**Documento de Programa**

**Tools**

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

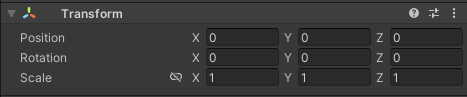
# **Objetos**

## 1. Player

### 1.1 Player\_GameObject (Principal)

#### 1.1.1 - Transform

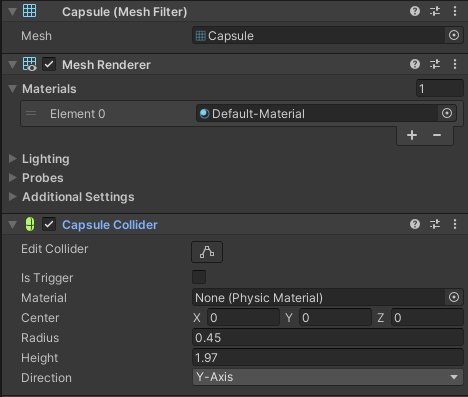
Determina la posición, rotación y escala del objeto.



#### 1.1.2 – Capsule Collider y Renderer

Determina el espacio que ocupa el personaje. Solo aspectos visuales son en 2D, pero la forma del jugador es en 3D.

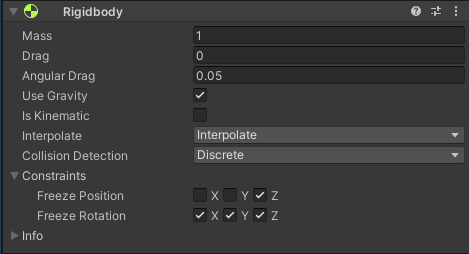
Usamos *Radius* y *Height* para mover los límites.





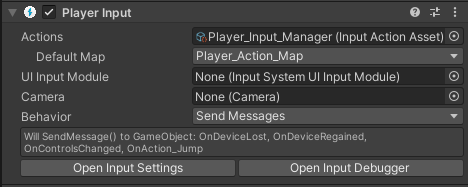
#### 1.1.3 - RigidBody

Proporciona la física a un objeto.



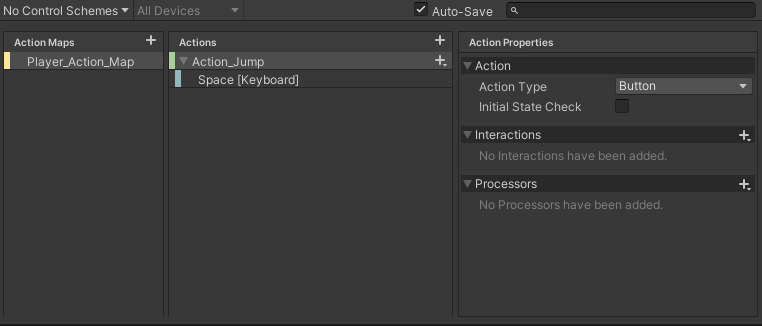
#### 1.1.4 – Player Input

Este componente es el encargado de manejar el Input System de Unity.



*Default Map* se refiere al mapa de acciones al que accedemos por defecto. El mapa de acciones contiene dentro las acciones y inputs que se realizan.

##### Input Action Window (Como crear Inputs / Modificar controles)



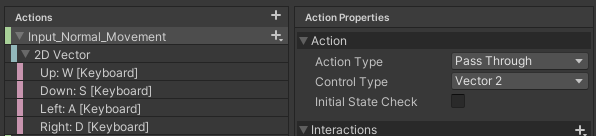
En esta ventana es donde se controlan y asginan los Inputs.

*Action Maps* se refiere al conjunto de accions que tiene un tipo de control, por ejemplo, se puede tener un tipo para las acciones del personaje y otra para cuando se navegan los menus.

*Actions* es cada accion individual que exista, cada una tiene un *Action Type*:

* *Button* - Para entradas ON/OFF (Ej: saltar con un botón).
* *Value* - Para valores continuos (Ej: dirección de un joystick).
* *Pass-Through* - Para flujos de datos sin filtrado.

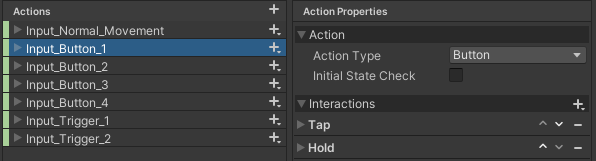
Debajo de las acciones se define el *Binding*, la tecla/boton la cual ejecuta la accion. *Space [Keyboard].*



* *Input\_Normal\_Movement* – Contiene los inputs necesarios para el movimiento.

*Actyon Type* es *Pass Through* esto significa que el valor del input se envía sin ningún tipo de procesamiento adicional. (Sin los eventos que se detectan en Button o Value).

*Control Type* es *Vector 2*, lo que significa que la acción espera dos valores flotantes (X, Y).



* *Input\_Button / Input\_Trigger* – Son los inputs para los botones principales del juego. Son *Action Type Button*, esto significa que en el codigo podemos detectar estados como "Started", "Performed" o "Canceled".

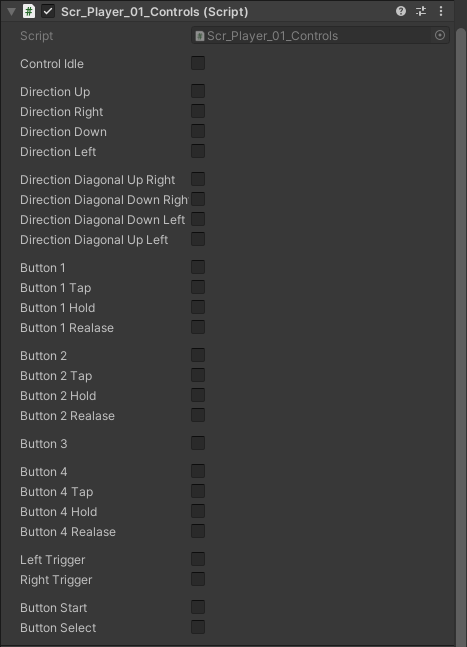
Hay dos tipos:

Button 1, 2 y 4 contienen *Interactions Tap* y *Hold*. Esto permite detectar en estos botones cuando se presiona rapidamente o se deja presionado.

Button 3 y Trigger 1 y 2 por el otro lado no cuenta con estas *Interactions*, lo importante es detectar cuando se usan, por lo que no importa como se presione.

#### 1.1.5 – Scr\_Player\_01\_Controls

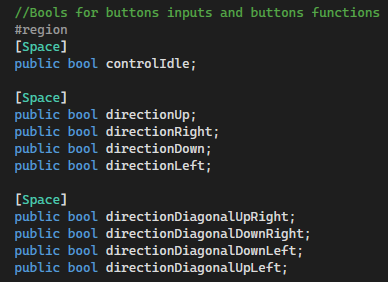
Este script se encarga de manejar el control que usa el jugador y sus inputs.



