DATA-DRIVEN

CHARACTER SYSTEM

Germán López Gutiérrez

Contenido

Índice de Figuras

Índice de Tablas

Motivación

Actualmente, para la codificación de los personajes de Howl of Iron se ha hecho uso de una jerarquía basada en **herencia**, donde existe un *HICharacter* que hereda de la clase *Character* por defecto de Unreal Engine 4. Esta clase *HICharacter* es la que contiene las funciones y variables comunes para todos los personajes del juego, tales como la vida o la velocidad de movimiento.

A raíz de esta clase, nacen el actor *HIWerewolf* (el personaje controlado por el jugador) y la clase *HIEnemy*. Esta última clase es la que da lugar a los diferentes enemigos del juego, y es la que contiene las funciones y variables concretas de los mismos.

A continuación, en la Figura 1 se muestra un diagrama UML con las diferentes ramificaciones.

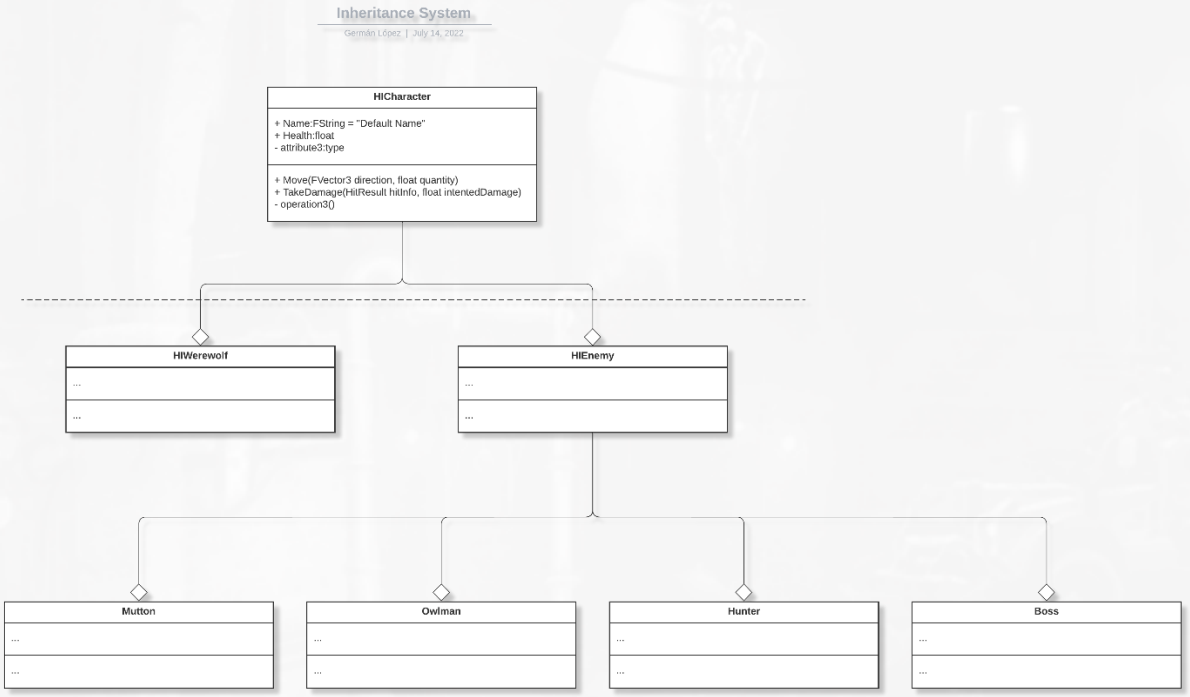


Figura 1 - Sistema de Herencia para los personajes

Sin embargo, la utilización de este sistema ha dado lugar a los siguientes problemas:

- Variables dispersas en diferentes lugares.
  - Al no estar todas las variables de los personajes englobadas en un único lugar (ya sea en una *DataTable* o en un *DataAsset*), al ser estas propias de la clase padre o únicas de los diferentes hijos, esto ha provocado que no exista un único lugar donde se encuentren las variables, complicando la labor de diseño.
  - Actualmente existen **150** variables en **32** lugares distintos (Más información aquí).
- Limitación en el prototipado
  - Muchos de los comportamientos de los personajes se encuentran programados en C++.
  - Si bien, muchos de los eventos son llamados en *Blueprints*, existe una serie de condiciones entre padres e hijos. Estos problemas se encuentran principalmente en el desarrollo de la IA ya que, al no estar utilizándose de base los componentes de Unreal Engine como el AI Perception, esto complica el prototipado de un nuevo personaje, requiriendo siempre del apoyo de un programador para hacer un personaje desde cero.

