**Documentos de Programación del Jugador**

**Tools**

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

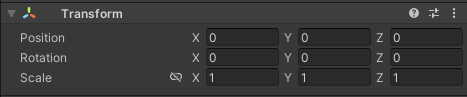
# **Objetos**

## 1. Player

### 1.1 Player\_GameObject (Principal)

#### 1.1.1 - Transform

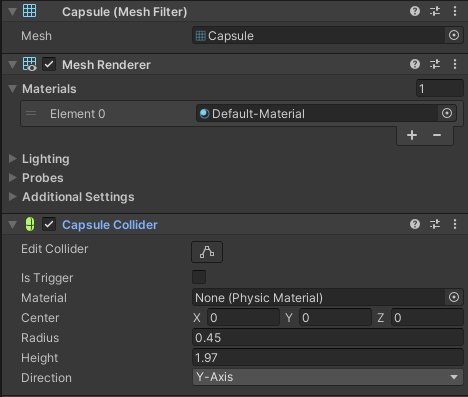
Determina la posición, rotación y escala del objeto.



#### 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.

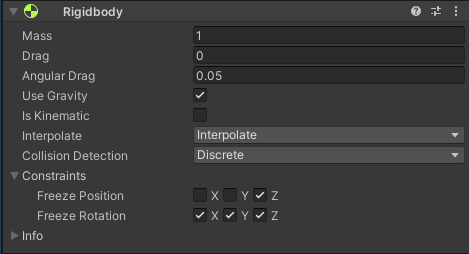
Usamos *Radius* y *Height* para mover los límites.





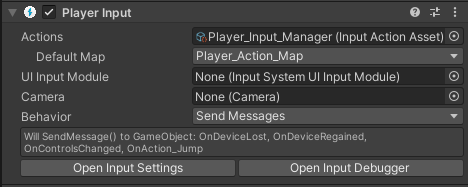
#### 1.1.3 - RigidBody

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)



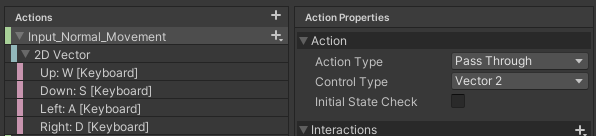
En esta ventana es donde se controlan y asginan los Inputs.

*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.

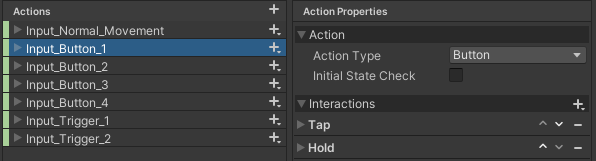
Debajo de las acciones se define el *Binding*, la tecla/boton la cual ejecuta la accion. *Roght Button [Mouse]. Button 4 [Controller].*



* *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".

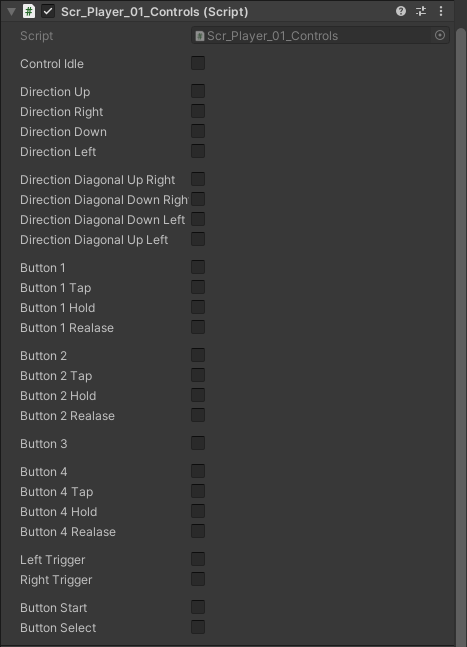
Hay dos tipos:

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

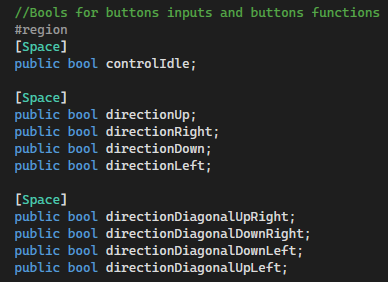
Este script se encarga de manejar el control que usa el jugador y sus inputs.



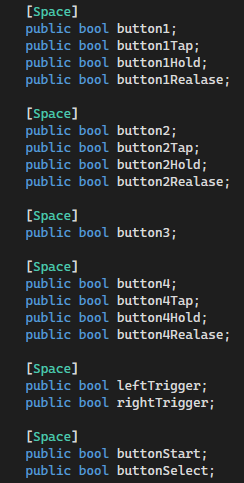
##### Variables



*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.

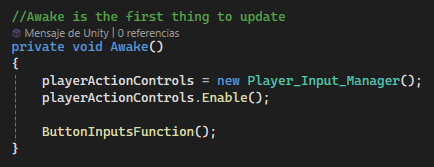


Las direcciones se componen de los cuatro ejes mas cuatro diagonales.



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

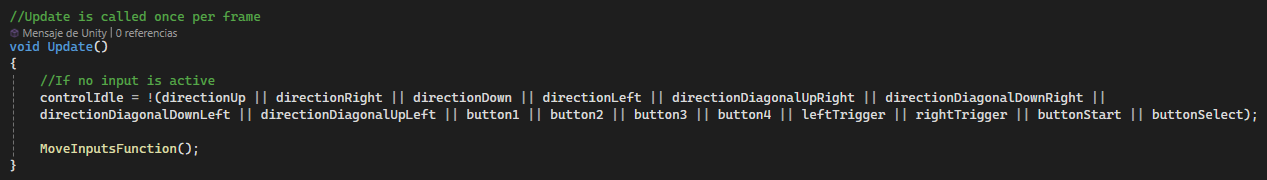
##### Awake()



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.

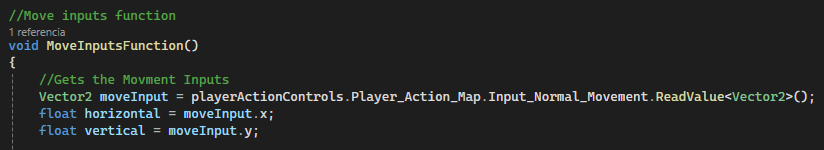
##### Update()



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

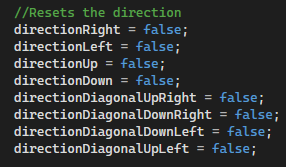
Se llama a la funcion *MoveInputsFunction()*, que actualiza los valores de dirección.

##### Funcion MoveInputsFunction()

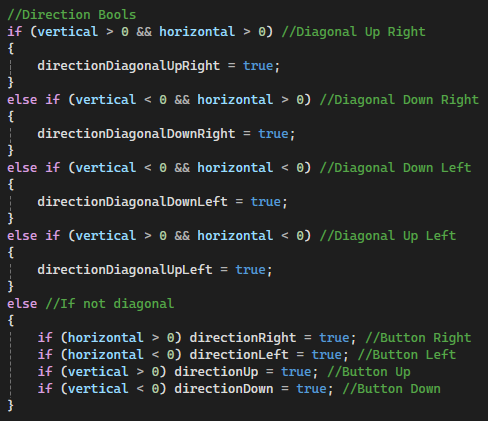


La funcion *MoveInputs()* se ejecuta en *Update()* y controla los Inputs utiliados para el movimiento.

*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.

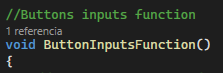


Esta seccion detecta el movimiento.

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.

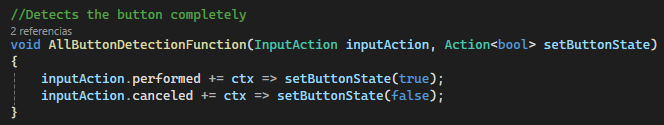
##### Funcion NewInputsFunction()



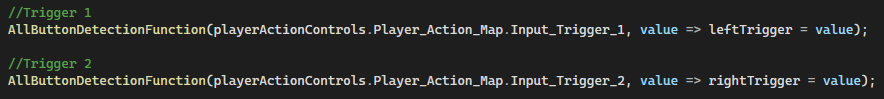
Esta funcion e encarga de asignar las funciones de detección a cada botón. Dentro de esta se llama a diferentes funciones que determinan el tipo de deteccion que tienen los botones.