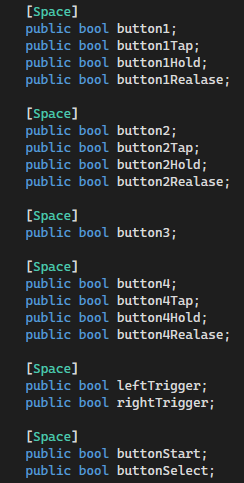
##### Variables



*playerActionControls* es una referencia al Input Asset generado con el nuevo sistema de Inputs. Permite acceder a los mapas de acción de *Player\_Action\_Map* y sus inputs.

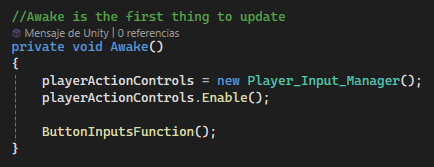


Las direcciones se componen de los cuatro ejes mas cuatro diagonales.



Para los inputs con *Tap y Hold* es necesario detectar *Tap, Hold y Realase.*

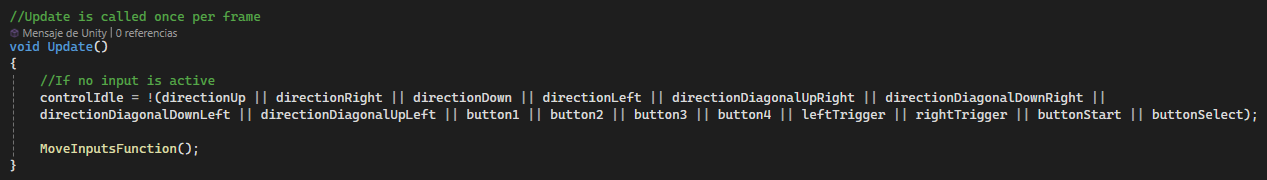
##### Awake()



Se crea una nueva instancia de *Player\_Input\_Manager* y se habilita.

Despues se llama a la funcion *ButtonInputsFunction()*, que asigna las funciones para detectar la entrada de botones.

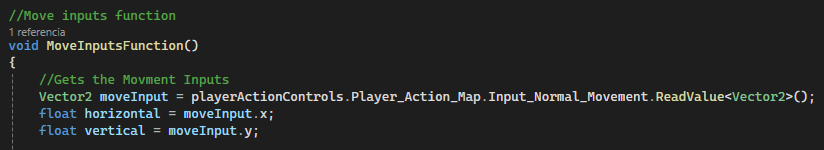
##### Update()



La bool *controlIdle* se activa cuando ningun otro boton esta activado.

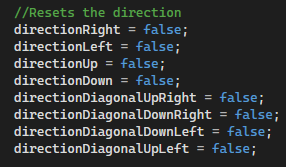
Se llama a la funcion *MoveInputsFunction()*, que actualiza los valores de dirección.

##### Funcion MoveInputsFunction()

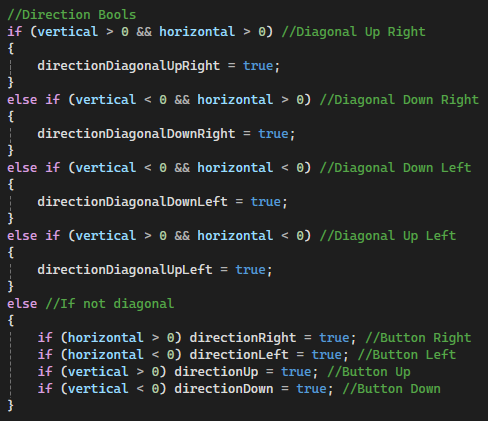


La funcion *MoveInputs()* se ejecuta en *Update()* y controla los Inputs utiliados para el movimiento.

*moveInput* lee el *Vector2* de los input en el *Action Map* (x = izquierda/derecha, y = arriba/abajo).



Se asegura de que las direcciones previas no se queden activas.

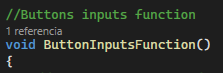


Esta seccion detecta el movimiento.

Si vertical y horizontal son ambos positivos o ambos negativos, significa que el stick se mueve en una dirección diagonal. Se activan las variables correspondientes (*directionDiagonalUpRight*, *directionDiagonalDownRight*, etc.).

Si el movimiento no es diagonal, activa solo una dirección (*directionRight*, *directionLeft*, etc.). Esto evita que se activen múltiples direcciones al mismo tiempo.

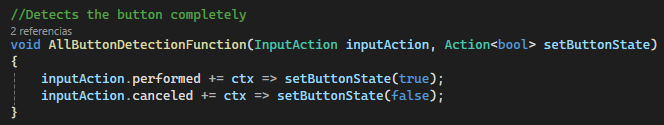
##### Funcion NewInputsFunction()



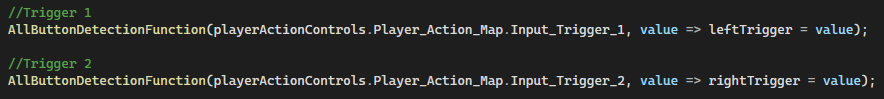
Esta funcion e encarga de asignar las funciones de detección a cada botón. Dentro de esta se llama a diferentes funciones que determinan el tipo de deteccion que tienen los botones.

##### Funcion AllButtonDetectionFunction()

Esta funcion es la deteccion basica del control. Activa una variable cuando se presiona (performed) y desactiva la variable cuando se suelta (canceled).



Se usa en botones simples como leftTrigger y rightTrigger.



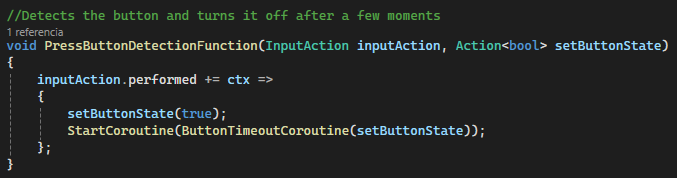
Se pasan los valores *InputAction inputAction* y *Action<bool> setButtonState.*

*inputAction* referencia al Input en el asset.

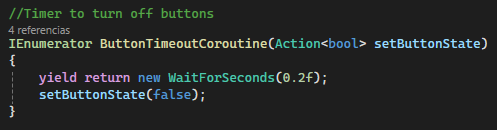
*setButtonState* la bool correspondiente a ese input en el script.

##### Funcion PressButtonDetectionFunction()

Esta funcion es una deteccion basica pero con un timer para apagarse despues de un tiempo.



Activa el botón cuando se presiona. Después de 0.2 segundos lo apaga la corutina ButtonTimeoutCoroutine().



Se usa en button3 (saltar), porque solo necesita activarse por un instante.

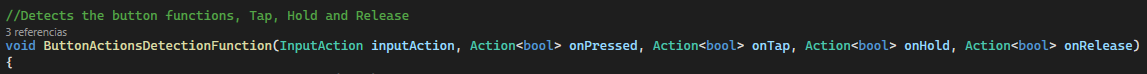


*inputAction* referencia al Input en el asset.