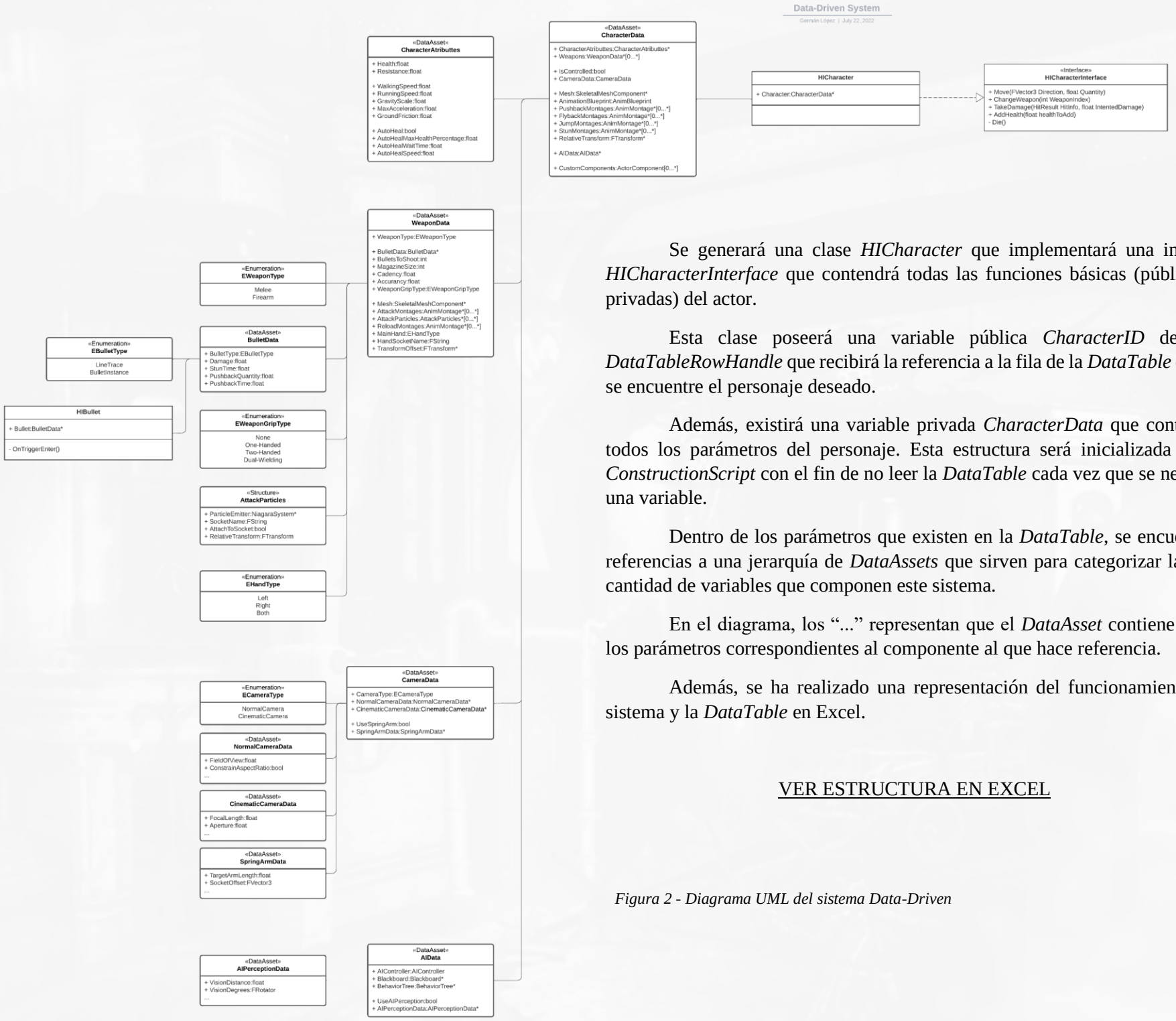
Por estas razones, en este documento se propone un sistema parcialmente diferente, donde se seguirá un flujo de trabajo **Data-Driven**, pero manteniendo algunas de las características del sistema actual.

Para ello, a continuación, se englobarán las **características de este sistema**, seguido por **la implementación del mismo** y, por último, analizando la **Pipeline** a seguir para este nuevo sistema.

Desarrollo del Sistema

Este sistema requiere de la creación de las siguientes variables por parte del equipo de programación:

