Germán López Gutiérrez

Content

Motivation....................5
System Development....................7
Variables....................9
Character Data....................9
Character Atributtes....................10
Weapon Data....................11
Bullet Data....................12
Attack Particles....................13
Camera Data....................13
AI Data....................14
Actor Implementation....................15
Construction Script....................15
Custom Events....................16
ExecuteCustomEvent....................17
CallCustomEvent....................17
CallAICustomEvent....................17
Actors with Player Controller....................17
Actors without Player Controller....................18
Pipeline....................19
Annexes and References....................21

Figures

Figure 1 - Inheritance system for the characters 5
Figure 2 - Data-Driven System UML Diagram 8
Figure 3 - Representation of CharacterData variables 9
Figure 4 - Bullet OnHit delegate........ 12
Figure 5 - ConstructionScript, Blueprint diagram 15
Figure 6 - ExecuteCustomEvent implementation 16
Figure 7 - CallCustomEvent implementation 16
Figure 8 - ExecutieAICustomEvent, behavior tree example 16
Figure 9 - Custom Task implementation........ 17
Figure 10 - Pipeline........ 20

Tables

Table 1 - CharacterData variables....10
Table 2 - CharacterAtributtes variables....10
Table 3 - WeaponData variables....11
Table 4 - BulletData variables....12
Table 5 - AttackParticles variables....13
Table 6 - CameraData variables....13
Table 7 - AIData variables....14
Table 8 - ExecuteCustomEvent variables....17
Table 9 - CallCustomEvent variables....17
Table 10 - CallAICustomEvent variables....17

Motivation

Currently, for the coding of the characters in Howl of Iron we have made use of a hierarchy based on inheritance, where there is a HICharacter that inherits from the default Character class of Unreal Engine 4. This HICharacter class is the one that contains the common functions and variables for all the characters in the game, such as life or movement speed.

From this class, the HIWerewolf actor (the player-controlled character) and the HIEnemy class are born. This last class is the one that gives rise to the different enemies in the game, and it is the one that contains the specific functions and variables of the enemies.

A UML diagram with the different branches is shown in Figure 1 below.