##### 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.



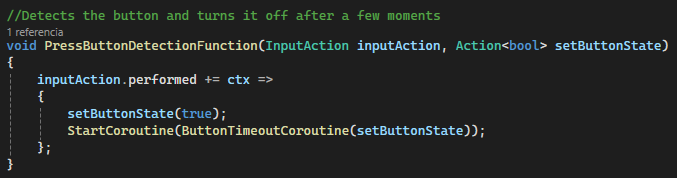
Se pasan los valores *InputAction inputAction* y *Action<bool> setButtonState.*

*inputAction* referencia al Input en el asset.

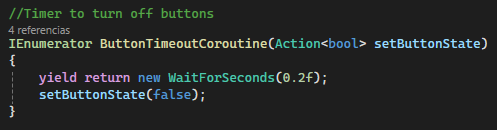
*setButtonState* la bool correspondiente a ese input en el script.

##### 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.

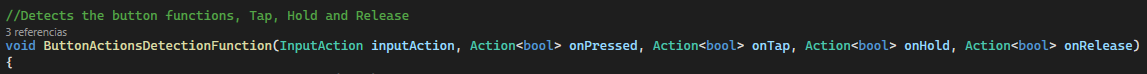


*inputAction* referencia al Input en el asset.

*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.

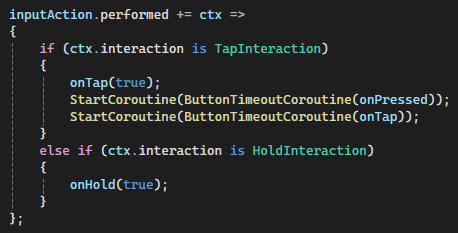
*onTap* se activa si el botón es presionado y soltado rápidamente.

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

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



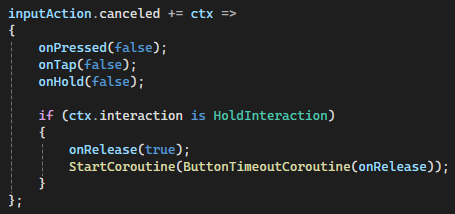
Esto se activa cuando el botón comienza a ser presionado (*inputAction.started*). Marca la variable bool correspondiente como true (ejemplo: button1 = true).



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.

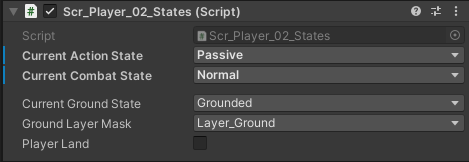


Se activa cuando el boton deja de usarse (*inputAction.canceled*).

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.

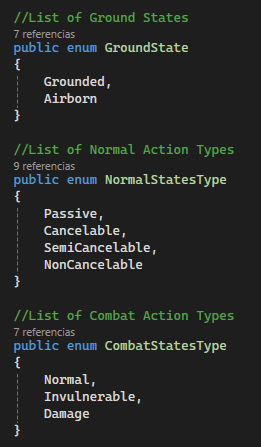
#### 1.1.6 – Scr\_Player\_02\_States

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



##### Enum

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

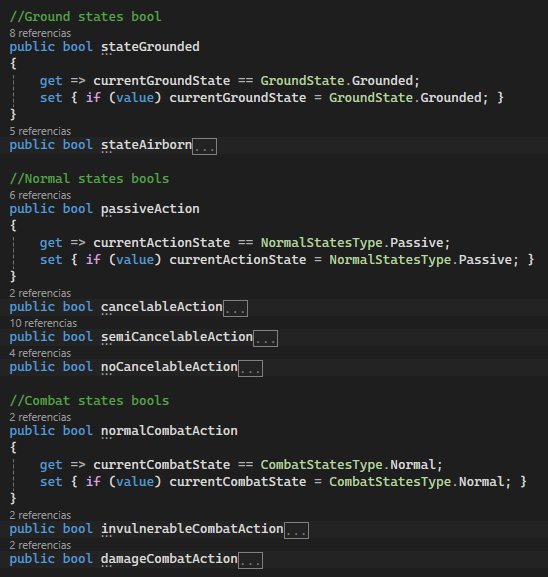


*GroundState* determina si el personaje esta en la superficie o no.

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

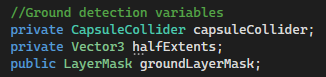
*CombatStatesType* determina el estado al estar en combate.

##### Variables



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

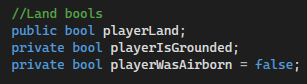
Las bools estan programadas con esta propiedad para setear el valor en true.



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

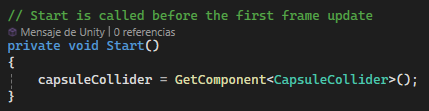
*halfExtents* define la distancia entre el personaje y la superficie para que detecte.

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



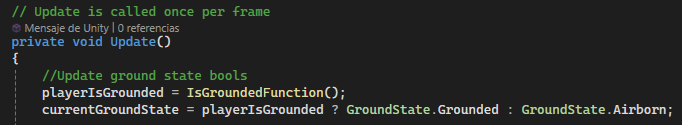
Bools utilizadas para el aterrizaje del jugador.

##### Start()



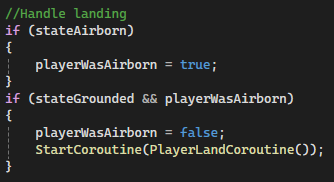
Obtiene el *CapsuleCollider* del personaje para usarlo en la detección de suelo.

##### Update()



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

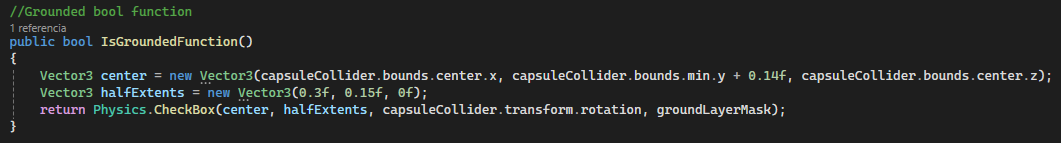
Luego, *currentGroundState* se actualiza con el estado correcto.



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()



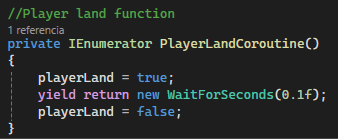
Esta funcion se encarga de detectar si el personaje esta sobre una superficie o no.

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).

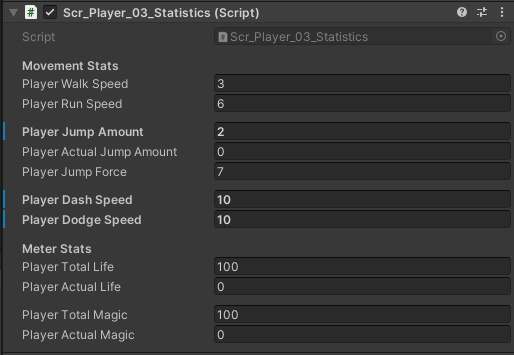
##### PlayerLandCouroutine()



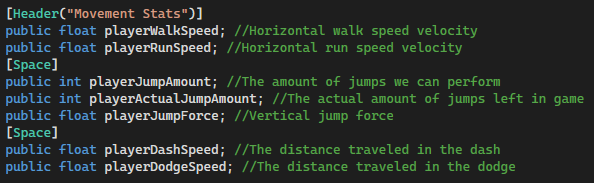
Cuando el jugador aterriza, *playerLand* se pone en true por 0.1 segundos y luego en false.

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

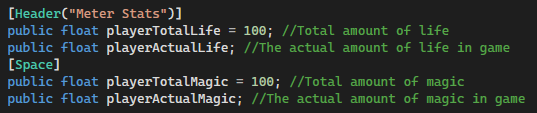
Este script contiene en un mismo lugar el valor numerico de las estadisticas del personaje.