- *CharacterData (DataTable)*
  - *DataTable* con todos los atributos que conforman un personaje.
  - El actor posee una variable pública que recibe una referencia a la *DataTable* que contiene los parámetros, los cuáles son procesados en el *ConstructionScript* dando lugar al personaje en cuestión.
- *CharacterAtributtes (DataAsset)*
  - *DataAsset* que posee todos los atributos de estado de un personaje (salud, velocidad de movimiento, gravedad, etc.).
- *WeaponData (DataAsset)*
  - *DataAsset* que posee todos los atributos respecto a un arma que pueda poseer un personaje.
- *EWeaponType (Enum)*
  - Enumerador que representa el tipo de arma a utilizar.
  - Estados: *Melee, Firearm*
- *EWeaponGripType (Enum)*
  - Enumerador que representa el manejo del arma.
  - Estados*: None, One-Handed, Two-Handed, Dual-Wielding*
- *EHandType (Enum)*
  - Enumerador que representa la mano con la que el actor agarra el arma.
  - Estados: *Left, Right, Both*
- *BulletData (DataAsset)*
  - *DataAsset* que posee todos los atributos referentes a la bala de un arma de fuego.
- *EBulletType (Enum)*
  - Enumerador que representa el tipo de bala que utiliza el arma para detectar enemigos.
  - Estados: *LineTrace, BulletInstance*
- *AttackParticles (Structure)*
  - Estructura de datos que contiene todos los atributos de un sistema de partículas.
- *CameraData (DataAsset)*
  - *DataAsset* con todos los atributos de la cámara del personaje en caso de que este sea controlado por el jugador.
- *ECameraType (Enum)*
  - Enumerador que representa el tipo de cámara que implementa el actor.
  - Estados: *NormalCamera, CinematicCamera*
- *NormalCameraData (DataAsset)*
  - *DataAsset* con los atributos de la cámara normal.
- *CinematicCameraData (DataAsset)*
  - *DataAsset* con los atributos de la cámara cinematográfica.
- *SpringArmData (DataAsset)*
  - *DataAsset* con los atributos del brazo de la cámara.
- *AIData (DataAsset)*
  - *DataAsset* que posee todos los parámetros y condiciones de la IA del personaje en caso de no ser controlado por el jugador.
- *AIPerceptionData (DataAsset)*
  - *DataAsset* que contiene toda la información referente al componente *AISense* de la IA.



Se generará una clase *HICharacter* que implementará una interfaz *HICharacterInterface* que contendrá todas las funciones básicas (públicas y privadas) del actor.

Esta clase poseerá una variable pública *CharacterID* de tipo *DataTableRowHandle* que recibirá la referencia a la fila de la *DataTable* donde se encuentre el personaje deseado.

Además, existirá una variable privada *CharacterData* que contendrá todos los parámetros del personaje. Esta estructura será inicializada en el *ConstructionScript* con el fin de no leer la *DataTable* cada vez que se necesite una variable.

Dentro de los parámetros que existen en la *DataTable*, se encuentran referencias a una jerarquía de *DataAssets* que sirven para categorizar la gran cantidad de variables que componen este sistema.

En el diagrama, los "..." representan que el *DataAsset* contiene todos los parámetros correspondientes al componente al que hace referencia.

Además, se ha realizado una representación del funcionamiento del sistema y la *DataTable* en Excel.

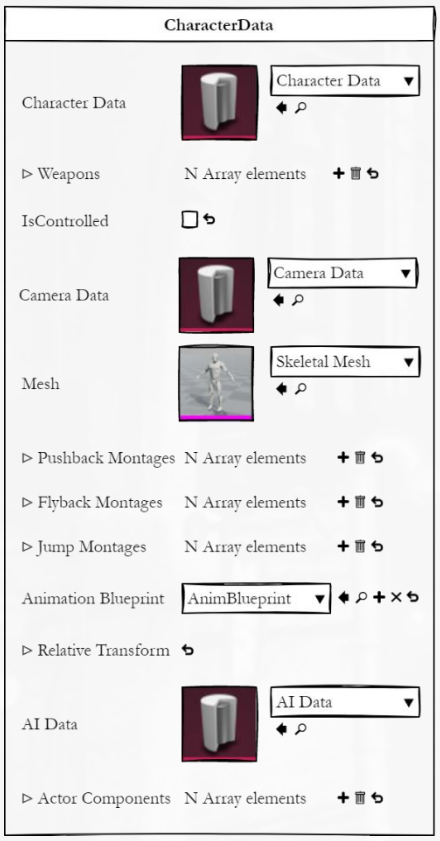
VER ESTRUCTURA EN EXCEL

Figura 2 - Diagrama UML del sistema Data-Driven

Variables

Character Data

CharacterData es una estructura de datos que contiene **todas** las variables de un personaje. A raíz de esta estructura se generará una *DataTable* que contendrá los distintos personajes del videojuego.



- **Main Data**
  - CharacterAtributtes
  - Weapons
- **Controller**
  - IsControlled
  - CameraData
- **Mesh Data**
  - Mesh
  - AnimationBlueprint
  - PushbackMontages
  - FlybackMontages
  - JumpMontages
  - RelativeTransform
- **AI Data**
  - AIData
- **Other Data**
  - CustomComponents

Figura 3 - Representación de las variables de CharacterData

A continuación, en la Tabla 1 se encuentran todas las variables desglosadas, junto a su tipo y una descripción.

