**Hit Box Program Document**

**General**

This document keeps track of the implementation of the game mechanics.

**Tools**

* Unity (ver. 2021.3.30f)
* Visual Studio 2022 (C#)

# **Player Object**

## 1. Principal Object - Character\_00\_Name\_Object

Is the main object of the object, it contains all the children and essential components for the character.

### 1.1 - Transform

Determines the position, rotation, and scale of the object.



### 1.2 - RigidBody

Provides the physics to an object.

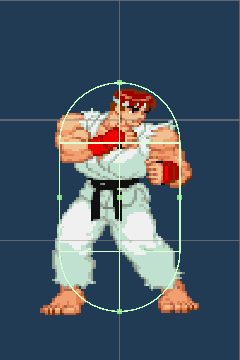
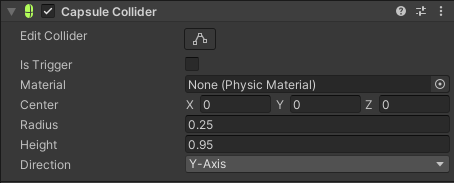


*Collision Detection –> Continuous:* The best method for accuracy in detections.

*Constraints:* The game is in 2D so we only need the X and Y Axis.

### 1.3 - Capsule Collider

Determines the space the character occupies. We use *Radius* and *Height* to move the boundaries.

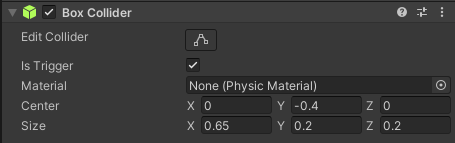


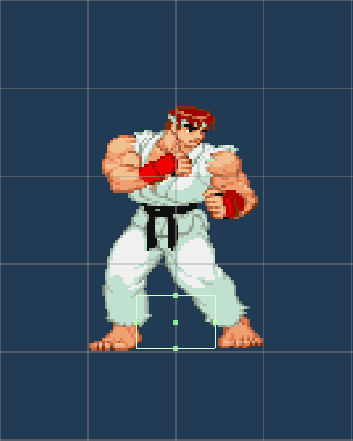
### 1.4 - Box Collider

Determines the space of the ground detection for the Scr\_02\_State\_Manager

The Collider is a Trigger because we don’t need to interact with solid objects.

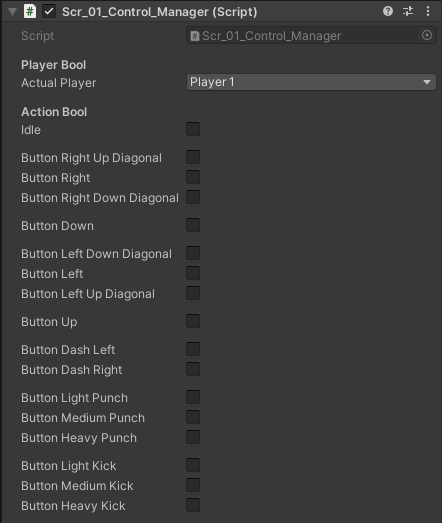
It always goes at the bottom of the character, but for detection purposes, it has to be smaller than the CapsuleCollider. We use *Center* and *Size* to move the boundaries.





### 1.5 - Scr\_01\_Universal\_Control\_Manager

Is a universal Scrpit that manages the inputs of Player 1 and Player 2.



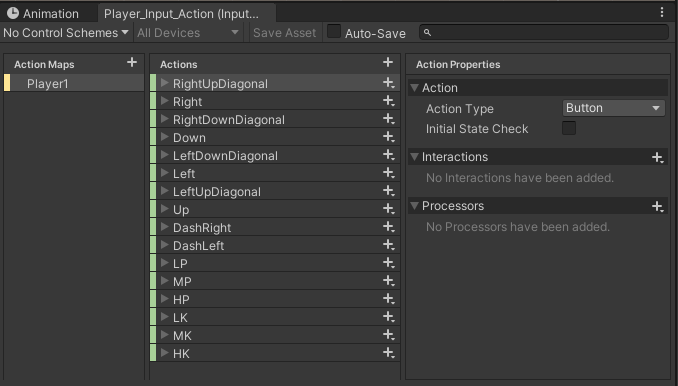
*Actual Player:* Determines the set of controls of that object, the options are *Player 1 and Player 2.*

*Action Bool:* Depending on the set of controls, when an input is detected, its respective box will be marked. When no input is detected the default box *Input* will be marked.

#### Input System Package

For the Inputs, we use the Input System Unity Package, which allows us to better handle the controls and assignations. This Package will generate a component and a script that we can reference to code.

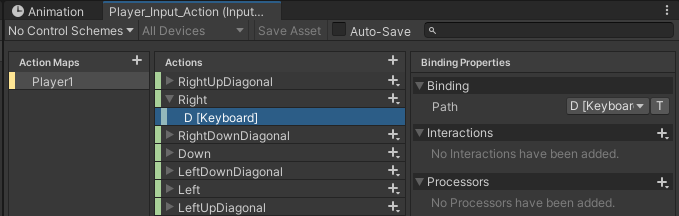




To add, modify or remove an input we use the Input Actions window in the Unity Editor.

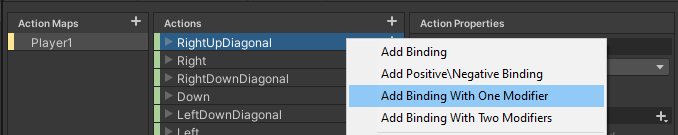
Action Maps refers to the set of controls that the game presents, in this case, Player1 refers to all the inputs of the playe 1 in the Keyboard.

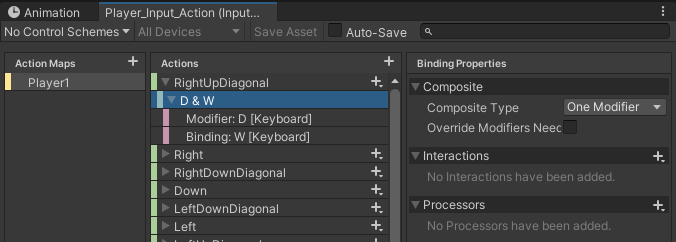
Actions are all the Inputs that the game has, click on the plus to create a new one.



