The Explorer: 2D

Components Documentation

Introduction

The 2D Game Kit allows you to create 2D platformer gameplay and puzzles in Unity without code. The following documents each component you can use in the Kit to create your game. You will find definitions for what the components do and what the settings in those components are.

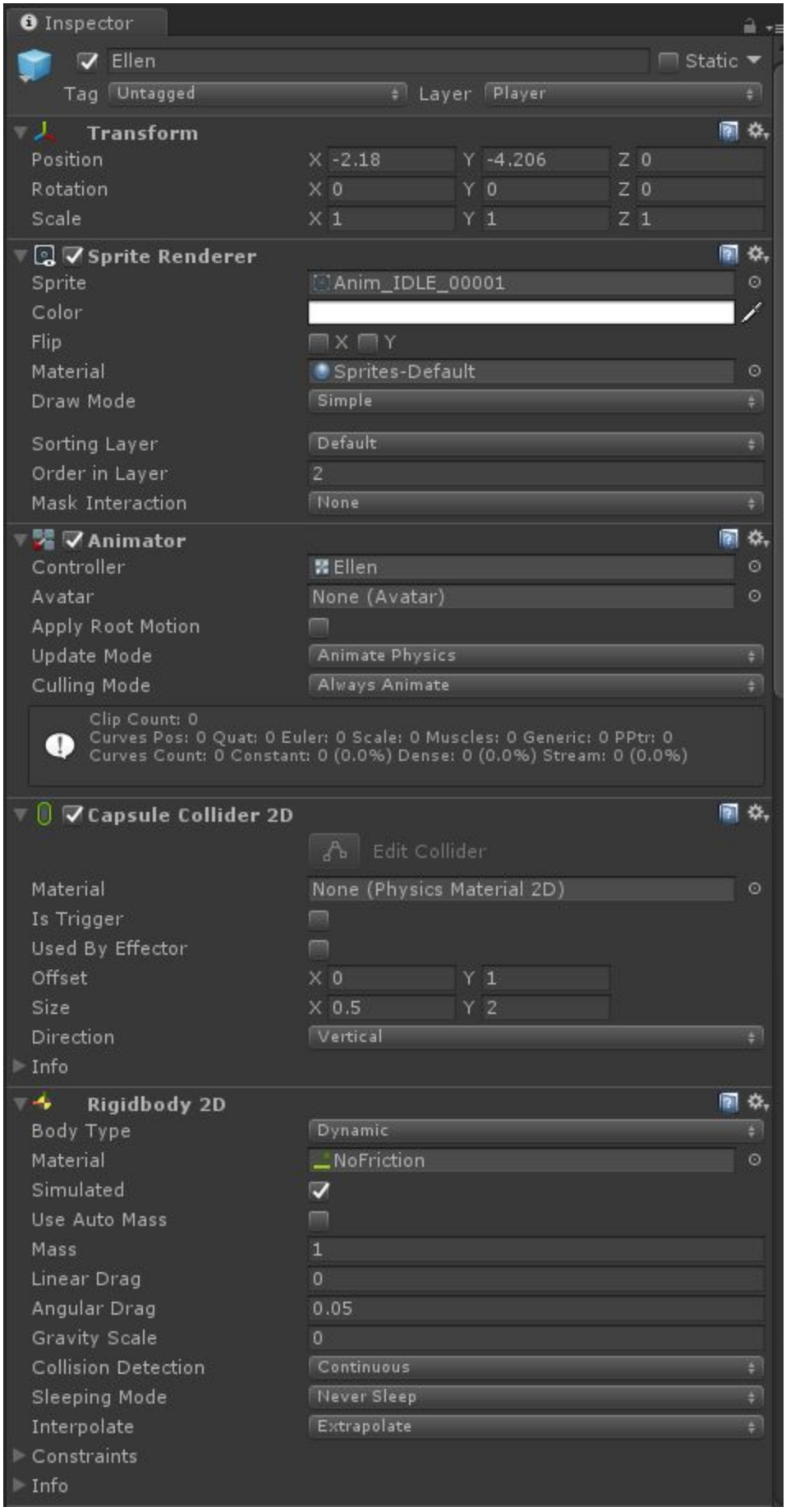
If you are new to Unity, we recommend you go through our Interactive Tutorials to familiarise yourself with the Unity interface and concepts. More information on Unity can be found on our Learn Site.

How to use this document

The best way to use this document is like a reference guide, searching by component or setting you want to know more about. To learn the basics of how to use the Game Kit, check the Getting Started Guide which will provide a good foundation on using the Kit.

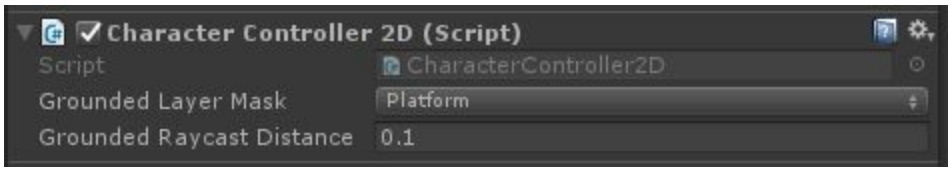
The Explorer example game included in this project uses all the components we provided for gameplay. To get inspiration or an example of how to use an object, check Zone 1 through 5 in the Scenes folder.

Important terms and concepts are highlighted with links if you would like to find out more information about them.


Inspector
Ellen
Static
Tag Untagged
Layer Player
Transform
Position X -2.18 Y -4.206 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Sprite Renderer
Sprite Anim_IDLE_00001
Color
Flip X Y
Material Sprites-Default
Draw Mode Simple
Sorting Layer Default
Order in Layer 2
Mask Interaction None
Animator
Controller Ellen
Avatar None (Avatar)
Apply Root Motion
Update Mode Animate Physics
Culling Mode Always Animate
Clip Count: 0
Curves Pos: 0 Quat: 0 Euler: 0 Scale: 0 Muscles: 0 Generic: 0 PPtr: 0
Curves Count: 0 Constant: 0 (0.0%) Dense: 0 (0.0%) Stream: 0 (0.0%)
Capsule Collider 2D
Edit Collider
Material None (Physics Material 2D)
Is Trigger
Used By Effector
Offset X 0 Y 1
Size X 0.5 Y 2
Direction Vertical
Info
Rigidbody 2D
Body Type Dynamic
Material NoFriction
Simulated
Use Auto Mass
Mass 1
Linear Drag 0
Angular Drag 0.05
Gravity Scale 0
Collision Detection Continuous
Sleeping Mode Never Sleep
Interpolate Extrapolate
Constraints
Info

Character Controller 2D

The **Character Controller 2D** script is used to move Ellen within the scene while obeying physics.



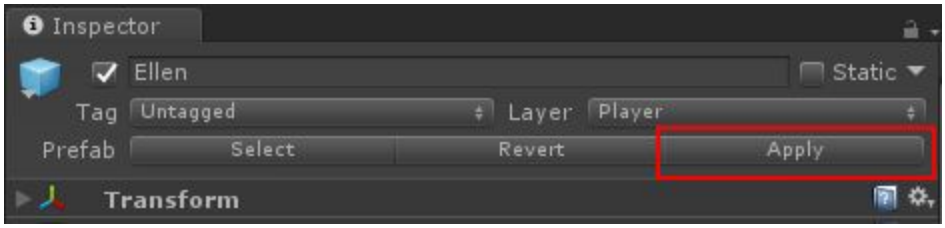
- **Grounded Layer Mask**:
  Default Setting: Platform
  These are the layers which the character can stand on. Having it set to the **Platform** layer allows Ellen to walk on anything else that is on that layer such as the ground.

- **Grounded Raycast Distance**:
  Default Setting: 0.1
  This determines whether Ellen is standing on the ground by using _____ from the bottom of her Collider. Increasing this number would make Ellen think the ground is higher and her jump landing will be affected. Lowering this number will make Ellen think the ground is lower and she will look like she is falling.

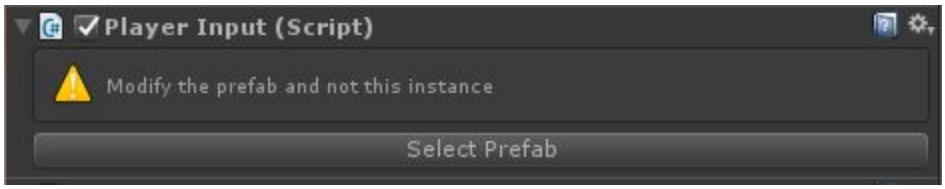
Player Input

In this project you can easily remap the input for a player, the **Player Input** script tells Unity's ***I M*** what controls your game should use. Whether it's keyboard and mouse or an Xbox One controller, you can change the keys or buttons for your game on this component.

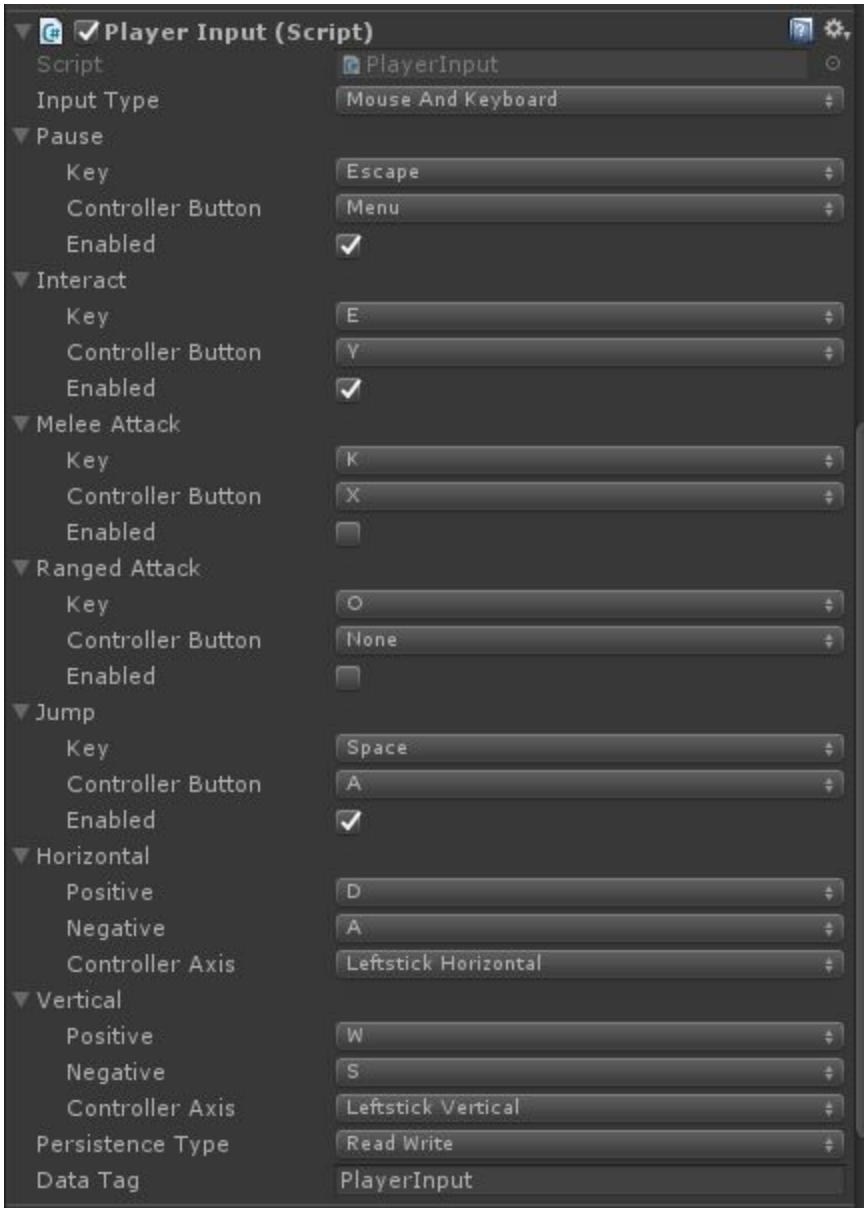
Please note that any changes to this component are per scene, as you are changing the copy ***( )*** of the Ellen Prefab. If you have multiple levels in your game and you have changed Ellen's **Player Input**, you must hit the **Apply** button at the top of the Ellen instance. This will change the settings on the original Prefab and insure any other Instance of Ellen in other levels have the same input.



We've provided a handy reminder to let you know if you're looking at an instance if you see this warning on the **Player Input** component, click **Select Prefab** which will take you to edit the prefab.



The settings for **Player Input** are;



- **Input Type**:
  Default Setting: Mouse And Keyboard
  The option for the type of input, the options are Mouse and Keyboard or Controller. The default controller type for this Kit is Xbox One.

- **Pause**: The key press or button used for pausing the game.

    - **Key:** The key on the Keyboard pressed for the action.

    - **Controller Button:** The controller button pressed for the action.
    - **Enabled:** Whether the action is on or off. Ticked is on, unticked is off.
- **Interact**: The key press or button used for interacting with the environment.
    - **Key:** The key on the Keyboard pressed for the action.
    - **Controller Button:** The controller button pressed for the action.
    - **Enabled:** Whether the action is on or off. Ticked is on, unticked is off.
- **Melee Attack**: The key press or button used for Ellen swinging her Staff.
    - **Key:** The key on the Keyboard pressed for the action.
    - **Controller Button:** The controller button pressed for the action.
    - **Enabled:** Whether the action is on or off. Ticked is on, unticked is off.
- **Ranged Attack**: The key press or button used for Ellen shooting her gun.
    - **Key:** The key on the Keyboard pressed for the action.
    - **Controller Button:** The controller button pressed for the action.
    - **Enabled:** Whether the action is on or off. Ticked is on, unticked is off.
- **Jump:** The key press or button used for jumping.
    - **Key:** The key on the Keyboard pressed for the action.
    - **Controller Button:** The controller button pressed for the action.
    - **Enabled:** Whether the action is on or off. Ticked is on, unticked is off.
- **Horizontal:** The key press or analogue stick to move left or right.
    - **Positive:** The key on the keyboard pressed to move the character right (positive on the X axis).

    - **Negative:** The key on the keyboard pressed to move the character left (negative on the X axis).

    - **Controller Axis:** The controller analogue stick or button to move left and right.

- **Vertical:** The key press or analogue stick to crouch or look up.