*setButtonState* la bool correspondiente a ese input en el script.

##### Funcion ButtonActionsDetectionFunction()

Esta funcion es para los botones mas complejos a los que se les tiene que detectar *Tap, Hold y Release.*



Parámetros:

*inputAction* referencia al Input en el asset.

*onPressed* se activa cuando el botón se empieza a presionar.

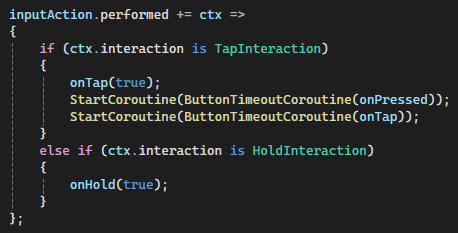
*onTap* se activa si el botón es presionado y soltado rápidamente.

*onHold*: se activa si el botón se mantiene presionado.

*onRelease* se activa cuando se suelta el botón después de haber sido mantenido.



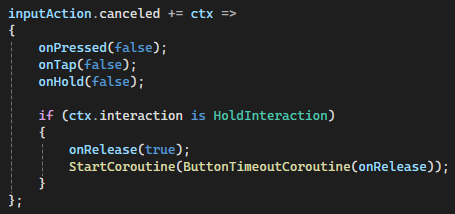
Esto se activa cuando el botón comienza a ser presionado (*inputAction.started*). Marca la variable bool correspondiente como true (ejemplo: button1 = true).



Se activa cuando el boton esta siendo presionado (*inputAction.performed*).

Si la interacción es Tap (presionar y soltar rápido), se activa *onTap(true)* y inicia corrutinas para apagar *onPressed* y *onTap*.

Si la interacción es un Hold (mantener presionado), se activa *onHold(true),* lo que indica que el botón está siendo sostenido.

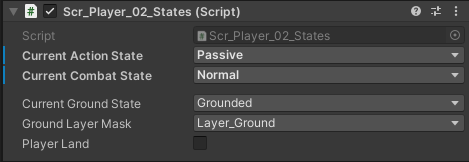


Se activa cuando el boton deja de usarse (*inputAction.canceled*).

Se desactivan todas las variables (*onPressed, onTap, onHold*). Si el botón estaba en modo Hold, al soltarlo se activa *onRelease(true),* la cual se apaga después un tiempol.

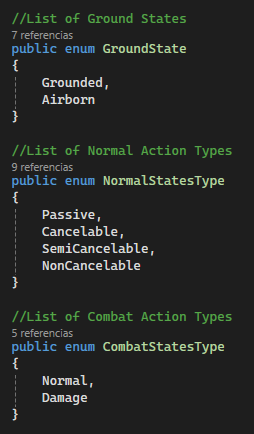
#### 1.1.6 – Scr\_Player\_02\_States

Este script controla los diferentes estados en el que se encuentra el jugador.



##### Enum

El script define tres *enum* para representar distintos estados.

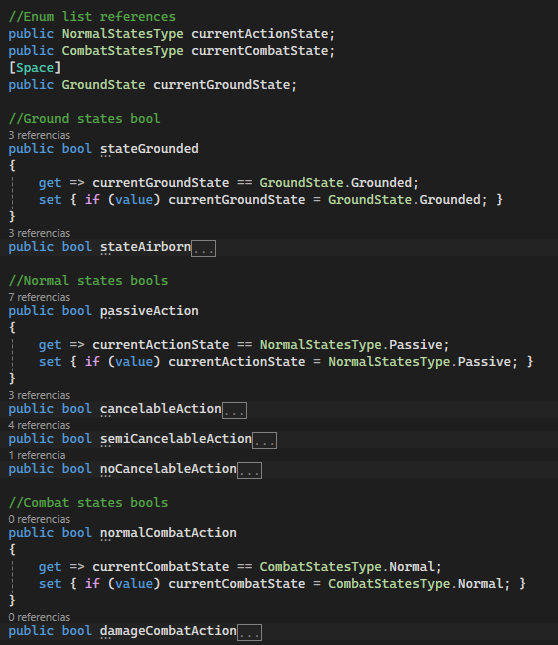


*GroundState* determina si el personaje esta en la superficie o no.

*NormalStatesType* determina el estado del personaje para realizar las acciones.

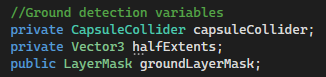
*CombatStatesType* determina el estado al estar en combate.

##### Variables



*Enum list references* son las referenicas en el editor de los enum.

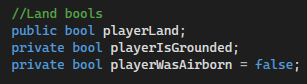
Las bools estan programadas con esta propiedad para setear el valor en true.



*capsuleCollider* referencia al componente que determina el collider del jugador.

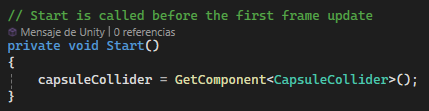
*halfExtents* define la distancia entre el personaje y la superficie para que detecte.

*groundLayerMask* indica en qué Layers de Unity se detectará como superficie.



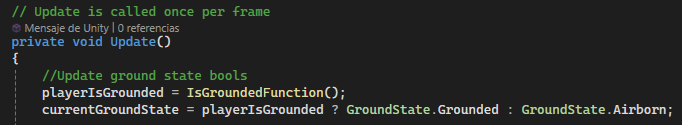
Bools utilizadas para el aterrizaje del jugador.

##### Start()



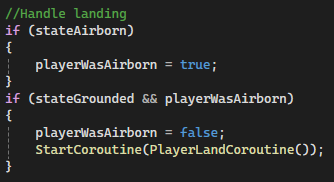
Obtiene el *CapsuleCollider* del personaje para usarlo en la detección de suelo.

##### Update()



*playerIsGrounded* se actualiza llamando a *IsGroundedFunction(),* que verifica si el personaje está tocando el suelo.

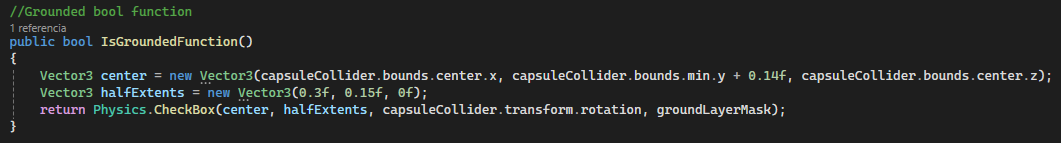
Luego, *currentGroundState* se actualiza con el estado correcto.



Si el personaje esta en el aire (*stateAirborn*), *playerWasAirborn* se pone en true.

Si luego aterriza (*stateGrounded*) y *playerWasAirborn* es true, se inicia la corutina *PlayerLandCouroutine()* para manejar el aterrizaje.

##### IsGroundedFunction()



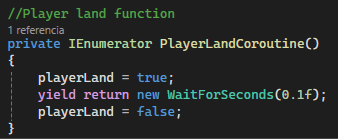
Esta funcion se encarga de detectar si el personaje esta sobre una superficie o no.