##### Variables

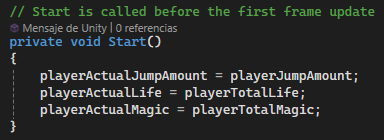


* **playerWalkSpeed** – La velocidad al caminar.
* **playerRunSpeed** – La velocidad al correr.
* **playerJumpAmount** – La cantidad total de saltos.
* **playerActualJumpAmoun** – La cantidad de saltos restantes in game.
* **playerJumpForce** – La fuerza de salto.
* **playerDashSpeed** – La distancia 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.

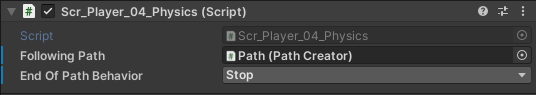
##### Start()



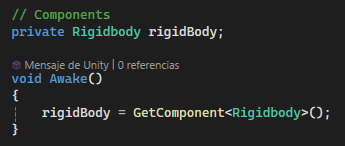
Se asigna la cantidad maxima a la cantidad actual.

#### 1.1.8 – Scr\_Player\_04\_Physics

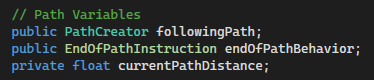
Este script contiene las funciones para el movimiento del objeto del jugador.



##### Variables



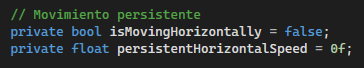
Para mover al jugador usamos el componente *rigidBody* combinado con las estadisticas que tenemos almacenadas en *playerStatistics*.



*pathCreator* referencia al objeto que contiene el camino que debe seguir el jugador.

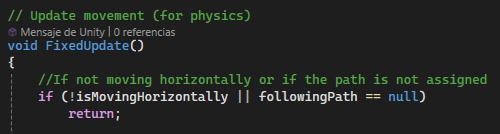
*endOfPathBehavior* indica el comportamiento del jugador al llegar al final del camino.

*currentPathDistance* guarda la distancia actual recorrida a lo largo del path.

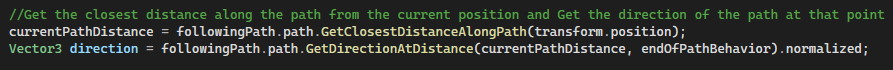


Controla si el jugador debe moverse horizontalmente y a qué velocidad.

##### FixedUpdate

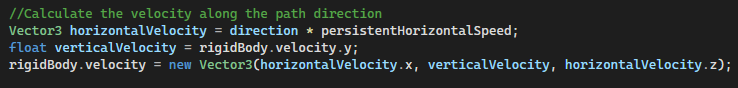


Si el objeto no debe moverse (*isMovingHorizontally* es false) o no se ha asignado un PathCreator (followingPath == null), entonces no hace nada y sale del método.



*GetClosestDistanceAlongPath* calcula cuánto ha avanzado el objeto en el path, según su posición actual.

*Vector3 direction* usa *GetDirectionAtDistance* que obtiene la dirección del camino en ese punto. Esto indca hacia dónde debe moverse el objeto.



*Vector3 horizontalVelocity* calcula la velocidad horizontal a lo largo de la dirección del path, usando la velocidad guardada (*persistentHorizontalSpeed*). *verticalVelocity* guarda la velocidad vertical actual.

*rigidBody.velocity* combina la velocidad horizontal con la vertical y se la aplica al Rigidbody.

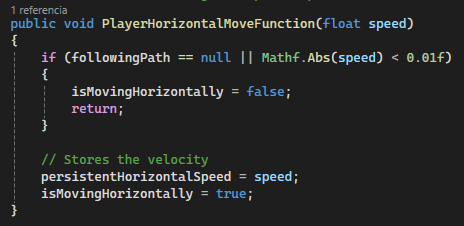


A veces, la física puede desalinear al objeto del camino, este código corrige la posición para que el objeto quede exactamente centrado sobre el path.

Con *GetPointAtDistance* se calcula la posición exacta en el path para la distancia actual. Se reemplaza la posición X y Z del objeto por las del path, pero se conserva la altura (Y), para no interferir con la física vertical.

##### PlayerVerticalMoveFunction

##### PlayerHorizontalMoveFunction



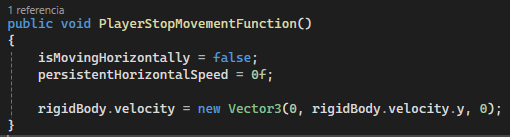
El objeto se mueve en horizontal referenciando esta funcion.

Si no hay un path asignado o si la velocidad es muy cercana a cero, se considera que el jugador no debe moverse. Si se cumple cualquiera se detiene el movimiento y sale del método sin continuar.

Sino guarda la velocidad proporcionada (*speed*) como un la velocidad (*persistentHorizontalSpeed*) para usarla luego en FixedUpdate().

Tambien activa el movimiento horizontal con isMovingHorizontally = true, esto le indica al sistema que, en el próximo FixedUpdate(), el jugador sí debe moverse.

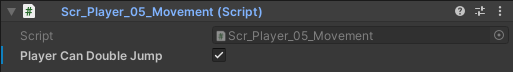
##### PlayeStopMovementFunction



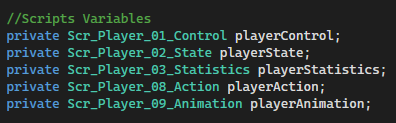
Reinicia *isMovingHorizontally* y *persistentHorizontalSpeed* mientras detiene el movimiento actual del personaje en X y Z.

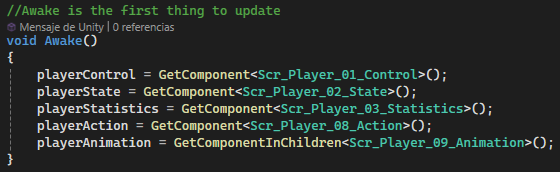
#### 1.1.9 – Scr\_Player\_05\_Movement

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



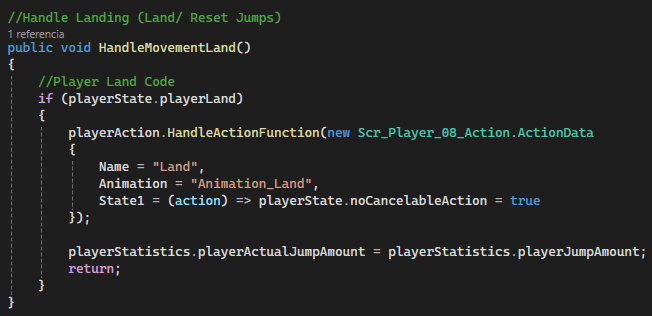
##### Variables





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.

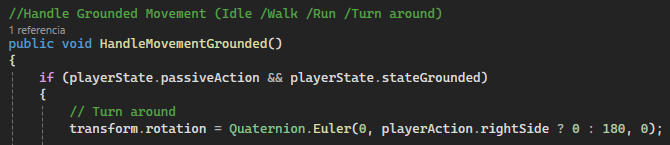
##### HandleMovementLand()



Esta funcion maneja el aterrizaje del personaje. Si el jugador esta aterrizando (*playerLand*) enotnces se ejecuta la funcion *HandleActionFunction* con la informacion de la accion.

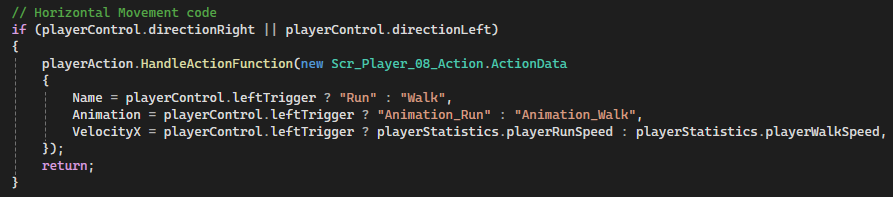
Tambien reinicia la cantida de saltos que se pueden ejecutar.

##### HandleMovementGrounded ()



Esta funcion maneja el movimiento en tierra del personaje.