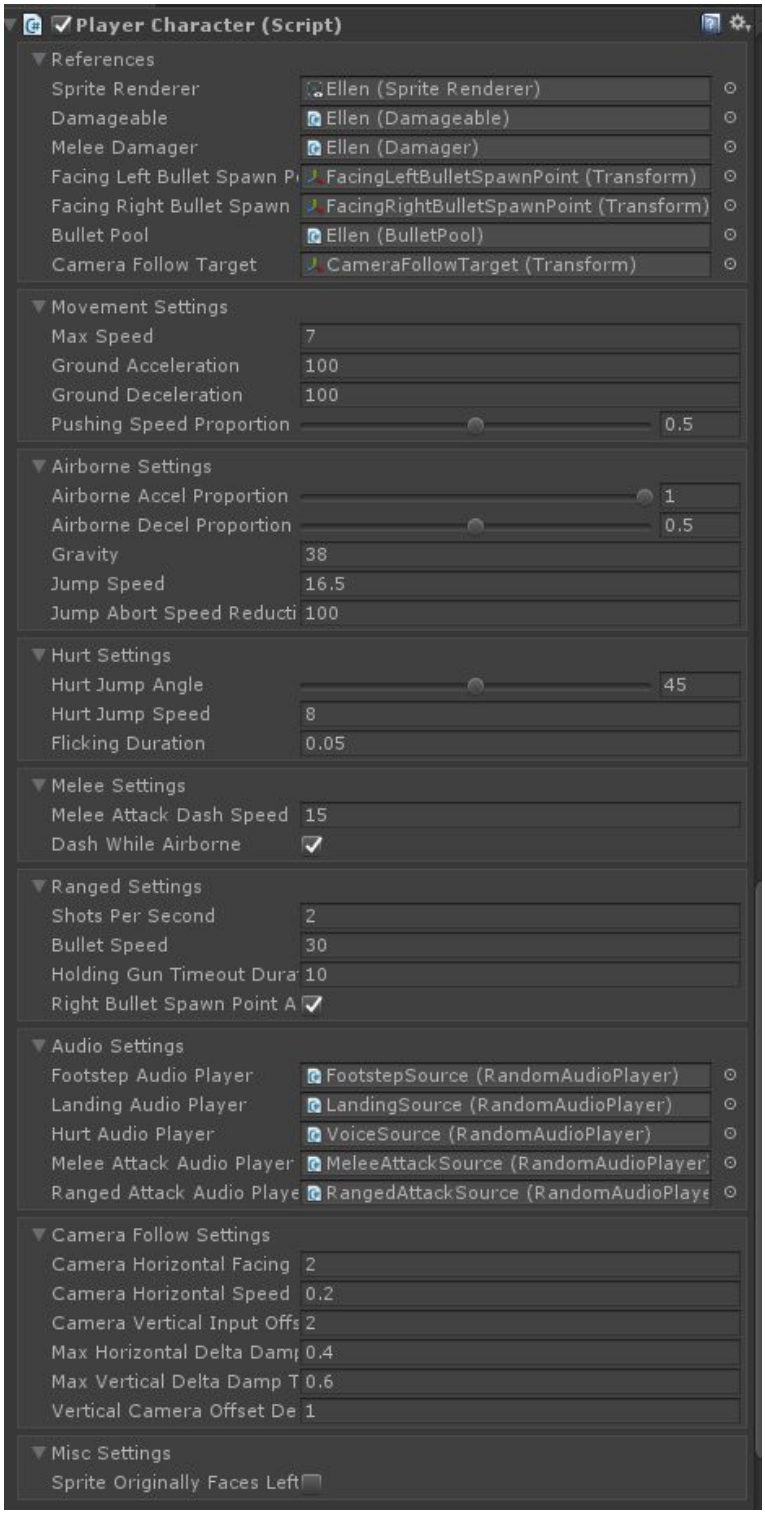
    - **Positive:** The key on the keyboard pressed to move the camera to look up (positive on the Y axis).

    - **Negative:** The key on the keyboard pressed to crouch (negative on the Y axis).

    - **Controller Axis:** The controller analogue stick or button to look up and crouch.

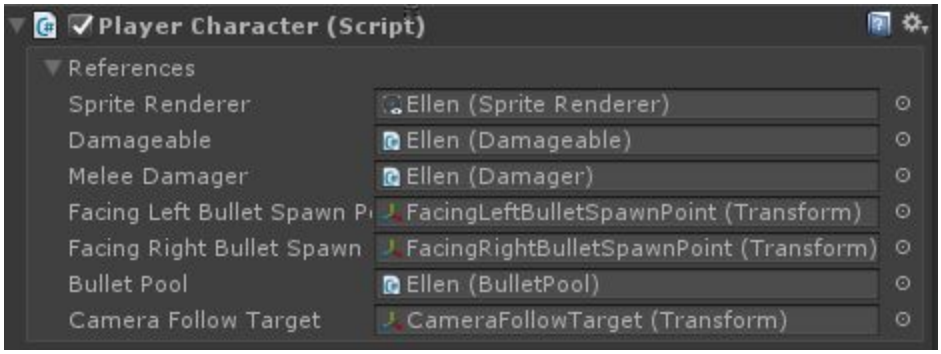
- **Persistence Type and Data Tag**: For information on how the Data Persistence system works see the **Data Persistence** section. Whether Melee Attack and Ranged Attack are enabled is the data persisted by this class.

Player Character

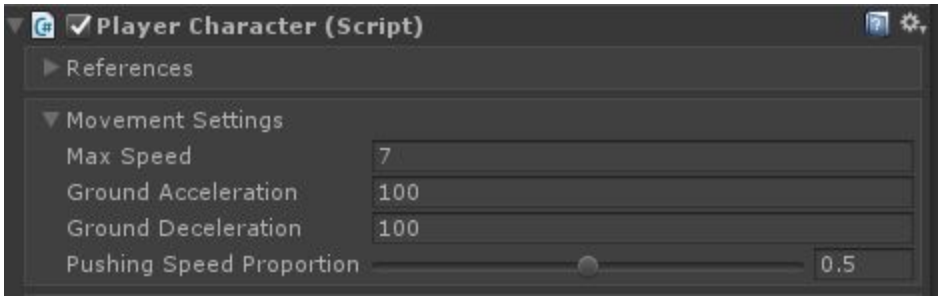


The **Player Character** script holds all the information for how Ellen behaves in the game, the settings here affect movement, Audio and Cameras.

If you change any settings it will only be applied to the Prefab instance in that scene, if you would like the changes to happen across all levels in your game, click **Apply** at the top of the instance for the change to be applied to the Ellen Prefab.



- **References**: By default you should not need to adjust this.These are all the references the script requires to function they are entirely found on the Ellen prefab.
    - **Sprite Rendere**r: Used to determine which way the character is facing.
    - **Damageable**: Used to determine the direction of motion when the character is hurt.
    - **Melee Damager**: Used to enable/disable damage during appropriate animations.
    - **Facing Left/Right Bullet Spawn Point**: Used as locations to spawn bullets from when firing.
    - **Bullet Pool**: Used when spawning bullets.
    - **Camera Follow Target**: Used to control camera positioning relative to the character, which allows behaviour such as maintaining a lead from the character when moving fast for example.



- **Movement Settings**: These are the settings for controlling how fast Ellen moves on the ground.

    - **Max Speed**:
      Default Setting: 7
      How fast Ellen can run.

    - **Ground Acceleration**:
      Default Setting: 100
      How fast Ellen gets to her Max Speed when on the ground.

    - **Ground Deceleration**:
      Default Setting: 100
      How fast Ellen slows to a stop when there is no input.

    - **Pushing Speed Proportion**:
      Default Setting: 0.5
      What proportion of the Max Speed Ellen moves at when pushing a box.

- **Airborne Settings**: These are the settings for controlling how fast Ellen moves whilst not grounded (in the air).

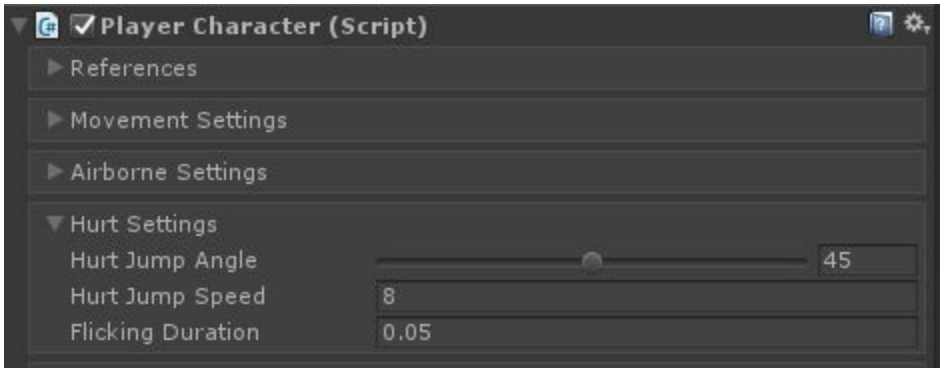
    - **Airborne Accel Proportion**:
      Default Setting: 1
      What proportion of the Ground Acceleration Ellen uses when not grounded.

    - **Airborne Decel Proportion**:
      Default Setting: 0.5
      What proportion of the Ground Deceleration Ellen uses when not grounded.

    - **Gravity**:
      Default Setting: 38
      How fast Ellen accelerates towards the ground. This also affects the jump height.

    - **Jump Speed**:
      Default Setting: 16.5
      How fast Ellen takes off from a jump. This affects the jump height.

    - **Jump Abort Speed Reduction**:
      Default Setting: 100
      The speed at which the jump value is reduced when the jump button is released. This affects jump height.

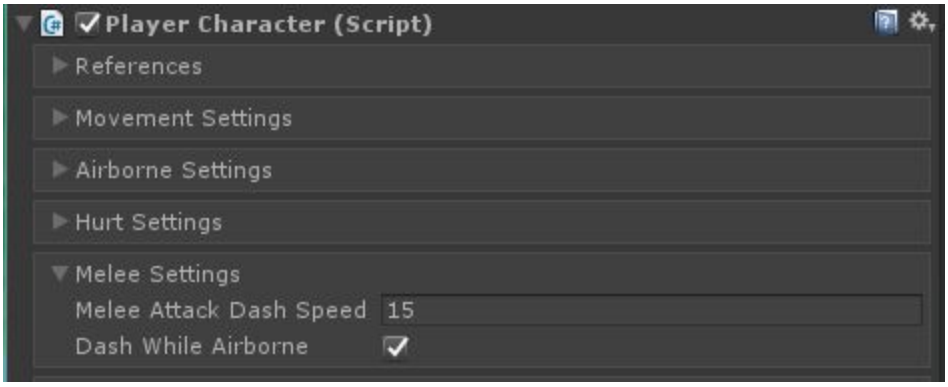


- **Hurt Settings**: These settings determine how Ellen moves and appears when she is hurt. For other settings to do with how Ellen reacts to damage see the section of this document on the Damage System.

    - **Hurt Jump Angle**:
      Default Setting: 45
      When Ellen is hurt she recoils by jumping into the air away from the source of the damage. This is the angle from horizontal at which she jumps.

    - **Hurt Jump Speed**:
      Default Setting: 8
      How fast Ellen takes off when recoiling from damage.

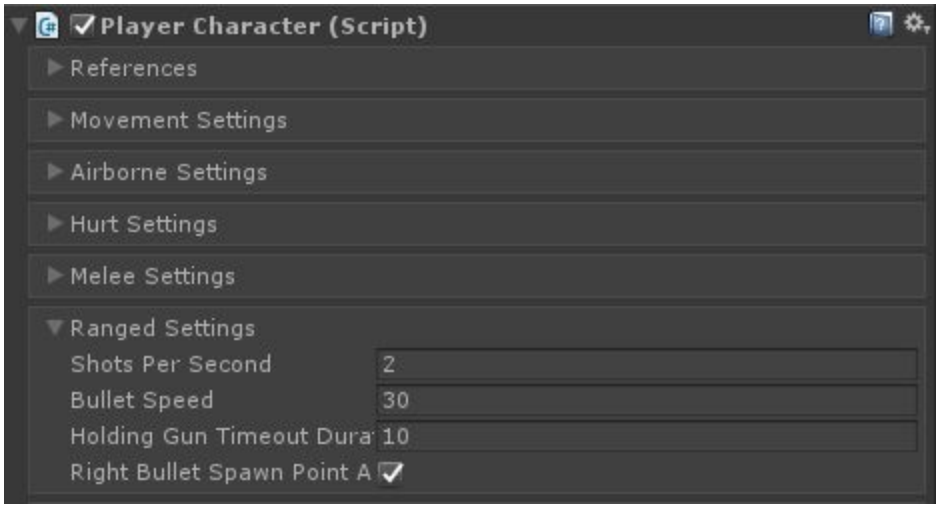
    - **Flickering Duration**:
      Default Setting: 0.05
      When Ellen is hurt her sprite turns on and off quickly to produce a flickering effect. This duration sets how long the sprite stays on and off for whilst flickering.



- **Melee Settings**: These are settings for how Ellen moves when she swings her staff. For other settings to do with how Ellen damages things see the section of this document on the Damage System.

    - **Melee Attack Dash Speed**:
      Default Setting: 15
      The distance at which Ellen dashes horizontally when she attacks with a Staff.

    - **Dash While Airborne**:
      Default Setting: Enabled
      An optional setting to enable a dash during an airborne attack. This will set horizontal movement and not increment it, if Ellen is already moving horizontally while airborne, this setting may not be noticeable.



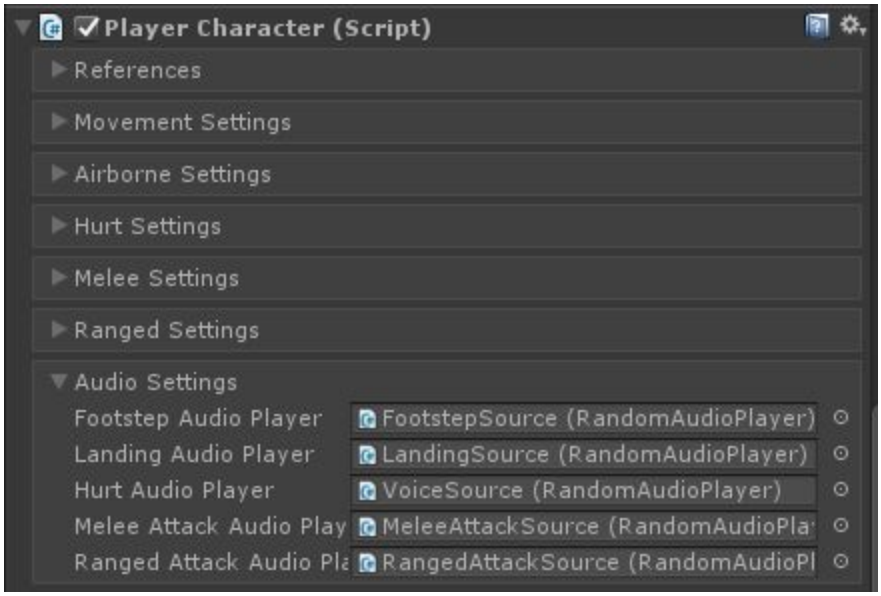
- **Ranged Settings**: These are settings to do with the bullets that are spawned when Ellen is firing her gun.

    - **Shots Per Second**:
      Default Setting: 2
      The maximum number of bullets that can be spawned per second.