Usa *Physics.CheckBox()* para detectar si el personaje está tocando el suelo.

*Vector3 center* define la posición del centro de la caja de detección, ubicada un poco por encima de la base del *CapsuleCollider*. *halfExtents* define el tamaño del área de detección.

Si la caja detecta colisión con un objeto con *groundLayerMask* (Ground Layer), retorna true (el personaje está en el suelo).

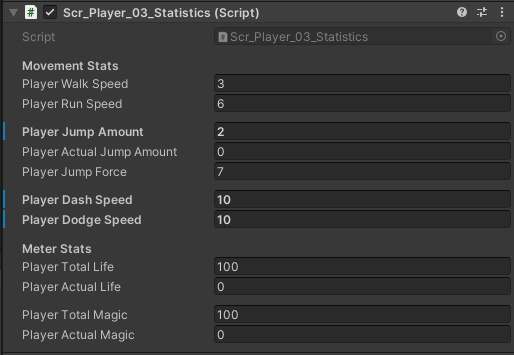
##### PlayerLandCouroutine()



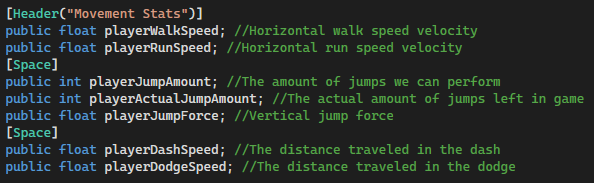
Cuando el jugador aterriza, *playerLand* se pone en true por 0.1 segundos y luego en false.

#### 1.1.7 – Scr\_Player\_03\_Statatics

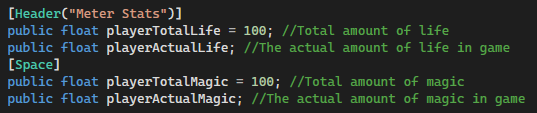
Este script contiene en un mismo lugar el valor numerico de las estadisticas del personaje.



##### Variables

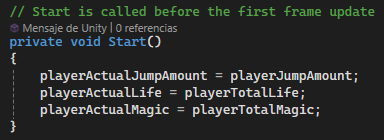


* **playerWalkSpeed** – La velocidad al caminar.
* **playerRunSpeed** – La velocidad al correr.
* **playerJumpAmount** – La cantidad total de saltos.
* **playerActualJumpAmoun** – La cantidad de saltos restantes in game.
* **playerJumpForce** – La fuerza de salto.
* **playerDashSpeed** – La distancia que reocrre al ejecutar un dash.
* **playerDodgeSpeed** – La distancia que recorre al esquivar.



* **playerTotalLife** – El total maximo de la vida del jugador.
* **playerActualLife** – La vida actual del jugador in game.
* **playerTotalMagic** – El total maximo de la magia del jugador.
* **playerActualMAgic** – La magia actual del jugador in game.

##### Start()

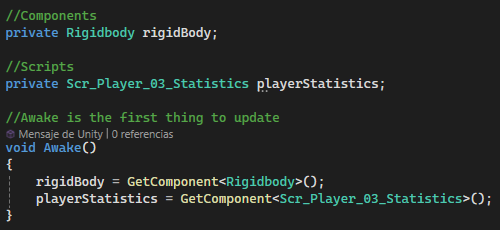


Se asigna la cantidad maxima a la cantidad actual.

#### 1.1.8 – Scr\_Player\_04\_Physics

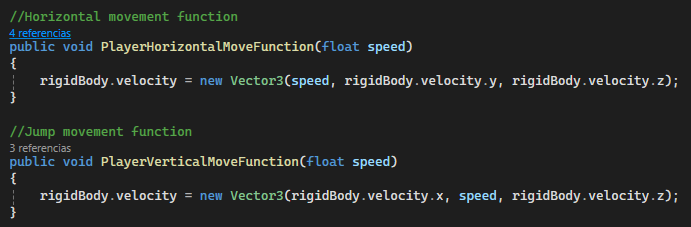
Este script contiene las funciones para el movimiento del objeto del jugador.

##### Variables



Para mover al jugador usamos el componente *rigidBody* combinado con las estadisticas que tenemos almacenadas en *playerStatistics*.

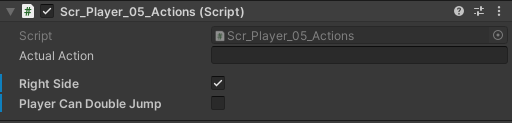
##### PlayerMovesFunctions



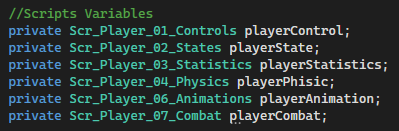
El objeto se mueve en referencia a estas funciones, las cuales pasan un valor y lo pasan al X o Y de un *Vector3*.

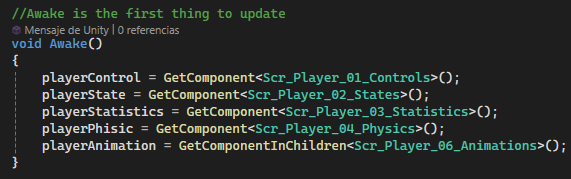
#### 1.1.9 – Scr\_Player\_05\_Actions

Este maneja el movimiento del personaje,utilizando funciones de los scripts principales, conformando asi las acciones del jugador.



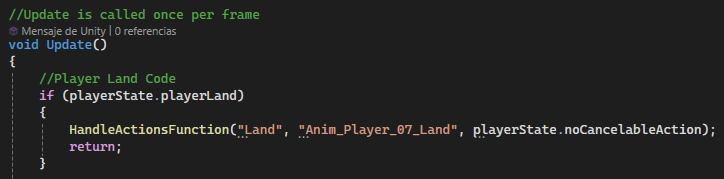
##### Variables



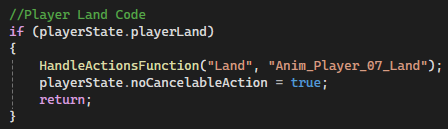


Se crean referencias a cada script, dependiendo de si estan en el mismo objeto (*GetComponent*) o en un objeto hijo (*GetComponentInChildren*) como se tiene que referenciar.

##### Update()



Las acciones tienen cierta estructura para su mejor funcionamiento. Al final de cada pedazo de codigo de accion esta *return;* esto para que el codigo no pueda evaluar otra accion en el mismo frame. Esto tambien causa que el orden en el que esten importa, mientras mas arriba, ma prioridad tiene la accion. Si la accion no tiene *return;* es porque tiene muy poca preoridad ej *Idle* o *Fall*.

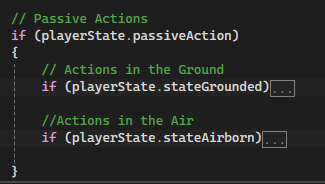


Aterrizar solo se ejecuta si *playerState.playerLand*, esto es porque esta accion se tiene que ejecutar siempre que el jugador aterrize, sin importar si esta haciendo otra accion.

El playerState se define en esta parte porque dentro de la funcion se activa por un segundo.