Inside of each action is the Binding, the key or input that we use to execute the action.

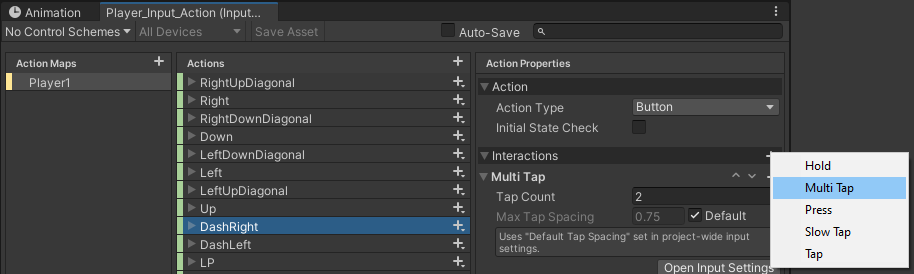
**Two Key Actions**

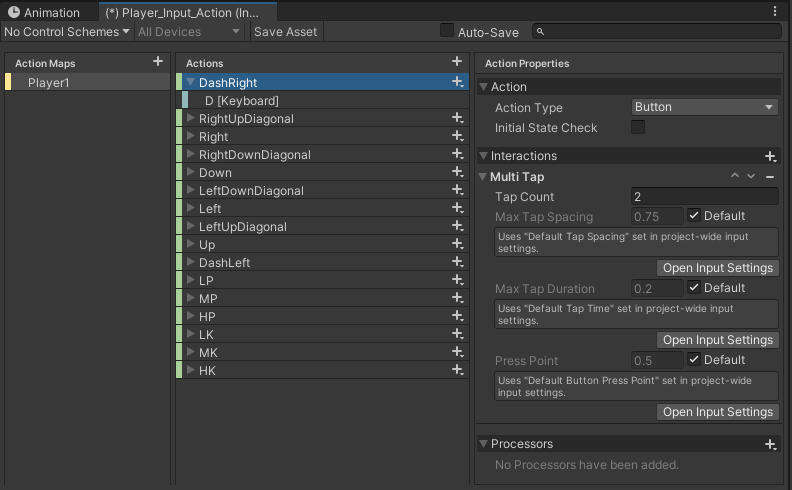




To make an action execute with only two keys, we right-click in the action and choose Add Binding With One Modifier, which will add us to Binding with the space to add two Keys.

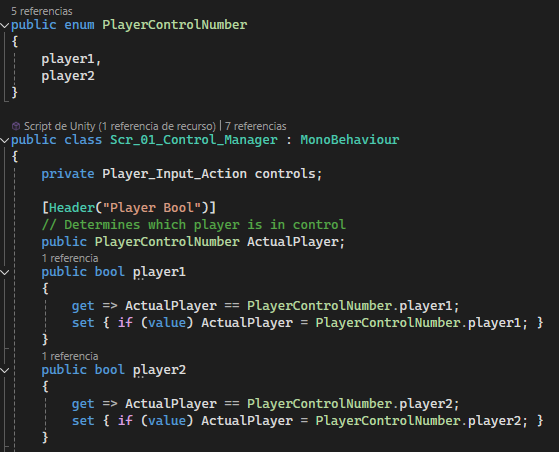
**Double Tap Actions**





To make an action activate only when the input is pressed twice, in the action properties, in interactions we add Multi-Tap, by default has the tap count in 2, but the parameters can be adjusted.

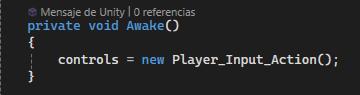
#### Code: Player Assignation



PlayerControlNumber contains the number of control configurations that may exist.

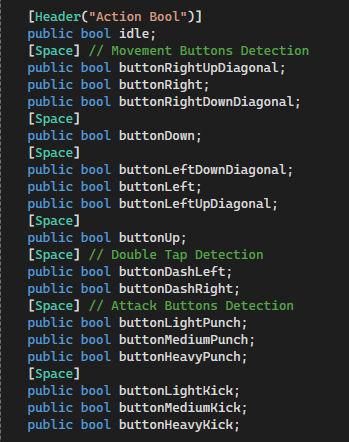
ActualPlayer Determines the set of controls of that object.

The rest of the code works so that it appears in the Unity UI and that only one option can be chosen.

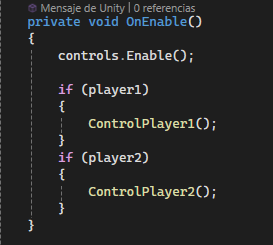


Controls is the reference to the Player\_Input\_Action script, to extract the inputs and the actions.

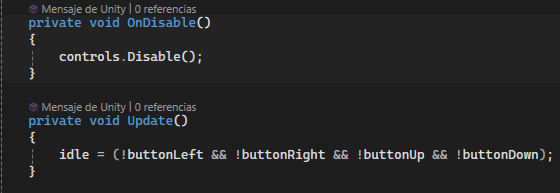
#### Code: Inputs



Action Bools refers to all the buttons on the control.

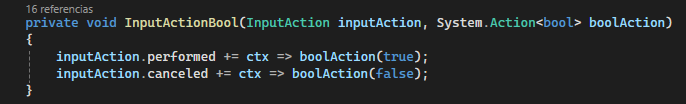


We use the OnEnable method to enable the controls when in use, and to check what bool is active (player1 or player2) and assign the controls stored in methods ControlPlayer1() or ControlPlayer2().