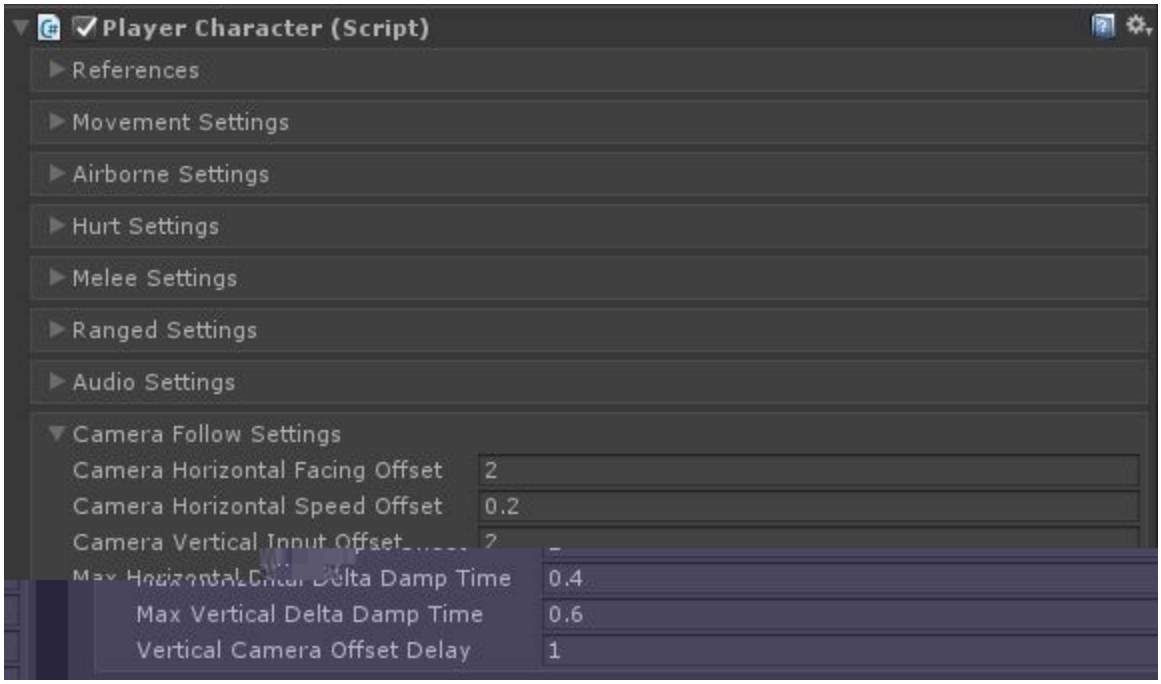
    - **Bullet Speed**:
      Default Setting: 30
      The speed given to bullets when they are spawned.

    - **Holding Gun Timeout Duration**:
      Default Setting: 1.5
      How long Ellen continues to hold her gun out after she has stopped firing.

    - **Right Bullet Spawn Point Animated**:
      Default Setting: Enabled
      The bullet spawn point is animated to follow the gun position, by default the Ellen asset has a bullet spawn position on the right. When she is facing left we mirror the spawn point animation. If you would like to add your own animated sprite facing to the left, then uncheck this box.



- **Audio Settings**: We randomise the audio played for Ellen's various actions so that it does not sound repetitive. These are the references to the **RandomAudioPlayers** which play the audio. These should be left unedited.
    - **Footstep Audio Player**: The RandomAudioPlayer used when Ellen makes a step.
    - **Landing Audio Player**: The RandomAudioPlayer used when Ellen becomes grounded after being airborne.
    - **Hurt Audio Player**: The RandomAudioPlayer used when Ellen is damaged.
    - **Melee Audio Player**: The RandomAudioPlayer used when Ellen does a melee attack.
    - **Ranged Attack Audio Player**: The RandomAudioPlayer that is used when Ellen does a ranged attack.



- **Camera Follow Settings**: Rather than following Ellen, the camera follows a target offset from Ellen's location. These settings control the offset.

    - **Camera Horizontal Facing Offset**:
      Default Setting: 2
      The amount the target is offset horizontally in front of Ellen.

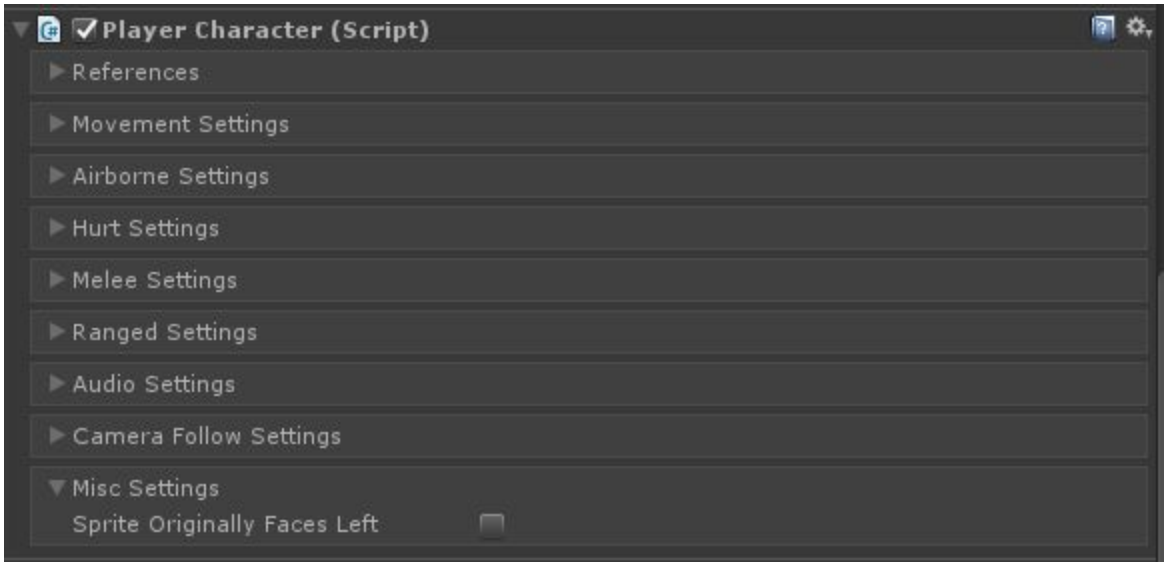
    - **Camera Horizontal Speed Offset:**
      Default Setting: 0.2
      The amount the target is shifted based on Ellen's horizontal speed.

    - **Camera Vertical Input Offset**:
      Default Setting: 2
      How much the target is offset vertically based on the player's input.

    - **Max Horizontal Delta Damp Time**:
      Default Setting: 0.4
      The amount of time it takes for the target to move horizontally from no offset to its desired horizontal offset.

    - **Max Vertical Delta Damp Time**:
      Default Setting: 0.6
      The amount of time it takes for the target to move vertically from no offset to its desired vertical offset.

- **Vertical Camera Offset Delay**:
  Default Setting: 1
  The amount of time that the up or down input keys need to be held before the target starts to move vertically, so that the player can ‘look up’ or ‘down’.

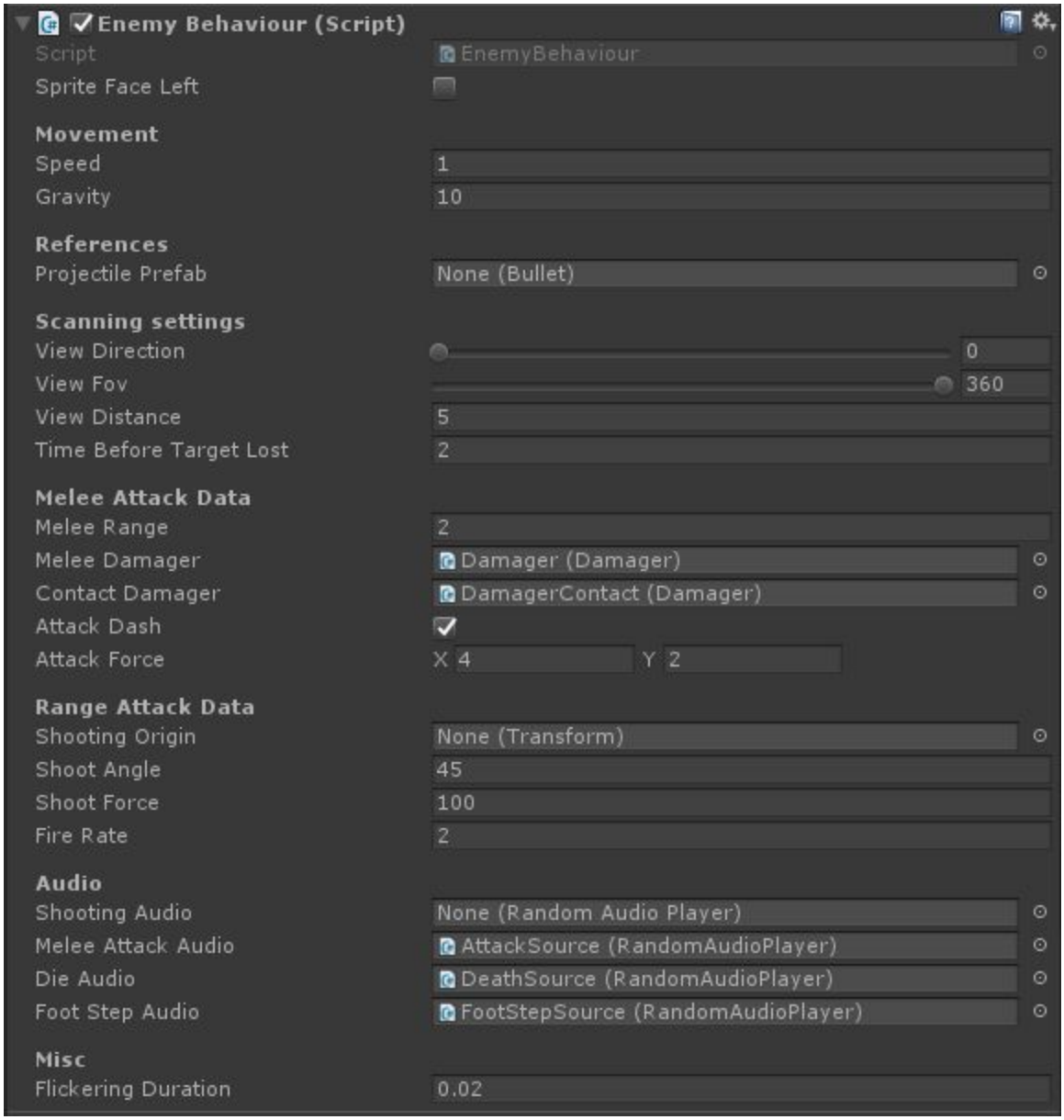


- **Misc Settings**: A folder for other settings.
  - **Sprite Originally Faces Left**:
    Default Setting: Disabled
    This refers to the sprite asset itself without any flipping. It is used to determine things like which spawn point bullets should spawn from.

Enemy Behaviour

The Enemy Behaviour Component is applied to Chomper and Spitter, our two enemies in the game. These enemies can be found in **Prefabs** < **Enemies**

Both enemies use the same component, they just use different settings.



- **Sprite Faces Left:** Checkbox if the Sprite faces Left

- **Movement**
    - **Speed:** The speed of the enemy.

    - **Gravity:** How fast it falls.

- **References**
    - **Projectile Prefab:** The Prefab of the projectile the enemy is to fire (only for projectile enemy).

- **Scanning Settings**
    - **View Direction:** The direction the enemy can detect the player. This is depicted with a green circle in the **Scene View**

    - **View FOV:** The Field of View the enemy has. This is depicted with a green circle in the **Scene View**

    - **View Distance:** How far away the enemy can see the player. This is depicted with a green circle in the **Scene View**

    - **Time Before Target Lost:** The amount of time before the enemy will stop chasing or searching the player once the player is out of range.

- **Melee Attack Data** Melee attack information, only for Melee enemy like Chomper.
    - **Melee Range:** The distance the away from itself the enemy can hit the player.

    - **Melee Damager:** The **Damager** Script on the enemy. See **Damage System** for more information.

    - **Contact Damager:** The **Contact Damager** Script on the enemy, this script causes Damage by touching the enemy.

    - **Attack Dash:** Enables or Disables a dash forward when the enemy attacks.

    - **Attack Force:** The force applied to the Player when they're attacked, this is what pushes the player back after being attacked.

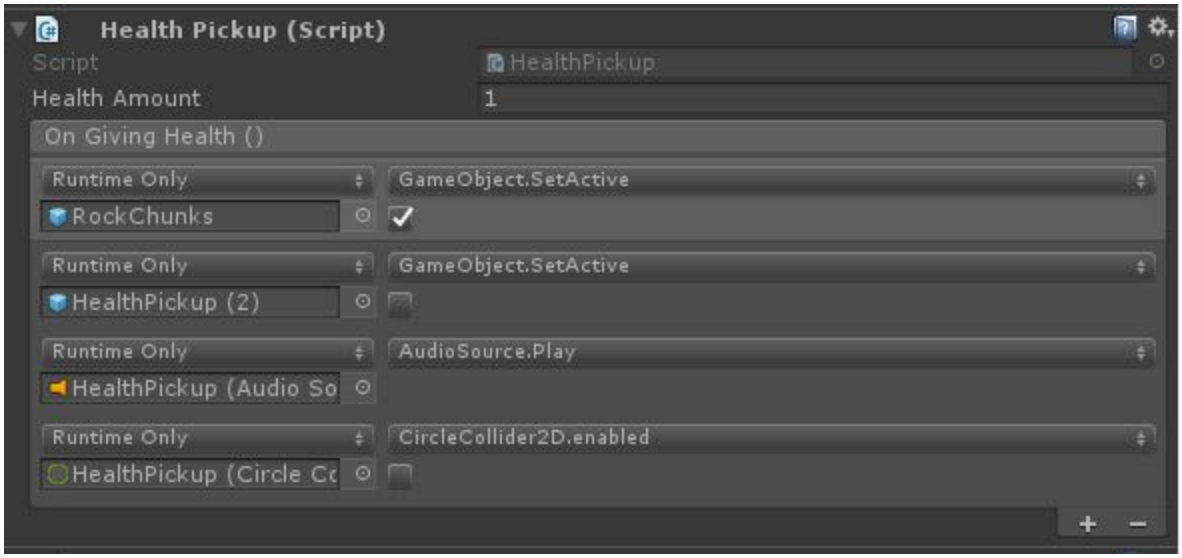
- **Ranged Attack Data** Ranged attack information, only for ranged enemy like Spitter.
    - **Shooting Origin:** The position the projectile is shot from.

    - **Shoot Angle:** The angle at which the projectile is shot form the origin.

    - **Shoot Force:** The force applied to the projectile.
    - **Fire Rate:** How fast the projectiles are shot.
- **Audio**
    - **Shooting Audio:** The **RandomAudioPlayer** used when a shot is fired.
    - **Melee Attack Audio:** The **RandomAudioPlayer** used when a Melee attack occurs.
    - **Die Audio:** The **RandomAudioPlayer** used when the enemy dies.
    - **Foot Step Audio:** The **RandomAudioPlayer** used for the enemy's footsteps.
- **Misc**
    - **Flickering Duration:** If the enemy's health is more than 1, when it is hit, it will flicker to indicate a hit. This value is the duration of time the flicker lasts.