Si el jugador esta en tierra (*stateGround*) y puede moverse (*passiveAction*) se ejecuta esta linea que se encarga de girar al personaje dependiendo *rightSide*.



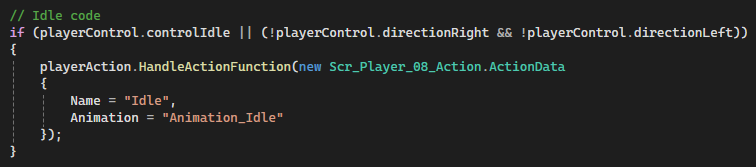
El movimiento horizontal se ejecuta utilizando uno de dos imputs (*directionRight y directionLeft*).

Esta parte del codigo utiliza varias veces la siguiente estructrua:

*playerControl.leftTrigger ? value1 : value2*.

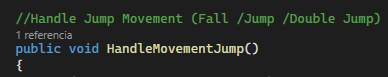
Esto signfica que dependiendo si *leftTrigger* es true, que valor se usa. Si es true se usa *vaule1* y si es false *vaule2*.

Por ejemplo, *Name* es *Run*  si *leftTrigger* es true y *Walk* si es false.



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.

##### HandleMovementJump()



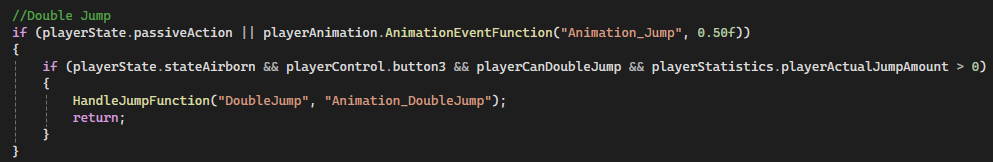
Esta funcion maneja el satlo del personaje, tanto en tierra como en aire.



Si el personaje se puede mover (*passiveAction*):

En tierra (*stateGrounded*), si se presiona el boton (*button3*) mientras la cantidad de saltos disponibles (*playerActualJumpAmount*) sea mayor que 0 entonces el personaje salta.

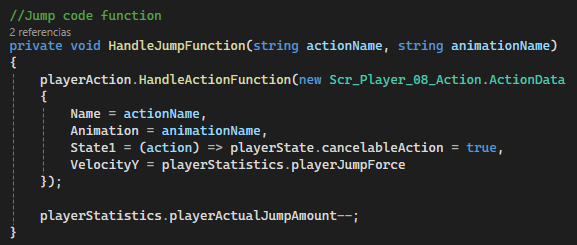
En el aire (*stateAirborn*) y la accion actual no es *Jump* o *DoubleJump*, entonces el personaje cae.



El doble salto se puede ejecutar con dos condicines principales, si el jugador se puede mover (*passiveAction*), y a la mitad de la animacion de saltar(*“Animation\_Jump”, 0.50f*).

La accion se ejecuta si se presonaje esta en el aire (*stateAirborn*), se presiona el boton (*button3*), el personaje tiene la habilidad (*playerCanDoubleJump*) y la cantidad de saltos saltos disponibles (*playerActualJumpAmount*) sea mayor que 0.

##### HandleJumpFunction()



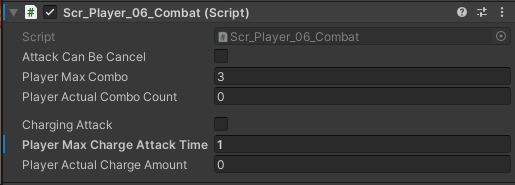
Esta funcion se usa para ejecutar los salto. Es una accion mas asi que usa la funcion *HandleActionsFunction*.

Para saltar se usa la estadistca *PlayerJumpForce* del script de fisicas (*Scr\_Player\_03\_Statistics*).

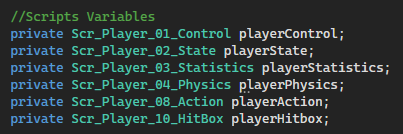
A lo ultimo le restamos a *playerActualJumpAmount*, esto es para controlar los saltos que ejecutamos.

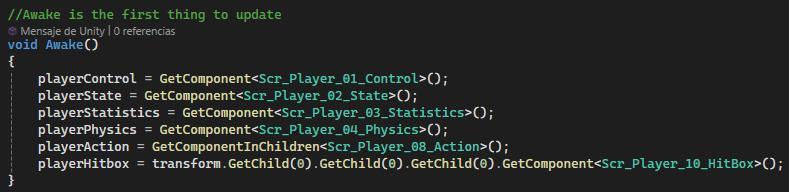
#### 1.1.10 – Scr\_Player\_06\_Combat

Este script se encarga de las acciones de combate.

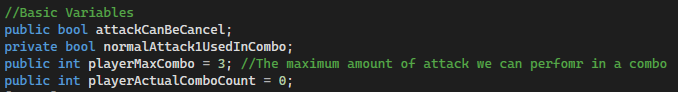


##### Variables





Se crean referencias a cada script, dependiendo de si estan en el mismo objeto (*GetComponent*), en un objeto hijo (*GetComponentInChildren*) o en el primer objeto hijo del objeto hijo (*transform.GetChild(0).GetChild(0).GetChild(0)*) como se tiene que referenciar.



El primer set de variables controla los combos.

*attackCanBeCancel* indica cuando se puede cancelar un ataque, *normalAttack1UsedInCombo* se usa para una cuestion de coordinacion entre el ataque normal 1 y 2.

La int *playerMaxCombo* define la cantidad maxima de ataques que se pueden realizar en un combo (3). *playerActualComboCount* indica el numero de ataques actuales que realizamos.

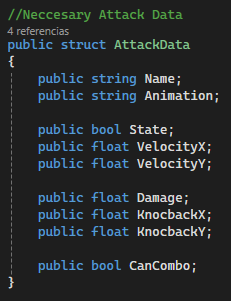


El segundo set de variabels controla los ataques cargados.

*chargingAttack* indica cuando un ataque se esta cargando. La int *playerMaxChargeAttackTime* define la cantidad maxima de tiempo que se puede cargar un ataque antes de ejecutarse automaticamente (1 segundos). *playerActualChargeAmount* indica el tiempo que lleva un ataque cargado actualmente.



*Bools* que determinan que tipo de ataque se puede realizar en relacion al escenario.



Esta estructura es usada para almacenar los datos de los ataque:

*string Name* - El nombre del ataque.

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

*bool State* – El estado que se activa en *Scr\_Player\_02\_State*

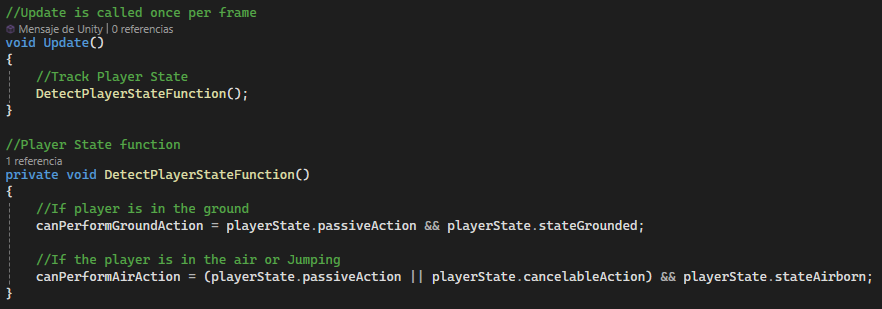
*float VelocityX y float VelocityY* - La velocidad de movimiento del ataque en los ejes X y Y.

*float Damage* – La cantidad de daño que genera el ataque.

*float KnocbackX y float KnocbackY* – El movimiento que genera el ataque en el enemigo en los ejes X y Y.

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

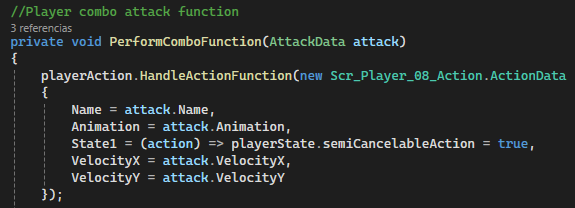
##### DetectPlayerStateFunction()



Este metodo detecta el estado del jugador para el uso de otras funciones. Lo hace a traves de la bool *canPerformGroundAction* y *canPerformAirAction*.