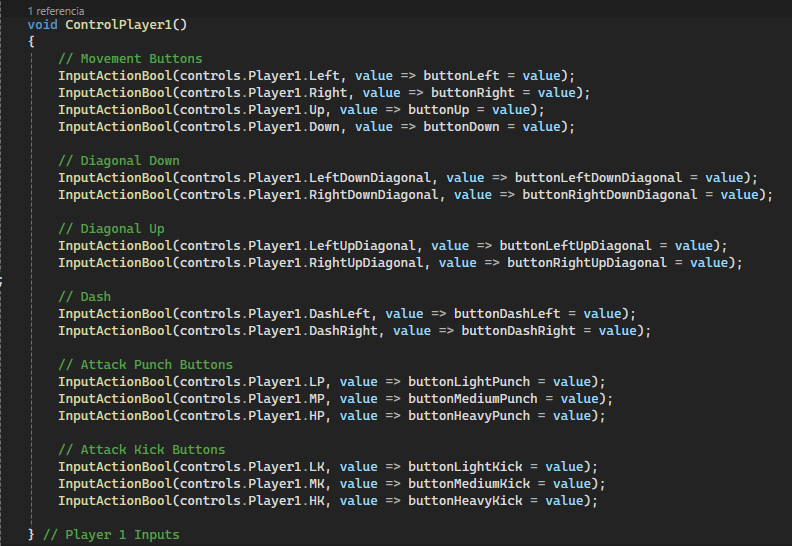
The method OnEnable and controls.Enable() is called when the script is use, the OnDisable, and controls.Disable() is calld when the script isnt in use.

Idle is activated when the main inputs are not activated, this code is in the Update(), because otherwise it won't work properly.



The InputActionBool() function is what connects the Player\_Input\_Action script with the action bools.

inputAction detects the action in the Player\_Input\_Action script, and depending on his state, it turns the boolAction.

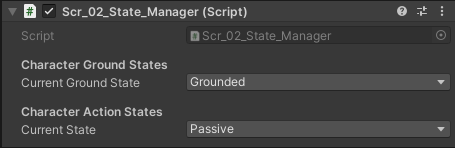


The ControlPlayer1 and ControlPlayer2 method is where the control inputs are gathered.

Assigning to the InputActionBool function the input and the bool of the corresponding action.

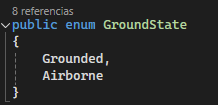
### 1.6 - Scr\_02\_Universal\_State\_Manager

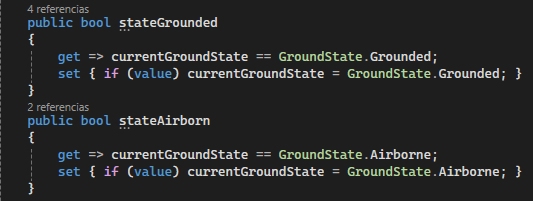
Is a universal Scrpit that manages states of the Object.



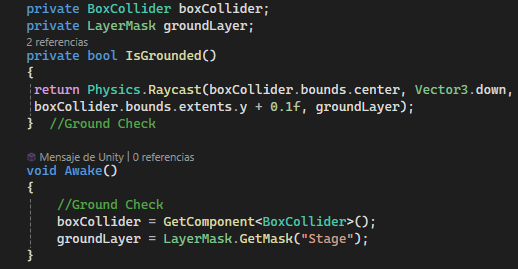
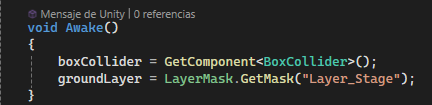
*Current Ground Check* determines if the player is *Grounded* or *Airborn*.

#### Code: Ground Check

**

**

The ground check works with two bools stateAirborn and stateGrounded, these activate depending on what state the object is in.

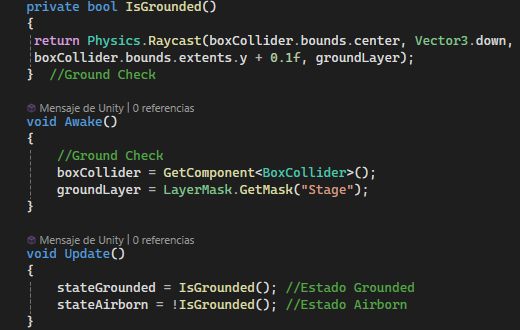
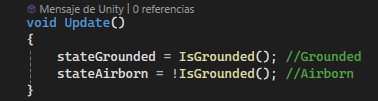
 

The detection works by combining the variable boxCollider and groundLayer.

The variable boxCollider is assigned to the component 1.4 - Box Collider, which determines the location and size of the detection box.

The variable groundLayer determines which layer the boxCollider should interact with to activate and deactivate bools, in this case, "Layer\_Stage".

The bools stateAirborn and stateGrounded will work with every object with the Layer “Layer\_Stage”.

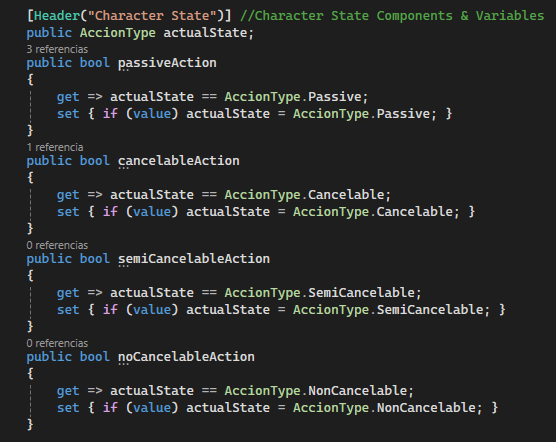
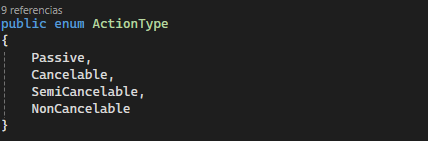
 

IsGrounded() is the principal method of the ground check.

It performs a ray cast from the center of the boxCollider downwards that detects objects with the Layer “Layer\_Stage”.

If this method is true, the bool stateGrounded is true, if the method is false, the bool stateAirborn is true.

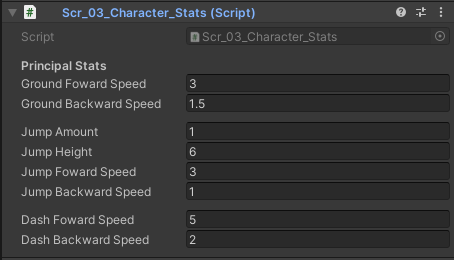
#### Code: Action State



The ActionType is a list of all the character states, te explanation of each state is in **[04 – Program Outlines –> 2 - State Manager]**.

### 1.7 - Scr\_03\_Universal\_Character\_Stats

Is a universal Scrpit that manages the numeric value of every stat the character has.