Health Pickup

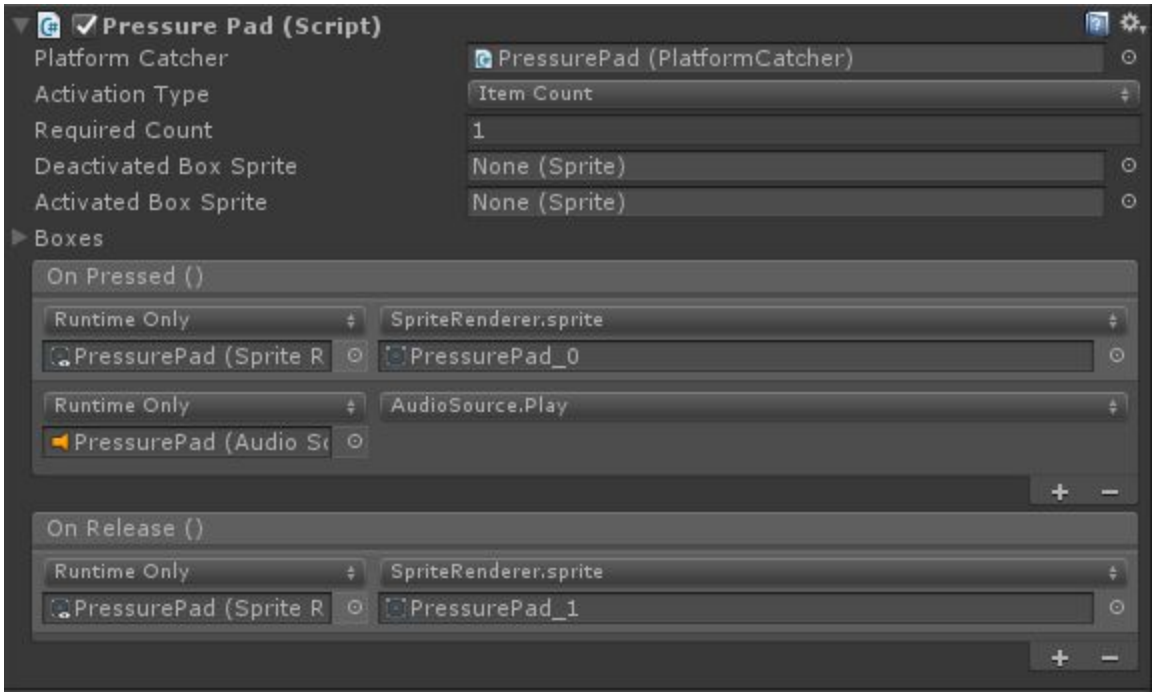
The Health Pickup is a prefab that gives health when the player walks through it. It can be found in **Prefabs** < **Interactables**



- **Health Amount:** The amount of health the box provides
- **On Giving Health ():** The events triggered when the box is collided with and health is given to the player.

Pressure Pad

The pressure pad is automatically set up to detect the player walking on to it, it will light up and play a sound. You can connect the pressure pad to trigger events like opening a door.



- **Activation Type:** How the button is activated. The option below will change depending on this setting.
    - **Item Count:** The amount of objects on the pressure pad to activate it.
    - **Mass:** The weight of an object required to activate the pad.

- **Required Count:** The required number of items for the pad to trigger the Event in On Pressed & On Release. This changes to **Required Mass** if **Activation Type** is changed.

- **Deactivated Box Sprite:** (Optional) The sprite to change to, when the platform is not pressed. Only needed if the **Boxes** option has a **Sprite Renderer**

- **Activated Box Sprite:** (Optional)The sprite to change to, when the platform is pressed. Only needed if the **Boxes** option has a **Sprite Renderer**

- **Boxes:** (Optional) A list of boxes expected to be pushed onto the **PressurePad**, when they are pushed on, the sprites will change to those in **Deactivated Box Sprite** and **Activated Box Sprite**.

- **On Pressed ():** Events that happen when the Pressure Pad is pressed, this is dependant on the correct **Required Count** or **Required Mass**

- **On Release ():** Events that happen when the Pressure Pad is released, this is dependant on the correct **Required Count** or **Required Mass**

Damage System.

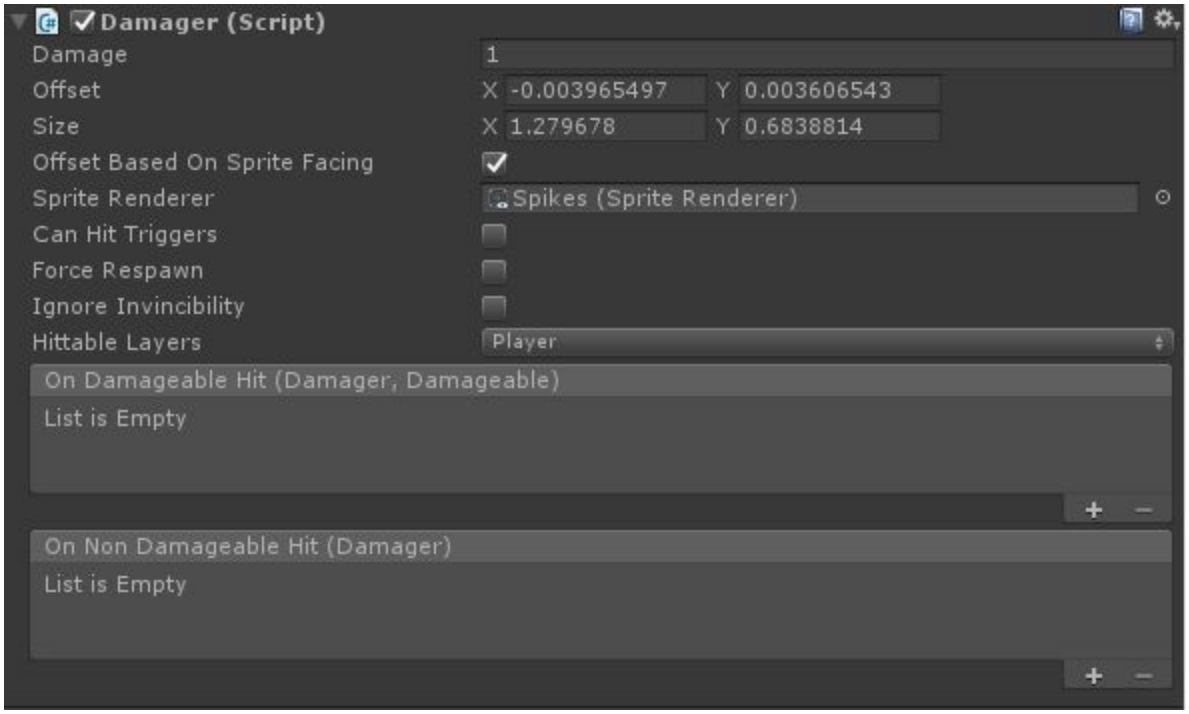
The damage system in the kit is composed of two components: **Damager** and **Damageable** scripts.

Adding a **Damager** script to an object makes it able to inflict damage to any object that has a **Damageable** component, as long as their settings match.

Damager

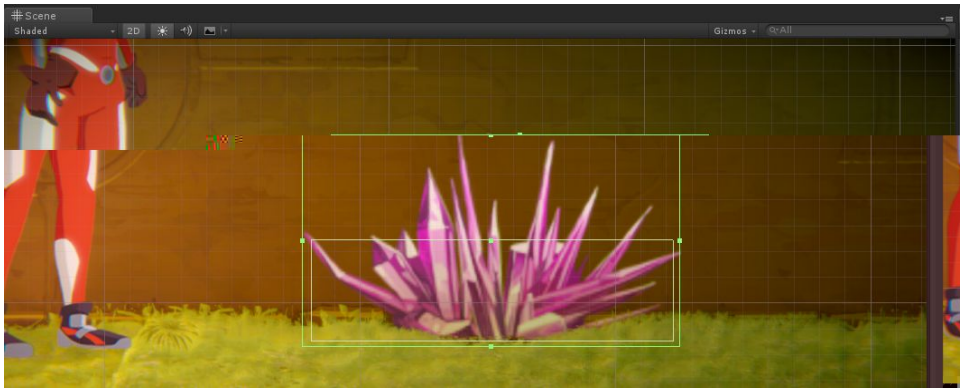
The **Damager** Component has events for collision with **Damageable** components and non Damageable components. The health calculation is done automatically and these events are used for triggering gameplay changes or visual effects.

The **Damager** component does not require a collider, it will display a box gizmo in the Scene View you can use to tweak its size and position.



- **Damage**: The amount of damage it will inflict on a Damageable (e.g Ellen or Chomper) when hit.

- **Offset** and **Size**: these are the two parameters that define the Damager collision box. This is the outer box displayed on the Spikes in the Scene View as the outer box, it will inflict damage when this is collided with.



- **Offset Based On Sprite Facing**: This setting allows the Damager to change position based on which direction a sprite is facing. E.g. This is used on Ellen allowing the Damager to lip left and right depending on which way she is facing.

- **Sprite Renderer**: The Sprite Renderer that the above setting uses.

- **Can Hit Triggers**: Toggles whether the Damager will collide with triggers if checked or totally ignore triggers if unchecked.

- **Force Respawn**: If this is enabled, the object containing a Damageable will immediately respawn to the latest checkpoint (the Acid prefab uses this when the player falls into the water).

- **Ignore Invincibility**: If enabled, the Damager will still register a ‘hit’ to the Damageable but not remove health (the Acid uses this so that the player will always respawn even if they are invincible from a previous enemy hit).

- **Hittable Layers**: The Layers which the Damager will inflict damage. Layers that are not selected will not receive damage.

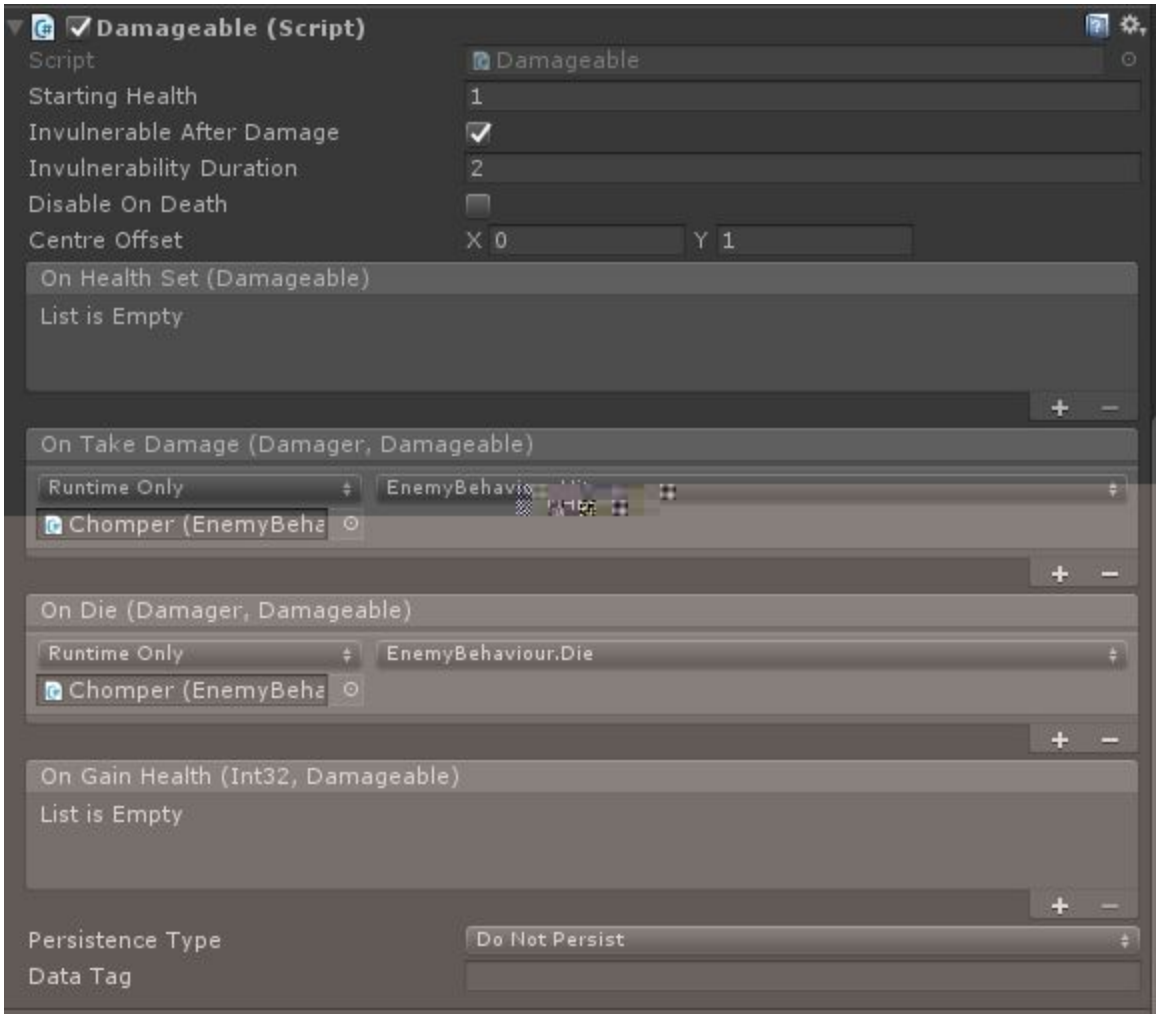
- **On Damageable Hit:** Events here will play when the object is hit and it has a Damageable component.

- **On Non Damageable Hit:** Events here will play when the object is hit but does not have a Damageable component.

Damageable

The **Damageable** Component receives damage from collisions of objects with the **Damager** Component attached to them.
There are events for when damage is received, health is restored, health changes, or complete depletion of health (death) occurs.

Contrary to the **Damager**, the **Damageable** component relies on colliders attached to the same GameObject, so in addition to adding a **Damageable** Script to your object, don't forget to add colliders to define hittable zones.



- **Starting Health**: Health level. Damagers will remove their health at each impact. Damageables die when this reaches 0.

- **Invulnerable After Damage**: This Damageable becomes invulnerable for a given time after impact.

- **Invulnerability Duration**: Length of time (in seconds) that this Damageable is considered invulnerable after impact (only applicable if previous setting is set).

- **Disable On Death**: This setting will disable the GameObject on death.

- **Center Offset**: This is used to define where the centre of the Damageable is (an offset of (0,0,0) means it is on the GameObject position). Used to compute distance to Damager.

- **On Take Damage:** Events triggered when the object receives damage.

- **On Die:** Events triggered when the object

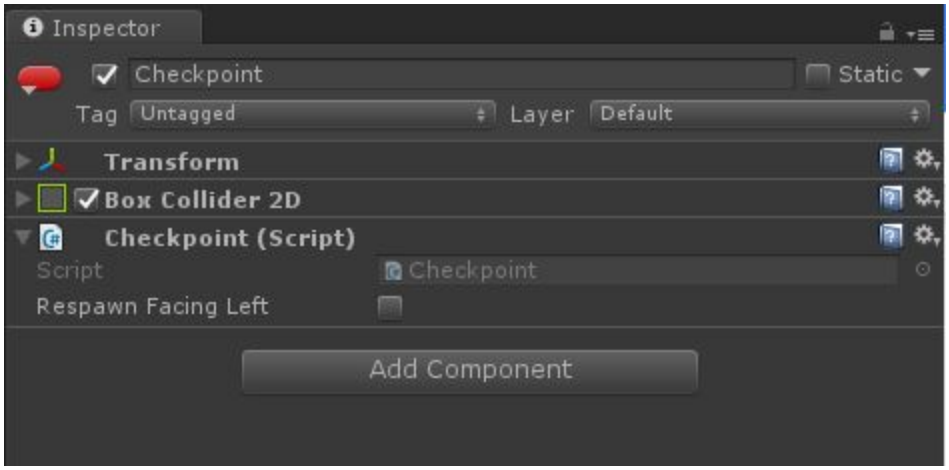
- **On Gain Health:** Events triggered when an object gains new health.