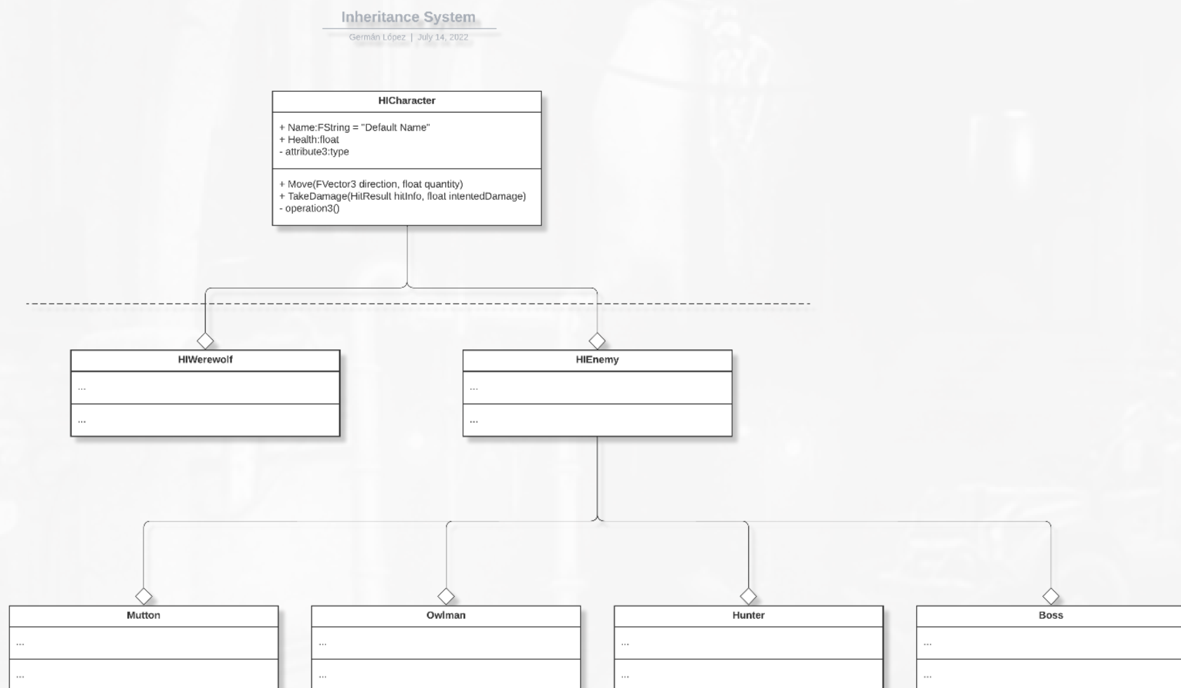


Figure 1 - Inheritance system for the characters

However, the use of this system has led to the following problems:

- Variables scattered in different locations.
  - As all the variables of the characters are not included in a single place (either in a DataTable or in a DataAsset), as they are specific to the parent class or unique to the different children, this has caused that there is not a single place where the variables are located, complicating the design work.
  - Currently there are 150 variables in 32 different places (More information here).
- Limitation in prototyping
  - Many of the character behaviors are programmed in C++.
  - While, many of the events are called in Blueprints, there are a number of conditions between parents and children. These problems are mainly found in the development of the AI since, as Unreal Engine components such as AI Perception are not being used as a base, this complicates the prototyping of a new character, always requiring the support of a programmer to make a character from scratch.

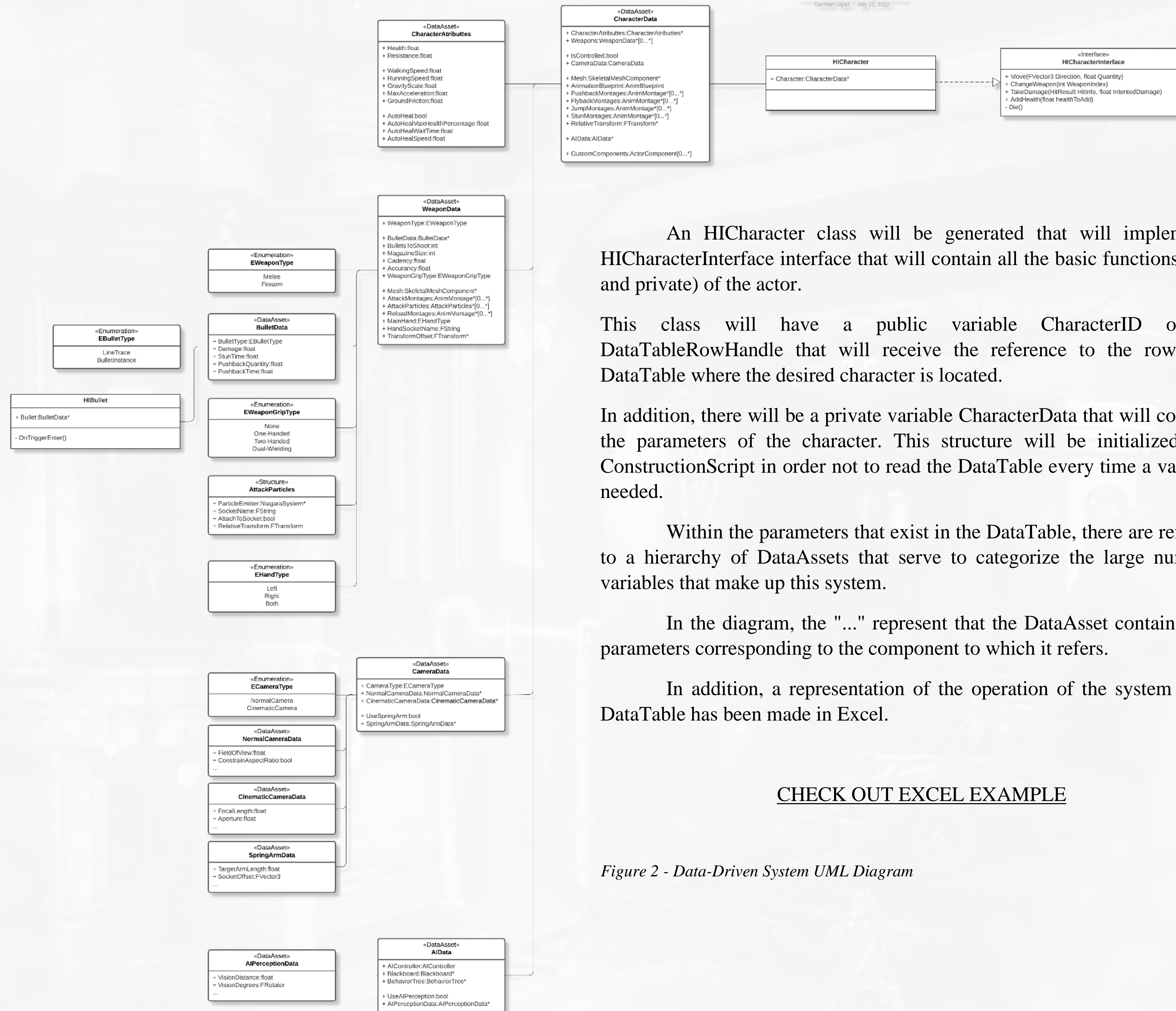
For these reasons, this document proposes a partially different system, where a Data-Driven workflow will be followed, but keeping some of the characteristics of the current system.