Es la unica que esta en *Update()* porque requiere que se actualice.

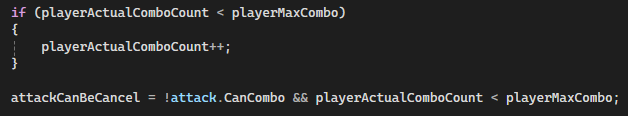
##### PerformComboFunction()



Esta función maneja los ataques en un combo. Los datos que pasemos de *AttackData* se pasaran a la funcion *HandleActionFunction* para realizar la accion.



Se le pasa los valores del ataque correspondiente (Daño y Knockback) a la HitBox.

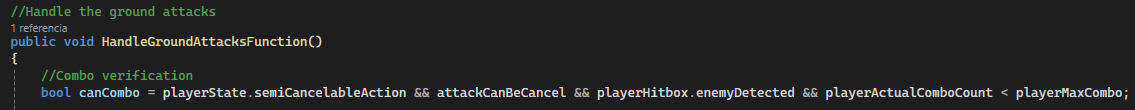


Si el contador de combos no ha alcanzado el límite máximo, el contador de combos se incrementa.

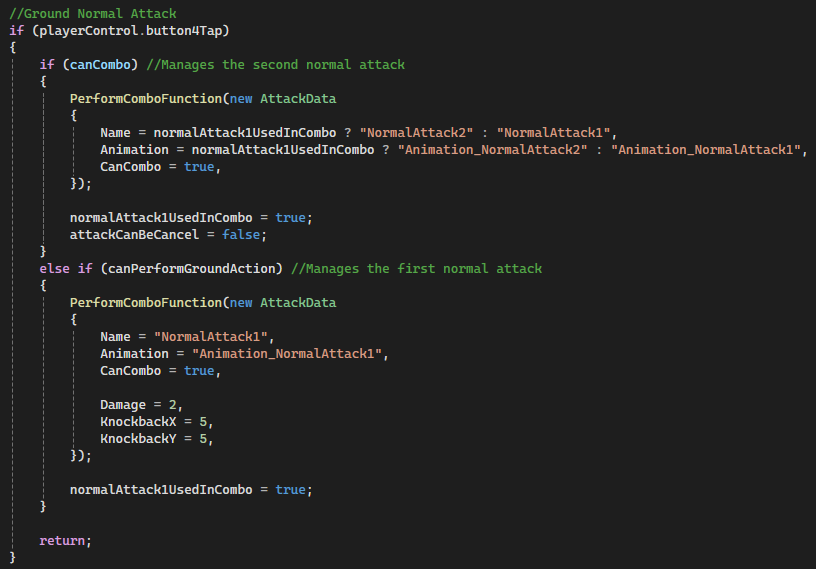
La ultima linea de alguna manera hace que el contador de combos funcione correctamente.

##### HandleGroundAttacksFunction()

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



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.

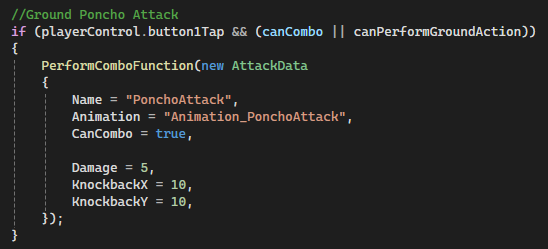


Si se presiona *button4Tap*, se ejecuta un combo o un ataque simple dependiendo de si el jugador ya ha usado un ataque en el combo.

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*).

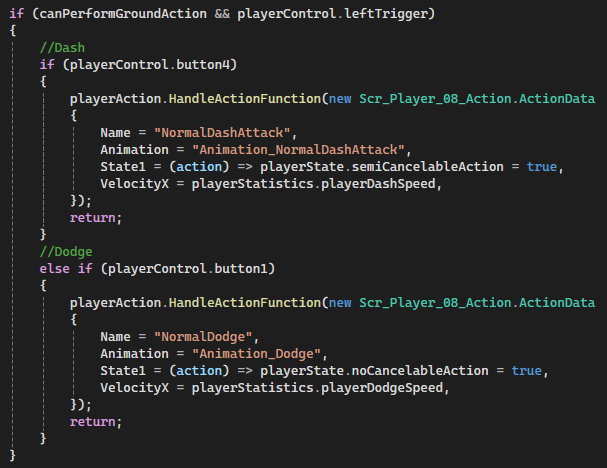
Si es false y canPerformAttack es true, lo que significa que es el primer ataque, entonces se realiza una llamada a *PerformComboAttackFunction* y se pasa solo la informacion de "*NormalAttack1*".



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

##### HandleMovementAttackFunction()

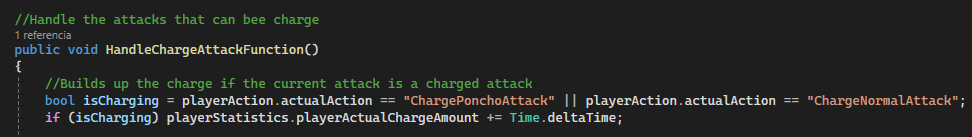
Esta función maneja los ataques que involucran movimiento, como el Dash y el Dodge. Tambien usa *bool canperformGroundAction* que se prende si el jugador con tiene la capacidad de moverse y esta en tierra.



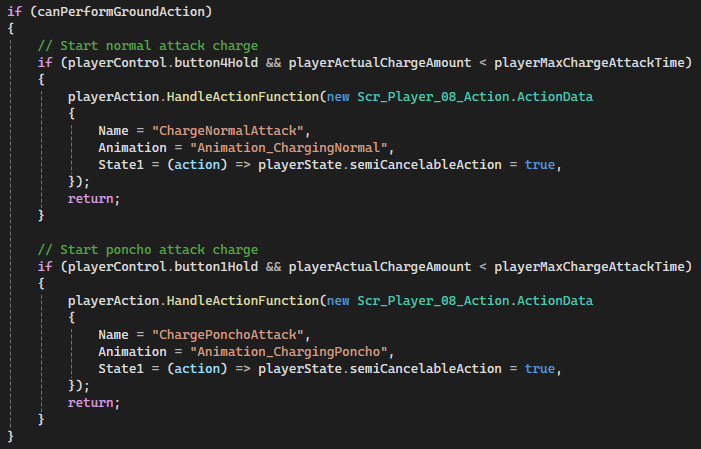
Si presiona los botones correspondientes (*leftTrigger* + *button 1 / 4*), se llama a *HandleActionFunction* y le pasamos la estadistica correspondiente en *VelocityX*.

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

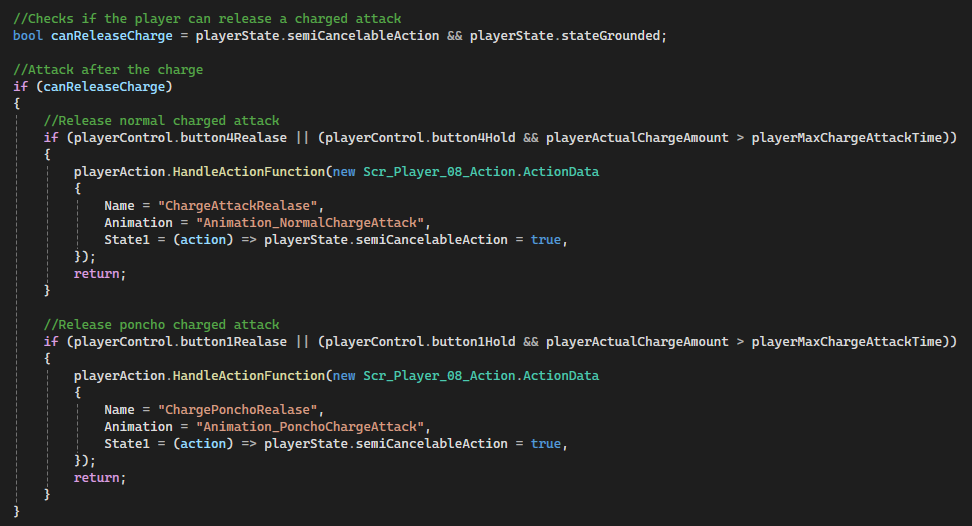
##### HandleChargeAttackFunction()



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



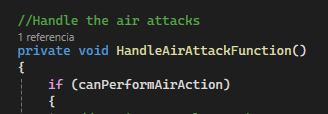
Si el jugador mantiene *button4Hold* o *button1Hold* y la carga actual es menor al tiempo maximo de carga, se comienza a cargar el ataque.