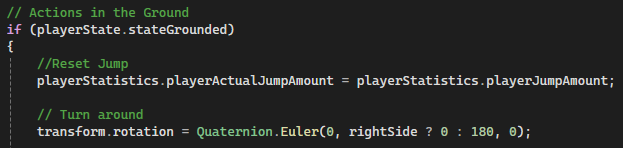


Esta funcion contiene una funcion de reseteo de estado para cada accion que la necesite.

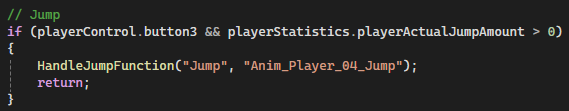


Todas las acciones de este codigo se ejecutan solo si el personaje esta en *passiveAction*, la cual tambien esta separada en dos dependiendo de en donde se ejecute, en tierra (*stateGrounded*) o aire (*stateAirborn*).

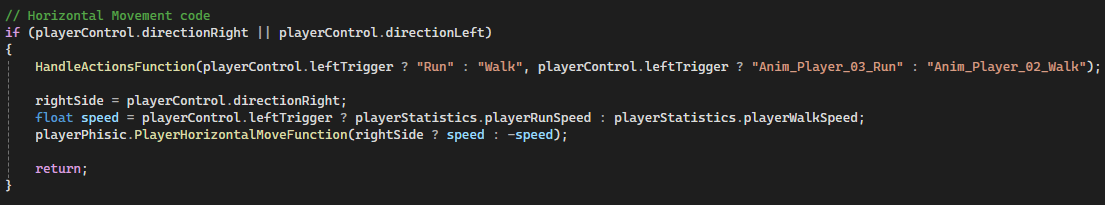
##### playerState.stateGrounded



Si el jugador esta en tierra se ejecutan estas dos lineas, uno reseta la cuenta de salto para poder volverlo a hacer y la otra es la que se encarga de girar al personaje dependiendo *rightSide*.



Para saltar en tierra hay que presionar el boton y tener que la cantidad de saltos actual (*playerActualJumpAmount*) sea mayor que 0.



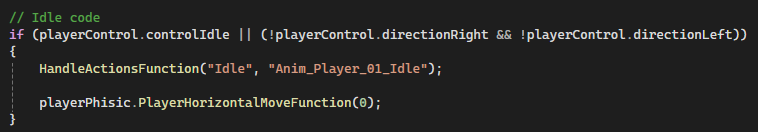
El movimiento horizontal se ejecuta utilizando uno de dos imputs (*directionRight y directionLeft*).

Esta parte del codigo utiliza varias veces la estructrua *playerControl.leftTrigger ? value1 : value2* , esto signfica que dependiendo si *leftTrigger* es true, que valor se usa. Si es true se usa *vaule1* y si es false *vaule2*.

Por ejemplo, *float speed* es *playerRunSpeed* si *leftTrigger* es true y *playerWalkSpeed* si es false.

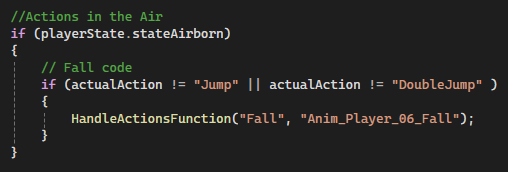
*rightSide* es true si se detecta *directionRight*, asi es como controlamos a que lado esta mirando.

Para el movimiento mismo se usa la funcion *PlayerHorizontalMoveFunction* de fisicas (*Scr\_Player\_04\_Physics*). Se le pasa *speed* que dependiendo de si *rightSide* es true es positivo o negativo.



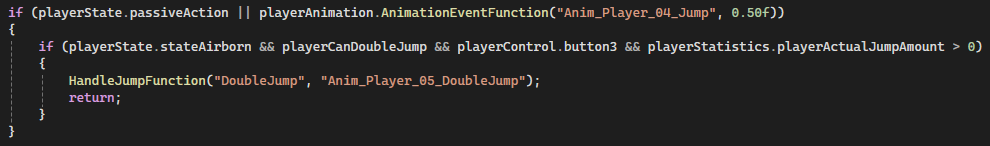
Idle se activa en dos casos, si el *controlIdle* es true, el cual lo es si no se presiona ningun boton o si ninguna de las direcciones se esta presionando.

##### playerState.stateAirborn

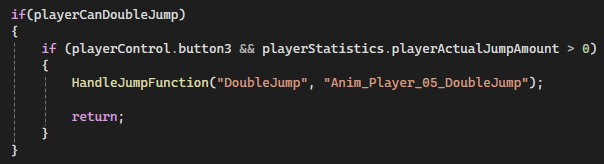


Caer se ejecuta si la accion actual no es *Jump* o *DoubleJump* y si se termina esas respectivas animaciones.



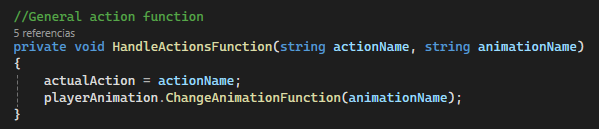


Esta funcion esta por fuera de *passiveAction*, pero es por tener el codigo mas limpio. El doble salto se puede ejecutar en *passiveAction* y en otro estado muys especifico, despues de cierto tiempo de saltar *AnimationFinished("Anim\_Player\_04\_Jump", 0.50f).*



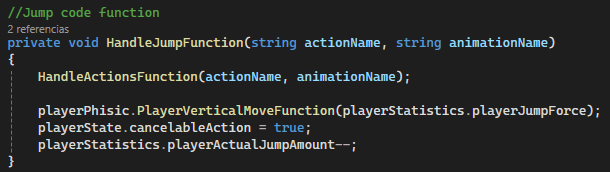
El codigo para saltar en el aire es el mismo que en el de la superficie, con la diferencia que para poder ejecutarse *playerCanDoubleJump* tiene que ser true, esta bool existe para poder darle esta habilidad al personaje en cierto punto.

##### HandleActionsFunction()



Esta funcion es la que usan la mayoria de acciones, dependiendo de los valores que se pase asigna la accion (*actionName*) y ejecuta la animacion (*animationName*).

##### HandleJumpFunction()

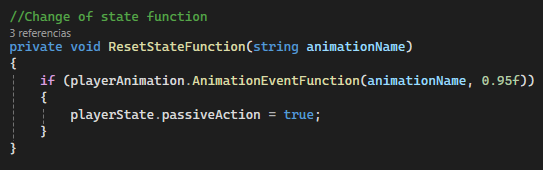


Esta funcion se usa para ejecutar los salto. Es una accion mas asi que usa la funcion *HandleActionsFunction* excepto que no pasa la bool porque la activamos dentro de esta funcion.