For this purpose, the characteristics of this system will be discussed below, followed by its implementation and, finally, analyzing the Pipeline to be followed for this new system together with different practical examples.

System Development

This system requires the creation of the following variables by the programming team:

- *CharacterData (DataTable)*
  - DataTable with all the attributes that make up a character.
  - The actor has a public variable that receives a reference to the DataTable containing the parameters, which are processed in the ConstructionScript giving rise to the character in question.
- *CharacterAtributtes (DataAsset)*
  - DataAsset that possesses all the status attributes of a character (health, movement speed, gravity, etc.).
- *WeaponData (DataAsset)*
  - DataAsset that has all the attributes with respect to a weapon that a character may possess.
- *EWeaponType (Enum)*
  - Enumerator representing the type of weapon to be used.
  - States: *Melee, Firearm*
- *EWeaponGripType (Enum)*
  - Enumerator representing the handling of the weapon.
  - States*: None, One-Handed, Two-Handed, Dual-Wielding*
- *EHandType (Enum)*
  - Enumerator representing the hand with which the actor grasps the weapon.
  - States: *Left, Right, Both*
- *BulletData (DataAsset)*
  - DataAsset that has all the attributes relating to a firearm bullet.
- *EBulletType (Enum)*
  - Enumerator representing the type of bullet used by the weapon to detect enemies.
  - Estados: *LineTrace, BulletInstance*
- *AttackParticles (Structure)*
  - Data structure containing all the attributes of a particle system.
- *CameraData (DataAsset)*
  - DataAsset with all the attributes of the character's camera in case the character is controlled by the player.
- *ECameraType (Enum)*
  - Enumerator representing the type of camera implemented by the actor.
  - States: *NormalCamera, CinematicCamera*
- *NormalCameraData (DataAsset)*
  - DataAsset with the attributes of the normal camera.
- *CinematicCameraData (DataAsset)*
  - DataAsset with film camera attributes.
- *SpringArmData (DataAsset)*
  - DataAsset with the attributes of the camera arm.
- *AIData (DataAsset)*
  - DataAsset that has all the parameters and conditions of the character's AI in case it is not controlled by the player.
- *AIPerceptionData (DataAsset)*
  - DataAsset that contains all the information concerning the AISense component of the AI.



An HICharacter class will be generated that will implement an HICharacterInterface interface that will contain all the basic functions (public and private) of the actor.

This class will have a public variable CharacterID of type DataTableRowHandle that will receive the reference to the row of the DataTable where the desired character is located.

In addition, there will be a private variable CharacterData that will contain all the parameters of the character. This structure will be initialized in the ConstructionScript in order not to read the DataTable every time a variable is needed.

Within the parameters that exist in the DataTable, there are references to a hierarchy of DataAssets that serve to categorize the large number of variables that make up this system.

In the diagram, the "..." represent that the DataAsset contains all the parameters corresponding to the component to which it refers.

In addition, a representation of the operation of the system and the DataTable has been made in Excel.