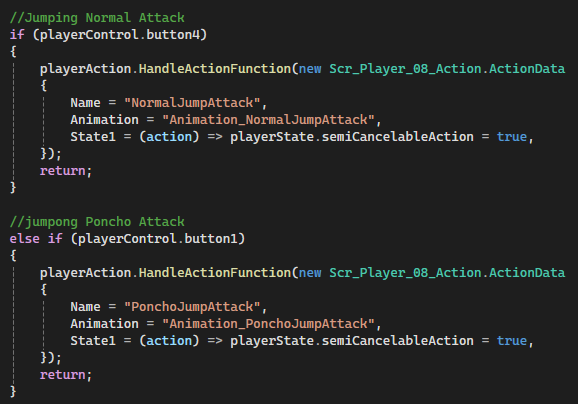
La *bool* *canReleaseCharge* esta para poder integrar ataques cargados a los combos, se prende si el jugador esta en una accion semi cancelable (*semiCancelableAction*) y esta en tierra (*stateGrounded*).

Si el jugador suelta el botón (*button4Realase, button1Realase*), o si sigue presionándolo pero ya superó el tiempo máximo (*playerMaxChargeAttackTime*), se ejecuta el ataque.

##### HandleAirAttackFunction()

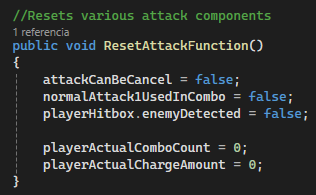


Maneja los ataques del jugador cuando está en el aire. Similar a los ataques en el suelo, esta función verifica si se puede ejecutar el ataque a traves de la bool *canPerformAirAction*.



Si el jugador presiona el botón *button4* o *button1* mientras está en el aire, ejecuta un ataque.

##### ResetAttackFunction()



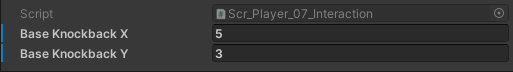
Esta función se encarga de restablecer varios componentes de ataque después de que se complete su animacion.

Resetea varios estados:

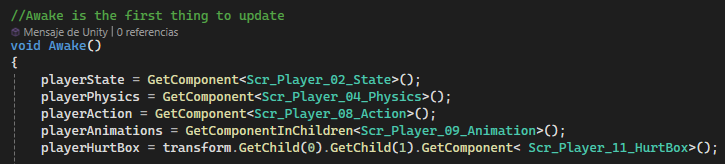
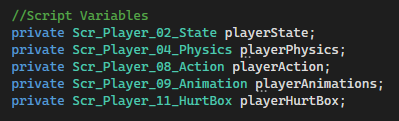
* Se desactiva la posibilidad de cancelar el ataque.
* Se desactiva la detección de enemigos,.
* Se reinicia el contador de combos.
* Se reinicia la carga de los ataques.

#### 1.1.11 – Scr\_Player\_07\_Interaction

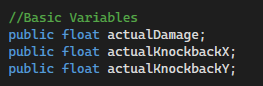
Este maneja las interacciones del jugador con el mundo y otras entidades (Enemigos, Bosses, NPCs)



##### Variables



Se crean referencias a cada script, dependiendo de si estan en el mismo objeto (*GetComponent*), en un objeto hijo (*GetComponentInChildren*) o en el segundo objeto hijo del objeto hijo (*transform.GetChild(0).GetChild(1)*) como se tiene que referenciar.



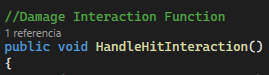
Estos floats determinan el daño y las idistancias actuales de recibe el personaje al ser dañado.

##### Update()

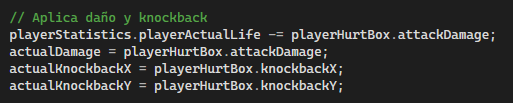


La bool *invulnerableCombatAction* de *playerState* controla la colision entre las capas del jugador (*Layer\_Player*) y los enemigos (*Layer\_Enemy*), si es falsa colsiona si es true se ignoran.

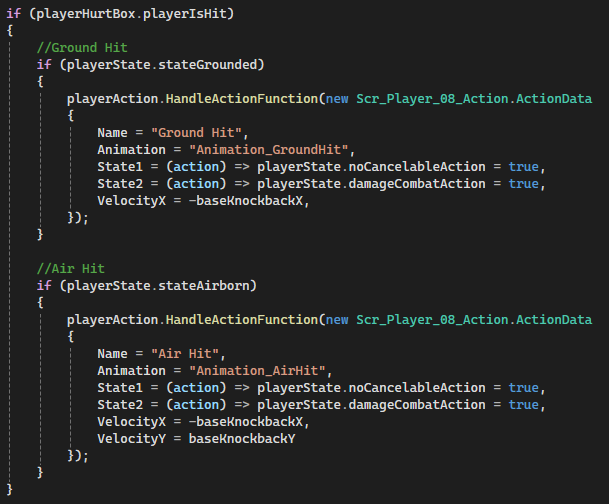
##### HandleHitInteraction()



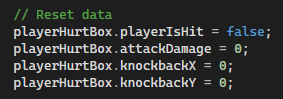
Esta se encarga de manejar el comportamiento al ser golpeados por un enemigo.



Se guarda la información del golpe recibido por playerHurtBox para usarla. Se resta la vida y se aplica el Knocback.

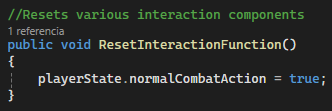


Si el jugador es golpeado (*playerIsHit*) se ejecuta la accion correspodiente si esta en tierra (*stateGround*, solo se le aplica *VelocityX*) o en el aire (*stateAirborn*, se le aplica *VelocityX* e *VelocityY*).



Resetea los datos del ataque para que no se manden varias veces.

##### ResetInteractionFunction()



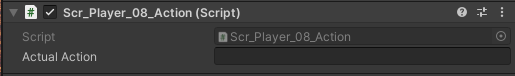
Esta función se encarga de restablecer varios componentes de las interaciones después de que se complete su animacion.

Resetea varios estados:

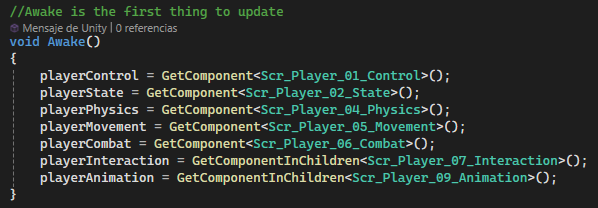
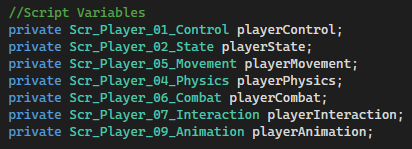
* Se resetea el *Current Combat State* a *normalCombatAction*.

#### 1.1.12 – Scr\_Player\_08\_Action

Este maneja las interacciones del jugador con el mundo y otras entidades (Enemigos, Bosses, NPCs)



##### Variables

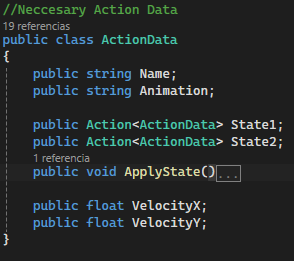


Se crean referencias a cada script, dependiendo de si estan en el mismo objeto (*GetComponent*) o en un objeto hijo (*GetComponentInChildren*).