Variable	Tipo	Descripción
CharacterAtributtes*	CharacterData	*DataAsset* que posee todos los atributos de estado de un personaje (salud, velocidad de movimiento, gravedad, etc.).
Weapons*	WeaponData [0...*]	*DataAsset* que posee todos los atributos respecto a un arma que pueda poseer un personaje.
IsControlled	Bool	
CameraData*	CameraData	*DataAsset* con todos los atributos de la cámara del personaje en caso de que este sea controlado por el jugador.
Mesh*	SkeletalMesh	*SkeletalMesh* del personaje.
PushbackMontages*	AnimMontage [0...*]	*Montages* para cuando inicie un *Pushback*.
FlybackMontages*	AnimMontage [0...*]	*Montages* para cuando inicie un *Flyback*.

AnimationBlueprint*	AnimBlueprint	*AnimationBlueprint* del personaje, con toda la lógica y variables para las transiciones entre estados y sistema locomoción.
RelativeTransform	Transform	*RelativeTransform* del *SkeletalMesh* del personaje.
AIData*	AIData	*DataAsset* que posee todos los parámetros y condiciones de la IA del personaje en caso de no ser controlado por el jugador.
ActorComponents*	ActorComponent [0...*]	Componentes adicionales para el actor.

Tabla 1 - Variables de CharacterData

Character Atributtes

CharacterAtributtes es un *DataAsset* que contiene todos los atributos **de estado** del personaje. Los parámetros de esta variable se categorizan en tres tipos:

- Variables básicas
  - Salud, resistencia.
- Variables de movimiento
  - Velocidad de andar, de correr, fricción, etc.
- Variables de curación
  - Velocidad de curación, máxima cantidad de salud curada, etc.

A continuación, en la Tabla 2 se encuentran todas las variables desglosadas, junto a su tipo y descripción.

Variable	Tipo	Descripción
Health	float	Float que determina la salud del personaje.
Resistance	float	Float que determina la resistencia a los daños del personaje.
WalkingSpeed	float	Float que determina la velocidad **al caminar** del *CharacterMovement*.
RunningSpeed	float	Float que determina la velocidad **al correr** del *CharacterMovement*.
GravityScale	float	Float que determina cuánto impacto posee la gravedad sobre el personaje.
MaxAcceleration	float	Float que determina la aceleración máxima del movimiento del personaje.
GroundFriction	float	Float que determina la fricción con otros elementos estáticos.
AutoHeal	bool	Bool que define si el personaje recibe una auto-curación.
AutoHealMaxHealthPercentage	float	Float que determina cuál es el porcentaje máximo de salud al que puede curar.
AutoHealWaitTime	float	Float que determina el tiempo de espera desde que el personaje recibió el último daño hasta que empieza a curarse.
AutoHealSpeed	float	Float que determina la velocidad en la que la salud se recupera.

Tabla 2 - Variables de CharacterAtributtes

Weapon Data

WeaponData es un *DataAsset* que contiene todos los parámetros de **un arma**. Los parámetros de esta variable se categorizan en cuatro tipos:

- Variables básicas
  - Tipo de arma (de fuego o cuerpo a cuerpo).
- Variables de disparo
  - Tamaño del cargador, cadencia, etc.
- Variables de apariencia
  - *Mesh*, partículas de ataque, etc.
- Variables de posicionamiento.
  - Nombre del socket, offset del *transform*.

A continuación, en la Tabla 3 se encuentran todas las variables desglosadas, junto a su tipo y descripción.

Variable	Tipo	Descripción
WeaponType	EWeaponType	Enumerador que determina el tipo de arma.
BulletData	BulletData*	*DataAsset* que determina las características de la bala instanciada.
BulletsToShoot	int	Entero que determina la cantidad de balas por disparo (1 en el caso de una pistola, varias en el caso de una escopeta).
MagazineSize	int	Entero que determina el tamaño de un cargador.
Cadency	float	Float que determina la velocidad de disparo del arma.
Accurancy	float	Float que determina la dispersión de cada disparo.
WeaponGripType	EWeaponGripType	Enumerador que determina el tipo de agarre.
Mesh	SkeletalMeshComponent*	*SkeletalMesh* del arma.
AttackMontages	AnimMontage* [0...*]	Montages de los ataques.
AttackParticles	AnimParticles* [0...*]	Partículas de los ataques.
ReloadMontages	AnimMontages* [0...*]	Montages de recarga.
MainHand	EHandType	Enumerador que determina la mano con la que se sostiene el arma.
HandSocketName	FString	Nombre del socket que sostiene el arma.
TransformOffset	FTransform	Transform del arma respecto al socket.

Tabla 3 - Variables de WeaponData

En la Tabla 3 se muestran tres variables de tipo enumerador. Los valores posibles en ellos se encuentran a continuación.