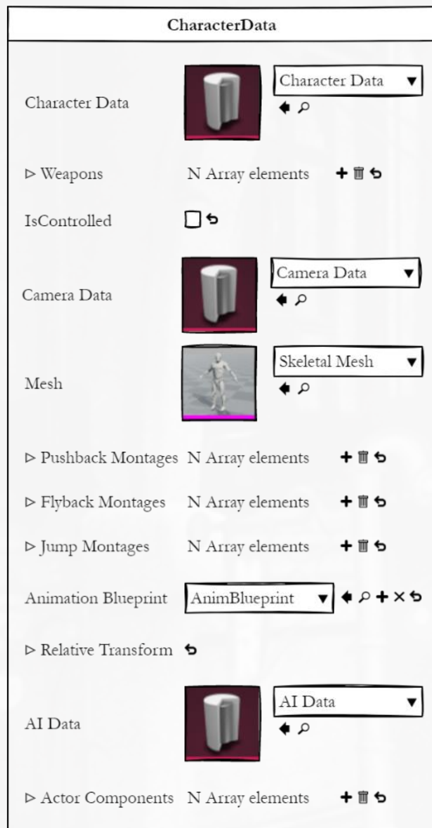
CHECK OUT EXCEL EXAMPLE

Figure 2 - Data-Driven System UML Diagram

Variables

Character Data

CharacterData is a data structure that contains all the variables of a character. From this structure a DataTable will be generated containing the different characters of the videogame.



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

Figure 3 - Representation of CharacterData variables

Table 1 below shows all the variables broken down, together with their type and a description.

Variable	Type	Description
CharacterAtributtes*	CharacterData	DataAsset that has all the status attributes of a character (health, movement speed, gravity, etc.).
Weapons*	WeaponData [0...*]	DataAsset that has all the attributes with respect to a weapon that a character may possess.
IsControlled	Bool	Bool that determines if the carácter is controlled by a player.
CameraData*	CameraData	DataAsset with all the attributes of the character's camera in case the character is controlled by the player.
Mesh*	SkeletalMesh	*SkeletalMesh of the character.*
PushbackMontages*	AnimMontage [0...*]	Montages for when you initiate a Pushback.
FlybackMontages*	AnimMontage [0...*]	Montages for when starting a Flyback.

AnimationBlueprint*	AnimBlueprint	AnimationBlueprint of the character, with all the logic and variables for transitions between states and locomotion system.
RelativeTransform	Transform	RelativeTransform of the character's SkeletalMesh.
AIData*	AIData	DataAsset that has all the parameters and conditions of the character's AI in case it is not controlled by the player.
ActorComponents*	ActorComponent [0...*]	Additional components for the actor.

Table 1 - CharacterData variables

Character Atributtes

CharacterAtributtes is a DataAsset containing all the character state attributes. The parameters of this variable are categorized into three types:

- Basic Variables
  - Health, endurance.
- Motion variables
  - Walking speed, running speed, friction, etc.
- Healing variables
  - Healing speed, maximum amount of healed health, etc.

Table 2 below shows all the variables broken down, together with their type and description.

Variable	Type	Description
Health	float	Float that determines the character's health.
Resistance	float	Float that determines the character's resistance to damage.
WalkingSpeed	float	Float that determines the walking speed of the CharacterMovement.
RunningSpeed	float	Float that determines the running speed of the CharacterMovement.
GravityScale	float	Float that determines how much impact gravity has on the character.
MaxAcceleration	float	Float that determines the maximum acceleration of the CharacterMovement.
GroundFriction	float	Float that determines the friction with other static elements.
AutoHeal	bool	Bool that defines if the character receives a self-healing.
AutoHealMaxHealthPercentage	float	Float that determines the maximum percentage of health at which the character can heal.
AutoHealWaitTime	float	Float that determines the waiting time from when the character received the last damage until it starts healing.
AutoHealSpeed	float	Float that determines the speed at which health recovers.

Table 2 - CharacterAtributtes variables

Weapon Data

WeaponData is a DataAsset that contains all the parameters of a weapon. The parameters of this variable are categorized into four types:

- Basic variables
  - Type of weapon (firearm or melee).
- Shooting variables
  - Magazine size, cadence, etc.
- Appearance variables
  - Mesh, attack particles, etc.
- Positioning variables
  - Socket name, transform offset.