*actualAction* guarda la acción actual que está ejecutando el jugador.

*rightSide* define si el personaje está mirando a la derecha (*true*) o a la izquierda (*false*).



Esta estructura es usada para almacenar los datos de las acciones:

*string Name* - El nombre de la accion.

*string Animation* - El nombre de la animación asociada con a la accion.

*bool State1 y State2* – El estado que se activa en *Scr\_Player\_02\_State*

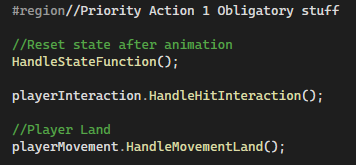
*float VelocityX y float VelocityY* - La velocidad de movimiento del ataque en los ejes X y Y.

##### Update()

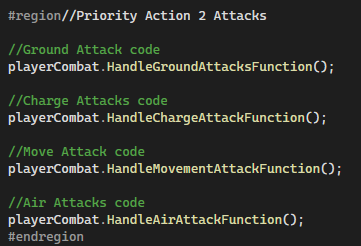


Dese este update es que se controlan todas las acciones, llamandolas desde su funcion en los scripts *05\_Movement, 06\_Combat* y *07\_Interaction*.

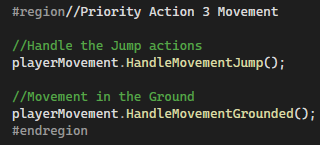
El orden de las funciones importa porque la mayoria de acciones terminan con un *return;* lo que significa que no evaluara mas acciones en ese frame. Mientras mas arriba este una funcion, mas preoridad tiene la accion.



Priority Action 1 es para las acciones que son de ejecucion obligatoria, las cuales se tienen que ejecutar si o si, como ser golpeado (*HandleHitInteraction*) o aterrizar (*HandleMovementLand*).



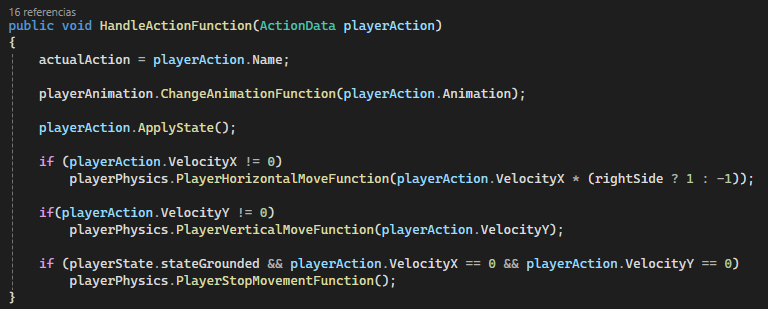
Priority Action 2 es para las acciones que pueden cancelarse bajo ciertos aspectos y rquieren input. Todas las funciones de ataque se encuentran en esta rergion.



Priority Action 3 es para las acciones que pueden cancelarse facilmente o no rquieren input. La funcion de salto (*HandleMovementJump*) y movimiento general (*HandleMovementGrounded*) se encuentra en esta parte.

##### HandleActionFunction()

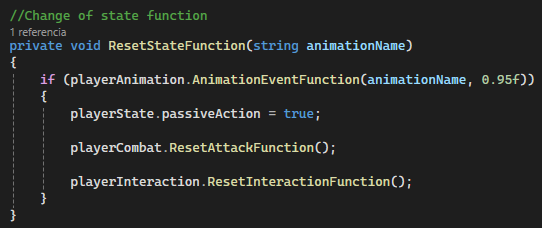
Esta funcion se encarga de ejecutar todas las partes necesarias para la accion, es la que referencian los demas scripts a la hora de hacer una.



Ejecuta una acción específica basada en los datos que se pasen en *playerAction*:

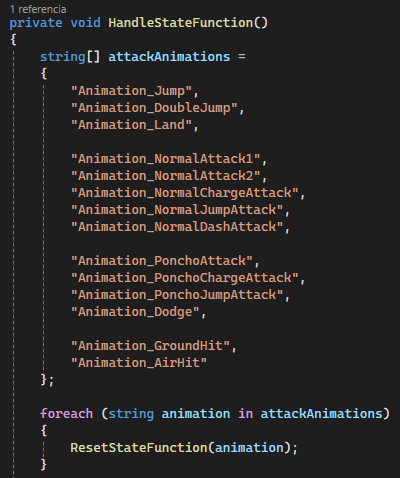
* Actualiza la acción actual (*playerAction.Name*)
* Cambia la animación (*ChangeAnimationFunction(playerAction.Animation*))
* Aplica cambios de estado. (*playerAction.ApplyState()*)
* Si *VelocityX* no es 0 llama a *PlayerHorizontalMoveFunction*() y ajusta la dirección dependiendo de *rightSide*.
* Si *VelocityY* no es 0 llama a *PlayerVerticalMoveFunction*().
* Si *VelocityX* y *VelocityY* es 0 y el personaje está en el suelo, llama a *PlayerStopMovementFunction*().

##### ResetStateFunction()



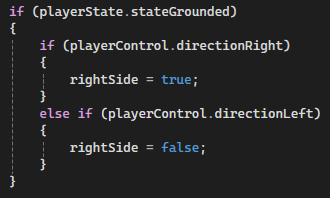
Esta funcion se encarga de resetear aspectos claves de los scripts al finalizar la animacion de una accion.

##### HandleStateFunction()



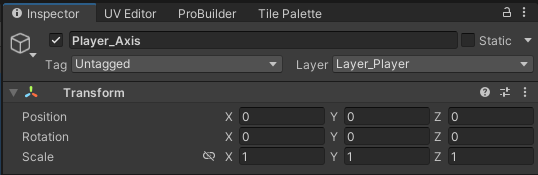
Esta funcion controla aspectos extras necesarios para el funcionamiento de las acciones.

Contiene la lista de acciones (*attackAnimations*) que al finalizar necesitan un reseteo.



Tambien ajusta la direccion al a que mira el jugador dependiendo de la ultima tecla que se presiono en tierra.

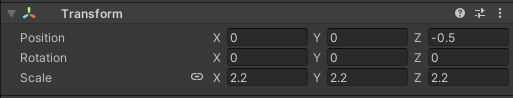
### 1.2 Player\_Axis (Hijo)



Este objeto determina el axis en el que se ve el Sprite.

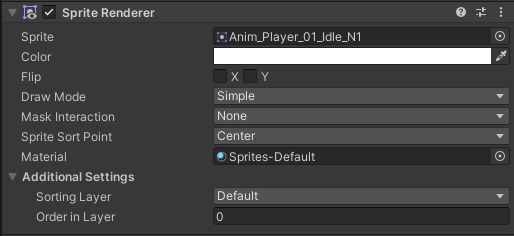
### 1.3 Player\_Sprites (Hijo)

#### 1.3.1 - Transform



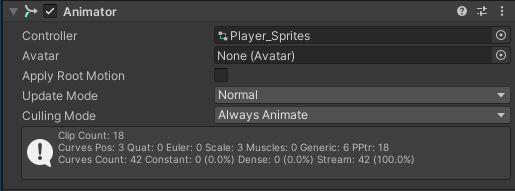
Determina la posición, rotación y escala del objeto.

#### 1.3.2 – Sprite Renderer



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

#### 1.3.3 - 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*



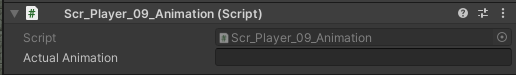
Luego, necesitamos los sprites que componen cada fotograma de animación, representados por rombos en la línea de tiempo.

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

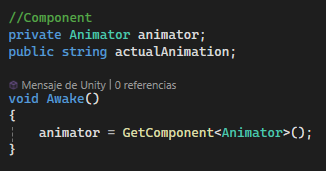


#### 1.3.4 – Scr\_Player\_09\_Animations

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



##### Variables

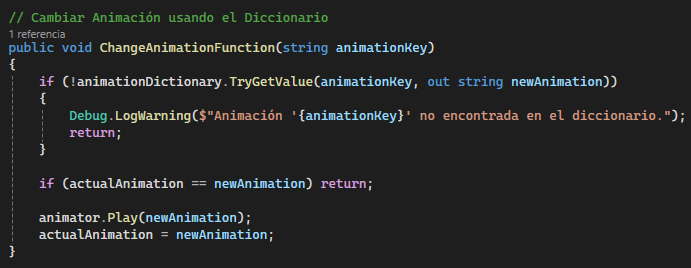


Se hace referencia al componente *Animator* de Unity y se hace un *string* que determina cual es la animacion actual que se esta ejecutando.