*Ground Foward Speed:* Determines the velocity of the character moving forward in the ground.

*Ground Backward Speed:* Determines the velocity of the character moving backward in the ground.

*Jump Amount:* Determines the amount of jumps a character has, the minimum is 1.

*Jump Height:* Determines height and distances of the jumps.

*Jump Forward Speed*: Determines the velocity of the character jumping forward.

*Jump Backward Speed:* Determines the velocity of the character jumping bacward.

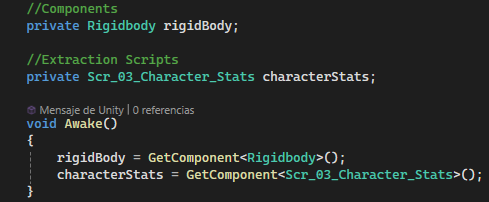
*Dash Foward Speed:* Determines the velocity of the character dashing forward.

*Dash Backward Speed:* Determines the velocity of the character dashing backward.

### 1.8 - Scr\_04\_Universal\_Physics\_Manager

Is a universal Scrpit that manages the basic physics of a character, those that everyone shares without exception.

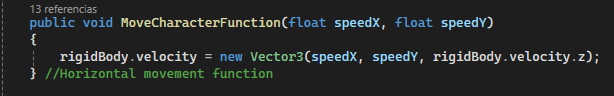
#### Code: Components



To control the object's movement, we need to manipulate the X and Y axis of the rigidBody.

The characterStats determines the parameters that will be used to control the character.

#### Code: Character Move



The MoveCharacterFunction() function is what makes the character move horizontally.

Take the SpeedX ​​parameter and determine the movement in the X axis of rigidBody.velocity.

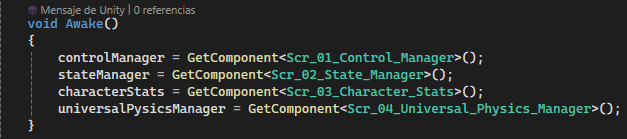
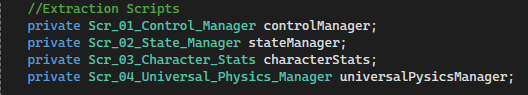
Take the SpeedY ​​parameter and determine the movement in the Y axis of rigidBody.velocity.

The Z axis stays the same.

### 1.9 - Scr\_05\_Universal\_Action\_Manager

Is a universal Scrpit that manages the basic actions of a character, those that everyone shares without exception.

#### Code: Components



The controlManager detects which key is being pressed, determining the character's action.

The stateManager determines the state of the character.

The characterStats determines the parameters that will be used to control the character.

The universalPysicsManager manages the physics of actions shared between all characters.

#### Code: Actual Action

The variable actualAction is a string that shows what action is being performed by the character.



We have to update it since other scripts are going to use the variable. The most important actions are documented **[03 – List of Labels -> 03 – Character Actions].**

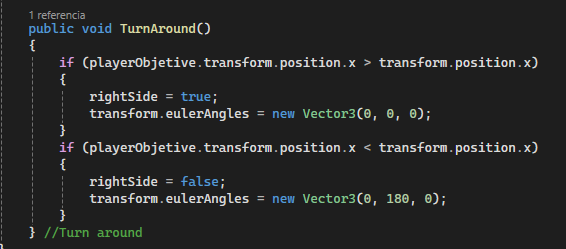
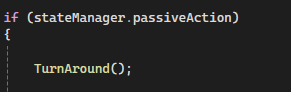
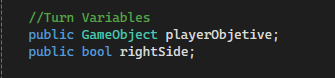
#### Code: Void

We use the method FixedUpdate() to code everything related to the character's physics.

#### Code: State Grounded and Passive Action

We put our code in stateManager.stateGrounded and stateManager.passiveAction if the action is of type Passive on the ground.

***Turn Around***

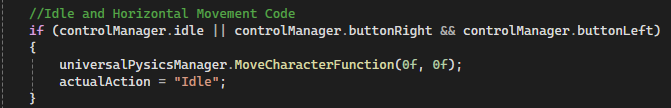


To turn around we change the value of Vector3 Y, and we use two vriables, the playerObject is the objet that the player is goin to follow, and the rightSide bool si on if the character is facing right.

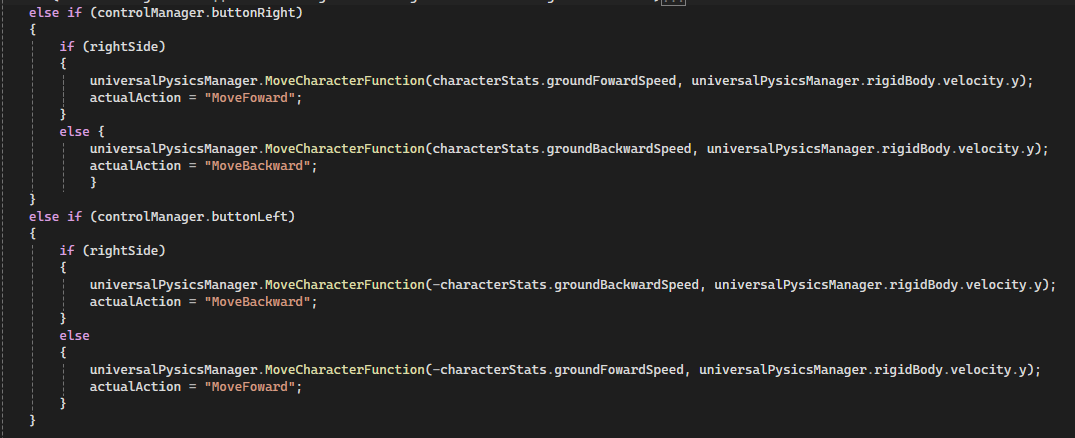
The charcter only turns around if its not performing any actions in the ground.

***Idle and Basic Horizontal Movment***

The horizontal movement is composed of a series of If statements.



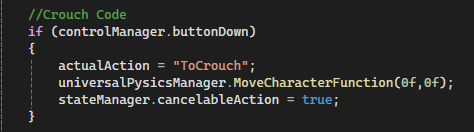
The first if prevents the character from moving if both or none inputs are pressed.