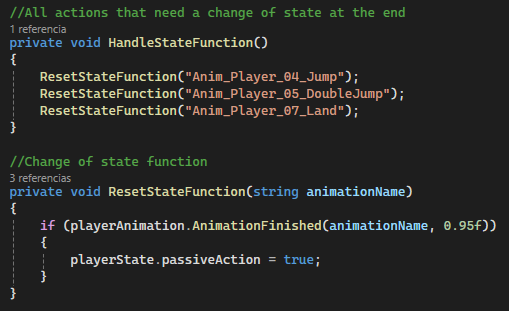
Para saltar se usa la funcion *PlayerVerticalMoveFunction* del script de fisicas (*Scr\_Player\_04\_Physics*).

A lo ultimo le restamos a *playerActualJumpAmount*, esto es para controlar los saltos que ejecutamos.

##### ResetStateFunction()



Algunas acciones requieren que se indique cuando terminan, prendiendo la bool de passiveAction. Se usa la funcion bool *AnimationEventFunction*, le pasamos *animationName* el nombre exacto de la animacion junto al float *0.95f* que indica la altura de la animacion a la que se debe prender.

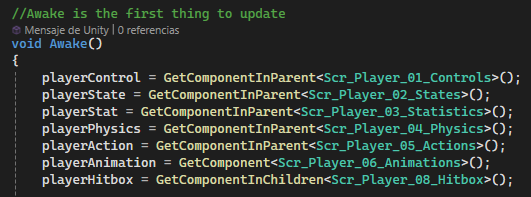
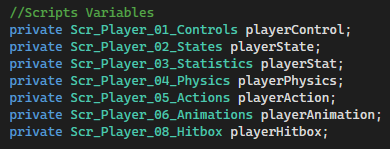


Esto se actualiza en update, para mejorar el manejo de todas las acciones que requieren de esta funcion, las juntamos en otra, *HandleStateFunction()*.

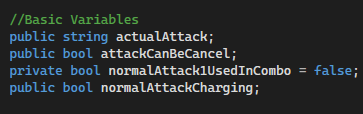
#### 1.1.10 – Scr\_Player\_07\_Combat

Este script es como el 05\_Actions pero solo se encarga de las acciones de combate.

##### Variables

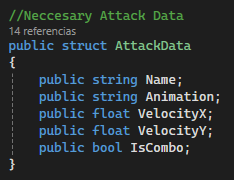


Se crean referencias a cada script, dependiendo de si estan en el mismo objeto (*GetComponent*) o en un objeto hijo (*GetComponentInChildren*) como se tiene que referenciar.



El *string actualAttack* nos indica en el inspector cual fue el ultimo ataque realizado, tambien se actualiza en el script *Scr\_Player\_05\_Actions*, pero en este solo se muestran los ataques.

Las bools indican distintos atributos de los ataques. *attackCanBeCancel* indica cuando se puede cancelar un ataque, *normalAttack1UsedInCombo* se usa para una cuestion de coordinacion entre el ataque normal 1 y 2, y *normalAttackCharging* indica cuando un ataque se esta cargando.



Esta estructura es usada para almacenar los datos del ataque:

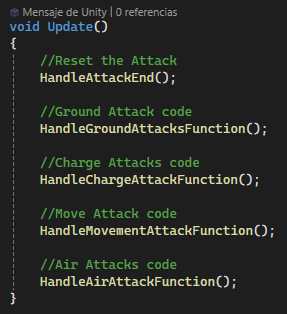
*string Name* - El nombre del ataque.

*string Animation* - El nombre de la animación asociada con el ataque.

*float VelocityX y float VelocityY* - La velocidad de movimiento del ataque en los ejes X y Y.

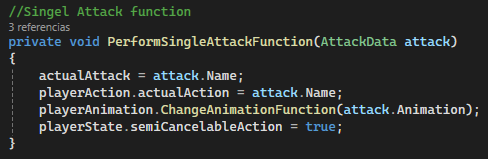
*bool IsCombo* - Un booleano que indica si el ataque forma parte de una secuencia de combos.

##### Update()



El metodo Update esta conformado por funciones para simplificar la legibilidad del codigo.

##### PerformSingleAttackFunction()

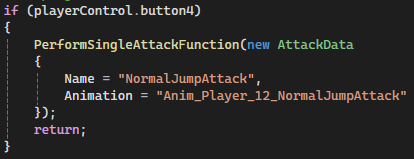


Esta función se encarga de ejecutar un solo ataque básico. Le pasamos la *AttackData* como parámetro, que contiene información sobre el el ataque.

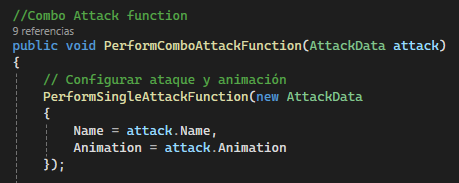
*actualAttack* y *actualAction* se configuran con el nombre del ataque.

Se cambia la animación del jugador en *ChangeAnimationFunction*.

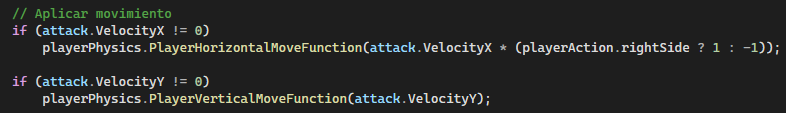
playerState.semiCancelableAction = true permite que el ataque sea cancelable después de un cierto tiempo.

Ej:  


##### PerformComboAttackFunction()

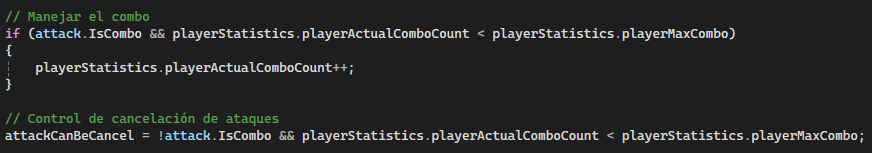


Esta función maneja los ataques en un combo. Usa la funcion *PerformSingleAttackFunction* para asignar el *Name* y *Animation* y se le aplica ciertos movimientos y controles adicionales.



Si hay un valor para VelocityX o VelocityY, se aplican los movimientos horizontales y verticales del jugador a través de *PlayerHorizontalMoveFunction* y *PlayerVerticalMoveFunction*.

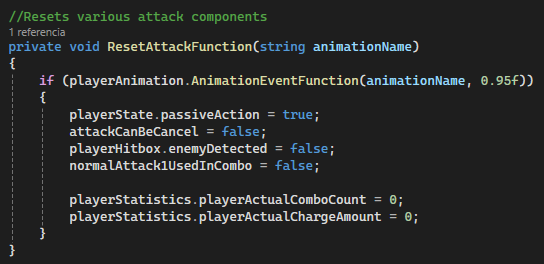
Dejar estos valores en 0 en vez de no ponerlos, ayuda a detener el movimiento si se ejecuta la accion.



Si el ataque es parte de un combo (*attack.IsCombo*) y el contador de combos no ha alcanzado el límite máximo, el contador de combos se incrementa.

La ultima linea de alguna manera hace que el contador de combos funcione correctamente.