EWeaponType
Melee
Firearm

EWeaponGripType
None
One-Handed
Two-Handed
Dual-Wielding

EHandType
Left
Right
Both

Bullet Data

BulletData es una estructura de datos que contiene todos los parámetros de un disparo proveniente de un arma. Los parámetros de esta variable se categorizan en tres tipos:

- Variables básicas
  - Tipo de bala, daño.
- Variables de *stun*
  - Tiempo de *stun*, cantidad de *pushback*, etc.

A continuación, en la Tabla 4 se encuentran todas las variables desglosadas, junto a su tipo y descripción.

Variable	Tipo	Descripción
BulletType	EBulletType	Enumerador que determina el tipo de bala/disparo.
Damage	float	Float que determina el daño que inflige el arma a otros personajes.
StunTime	float	Float que determina el tiempo que se encuentra en estado de *stun* el actor que ha recibido el disparo.
PushbackQuantity	float	Float que determina la cantidad de empuje que tiene el disparo contra el actor impactado.
PushbackTime	float	Float que determina la cantidad de tiempo que dura el empuje del disparo contra el actor impactado.

Tabla 4 - Variables de BulletData

En la Tabla 4 se muestra un tipo de variable de tipo enumerador. Los valores posibles se encuentran a continuación.

EWeaponType
LineTrace
BulletInstance

Para las balas que no sean un rayo, será requerida una clase *Bullet*, que reciba como parámetro el *DataAsset* de tipo *BulletData* para inicializar las variables correspondientes y también requerirá de un delegado *OnHit*, para cuando detecte a un actor que implemente la interfaz *HICharacterInterface*, llame a la función *TakeDamage*.

En la Figura 4 se encuentra representado el delegado y la llamada a la función *TakeDamage* del actor impactado.

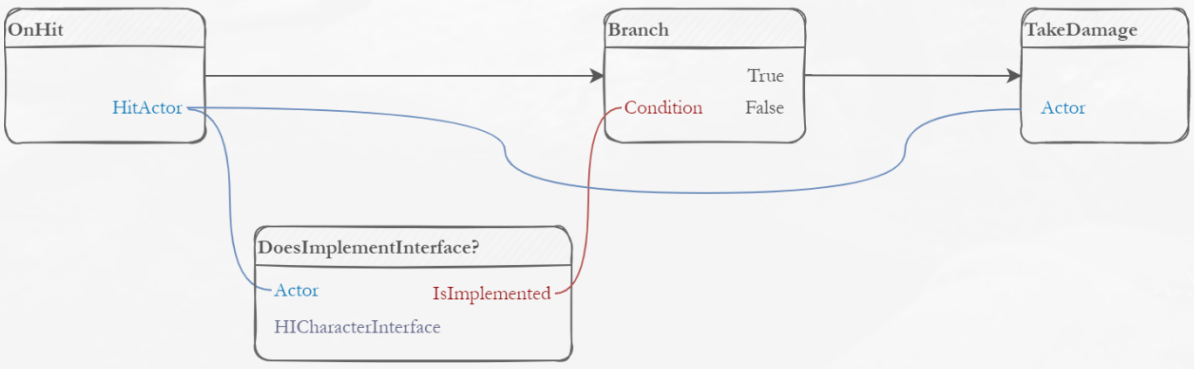


Figura 4 - Delegado OnHit de una bala

Attack Particles

AttackParticles es una estructura de datos que contiene todos los parámetros correspondientes al sistema de partículas de un ataque. Los parámetros de esta variable se categorizan en dos tipos:

- Variables básicas
  - Sistema de partículas
- Variables de posicionamiento
  - Nombre del *socket* al que se encuentra anclada la partícula, *transform* respecto a la zona en la que se instancia, etc.

A continuación, en la Tabla 5 se encuentran todas las variables desglosadas, junto a su tipo y descripción.

Variable	Tipo	Descripción
ParticleEmitter	NiagaraSystem	Sistema de partículas de *Niagara* a instanciar.
SocketName	FString	Nombre del socket al que se encuentra sujeta la partícula.
AttachToSocket	bool	Bool que determina si la partícula se encuentra anclada al *socket*.
RelativeTransform	FTransform	Transform respecto a la zona en la que se instancia el sistema de partículas.

Tabla 5 - Variables de AttackParticles

A la hora de llamar a la función de ataque, se puede llamar al sistema a uno de los sistemas de partículas del *array* de *AttackParticles*.

Camera Data

CameraData es un *DataAsset* que contiene todos los parámetros correspondientes a la cámara. Este *DataAsset* solo será necesario en caso de que el personaje sea controlado por el jugador. Los parámetros de esta variable se categorizan en dos tipos:

- Variables básicas
  - Tipo de cámara, configuración de la cámara.
- Variables de posicionamiento y movimiento
  - Uso de un *SpringArm*, longitud.