Table 3 shows all the variables broken down, together with their type and description.

Variable	Type	Description
WeaponType	EWeaponType	Enumerator that determines the type of weapon.
BulletData	BulletData*	DataAsset that determines the characteristics of the instantiated bullet.
BulletsToShoot	int	Integer that determines the number of bullets per shot (1 in the case of a pistol, several in the case of a shotgun).
MagazineSize	int	Integer that determines the size of a magazine.
Cadency	float	Float that determines the rate of fire of the weapon.
Accurancy	float	Float that determines the dispersion of each shot.
WeaponGripType	EWeaponGripType	Enumerator that determines the type of grip.
Mesh	SkeletalMeshComponent*	SkeletalMesh of the weapon.
AttackMontages	AnimMontage* [0...*]	Montages of the attacks.
AttackParticles	AnimParticles* [0...*]	Particles of the attacks.
ReloadMontages	AnimMontages* [0...*]	Montages of reload.
MainHand	EHandType	Enumerator that determines the hand with which the weapon is held.
HandSocketName	FString	Name of the socket holding the weapon.
TransformOffset	FTransform	Transform of the weapon with respect to the socket.

Table 3 - WeaponData variables

Table 3 shows three enumerator type variables. The possible values in them are as follows.

EWeaponType
Melee
Firearm

EWeaponGripType
None
One-Handed
Two-Handed
Dual-Wielding

EHandType
Left
Right
Both

Bullet Data

BulletData is a data structure that contains all the parameters of a shot coming from a gun. The parameters of this variable are categorized into three types:

- Basic variables
  - Bullet type, damage.
- Stun variables
  - Stun time, amount of pushback, etc.

Table 4 below shows all the variables broken down, together with their type and description.

Variable	Type	Description
BulletType	EBulletType	Enumerator that determines the type of bullet/shot.
Damage	float	Float that determines the damage inflicted by the weapon to other characters.
StunTime	float	Float that determines how long the actor that has been shot is stunned.
PushbackQuantity	float	Float that determines the amount of thrust the shot has against the actor hit.
PushbackTime	float	Float that determines the amount of time that the thrust of the shot lasts against the hit actor.

Table 4 - BulletData variables

Table 4 shows an enumerator type variable. The possible values are listed below.

EWeaponType
LineTrace
BulletInstance

For bullets that are not raycasted, a Bullet class will be required, which receives as parameter the DataAsset of BulletData type to initialize the corresponding variables and will also require an OnHit delegate, so that when it detects an actor that implements the HICharacterInterface interface, it calls the TakeDamage function.

Figure 4 shows the delegate and the call to the TakeDamage function of the impacted actor.

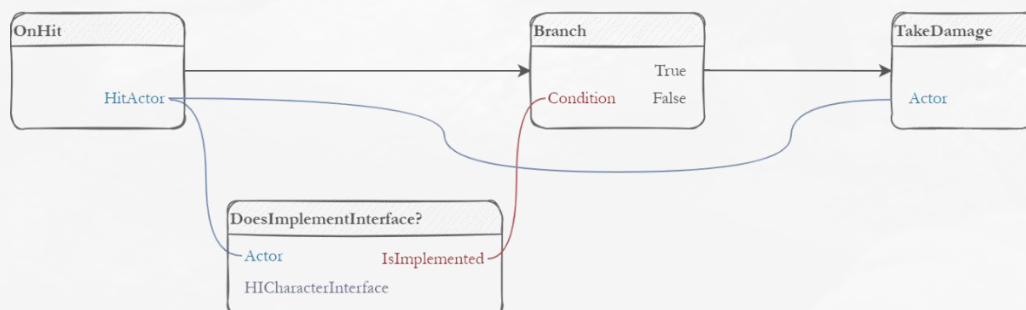


Figure 4 - Bullet OnHit delegate

Attack Particles

AttackParticles is a data structure containing all the parameters corresponding to the particle system of an attack. The parameters of this variable are categorized into two types:

- Basic variables
  - Particle system
- Positioning variables
  - Name of the socket to which the particle is anchored, transform with respect to the area in which it is located, etc.

Table 5 below shows all the variables broken down, together with their type and description.