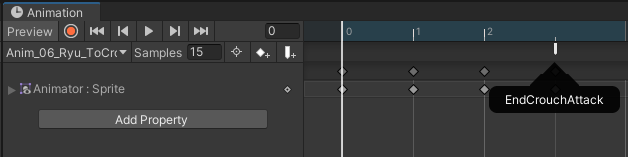
The two else if, check what button is being pressed, if controlManager.buttonWright or controlManager.buttonLeft. Inside of this statmens is other that determines the animation and velocity depending on what direction the character is facing.

***Crouch***

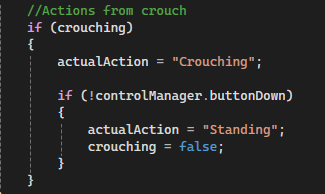
The Crouch code is divided in three.



The first part is inside of if (stateManager.stateGrounded && stateManager.passiveAction). When activated the character stops his movment and changes to cancelableAction.

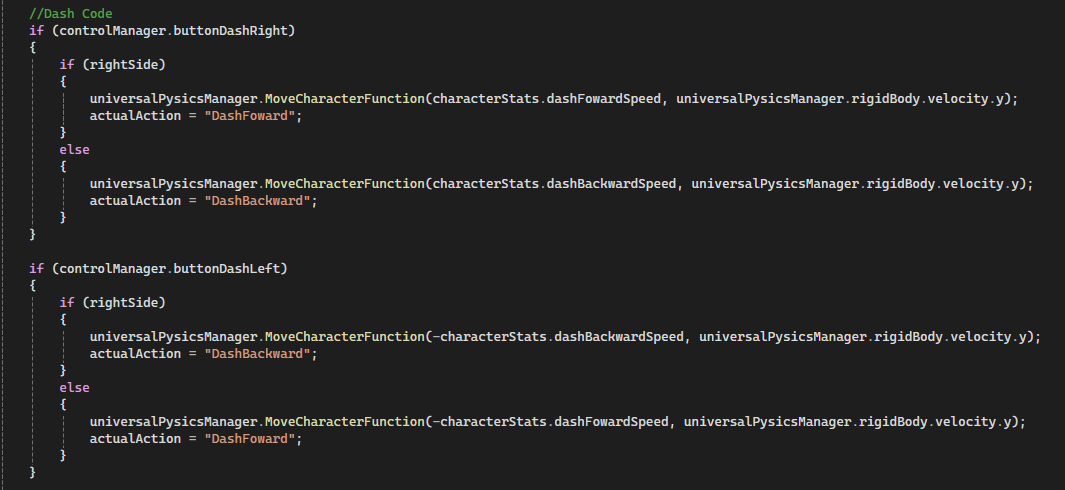


At the end of the ToCrouch animation, the EndCrouchAttack event activate, this changes the bool crouching to true and change the stateManager to cancelableAction.

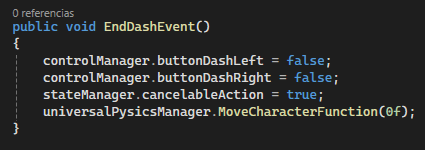
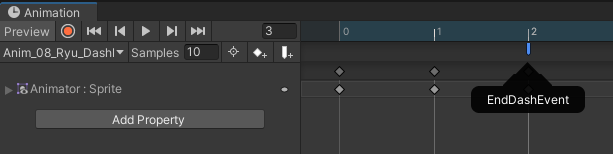


The second part is inside of if (stateManager.stateGrounded && stateManager.cancelableAction). If crouching is true, the actualAction is “Crouching”, if we stop pressing the button it will change to “Standing” and crouching will change to false.

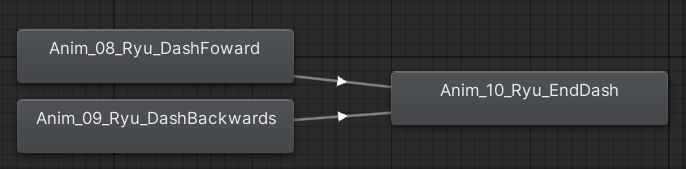
***Dash***

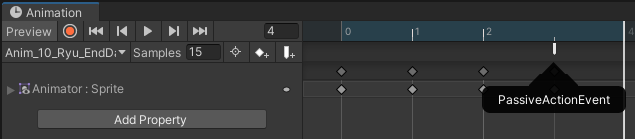


The dash movement is composed of a series of If statements, the same as horizontal movment. The differences is in some extra steps.



Both dash have at the end of their animation the End Dash Event, this one shuts down both control dash variables, it changes the state to Cancelable [For now], and stop the horizontal movment. This is because after a dash the character has to recover.

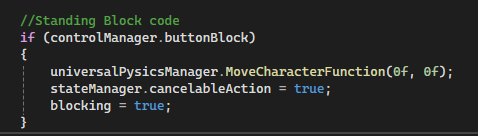




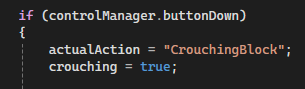
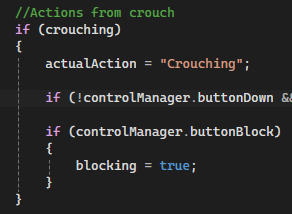
The recovery activates after the dash animation ends, at the end of this one is the Passive Action Event so the character can move again.

***Block***

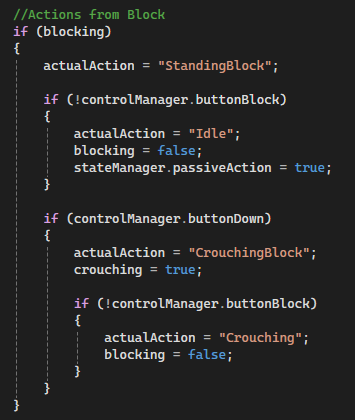
The Block code is divided in three.



Standing Block is inside of if (stateManager.stateGrounded && stateManager.passiveAction). When activated the character stops his movment and changes to cancelableAction.



Crouching Block is inside of if (crouching).