- **Persistent Type:** See Persistent Data

- **Data Tag:** See Persistent Data

Checkpoints

Checkpoints are found in the folder **Prefabs** < **SceneControl**



Each time the Player enters a checkpoint collider, this checkpoint becomes active. When they fall into the water (or get hit by any **Damager** that has the **Force Respawn** setting) they will reappear at that checkpoint.

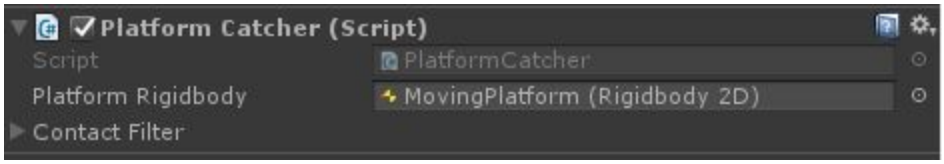
Note: The Player will reappear at the GameObject position, so make sure it is above ground.

Moving Platform

A Prefab located in the Project View in ***< I***

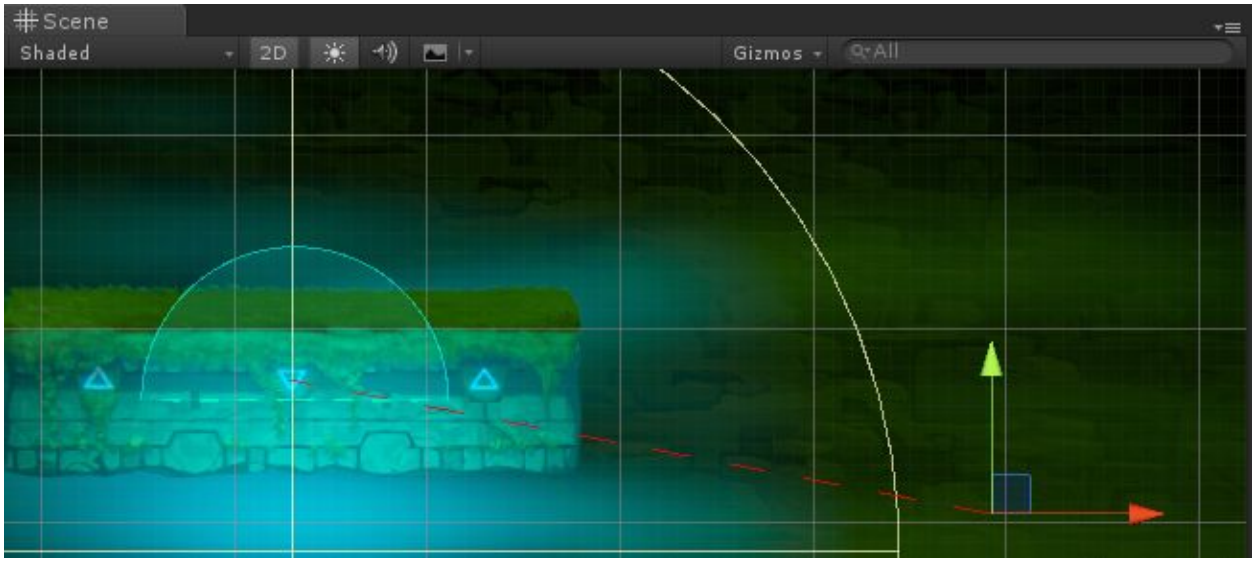
To create a **Moving Platform** in your scene, drag a **MovingPlatform** Prefab from the Project View into the **Scene View**.

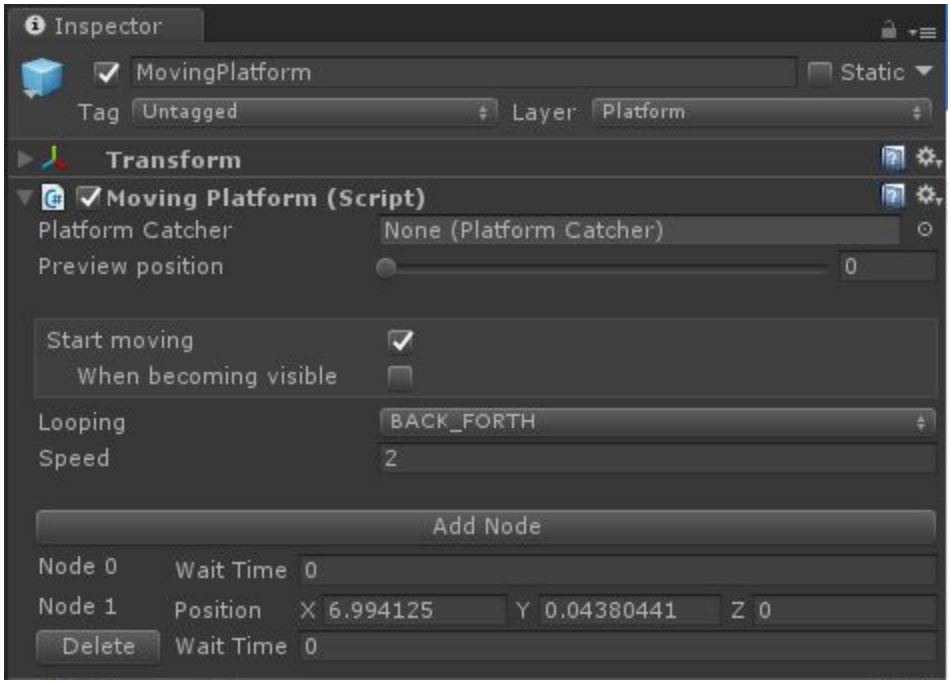
It includes a **Moving Platform Script**, Box Collider 2D, Rigidbody 2D, Platform Effector 2D and a **Platform Catcher Script.**



The **Platform Catcher Script** allows objects on top of the platform to move freely. And does not need to be adjusted.

In the **Scene View**, you will see red dotted line with a Transform Tool on the end, this indicates the path the platform will take and it's destination. Drag the Transform Tool in any direction to change the path and destination.





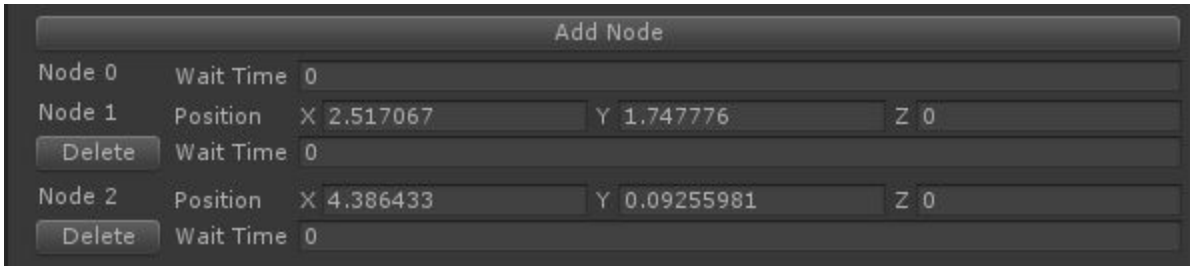
- **Platform Catcher**: This automatically finds the Platform Catcher Script on the Platform, there is no need to change this.

- **Preview Position**: Allows you to scrub through and preview where the platform will go in the Scene View.

- **Start Moving**: When enabled, the platform will start moving as soon as a level is loaded. If unticked, it will need to be triggered by an event.
    - **When becoming visible**: This option will only appear if the platform is set to Start Moving. If this is enabled, the platform will start moving only when it becomes visible on screen, not when the level is loaded.

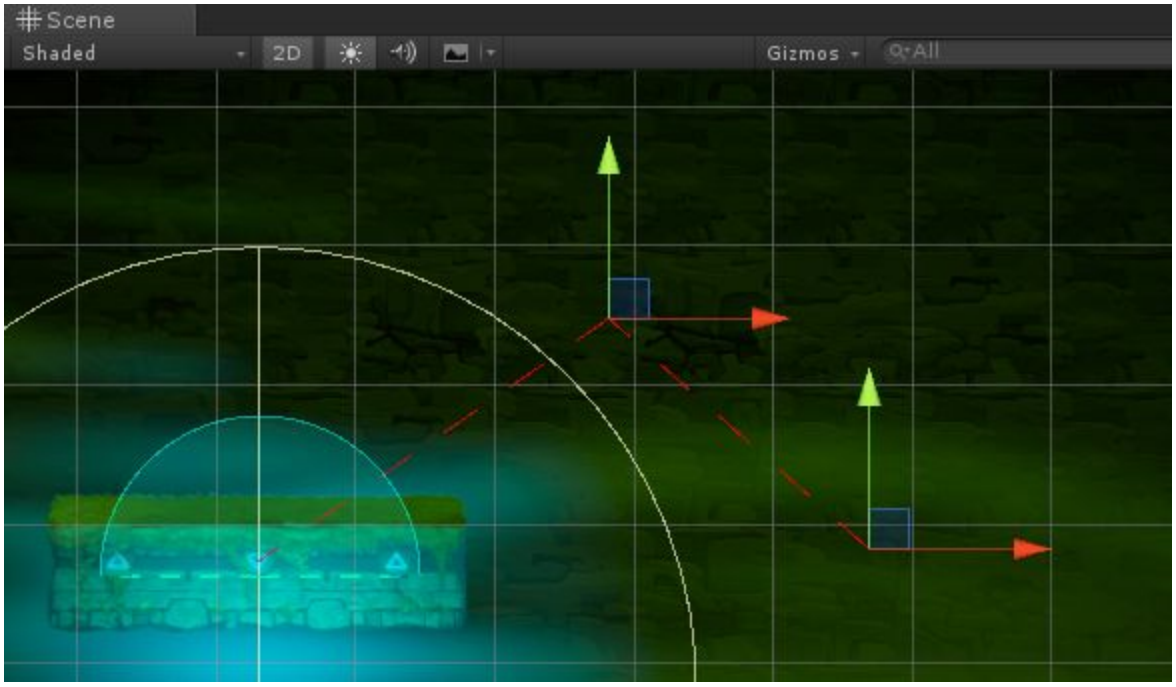
- **Looping**: Defines how the platform reacts once it reaches the end of its path.
    - ***BACK F H:*** platform moves back and forth between a start and end point.
    - ***L :*** Once the platform reaches the last point of it's path, it will move towards the start point in a straight line to restart the cycle, making a loop.
    - ***CE:*** The platform stops when it reaches the final point.

- **Speed**:
  Default Setting: 2
  The speed at which the platform moves (in units per second).



- **Add Node:** This button will add another node, or destination point to your platform's path. A platform moves from Node to Node.
    - **Node 0:** The platform position. Cannot be deleted.
      There is no position as this cannot be altered.
        - **Wait Time:** The amount of time the platform will wait to move on to the next node.
    - **Node 1:** The first path and destination, this is always included by default.
        - **Position:** The position of the Node (local Space).
        - **Delete:** The button to delete a Node.
        - **Wait Time:** The amount of time the platform will wait to move on to the next node.

A platform with two **Nodes** can be seen like this in the Scene View.



Interaction System

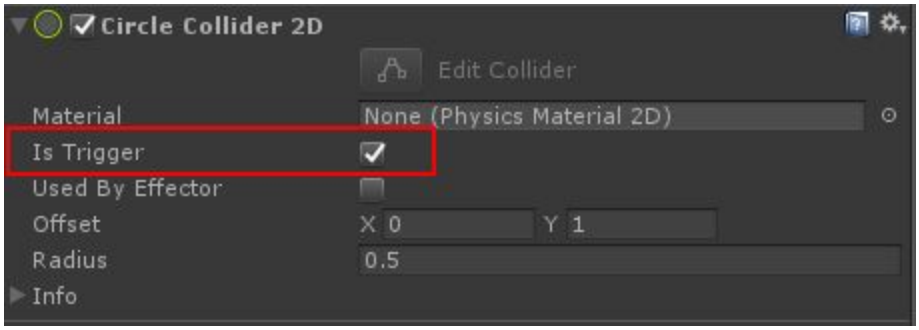
Interaction with GameObject by the player is made possible through multiple components all named **InteractOn*:**

- **InteractOnCollision2D**
- **InteractOnTrigger2D**
- **InteractOnButton2D**

Interactions are dependant on objects having a **Collider 2D** Component on them. A Collider is an invisible area around an object that can detect when something touches or enters it. The main ones we use in the Kit are Box (**Box Collider 2D**), Capsule (**Capsule Collider 2D**) and Circle (**Circle Collider 2D**).

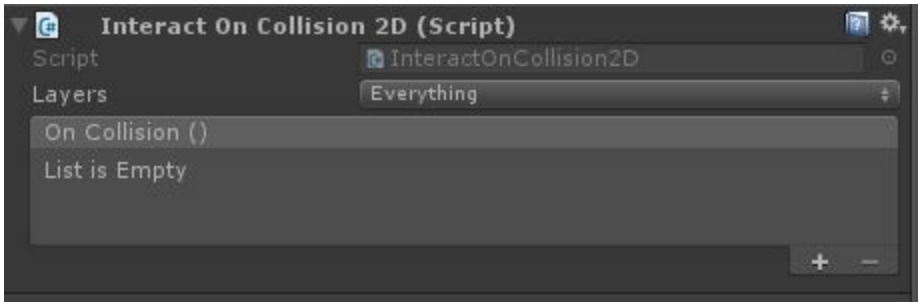
These are all set up for you in the various Prefabs provided in the Kit.

If you wish to add new types of interactions on objects or create your own, depending on which of the following **InteractOn** components you use, you must take note of the **Is Trigger** checkbox on the Collider of the object you'd like to make interactable. We will note what state the Collider will need to be in for a particular **InteractOn** component.



Interact On Collision 2D

This is used to trigger Events when object Colliders touch.