Variable	Tipo	Descripción
CameraType	ECameraType	Tipo de cámara que utilizará el actor.
NormalCameraData	NormalCameraData*	*DataAsset* con **todos** los ajustes de la cámara normal en caso de ser esta la utilizada.
CinematicCameraData	CinematicCameraData*	*DataAsset* con **todos** los ajustes de la cámara cinematográfica en caso de ser esta la utilizada.
UseSpringArm	bool	Bool que determina si la cámara se encuentra anclada a un SpringArm.
SpringArmData	SpringArmData*	*DataAsset* con **todos** los ajustes del *SpringArm* en caso de ser utilizado.

Tabla 6 - Variables de CameraData

En la Tabla 6, se muestran tres *DataAssets* que no van a contar con un apartado propio, ya que su implementación solo requiere de copiar e implementar las variables del componente al que hacen referencia. Por otro lado, se muestra un tipo de variable de tipo enumerador. Los valores posibles se encuentran a continuación.

ECameraType
NormalCamera
CinematicCamera

AI Data

AIData es un DataAsset que contiene todos los parámetros en lo referente a la IA del actor. Los parámetros de esta variable se categorizan en dos tipos:

- Variables básicas
  - Controlador de la IA, referencia a la *Blackboard* y el *BehaviorTree*.
- Variables de percepción
  - Parámetros del componente *AIPerception*.

A continuación, en la Tabla 7 se encuentran todas las variables desglosadas, junto a su tipo y descripción.

Variable	Tipo	Descripción
AIController	AIController*	*Controlador* de la IA.
Blackboard	Blackboard*	*Blackboard* de la IA.
BehaviorTree	BehaviorTree*	*BehaviorTree* con el comportamiento y decisiones de la IA.
UseAIPerception	bool	Bool que determina si se utiliza el componente de Unreal Engine *AIPerception*.
AIPerceptionData	AIPerceptionData*	*DataAsset* con **todos** los ajustes del AIPerception en caso de ser utilizado.

Tabla 7 - Variables de BulletData

En la Tabla 7 se muestran un *DataAsset* que no va a contar con un apartado propio, ya que su implementación solo requiere de copiar e implementar las variables del componente al que hace referencia.

Implementación del Actor

En esta sección se detallará la implementación del sistema de creación personajes en Unreal Engine.

Construction Script

En el ConstructionScript es donde se inicializarán los parámetros del personaje para poder previsualizarlo en editor.

En primer lugar, se leerá la fila de la *DataTable* con la configuración del personaje y se almacenarán los datos en una estructura (variable privada). Esta estructura es de la que se **obtendrán** y donde se **actualizarán** todas las variables. A partir de aquí no será necesaria la referencia a la fila de la *DataTable*.

En segundo lugar, se configurará el personaje. Para ello, esta configuración estará repartida en 5 funciones distintas:

- *SetupController*
  - Configura los ajustes del *PlayerController*. En caso de ser utilizado, se configurarán los ajustes de la cámara.
- *SetupWeaponry*
  - Configura el arsenal del personaje, el arma equipada e inicializará cada una de las armas.
- *SetupMesh*
  - Configura todos los parámetros del *SkeletalMesh* y sus animaciones.
- *SetupAI*
  - Configura la IA en caso de utilizar una.
- *SetupCustomComponents*
  - Añade los componentes extra.

En la Figura 5 se muestra el esquema a seguir en formato de *Blueprints*.