The last part is inside of if (stateManager. cancelableAction). When the bool blocking is true, actualAction is “StandingBlock”, and if controlManager.buttonDown then actualAction is “CrouchingBlock”.

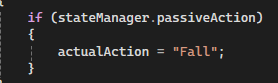
When we release the controlManager.buttonBlock, actualAction changes to the previous default state “Idle” or “Crouching”.

#### Code: State Airborn and Passive Action

We put our code in stateManager.stateAirborn and stateManager.passiveAction if the action is of type Passive on the air.

***Fall***

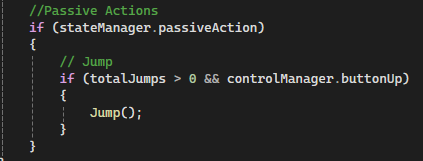
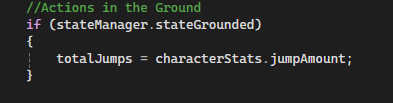
When not performing an action in the air, the character will Fall. It activates automatically after jumping.



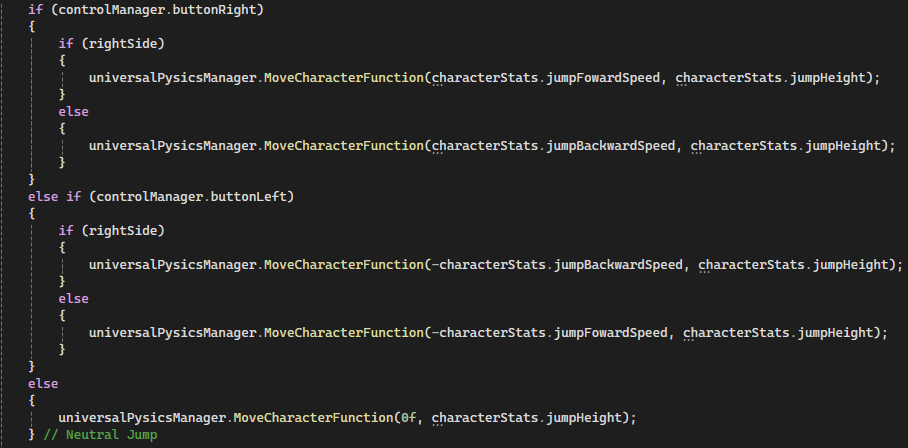
#### Code: Passive Action

We put our code in stateManager.passiveAction if the action is of type Passive.

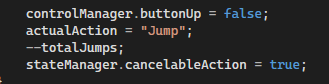
***Jump***



The jump can be performed in any state as long as totalJumps is more than zero. This variable resets every time the character is in the ground.



The tree else if, check what button is being pressed, to determine a neutral Jump, Forward jump o backward Jump. Inside of this statement is another that determines velocity depending on what direction the character is facing.



controlManager.buttonUp is set to false so that doesn't trigger the double jump right away.

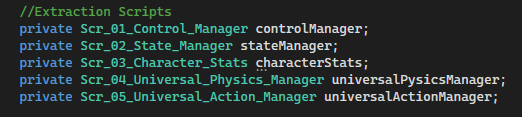
Every time the function executes, one is subtracted from the variable totalJumps, to keep track of how many jumps are left.

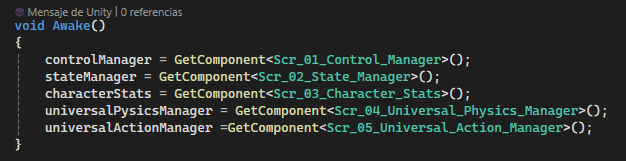
Jumps are Cancelable actions, so we put stateManager.cancelableAction = true.

### 1.10 - Scr\_08\_Character0\_Action\_Manager

Is a unique Scrpit that manages the Attacks, Command Inputs and Special Moves of a specific character.

#### Code: Components





We extract the Inputs from the Action Bools of controlManager.

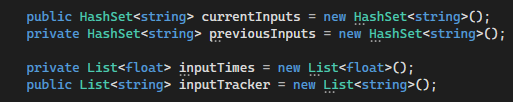
The stateManager determines the state of the character.

The characterStats determines the parameters that will be used to control the character.

The universalPysicsManager manages the physics of actions shared between all characters.

The universalActionManager manages the basic actions of the character.

#### Code: Storage of Inputs

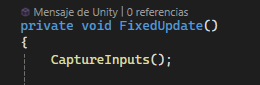


The first two, currentInputs and previousInputs, keep track of the current and previous frame input.

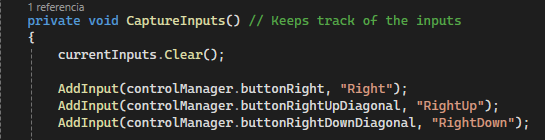
inputTimes keeps track of the time of each input, this is to check the right timing in special moves.

The last array inputTracker is the filtered list of inputs, the ones that are detected for special moves and appear in the Unity UI.

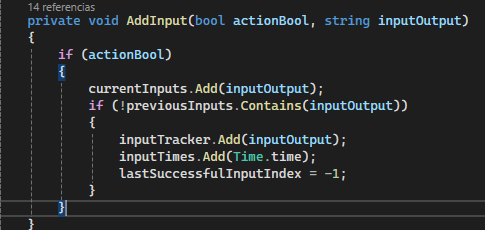
#### Code: Get Inputs



We use the function CaptureInputs() to track the controlManager Action Bools.



currentInputs.Clear() exists to clear current inputs at the start of each frame. To track each individual input we use the AddInput() function.

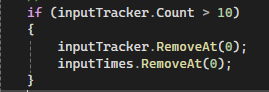


In this function we pass two variables, the action bool of the Scr\_01\_Universal\_Control\_Manger, and the string we want to us te be the output input.

When actionBool is correct, we add the input to the array with currentInputs.Add(inputOutput).

The next if ensures that an input is only added to the tracking system if it hasn’t been added before. It is also responsible for retaining the input time, to later take the timing into account when a Special Move is made.

The variable lastSuccessfulInputIndex is reset to -1. To keep track of the index in inputTracker corresponding to the last input that successfully triggered a special move or another important action.



At the end of the function, we have an if that determines that only the last ten inputs can be stored, if it's exceeded the oldest gets eliminated.