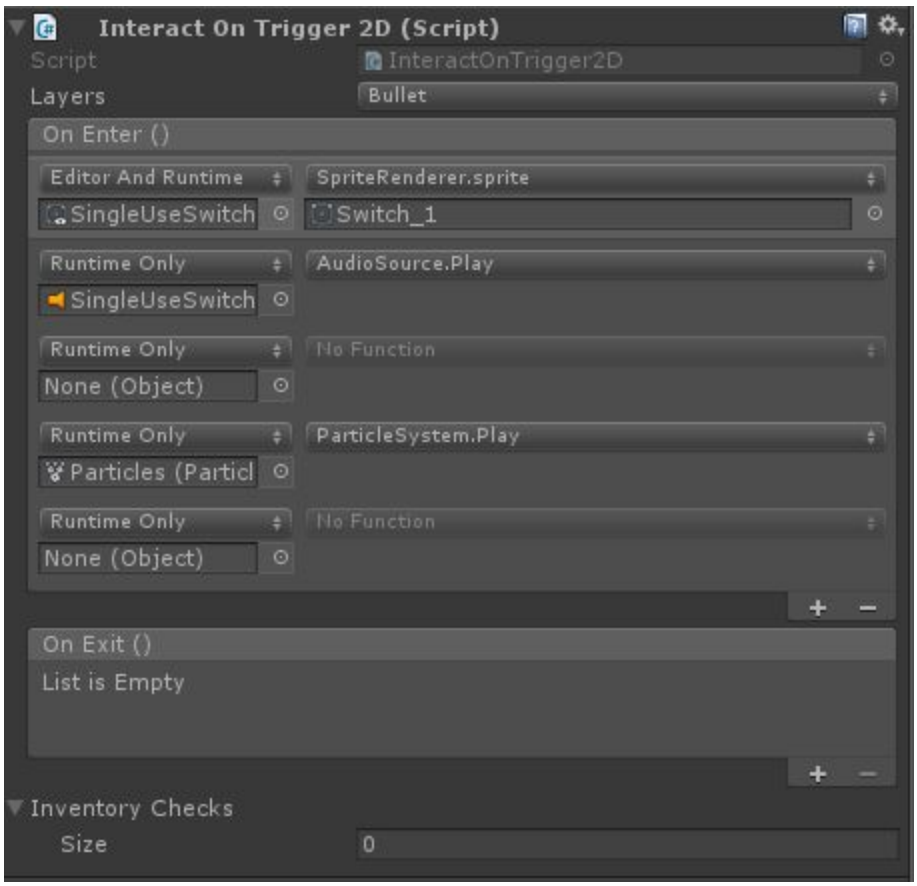
- **Script:** The name of the script being used (cannot be changed).
- **Layers:** Only objects on the selected **Layers** will trigger the events.
- **On Collision ():** Events listed here will happen when an object that belong to the set layer collide.

(If setting up your own, there is no need to enable **Is Trigger** on the Collider)

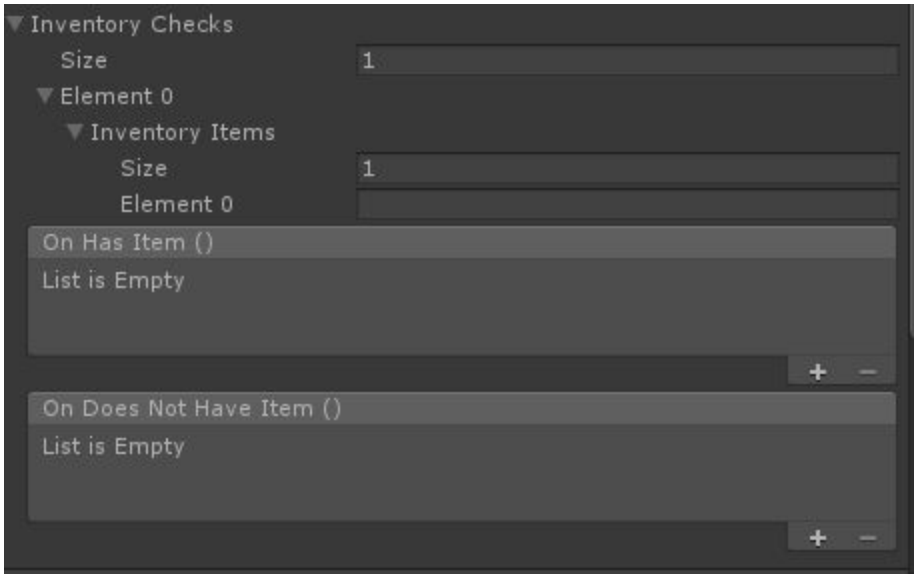
Interact On Trigger 2D

This is used to trigger Events when an object enters a collider and more events when an object exists a collider.

(If making your own, ensure **Is Trigger** is enabled on the Collider Component).



- **Script:** The name of the script being used (cannot be changed).
- **Layers:** Only objects on the selected **Layers** will trigger the events.
- **On Enter ():** Events listed here will happen when an object that belong to the set layer enter the Trigger.
- **On Exit ():** Events listed here will happen when an object that belong to the set layer Exits the Trigger.

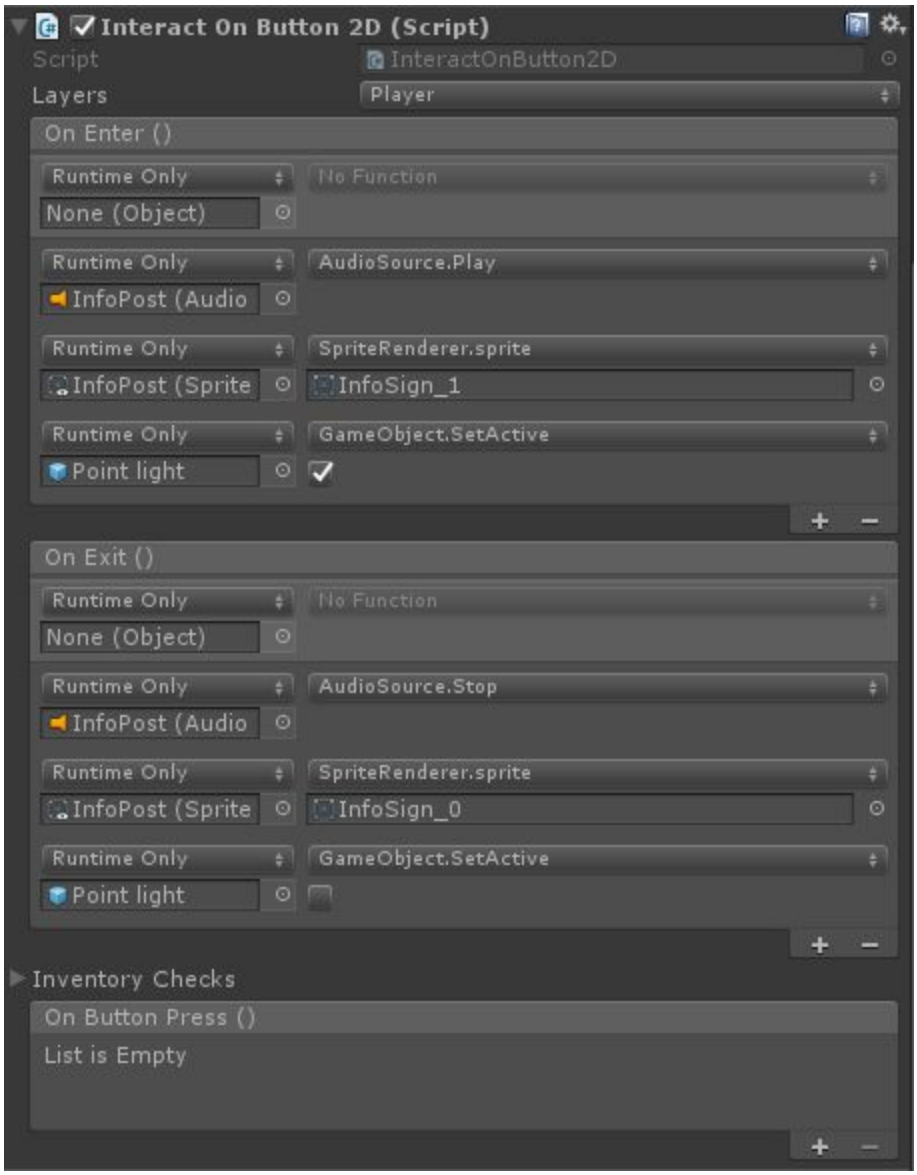


- **Inventory Checks:** This allows you to ensure Events only happen when an object entering the trigger has an object in its inventory. For example only opening a door once a player has a Key in their inventory when entering the trigger zone.
    - **Size:** Set to 0 by default, increase this number to increase the amount of Inventory the trigger should expect.
    - **Element 0:** The first Inventory Check.
        - **Inventory Items:** Objects it expects in the inventory of the object colliding with it.
            - **Size:** The number of items you would like the triggering object to carry in its inventory.
            - **Element 0:** The first inventory item needed. This must match the expected names, these are listed in the **Inventory System** section.
- **On has Item ():** Events triggered when object holds ALL the items set in the **Inventory Items**.
- **On Does Not Have Item ():** Events triggered when object does not hold ALL the items set in **Inventory Items**.

Interact On Button 2D

This is used to trigger Events when an object is inside the trigger zone and a player presses the Interact button in the game (see the **Player Input** section for more information on the Interact Button).

(If making your own, ensure **Is Trigger** is enabled on the Collider Component).



- **Script:** The name of the script being used (cannot be changed).

- **Layers:** Only objects on the selected **Layers** will trigger the events.

- **On Enter ():** Events listed here will happen when an object that belong to the set layer enter the Trigger.

- **On Exit ():** Events listed here will happen when an object that belong to the set layer Exits the Trigger.

- **Inventory Checks:** This allows you to ensure Events only happen when an object entering the trigger and pressing the interact button has an object in its inventory. For example only opening a door once a player has a Key in their inventory when entering the trigger and pressing interact zone.

    - **Size:** Set to 0 by default, increase this number to increase the amount of Inventory the trigger should expect.

    - **Element 0:** The first Inventory Check.

        - **Inventory Items:** Objects it expects in the inventory of the object colliding with it.

            - **Size:** The number of items you would like the triggering object to carry in its inventory.
            - **Element 0:** The first inventory item needed. This must match the expected names, these are listed in the **Inventory System** section.

- **On Button Press ():** Events that are triggered when the interact button is pressed.

Inventory System

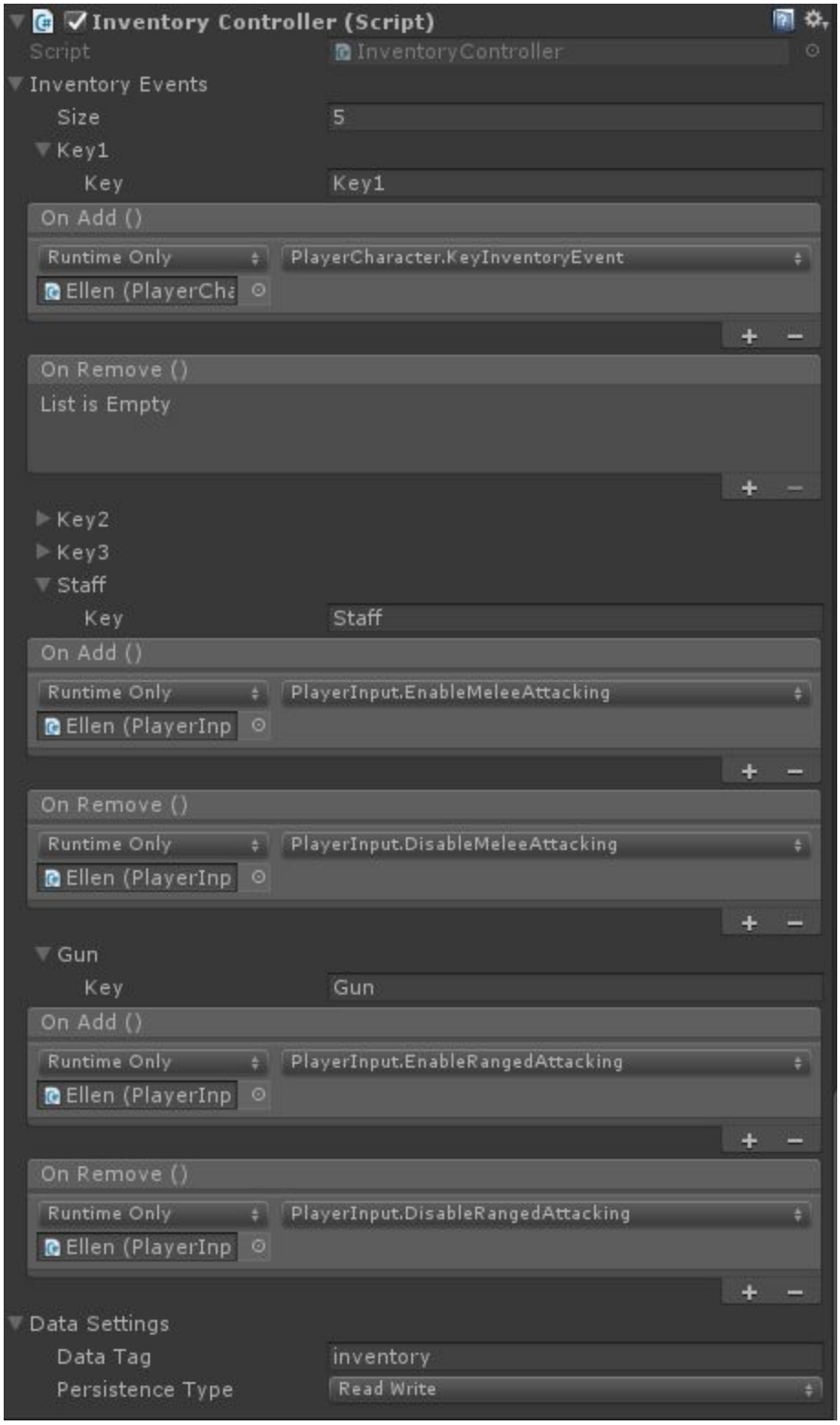
The inventory system is made up of two parts, the **Inventory Controller** Component and the **Inventory Item**.

The **Inventory Controller** should be added to an object holding the inventory. An **Inventory Item** script should be added to the item that is being collected. For example Ellen (**Inventory Controller**) will collect a Key (**Inventory Item**)

The **Inventory Controller** is able to automatically collect **Inventory Item** on contact with their colliders, and defines actions that happen when a given object is received or removed.

Inventory Controller

This Component has an entry for each item that can be carried in your inventory. The Component can exist on a character, a chest or a shop.


Inventory Controller (Script)
Script
InventoryController
Inventory Events
Size
5
Key1
Key
Key1
On Add ()
Runtime Only
PlayerCharacter.KeyInventoryEvent
Ellen (PlayerCha
On Remove ()
List is Empty
Key2
Key3
Staff
Key
Staff
On Add ()
Runtime Only
PlayerInput.EnableMeleeAttacking
Ellen (PlayerInp
On Remove ()
Runtime Only
PlayerInput.DisableMeleeAttacking
Ellen (PlayerInp
Gun
Key
Gun
On Add ()
Runtime Only
PlayerInput.EnableRangedAttacking
Ellen (PlayerInp
On Remove ()
Runtime Only
PlayerInput.DisableRangedAttacking
Ellen (PlayerInp
Data Settings
Data Tag
inventory
Persistence Type
Read Write

- **Inventory Events**: List of Events to occur on specific objects added to the inventory. Increase the number here to add new Events *(tip: you can delete an Event by right clicking on the name in the list and choosing **D A E**).*

    - **Key**: Is unique identifier that represents the **Inventory Item** e.g. Key, Gun, Staff. This must match the **Key** in the **Inventory Item** component on the item you wish to be collected. The inventory can only hold a single instance of each object; if an object with the same **Key** is already in the inventory, the add event in Inventory Controller won't be called.