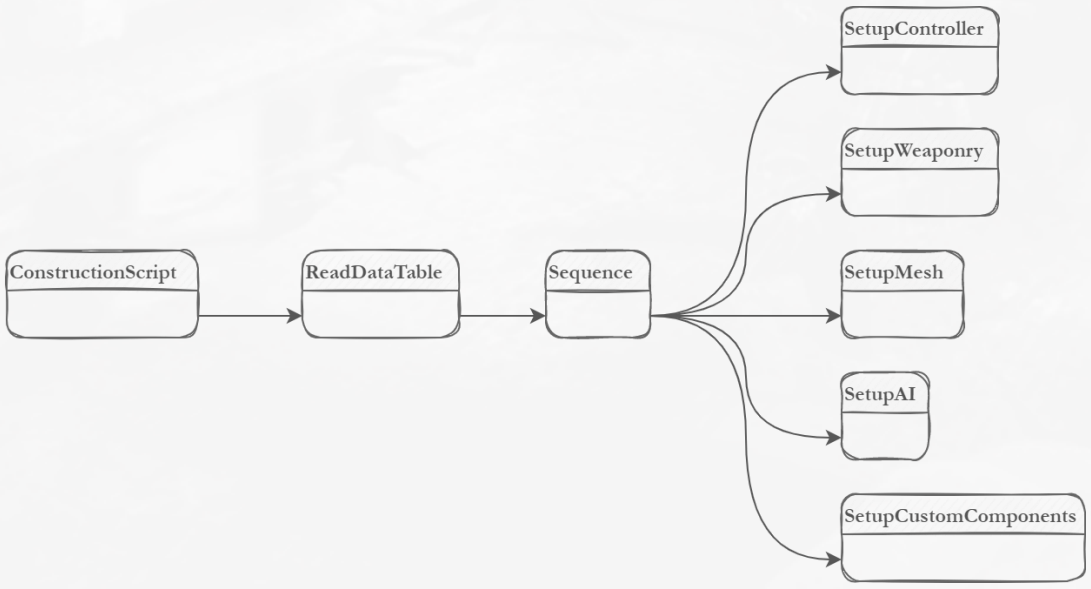


Figura 5 - ConstructionScript, diagrama del Blueprint

Eventos Personalizados

A la hora de crear un personaje, es normal que surja la necesidad por parte de diseño de crear acciones específicas para dicho personaje. Un *dash*, un salto personalizado, un tipo de ataque concreto. Todas esas acciones se gestionarán mediante la creación de un UObject en C++ llamado *CustomEvent*, que posea una función implementable en Blueprints llamada *ExecuteCustomEvent*.

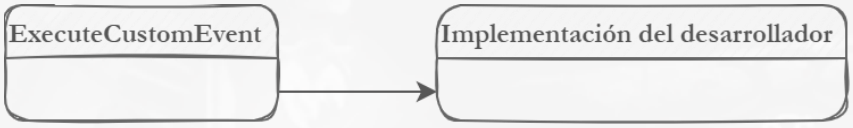


Figura 6 - Implementación de la función ExecuteCustomEvent

Toda la lógica de las funciones personalizadas se localizará en la función *ExecuteCustomEvent*. Por otro lado, este evento personalizado debe ser llamado desde algún lugar. Para ello, se creará una **macro** llamada *CallCustomEvent* que reciba una variable de tipo *CustomEventClass* que será el evento a llamar. Además, contará con dos salidas: *Execute* (la básica) y *OnFinished* (que se llama al terminar la ejecución del evento).

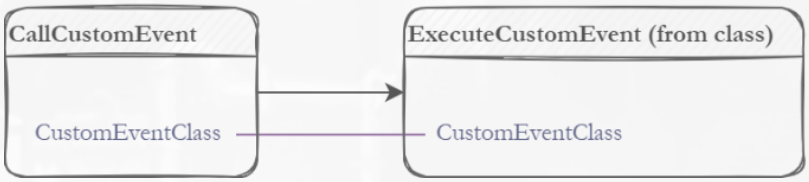


Figura 7 - Implementación de la macro CallCustomEvent

Esta función estática será llamada en el *PlayerController* en el caso de ser un actor controlado por el jugador.

Por otro lado, para las IAs, se creará una *Task* de nombre *ExecuteAICustomEvent* que recibirá una variable de tipo *CustomEventClass* y llamará a la función estática *CallCustomEvent*, introduciendo la variable que recibe la *Task*. Hecho esto, los diseñadores, a la hora de crear un personaje, prototiparán en *Blueprints* mediante los *CustomEvents* las acciones concretas del personaje y las llamará en el *PlayerController* o el *BehaviorTree*.

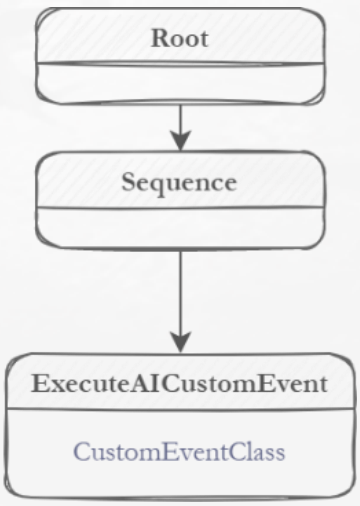


Figura 8 - Llamada en BehaviorTree a la Task ExecuteAICustomEvent

Aun así, los diseñadores tendrán libertad para implementar sus propias *Tasks* a ser llamadas en *BehaviorTrees*, siempre y cuando se siga la estructura de la Figura 9.

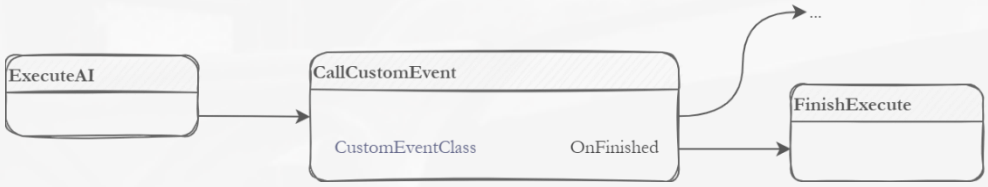


Figura 9 - Implementación de una Task personalizada

ExecuteCustomEvent

Los eventos personalizados cuentan con una serie de parámetros a tener en cuenta, que englobarán los posibles tipos de acciones. A continuación, en la Tabla 8 se encuentran referenciados dichos atributos.