Variable	Type	Description
ParticleEmitter	NiagaraSystem	Niagara particle system to instantiate.
SocketName	FString	Name of the socket to which the particle is attached.
AttachToSocket	bool	Bool that determines if the particle is anchored to the socket.
RelativeTransform	FTransform	Transform with respect to the zone in which the particle system is instantiated.

Table 5 - AttackParticles variables

When calling the attack function, the system can be called to one of the particle systems of the AttackParticles array.

Camera Data

CameraData is a DataAsset that contains all the parameters corresponding to the camera. This DataAsset will only be needed in case the character is controlled by the player. The parameters of this variable are categorized into two types:

- Basic variables
  - Camera type, camera configuration.
- Positioning and movement variables
  - Use of a SpringArm, length.

Variable	Type	Description
CameraType	ECameraType	Type of camera to be used by the actor.
NormalCameraData	NormalCameraData*	DataAsset with all the settings of the normal camera if used.
CinematicCameraData	CinematicCameraData*	DataAsset with all the settings of the cinema camera in case it is the one used.
UseSpringArm	bool	Bool that determines if the camera is anchored to a SpringArm.
SpringArmData	SpringArmData*	DataAsset with all SpringArm settings if used.

Table 6 - CameraData variables

Table 6 shows three DataAssets that will not have their own section, since their implementation only requires copying and implementing the variables of the component to

which they refer. On the other hand, an enumerator type variable is shown. The possible values are shown below.

ECameraType
NormalCamera
CinematicCamera

AI Data

AIData is a DataAsset that contains all the parameters regarding the actor's AI. The parameters of this variable are categorized into two types:

- Basic variables
  - AI controller, reference to Blackboard and BehaviorTree.
- Perception variables
  - Parameters of the AIPerception component.

Table 7 below shows all the variables broken down, together with their type and description.

Variable	Type	Description
AIController	AIController*	AI Controller.
Blackboard	Blackboard*	AI Blackboard.
BehaviorTree	BehaviorTree*	BehaviorTree with AI behavior and decisions.
UseAIPerception	bool	Bool that determines if the Unreal Engine AIPerception component is used.
AIPerceptionData	AIPerceptionData*	DataAsset with all the AIPerception settings if used.

Table 7 - AIData variables

Table 7 shows a DataAsset that will not have its own section, since its implementation only requires copying and implementing the variables of the component to which it refers.

Actor Implementation

This section will detail the implementation of the character creation system in Unreal Engine.

Construction Script

In the ConstructionScript is where the parameters of the character will be initialized in order to preview it in the editor.

First, the row of the DataTable with the configuration of the character will be read and the data will be stored in a structure (private variable). This structure is where all the variables will be obtained and updated. From this point on, the reference to the DataTable row will not be necessary.

Secondly, the character will be configured. For this purpose, this configuration will be divided into 5 different functions:

- *SetupController*
  - Configures the PlayerController settings. If used, the camera settings will be configured.
- *SetupWeaponry*
  - Configures the character's arsenal, the weapon equipped and will initialize each weapon.
- *SetupMesh*
  - Configure all SkeletalMesh parameters and animations.
- *SetupAI*
  - Configure the AI in case you use one.
- *SetupCustomComponents*
  - Add the extra components.

Figure 5 shows the scheme to be followed in Blueprints format.