##### ResetAttackFunction()



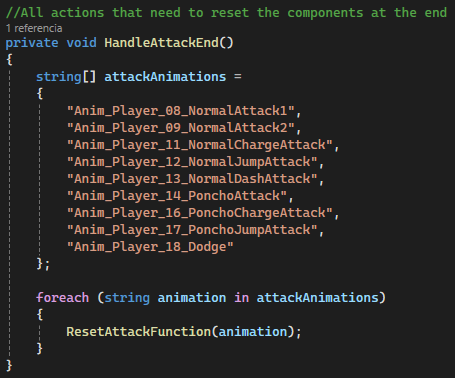
Esta función se encarga de restablecer varios componentes de ataque después de que se complete su animacion.

Usa *playerAnimation*.*AnimationEventFunction* para verificar si la animación ha llegado a un 95% de su finalización. Cuando se alcanza este umbral, se realiza el reset.

Resetea varios estados:

* Se devuelve al jugador a playerState.passiveAction.
* Se desactiva la posibilidad de cancelar el ataque.
* Se desactiva la detección de enemigos,.
* Se reinicia el contador de combos.
* Se reinicia la carga de los ataques.

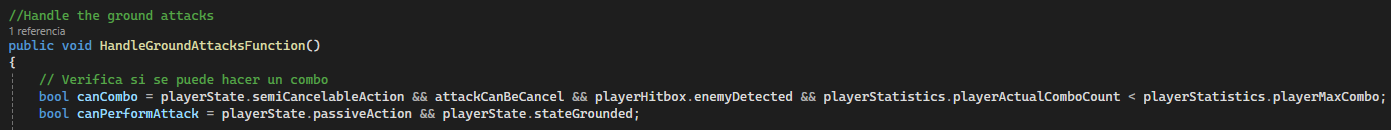
##### HandleAttackEnd()



Esta función llama a *ResetAttackFunction* para varias animaciones de ataque, asegurándose de que el estado de combate se resetea al final de cada animación.

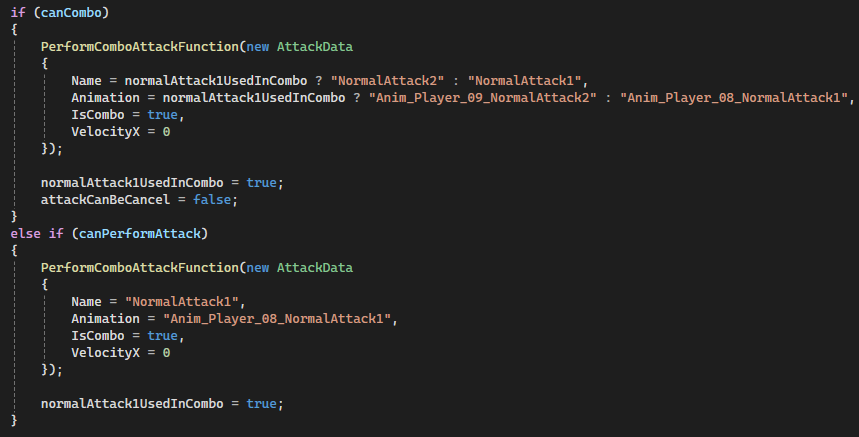
##### HandleGroundAttacksFunction()

Esta función maneja los ataques realizados cuando el jugador está en el suelo.

Primero, verifica si el jugador puede realizar un ataque basado en dos factores, que tambien diferencian si es el primer o segundo ataque.

La *bool canPerformAttack* se prende si el jugador con tiene la capacidad de moverse (*passiveAction*) y esta en tierra (*stateGround*). Si el ataque se ejecuta gracias a esta bool es porque es el primer ataque.

La bool canCombo se prende si el jugador esta en un accion que se puede cancelar en momentos concretos (*semiCancelableAction*), si el ataque puede ser cancelado (*attackCanBeCancel*), si la hitbox del ataque anterior colisiono con un enemigo (*playerHitbox.enemyDetected*) y si el numero actual de combo es menor al numero maximo (*playerActualComboCount < playerMaxCombo*). Si el ataque se ejecuta gracias a esta bool es porque es el segundo ataque.

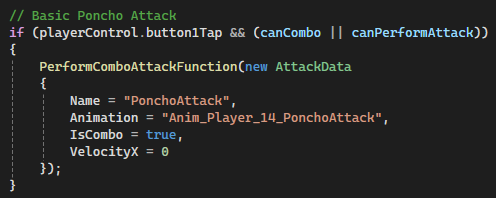


Si se presiona button4Tap, se ejecuta un combo o un ataque simple dependiendo de si el jugador ya ha usado un ataque en el combo.

El código comprueba primero si este es el primer ataque, lo hace con la condición *canCombo*, que solo es true si se paso del primer ataque.

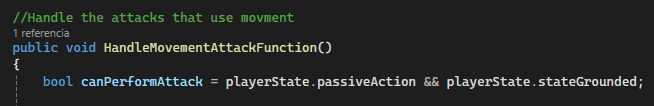
Si es true, se realiza una llamada a *PerformComboAttackFunction* y se elige entre "*NormalAttack1*" y "*NormalAttack2*" dependiendo de si el primer ataque del combo fue usado (*normalAttack1UsedInCombo*).

Si es false y canPerformAttack es true, lo que significa que es el primer ataque, entonces se realiza una llamada a *PerformComboAttackFunction* y se pasa solo la informacion de "*NormalAttack1*".

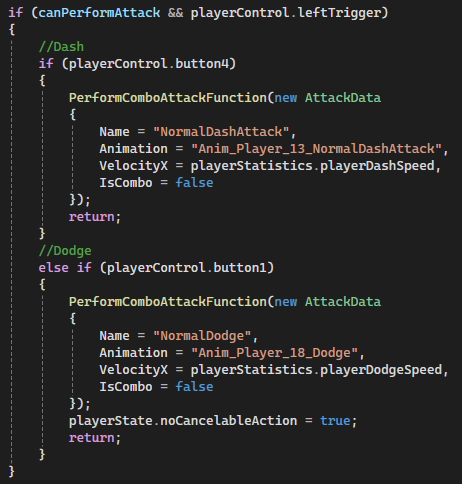


Si se presiona *button1Tap* y cualquiera de las bool de condiciones se cumplen, se realiza un ataque con el poncho.

##### HandleMovementAttackFunction()



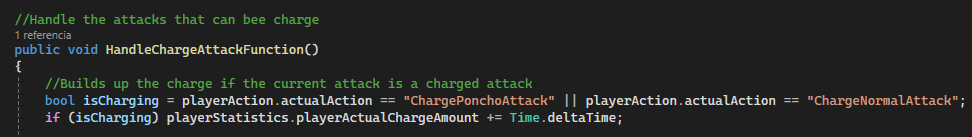
Esta función maneja los ataques que involucran movimiento, como el Dash y el Dodge. Tambien usa *bool canPerformAttack* que se prende si el jugador con tiene la capacidad de moverse (*passiveAction*) y esta en tierra (*stateGround*).