This part handles multiple input presses.

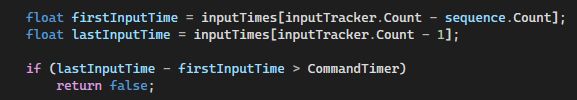
#### Code: Detecting Combinations



This function is passed a list containing the combination of inputs. The function is also a bool, and if it passes a series of loops it returns true.



This first part checks the length of the sequence, and if it is not the same as the sequence, the function returns false.



The second part is responsible for the timing of the combination, if it is not done in less than the CommandTimer value it returns false.

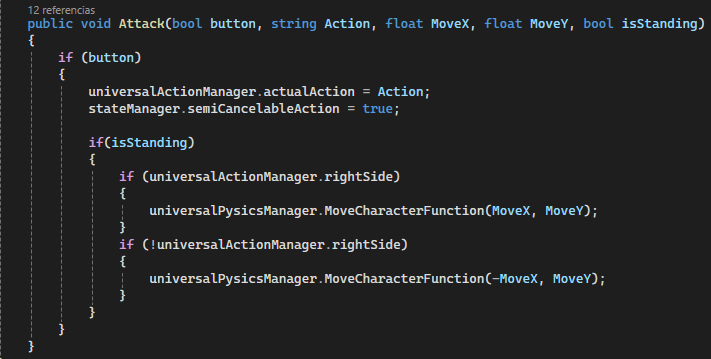


This check ensures that the same sequence doesn’t trigger the special move again unless new inputs have been added since the last successful sequence.



If the length, the timing and the combination of inputs passes all this loops then the function returns true.

#### Code: Attack Function



This function executes every attack and the necessary parts. It takes five variables.

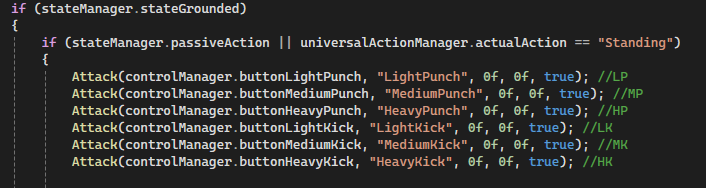
The bool button determines the button that will execute the attack.

The string Action determines the actualAction in the universalactionManager.

Floats MoveX and MoveY determines the movment of the attack in each axis, if it has one.

The bool isStanding determines if the character is standing or not, this will influence the movement, on the ground the attacks leave the player still while in the air he has to keep moving while executing it.

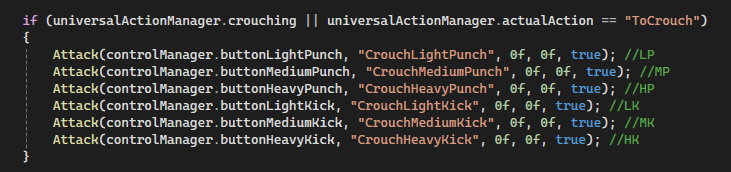
***Standing Attacks***



The standing attacks are executed when the character is in stateManager.stateGround and stateManager.passiveAction.

The last variable has to be true to enable the character movement If the attack has one or to freeze the character in place if X and Y are 0.

***Crouch Attacks***



The crouch attacks are executed when the character universalActionManager.crouching is true.

***Transition Attacks***

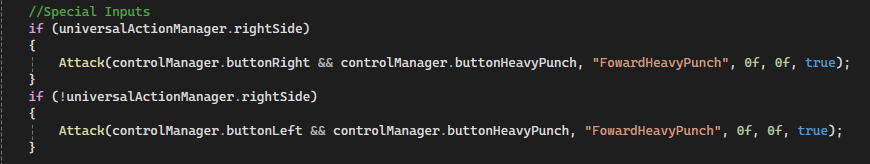




The character can attack while transitioning between states, the attacks that will come out are determined by the state that is going to.

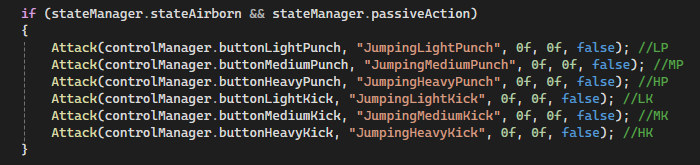
While standing to crouch are Crouch Attacks, and while crouch to standing are Standing Attacks.

***Command Inputs Attacks***



The Command Normals uses the Attack function butt with two bool variables. The function is inside two if with universalActionManager.rightSide because some inputs depend on where the character is facing.

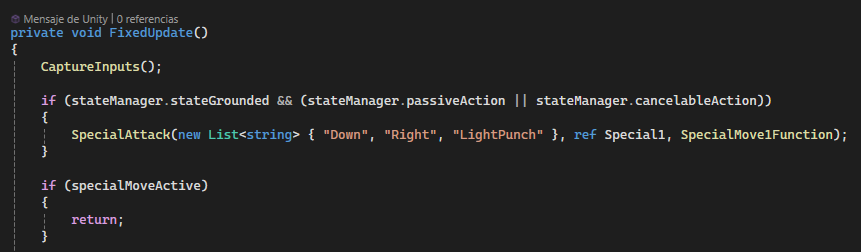
***Jump Attacks***



The jump attacks are executed when the character is in stateManager.stateAirborn and stateManager.passiveAction.

The last variable has to be false to enable the character to move while attacking in the air unless the move stops it.

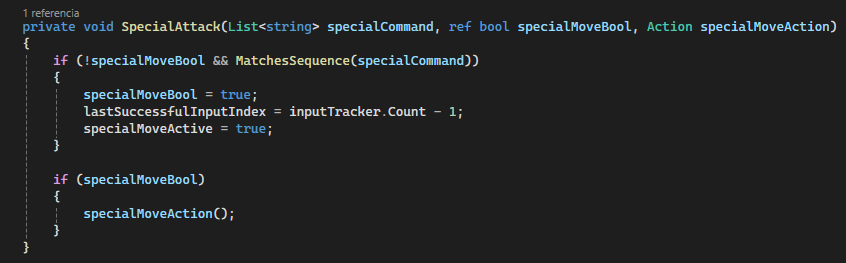
#### Code: Executing Special Moves



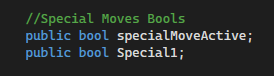
To execute special movements we first need to pass the conditions under which the attack is fulfilled in FixedUpdate().