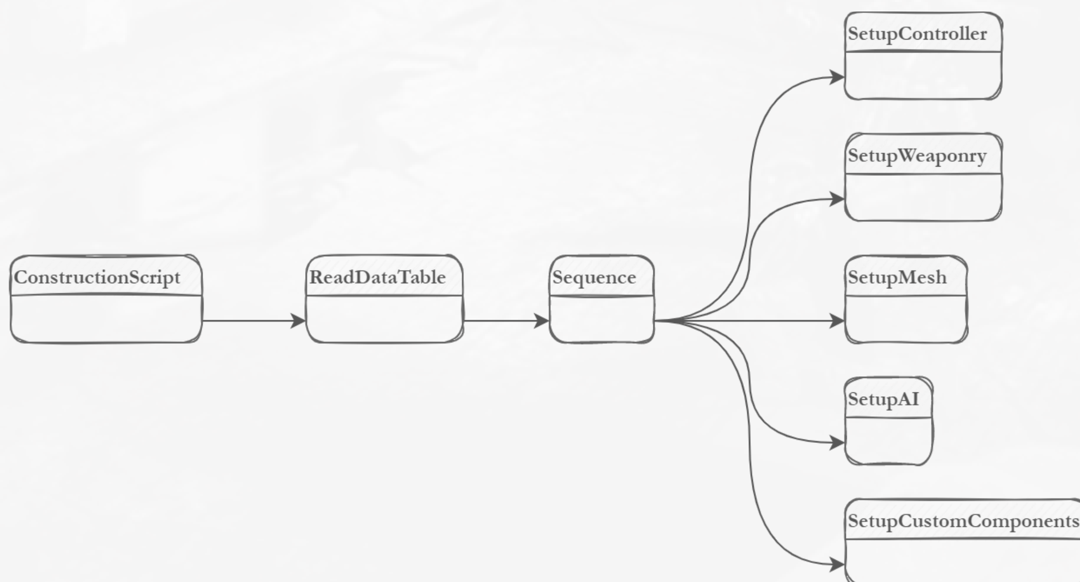


Figure 5 - ConstructionScript, Blueprint diagram

Custom Events

When creating a character, it is normal that the need arises for the design to create specific actions for that character. A dash, a custom jump, a specific type of attack. All these actions will be managed by creating a C++ UObject called CustomEvent, which has an implementable function in Blueprints called ExecuteCustomEvent.

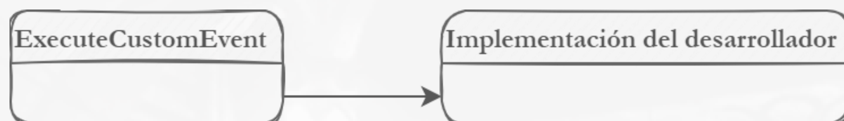


Figure 6 - ExecuteCustomEvent implementation

All the logic of the custom functions will be located in the ExecuteCustomEvent function. On the other hand, this custom event must be called from somewhere. To do this, a macro called CallCustomEvent will be created to receive a variable of type CustomEventClass which will be the event to be called. In addition, it will have two outputs: Execute (the basic one) and OnFinished (which is called at the end of the event execution).

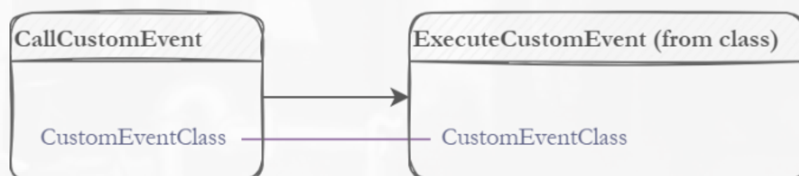


Figure 7 - CallCustomEvent implementation

This static function will be called in the PlayerController in the case of being an actor controlled by the player.

On the other hand, for AIs, a Task named ExecuteAICustomEvent will be created, which will receive a variable of type CustomEventClass and will call the static function CallCustomEvent, introducing the variable received by the Task. Once this is done, the designers, when creating a character, will prototype in Blueprints by means of the CustomEvents the concrete actions of the character and will call them in the PlayerController or the BehaviorTree.

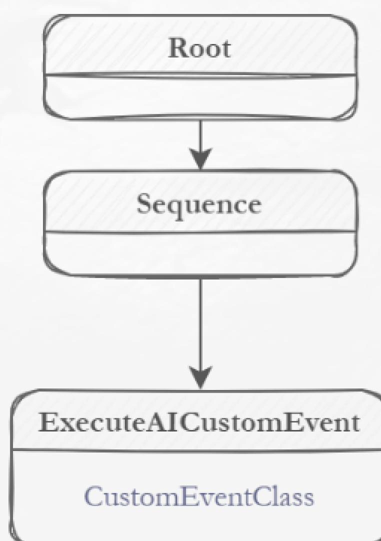


Figure 8 - ExecutieAICustomEvent, behavior tree example

However, designers are free to implement their own Tasks to be called in BehaviorTrees, as long as the structure of Figure 9 is followed.