    - **OnAdd**: Events triggered when the item is added to the inventory.

    - **OnRemove:** Events triggered when the item is removed from the inventory.

    - **Data Settings**: See the **Persistent Data** documentation for more info.

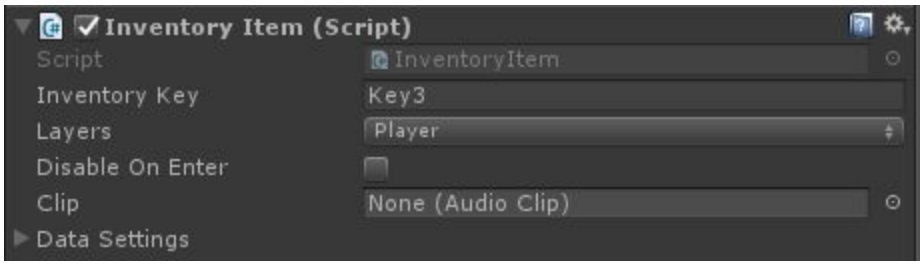
Do not confuse the **Key** of an item which is the unique identifier assigned to an item with a key, a collectible object we like to use in games. Mention within the documentation of the unique identifier is in bold to limit confusion.

Inventory Item

Any item that can be carried in the inventory needs an **Inventory Item** Component. This specifies the **Key** of the item, an **Audio Clip** and which Layers can pick the item up.

An **InventoryItem** is a component added to the **GameObject** representing the item. It also requires a **Circle Collider 2D** and will automatically add one if missing. That collider is set to trigger and is used to detect when the Player touches the object.

The state of the GameObject (active or not) is saved, so it remains the same when the scene is reloaded and the Player re-enters the zone.



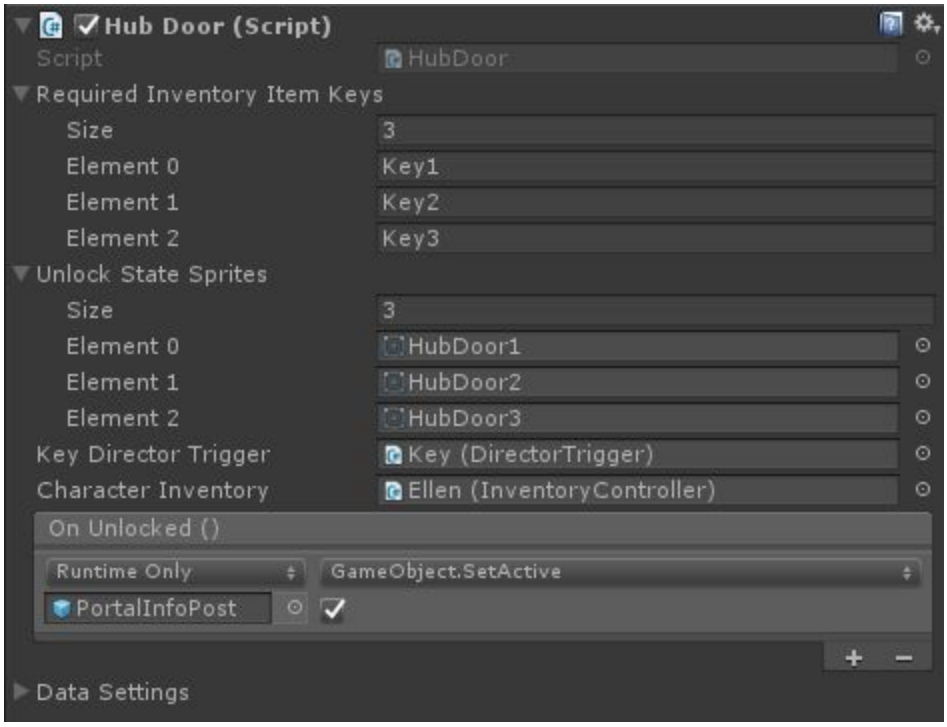
- **Inventory Key**: A unique name for the item (e.g. Key1), used by the inventory to track if an item is already owned or not.
- **Layers**: The Layer on which the object collecting the item is on.
- **Disable On Enter**: This setting disables the object once it has been collected.
- **Clip**: The Audio clip to be played when the item is collected. If empty, no clip will be played.
- **Data Settings**: See the **Persistent Data** documentation for more info.

HubDoor

This component is used to change the sprite of an object depending on how many inventory items have been collected.

Once all the items have been collected the **On Unlocked** function is called. In the below example the three inventory items to collect are Key1, Key2 and Key3. Each time one item in the inventory is collected the sprite will change.

Interact On Trigger 2D and Interact On Button 2D are also able to check if this component has been satisfied.



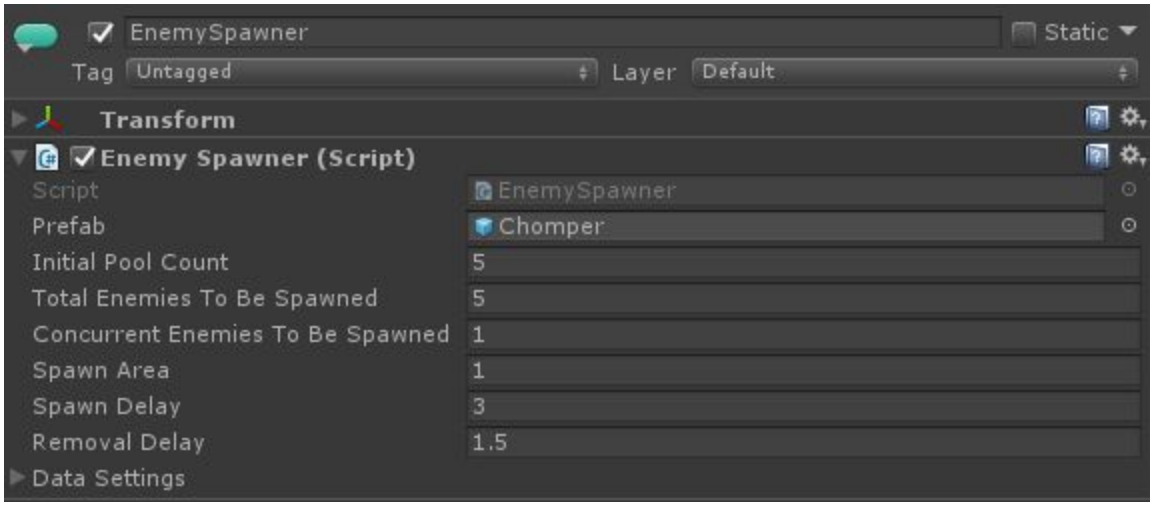
- **Required Inventory Item Keys:** The list of **Keys** of the objects that need to be collected. Once ALL are collected it will tq edhe

- **On Unlocked ():** The events triggered when all the items have been collected and sprites changed.

- **Data Settings:** See the **Persistent Data** documentation for more info.

Enemy Spawner

The **Enemy Spawner** prefab is located in **Prefabs** < **Enemies** this will control the spawning of multiple enemies in your scene at the location it is placed.



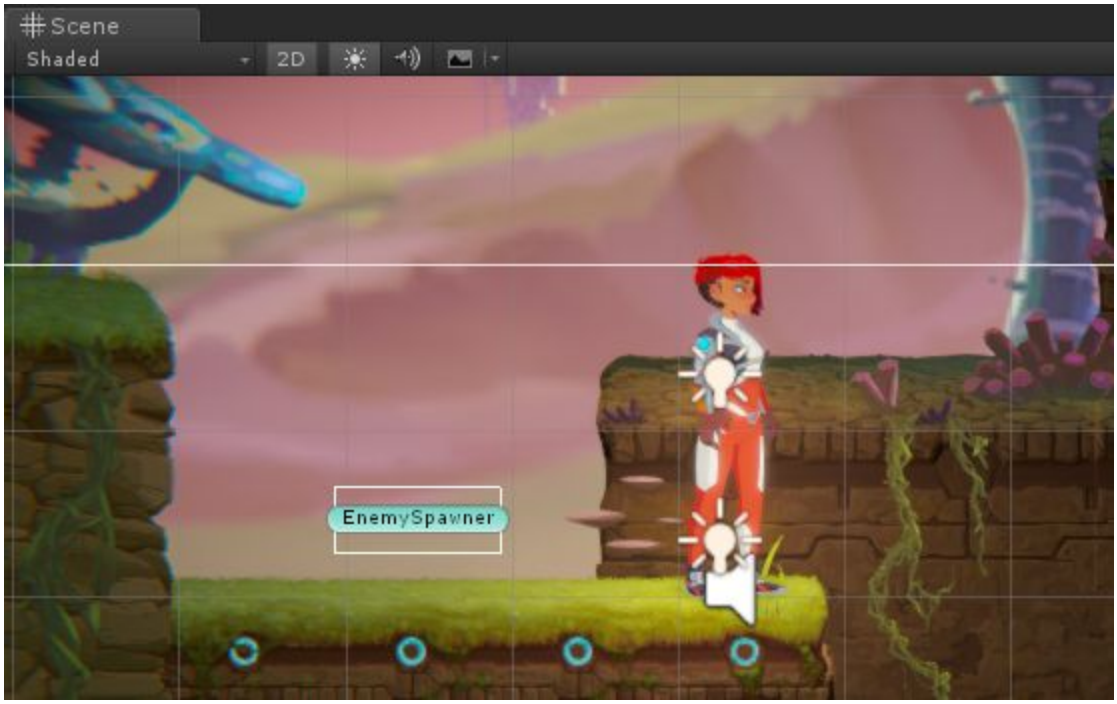
- **Prefab**: The enemy Prefab to be spawned.

- **Initial Pool Count**: This determines the amount of enemies which will be held in the pool. A good value to enter here is equal to the number of concurrent enemy that need to be spawned.

- **Total Enemies to be spawned**: The number of enemies that the spawner will spawn until it stops spawning enemies.

- **Concurrent Enemies to be spawned**: The number of enemies that the spawner will spawn in the world at the same time. *E.g. If this is 5 with 10 total enemies, the spawner will spawn 5 enemies. Once one is killed it will spawn a new one unless it has already spawned 10 in total.*

- **Spawn Area**: This is the extent of the spawn zone around the spawn point. Enemies will be spawned at random positions within the zone. This can be visualized in the scene view by the white rectangle gizmo around the spawn point.



- **Spawn Delay**: The delay between the death of an enemy and the spawning of the next one.

- **Removal Delay**: The time before an enemy is sent back to the pool (removed from the world).

Game Kit Advanced Topics

The following sections consist of deeper information on the Game Kit systems and may require more Unity knowledge, some topics require knowledge of C#.

RandomAudioPlayer

This script allows you to play a random sound chosen from a list of user-assigned clips. You can see it being used to control different sounds on the Ellen prefab (check under the SoundSources child GameObject of Ellen). For example, looking at FootstepSource, you can see that it defines a list of four footstep sounds that will be randomly picked by the footstep, with an added pitch randomization. That example also includes the use of overrides per specific tile - an override for the Alien ruin tile plays a different footstep sound than if Ellen was walking on a grassier surface.

To add overrides to objects that aren't a tilemap, check the AudioSurface section below.

Scripting side

On the scripting side, RandomAudioPlayer has a PlayRandomSound function that is called by other scripts when a sound needs to be played (e.g. the footstep sound is triggered by the function PlayFootstep in the PlayerController, while the function itself is called by the animation clip on the frame where the feet make contact with the ground.)

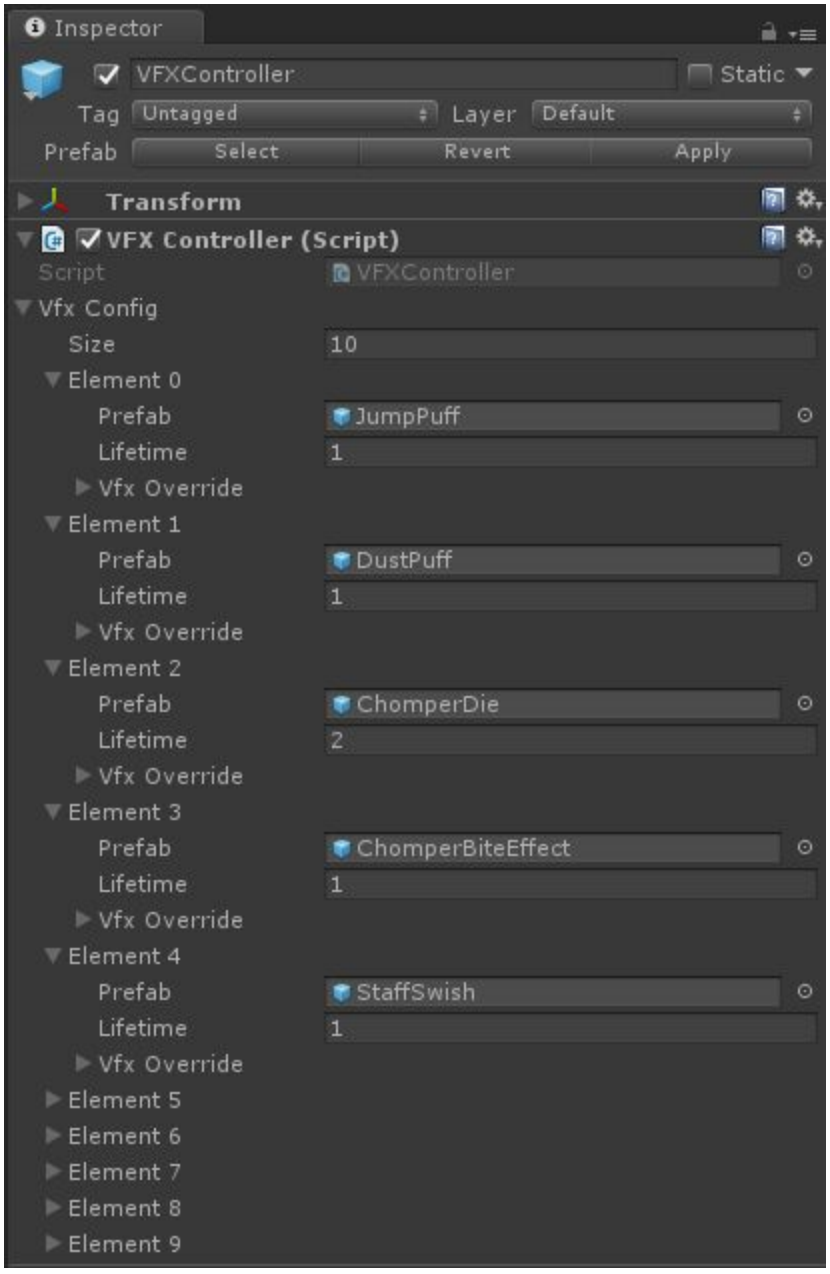
This function can take a TileBase to choose override sounds (e.g. footsteps use the current surface the Player is on, a bullet colliding will use the Tile of the surface it just collide with etc.)