Variable	Tipo	Descripción
CasterReference	Actor*	Referencia al actor que ha llamado al evento.
Duration	float	Duración del evento.
AppliedEffectQuantity	float	Cantidad del efecto aplicado: Daño, curación, etc.

Tabla 8 - Variables del evento ExecuteCustomEvent

CallCustomEvent

La macro de nombre *CallCustomEvent*, deberá replicar los mismos parámetros que el evento *ExecuteCustomEvent*, sumado a la referencia a la clase del evento que se desea invocar. A continuación, en la Tabla 9 se encuentran referenciados dichos atributos.

Variable	Tipo	Descripción
CustomEvent	CustomEventClass	Referencia a **la clase** del evento que se desea invocar.
CasterReference	Actor*	Referencia al actor que ha llamado al evento.
Duration	float	Duración del evento.
AppliedEffectQuantity	float	Cantidad del efecto aplicado: Daño, curación...

Tabla 9 - Variables de la macro CallCustomEvent

CallAICustomEvent

La *Task* de un *BehaviorTree* que llama a un CustomEvent, deberá replicar los mismos parámetros que el evento *ExecuteCustomEvent*, sumado a la referencia a la clase del evento que se desea invocar. A continuación, en la Tabla 10 se encuentran referenciados dichos atributos. La referencia al actor que llama al evento, se referenciará dentro del propio *Blueprint*.

Variable	Tipo	Descripción
CustomEvent	CustomEventClass	Referencia a **la clase** del evento que se desea invocar.
Duration	float	Duración del evento.
AppliedEffectQuantity	float	Cantidad del efecto aplicado: Daño, curación...

Tabla 10 - Variables de la Task CallAICustomEvent

Actores con Player Controller

Los actores que posean un *PlayerController*, implementarán sus acciones en la respuesta a los diferentes inputs del jugador. Desde el *Blueprint* del *PlayerController*, al

detectar un input, se hará una llamada a la macro *CallCustomEvent* y se introducirá la acción deseada (una vez esté creada).

Actores sin Player Controller

Tal y como se ha mencionado en el apartado Eventos Personalizados, los actores que no posean un PlayerController, llamarán a sus respectivos eventos en su *BehaviorTree*.

Existen dos formas en las que un desarrollador puede llamar a su *CustomEvent* en el *BehaviorTree*.

1. Llamando a la *Task* de nombre *CallAICustomEvent*
2. Creando una *Task* personalizada que llame a la macro *CallCustomEvent*.

De esta manera, se cuenta con una gran libertad para el desarrollo de comportamientos tanto por parte de programación como de diseño.

Pipeline de Trabajo

A la hora de trabajar con este sistema, se deben seguir una serie de **6 pasos**, siempre y cuando se esté creando un **personaje de cero**.

Los pasos que debe seguir el equipo una vez esté la parte de arte de dicho personaje terminada son:

1. Importar *Meshes*
    a. Se deberá importar el modelo realizado por el departamento de arte.
    b. Deberán incluirse su esqueleto y animaciones.

2. Generar *AnimBlueprints* y desarrollar los *Montages* de las animaciones
    a. A raíz del modelo y sus animaciones, se generará el *AnimBlueprint* del personaje, donde se gestionarán las transiciones a sus diferentes estados.
    b. Se realizarán los *Montages* de las animaciones como los ataques, saltos, etc.

3. Crear nueva fila en la *DataTable* de *CharacterData*
    a. Se añadirá una nueva fila a la *DataTable* que recibirá los parámetros nuevos del personaje.

4. Implementar nuevos *CustomEvents* y comportamientos
    a. No siempre será necesario crear un nuevo *CustomEvent*.
    b. En función de si es un personaje controlado o no, se realizará la implementación de las acciones en el *BehaviorTree* o en el *PlayerController*.

5. Ajustar los valores de la fila del personaje
    a. Se configurará el valor de los parámetros mencionados en el apartado Variables.

6. Añadir la fila de la *DataTable* al personaje
    a. Se añadirán los ajustes del personaje al actor deseado mediante la variable de tipo *DataTableRowHandle*.

En función del estado del personaje, no será necesario cumplir con todos los pasos anteriores. Por ejemplo, si el personaje se encuentra en un estado avanzado, pero se ha modificado una animación, solo será necesario los pasos 1 y 5.

A continuación, en la Figura 10 se encuentra esquematizado el Pipeline de trabajo para la utilización de este sistema y la creación de personajes.

PIPELINE

Figura 10 - Pipeline de trabajo

Anexos y Referencias

Simulación del sistema mediante Tablas de Datos.

Sistemas de detección (Visual y Auditiva) de los enemigos, atributos del Hombre Lobo y Encounters.

¿Dónde está cada atributo de Combate/Sistemas?