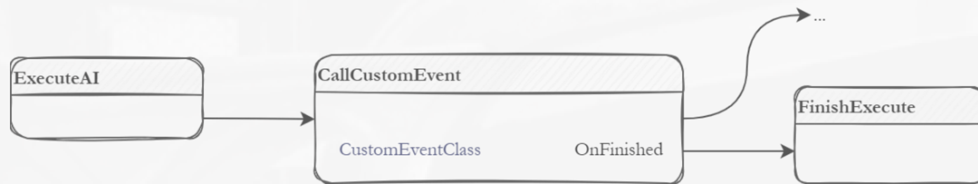


Figure 9 - Custom Task implementation

ExecuteCustomEvent

Custom events have a series of parameters to be taken into account, which encompass the possible types of actions. These attributes are referenced in Table 8 below.

Variable	Type	Description
CasterReference	Actor*	Reference to the actor who called the event.
Duration	float	Duration of the event.
AppliedEffectQuantity	float	Amount of effect applied: damage, healing, etc.

Table 8 - ExecuteCustomEvent variables

CallCustomEvent

The macro named CallCustomEvent must replicate the same parameters as the ExecuteCustomEvent event, plus the reference to the class of the event to be invoked. These attributes are referenced in Table 9 below.

Variable	Type	Description
CustomEvent	CustomEventClass*	Reference to the class of the event to be invoked.
CasterReference	Actor*	Reference to the actor that called the event.
Duration	float	Duration of the event.
AppliedEffectQuantity	float	Amount of the applied effect: damage, healing...

Table 9 - CallCustomEvent variables

CallAICustomEvent

The Task of a BehaviorTree that calls a CustomEvent must replicate the same parameters as the ExecuteCustomEvent event, plus the reference to the class of the event to be invoked. These attributes are referenced in Table 10 below. The reference to the actor that calls the event will be referenced within the Blueprint itself.

Variable	Type	Description
CustomEvent	CustomEventClass*	Reference to the class of the event to be invoked.
Duration	float	Duration of the event.
AppliedEffectQuantity	float	Amount of the applied effect: damage, healing...

Table 10 - CallAICustomEvent variables

Actors with Player Controller

The actors that have a PlayerController, will implement their actions in response to the different inputs from the player. From the PlayerController Blueprint, when an input is detected, a call will be made to the macro CallCustomEvent and the desired action will be introduced (once it is created).

Actors without Player Controller

As mentioned in the CustomEvents section, actors that do not have a PlayerController will call their respective events in their BehaviorTree.

There are two ways in which a developer can call their CustomEvent in the BehaviorTree.

1. Calling the Task named CallAICustomEvent.
2. Creating a custom Task that calls the macro CallCustomEvent.

In this way, there is a great deal of freedom for the development of behaviors by both programming and design.

Pipeline

When working with this system, a series of 6 steps must be followed, as long as a character is being created **from scratch**.

The steps to be followed by the team once the art part of the character is finished are:

1. Import Meshes
    a. The model made by the art department must be imported.
    b. Its skeleton and animations must be included.

2. Generate AnimBlueprints and develop Animation Montages
    a. As a result of the model and its animations, the AnimBlueprint of the character will be generated, where the transitions to its different states will be managed.
    b. Montages of animations such as attacks, jumps, etc. will be performed.

3. Create a DataAsset that inherits from CharacterData
    a. A DataAsset that inherits from CharacterData will be created.

4. Implementing new CustomEvents and behaviors
    a. It will not always be necessary to create a new CustomEvent.
    b. Depending on whether it is a controlled character or not, the actions will be implemented in the BehaviorTree or in the PlayerController.

5. Adjusting the character's DataAsset values
    a. The value of the parameters mentioned in the Variables section will be configured.

6. Add DataAsset to the character
    a. The character settings will be added to the desired actor.

Depending on the status of the character, it may not be necessary to complete all of the above steps. For example, if the character is in an advanced state, but an animation has been modified, only steps 1 and 5 will be necessary.

Figure 10 below shows a schematic of the working pipeline for the use of this system and the creation of characters.

PIPELINE

Figure 10 - Pipeline

Annexes and References

Simulation of the system using Data Tables.

Enemy detection systems, Werewolf attributes and Encounters.

Where is each Combat/Systems attribute?