scoring in a sports game).