Este diccionario se encarga de asociar una animacion a una string resumido para que sea mas facil la sintaxis del codigo a la hora de pasar una imagen.

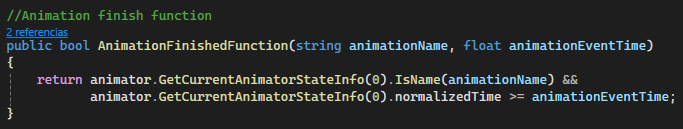
##### ChangeAnimationFunction()



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*.

Tambien verifica si la animacion existe en el diccionario y nos manda un mensaje si no es el caso.

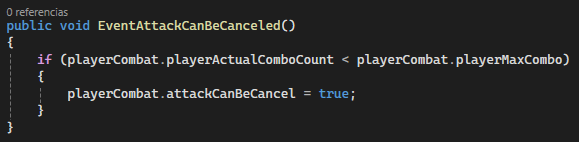
##### AnimationFinishedFunction()



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.

##### Animation Events

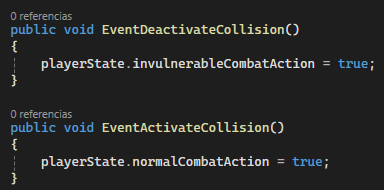
Estas funciones son usadas por el animador en ciertos frames especificos de la animaciones.



Indica el frame cuando un ataque puede ser cancelado.

Se usa en:

* Anim\_Player\_08\_NormalAttack1
* Anim\_Player\_09\_NormalAttack2
* Anim\_Player\_14\_PonchoAttack



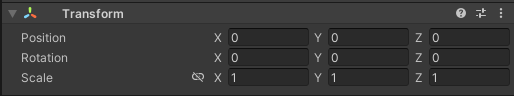
Activan y desactivan la colision entre jugador y enemigo.

Se usa en:

* Anim\_Player\_18\_Dodge

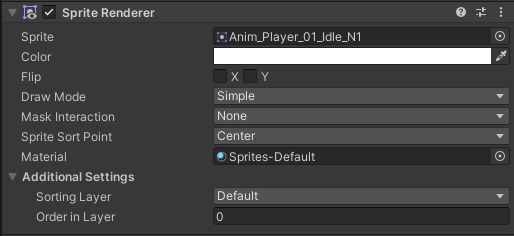
### 1.4 Player\_HitBox (Hijo)

#### 1.4.1 - Transform



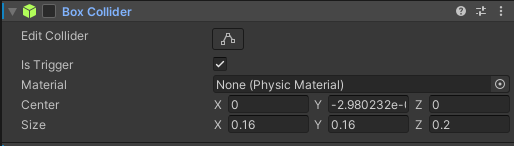
Determina la posición, rotación y escala del objeto.

#### 1.4.2 – Sprite Renderer



Este componente se usa para renderizar las imágenes en 2D. Se usa para ver visualmente la zona de colision.

#### 1.4.3 – Box Collider



Este compoente se encarga de demarcar la zona activa de nuestra hitbox en los ataques.

##### 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.

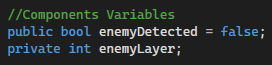


#### 1.4.4 – Scr\_Player\_10\_HitBox

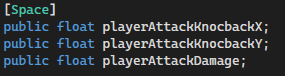


Este script controla la deteccion de la Hit Box.

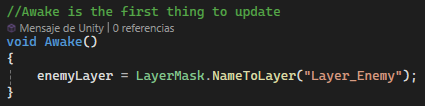
##### Variables



La bool *enemyDetected* se prende si la Hit Box detecta un enemigo.

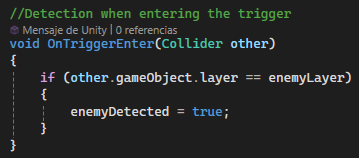


*playerAttackDamage* es la cantidad de daño que genera el ataque. *playerAttackKnocbackX* y *playerAttackKnocbackY* es la distancia en X e Y que recorre el enemigo al ser golpeado.



*enemyLayer* es la Layer que tiene que tener el objeto para detectarse como enemigo (*Layer\_Enemy*).

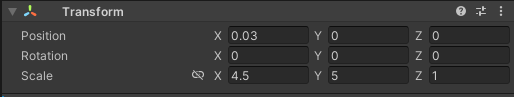
##### OnTriggerEnter()



Si se entra en el trigger de un objeto con la misma layer que *enemyLayer* entonces se prende la bool *enemyDetected*.

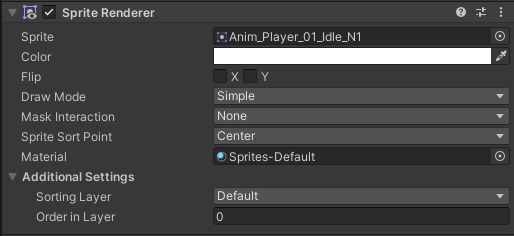
### 1.5 Player\_HurtBox (Hijo)

#### 1.5.1 - Transform



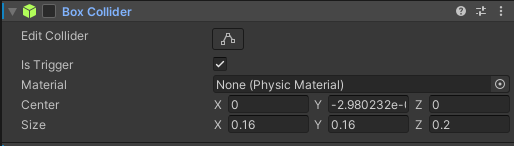
Determina la posición, rotación y escala del objeto.

#### 1.5.2 – Sprite Renderer



Este componente se usa para renderizar las imágenes en 2D. Se usa para ver visualmente la zona de colision.

#### 1.5.3 – Box Collider



Este compoente se encarga de demarcar la zona activa de nuestra hitbox en los ataques.

##### 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.

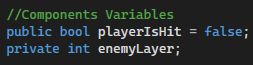


#### 1.5.4 – Scr\_Player\_11\_HurtBox

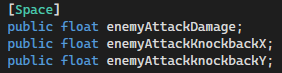


Este script controla la deteccion de la Hurt Box.

##### Variables

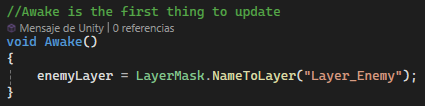


La bool *playerIsHit* se prende si la Hurt Box detecta un enemigo.



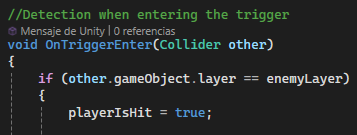
*attackDamage* guarda la cantidad de daño que el enemigo inflige.

*knockbackX* y *knockbackY* guarda la fuerza de empuje en X e Y del ataque.

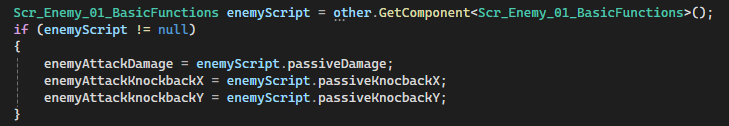


*enemyLayer* es la Layer que tiene que tener el objeto para detectarse como enemigo (*Layer\_Enemy*).

##### OnTriggerEnter()

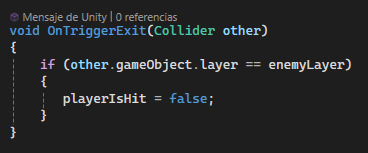


Si se entra en el trigger de un objeto con la misma layer que *enemyLayer* entonces se prende la bool *playerIsHit*.



Verifica que el objeto con el que se colisiona tenga el Script *Scr\_Enemy\_01\_BasicFunctions* y saca los datos correspondientes (Daño y Knocback).

##### OnTriggerExit()



Si se saca el trigger de un objeto con la misma layer que *enemyLayer* entonces se apaga la bool *playerIsHit*.