AudioSurface

AudioSurface is a script you can add to any object that is not a tilemap (e.g. a moving or passthrough platform, a moving box etc...) that allows you to define what Tile should be used when the audio Player looks for an override sound (e.g. a stone box would use an AudioSurface with the Alien Tile as the tile setup, so walking on it would trigger the "stone footstep" sound and not the normal "ground" sound.)

VFXController

This component is used to spawn VFX prefabs from a pool, deactivate them and return them to the pool after a lifetime has expired.



VFXController is an asset in the project folder (inside Resources folder) which allows you to list all the VFX the game will use. It will create a pool of instances of those VFX to be used, moving the cost of instantiating the VFX prefab at the beginning of the game instead of every time they are used.

Scripts like the **PlayerController** or the **StateMachineBehaviour TriggerVFX** will then trigger those VFX by name.

Like the Audio Player, it also allows you to define overrides per tile. E.g you can set an override to the VFX used for each footstep depending on which surface the footstep lands on.

(In the case of the VFXController shipped with this project, you can see an example of that on the DustPuff VFX that uses an override for when the Player walks on stone)

The overriding tile is given by the script that triggers the VFX (e.g. footstep will pass the current surface, a bullet hitting a surface will pass which tile is in that surface, etc.)

It is possible to setup the override tile on non-tilemap objects; refer to the sound documentation Sounds.txt for how to setup which override a GameObject should correspond to.

Data Persistence

The data persistence system allows you to save some data during a playthrough, to keep some information on what the Player already did.

Without it, as each zone is a new scene, entering a zone will load the "default" state of that zone (i.e. as it is designed in the editor) but any permanent action the Player took (like grabbing a key or a weapon) would be "undone".

Usage in Editor

The data system works through a scripting interface called IDataPersister.

Objects implementing this interface (like the built-in InventoryItem, HubDoor, PlayerInput, etc.) can write and read data from the PersistentDataManager

Monobehaviours that implement the interface have a Data Settings foldout that appears at the bottom of their inspector. The settings comprise of:

- **Data Tag**: this is a unique identifier for the object, used by the manager to link data to that object. It can be anything: some built-in components use an autogenerated Unique ID, but it could also be a manually typed name, like "Zone_3_key" or "Quest_Item_Card", etc.
- **Persistence Type**: the object can either:
    - Don't Persist: allows disabling persistence, useful for objects that need to be reset on scene change (e.g. you may want a door to close again when restarting a level)
    - Read Only: this object can only read data, not write to it. A way to use this is to have an object with Write Only (see below) with the same Data Tag. This object

will use the data that the other object writes for that tag but won't be able to write over it.
    - Write Only: the object can write data but can't read from it. See above Read Only for example of use.
    - Read Write: most common case of use, the object reads and writes data with its given Data Tag

Data Saving/Loading cycle

SaveData is called on all instances of IDataPersister in the scene before a scene transition out.

LoadData is called on all instances of IDataPersister in the scene after a new scene was loaded.

It is possible to manually save data at any time by calling PersistentDataManager.SetDirty(this) on any IDataPersister.

Example of use in code

The way that the data manager can be used in code is by reading its associated data when it's enabled/started and react to it.

For example, the inventory item writes its state to the persistent data (active or not). So when the scene is loaded, the inventory item retrieves the data associated to its tag. If a false value is saved, that means that the object had already been retrieved and it can disable itself.

Another example would be a door saving its state. When the scene is loaded and data is read, it can set itself to the desired state (e.g. open/closed).

SceneLinkedSMB

SceneLinkedSMB is a class that extends how StateMachineBehaviours work. It allows for Behaviour on states in the Animator state machine to keep a reference to the MonoBehaviour. Although SceneLinkedSMBs can be used wherever you need to reference a specific MonoBehaviour, they were designed with the idea of separating logic and functionality. An animator contm

Example

Enemy.cs

```
public class Enemy: MonoBehaviour
{
	Animator animator;

	void Start()
	{
		animator = GetComponent<Animator>();
		SceneLinkedSMB<Enemy>.Initialise(animator, this);
	}

	public void TrickerAttack()
	{
		// ...
	}
}
```

EnemyAttackState.cs

```
public class EnemyAttackState: SceneLinkedSMB<Enemy>
{
	public override void OnSLStateEnter (Animator animator, AnimatorStateInfo stateInfo, int
layerIndex)
	{
		// m_MonoBehaviour is of type Enemy
		m_MonoBehaviour.TrickerAttack ();
	}
}
```

Object Pooling

The Gamekit uses an extensible object pooling system for some of its systems. The following explanation is only relevant to those wishing to extend the system for their own use and is not required knowledge for working with the Gamekit.

In order to extend the object pool system you must create two classes - one which inherits from **ObjectPool** and the other from **PoolObject**. The class inheriting from ObjectPool is the pool itself whilst the class inheriting from PoolObject is a wrapper for each prefab in the pool. The two classes are linked by generic types. The ObjectPool and PoolObject must have the same two generic types: the class that inherits from ObjectPool and the class that inherits from PoolObject in that order. This is most easily shown with an example:

```
public class SpaceshipPool: ObjectPool<SpaceshipPool, Spaceship>
{
…
}

public class Spaceship: PoolObject<SpaceshipPool, Spaceship>
{
…
}
```

This is so that the pool knows the type of objects it contains and the objects know what type of pool to which they belong. These classes can optionally have a third generic type. This is only required if you wish to have a parameter for the PoolObject's WakeUp function which is called when the PoolObject is taken from the pool. For example, when our spaceships are woken up, they might need to know how much fuel they have and so could have an additional float type as follows:

```
public class SpaceshipPool: ObjectPool<SpaceshipPool, Spaceship, float>
{
…
}

public class Spaceship: PoolObject<SpaceshipPool, Spaceship, float>
{
…
}
```

By default a **PoolObject** has the following fields:

- **inPool**: This bool determines whether or not the PoolObject is currently in the pool or is awake.
- **instance**: This GameObject is the instantiated prefab that this PoolObject wraps.
- **objectPool**: This is the object pool to which this PoolObject belongs. It has the same type as the ObjectPool type of this class.

A **PoolObject** also has the following virtual functions:

- **SetReferences**: This is called once when the PoolObject is first created. Its purpose is to cache references so that they do not need to be gathered whenever the PoolObject is awoken, although it can be used for any other one-time setup.
- **WakeUp**: This is called whenever the PoolObject is awoken and gathered from the pool. Its purpose is to do any setup required every time the PoolObject is used. If the classes are given a third generic parameter then WakeUp can be called with a parameter of that type.
- **Sleep**: This is called whenever the PoolObject is returned to the pool. Its purpose is to perform any tidy up that is required after the PoolObject has been used.
- **ReturnToPool**: By default this simply returns the PoolObject to the pool but it can be overridden if additional functionality is required.

An **ObjectPool** is a MonoBehaviour and can therefore be added to GameObjects. By default it has the following fields:

- **Prefab**: This is a reference to the prefab that will be instantiated multiple times to create the pool.
- **InitialPoolCount**: The number of PoolObjects that will be created in the Start method.
- **Pool**: A list of the PoolObjects.

An **ObjectPool** also has the following functions:

- **Start**: This is where the initial pool creation happens. You should note that if you have a Start function in your ObjectPool it will effectively hide the base class version.
- **CreateNewPoolObject**: This is called when PoolObjects are created and calls their SetReferences and then Sleep functions. It is not virtual, so it cannot be overridden but it is protected so can therefore be called in your inheriting class if you wish.
- **Pop**: This is called to get a PoolObject from the pool. By default it will search for the first one that has the inPool flag set to true and return that. If none are true it will create a new one and return that. It will call WakeUp on whichever PoolObject is going to be returned. This is virtual and can be overridden.
- **Push**: This is called to put a PoolObject back in the pool. By default it just sets the inPool flag and calls Sleep on the PoolObject but it is virtual and can be overridden.

For a full example of how to use the object pool system please see the BulletPool documentation and scripts.

BulletPool

The **BulletPool** MonoBehaviour is a pool of **BulletObjects**, each of which wraps an instance of a bullet prefab. The BulletPool is used for both Ellen and enemies but is used slightly differently for each. For Ellen there is a BulletPool MonoBehaviour attached to the parent GameObject with the Bullet prefab set as its Prefab field. The other use of BulletPool is in the EnemyBehaviour class. It uses the GetObjectPool static function to use BulletPools without the need to actively create them.

The **BulletPool** class has the following fields:

- **Prefab**: This is the bullet prefab you wish to use.
- **Initial Pool Count**: This is how many bullets will be created in the pool to start with. It should be as many as you expect to be used at once. If more are required they will be created at runtime.
- **Pool**: This is the BulletObjects in the pool. This is not shown in the inspector.

The **BulletPool** class has the following functions:

- **Pop**: Use this to get one of the BulletObjects from the pool.
- **Push**: Use this to put a BulletObject back into the pool.
- **GetObjectPool**: This is a static function that will find an appropriate BulletPool given a specific prefab.

When getting a bullet from the pool it comes in the form of a **BulletObject**. The BulletObject class has the following fields:

- **InPool**: Whether or not this particular bullet is in the pool or being used.
- **Instance**: The instantiated prefab.
- **ObjectPool**: A reference to the BulletPool this BulletObject belongs to.
- **Transform**: A reference to the Transform component of the instance.
- **Rigidbody2D**: A reference to the Rigidbody2D component of the instance.
- **SpriteRenderer**: A reference to the SpriteRenderer component of the instance.
- **Bullet**: A reference to the Bullet script of the instance.

The **BulletObject** has the following functions:

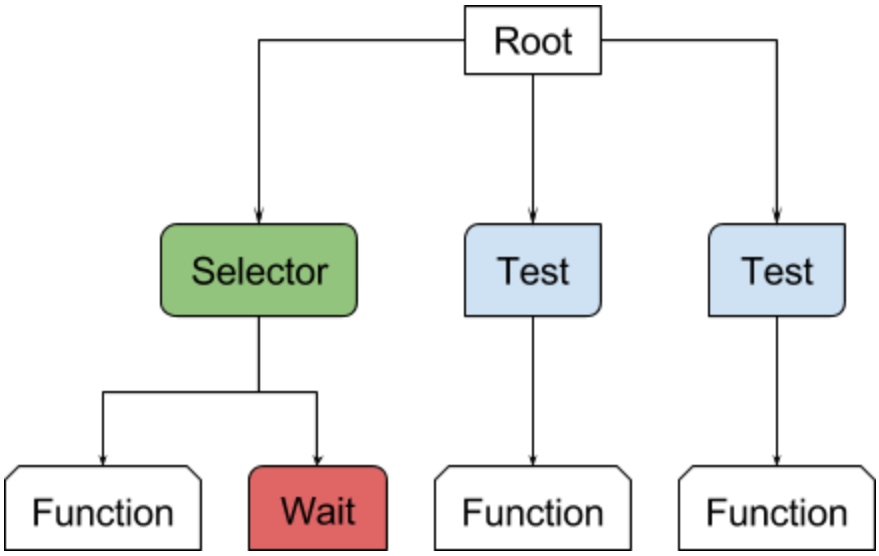
- **WakeUp**: This is called by the BulletPool when its Pop function is called.
- **Sleep**: This is called by the BulletPool when its Push function is called.
- **ReturnToPool**: This should be called when you are finished with a particular bullet. It will call the Push function of its BulletPool and so call its Sleep function.

Behaviour Tree

Note: This is a scripting only system, you need at least basic knowledge in how scripts and C# work in Unity to understand how it works and how to use it

As the name implies, Behaviour Trees are a way to code behaviour using a tree of actions. They are used in the Game Kit to control the behaviour of AI for enemies and the boss battle sequence.

A visual example of a Behaviour Tree is shown below:



Theory

During each update of the game, the tree is "ticked", meaning it goes through each child node of the root, going further down if said node has children, etc.

Each node will have actions associated, and will return one of three states to its parent:

- **Success**: the node successfully finished its task.
- **Failure**: the node failed the task.
- **Continue**: the node didn't finish the task yet.

Returned state is used by each node parent differently.

For example:

- *will make the next child active if the current one returned **F** or and keep the current one active if it returned **C** .*
- *node would call their child and return the child state if the test is true, and return **F** without calling their children if the test is false.*

Game Kit Implementation

The way Behaviour Trees are used in the game kit is through script. Here is an example of a very simple behaviour tree:

First, we need to add **using BTAI;** at the top of the file.

```
Root aiRoot = BT.Root();
aiRoot.Do(
 BT.If(TestVisibleTarget).Do(
   BT.Call(Aim),
   BT.Call(Shoot)
 ),
 BT.Sequence().Do(
   BT.Call(Walk),
   BT.Wait(5.0f),
   BT.Call(Turn),
   BT.Wait(1.0f),
   BT.Call(Turn)
 )
);
```

aiRoot should be stored in the class as a member, because you need to call **aiRoot.Tick()** in the **Update** function so that Tree actions can be executed.

Let's walk through how the **Update** will go in the **aiRoot.Tick()**:

- First, we test if the function **TestVisibleTarget** returns true. If it does, it goes on to execute the children, which are calling the functions **Aim** and **Shoot**
- If the test returns false, the **If** node will return **Failure** and the root will then go to the next child. This is a **Sequence**, which starts by executing its first child
    - This calls the function **Walk**. It returns **Success** so that the **Sequence** sets the next child as active and executes it
    - The **Wait** node executes. Since it has to wait for 5 seconds and was just called for the 1st time, it will not have reached the wait time, so it will return **Continue**

    - As the Sequence receives a **Continue** state from its active child, it won't change the active child, so it will start from that child on the next **Update**
- Once the Wait node has been updated enough to reach its timer, it will return **Success** so that the sequence will go to the next child.

Nodes List

Sequence

Execute child nodes one after another. If a child returns:

- **Success**: the sequence will tick the next child on the next frame.
- **Failure**: the sequence will return to the first child on the next frame.
- **Continue**: the sequence will call that node again on the next frame.

RandomSequence

Execute a random child from its list of children every time it is called. You can specify a list of weights to apply to each child as an int array in the constructor, to make some children more likely to be picked.

Selector

Execute all children in order until one returns **Success**, then exit without executing the remaining children nodes. If none return **Success** then this node will return **Failure.**

Call

Call the given function, which will always return **Success.**

If
Call the given function.

- If it returns true, it calls the current active child and return its state.
- Otherwise, it will return **Failure** without calling its children

While
Return **Continue** as long as the given function returns true (so the next frame when the tree is ticked, it will start from that node again without evaluating all the previous nodes).

Children will be executed one after another.

Will return **Failure** when the function returns false and the loop is broken.

Condition

This node returns **Success** if the given function returns true, and returns **Failure** if false.

Useful when chained with other nodes that depend on their children's result (e.g Sequence, Selector etc.)

Repeat
Will execute all child nodes a given number of times consecutively.

Always returns **Continue** until it reaches the count, where it returns **Success.**

Wait
Will return **Continue** until the given time has been reached (starting when first called), where it will then return **Success.**

Trigger
Allows the setting of a trigger (or unsetting of a trigger if the last argument is set to false) in the given animator. Always returns **Success.**

SetBool
Allows setting the value of a Boolean Parameter in the given animator. Always returns **Success**

SetActive
Set active/inactive a given GameObject. Always returns **Success.**