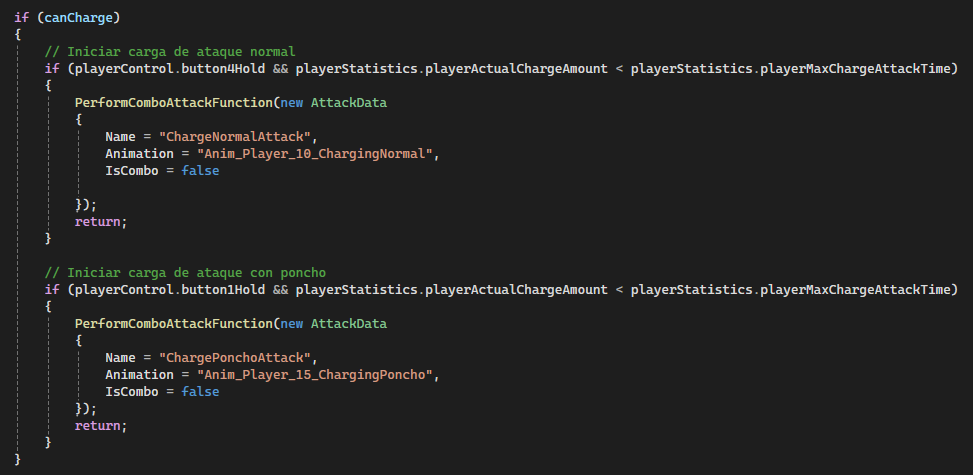
Si el jugador está en el suelo (*passiveAction* y *stateGround*) y presiona los botones correspondientes (*leftTrigger* + *button 1 / 4*), se llama a *PerformComboAttackFunction* y le pasamos la estadistica correspondiente en *VelocityX*.

El Dodge tiene *playerState*.*noCancelableAction* porque no se deberia ver interrumpida por otra accion.

##### HandleChargeAttackFunction()



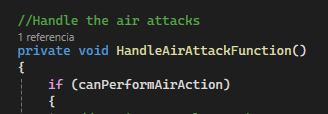
Esta funcion maneja los ataques cargados. La función acumula la carga (*playerActualChargeAmount += Time.deltaTime*) mientras el jugador mantiene presionado el botón correspondiente (*playerAcvtion.actualAction ==*)



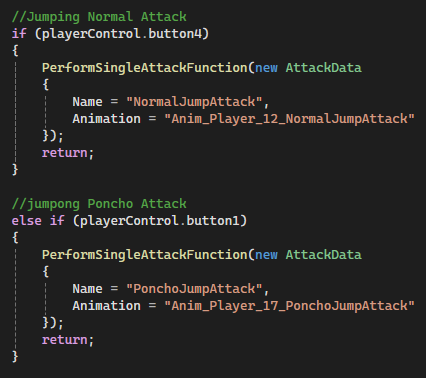
Si el jugador mantiene button4Hold o button1Hold, comienza la carga del ataque.

Si el jugador suelta el botón o la carga alcanza el máximo, ejecuta el ataque cargado.

##### HandleAirAttackFunction()



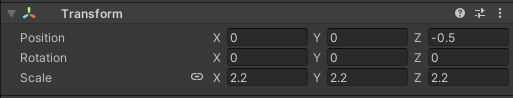
Maneja los ataques del jugador cuando está en el aire. Similar a los ataques en el suelo, esta función verifica si se puede ejecutar el ataque a traves de la bool *canPerformAirAction*.



Si el jugador presiona el botón *button4* o *button1* mientras está en el aire, ejecuta un ataque. *PerformSingleAttackFunction* no tiene *VelX* o *VelY* porque si se ejecuta el ataque mientras saltamos, queremos que se siga el mocimiento de salto.

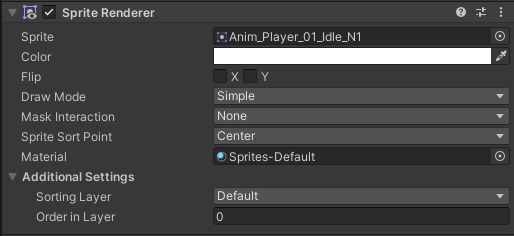
### 1.2 Player\_Sprites (Hijo)

#### 1.2.1 - Transform



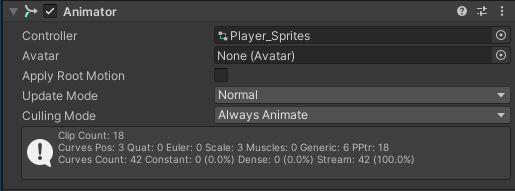
Determina la posición, rotación y escala del objeto.

#### 1.2.2 – Sprite Renderer



Este componente se usa para renderizar las imágenes en 2D.

#### 1.2.3 - Animator



Este compoente se encarga de las animaciones del jugador.

El tipo de Controller define el flujo de las animaciones y sus transiciones.

##### Animation Window (Como crear animaciones)

Las animaciones se hacen en la ventana Animation.



Para crear una nueva animacion tenemos que ir a la caja debajo de *Preview* y despues *Create New Clip.*



Cada animación es un archivo anim. Necesitamos crear uno con la estructura de nombre correcta:

*Anim\_[Number]\_[Character]\_[Animation Name]*

Ej: *Anim\_01\_Player\_Idle*



Luego, necesitamos los sprites que componen cada fotograma de animación, representados por rombos en la línea de tiempo.

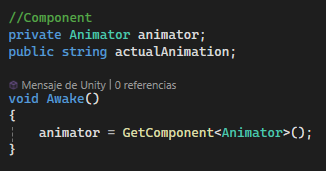
Los *Samples* representa la velocidad a la que se reproduce cada fotograma. La base es 60, pero la velocidad adecuada es 15.



#### 1.2.4 – Scr\_Player\_06\_Animations

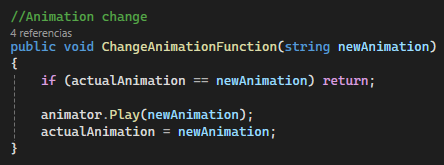
Este script contiene las funciones para la reproduccion de animaciones y los eventos que se pueden ejecutar.

##### Variables



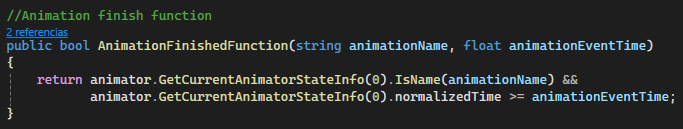
Se hace referencia al componente *Animator* de Unity y se hace un *string* que determina cual es la animacion actual que se esta ejecutando.

##### ChangeAnimationFunction()



Esta funcion es la que se usa para cambiar de animacion, le pasamos *newAnimation*, el cual es comparado a *actualAnimation* para evitar que se reinicie la animacion, y luego se ejecuta en el *animator* y cambia *actualAction*.

##### AnimationFinishedFunction()



A esta funcion es una bool que para ser true se le pasa dos valores, *animationName*, el nombre de la animacion, y *animationEventTime*, el tiempo dentro de esa animacion en el que pasa a true.