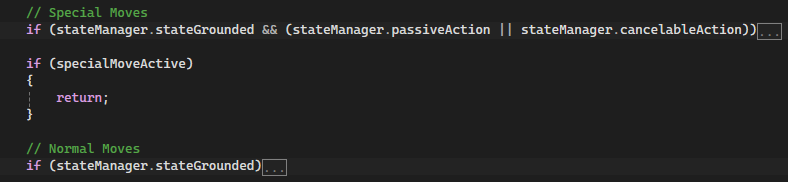
In this case, this move is executed only in the Grounded and If the player is in passiveAction or cancelableAction.

We use the function ExecuteSpecialMove() to do that.



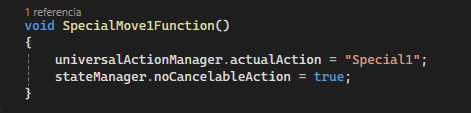
We pass three defining variables, specialCommand is the list of inputs we need to do the move, specialMoveBool refers to the designated bool, this is to confirm when the move is active and specialMoveAction is the function that contains the behavior of the move.

**



We activate the bool specialMoveActive to prevent the use of Normal Moves when using it in a combination.

Special Moves Function



Since each special move has its own characteristics we need to create a function for each on of them. Also depending on what this function needs the script components will change.

## 2. Objeto Hijo 1 - Animator

Is the object that contains all the components to manage animations.

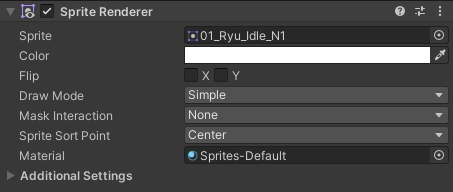
### 2.1 - Transform

Determines the position, rotation, and scale of the object.   
It needs to be in Position (0,0,0) to stay in the center of the main object at all times.



### 2.2 – Sprite Renderer

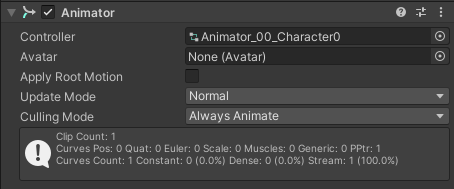
This component shows the sprites and images of the object.



### 2.3 – Animator

This component manages the animations of the object.

The *Controller* is the set of animations and connections of a character.



*How to set up animations:*

The Animations are done in the *Animation* window.



To create a new animation we go to the box below *Preview* and then *Create New Clip.*



Every animation is an anim file. We need to create one with the right name structure:  
*Anim\_[Number]\_[[Character]\_[Animation Name]*

Ej: *Anim\_01\_Character0\_Idle*



Then we need to bring the Sprites that make up each animation frame, which are represented by rhombuses on the timeline.

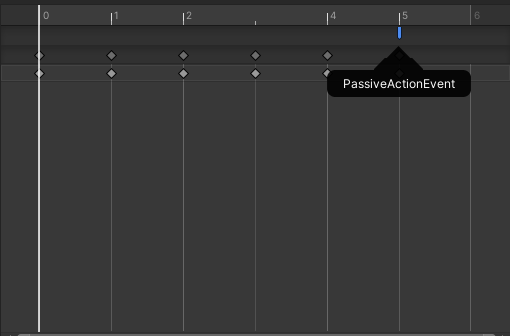
The *Samples* is the velocity at which every frame is reproduced. The base is 60 but the appropriate speed is 15.



### 2.4 - Scr\_06\_Universal\_Animation\_Events

Is a universal Scrpit that manages the events of animations, those that everyone shares without exception.

An event is a function that can be programmed to happen in a certain frame in an animation. The most important events are documented **[03 – List of Labels -> 06 – Universal Animation Events].**



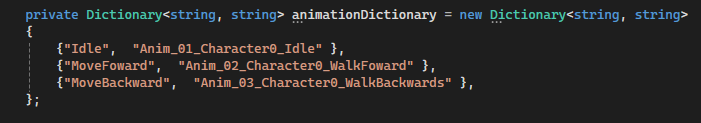
If the event has to be triggered on the last frame, then you have to duplicate it and place the event on the second one.

### 2.5 - Scr\_07\_Character0\_Animation\_Manager

Is a unique Scrpit that manages the animations of the character. Is unique because we have to change animation variables to match the character we want.



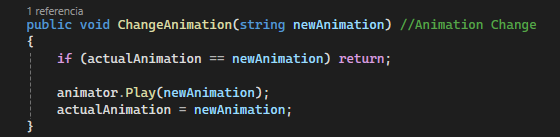
The characterAction determines the action that is being performed in order to assign the correct animation.



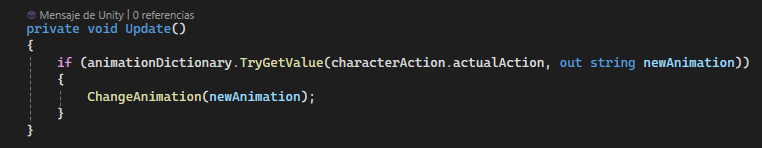
We use animationDictionary to get values ​​from universal scripts and translate them to what we want. For example, every character has Idel but the animation but the assigned animation is different.

On the left are the universal values ​​and on the right are the values ​​to which we want them to be translated.

The universal values are documented **[03 – List of Labels -> 02 – Animations].**

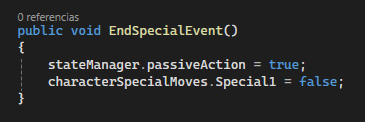


To change animations we use the method. It takes a string value, which must be the name of the animation that we want to play, and compares it with the current animation so that if it comes twice it does not stop. If it is different then it tells the animator to reproduce it.



We use this if to combine animationDictionary and ChangeAnimation(), it takes the universal value, translates it, and then sends it to be reproduced.

Special Moves Events



Since each character has its own scripts and characteristics, it is necessary to create the events in this script, since both will always be in the same character and can reference each other.