Blueprints Compendium

Volume 2

By Unreal Dev Grant Winner:
Marcos Romero
romeroblueprints.blogspot.com

About this document:

Blueprints is a visual scripting language that was created by Epic Games for Unreal Engine 4.

The first volume of Blueprints Compendium was focused on presenting some key blueprint nodes with examples.

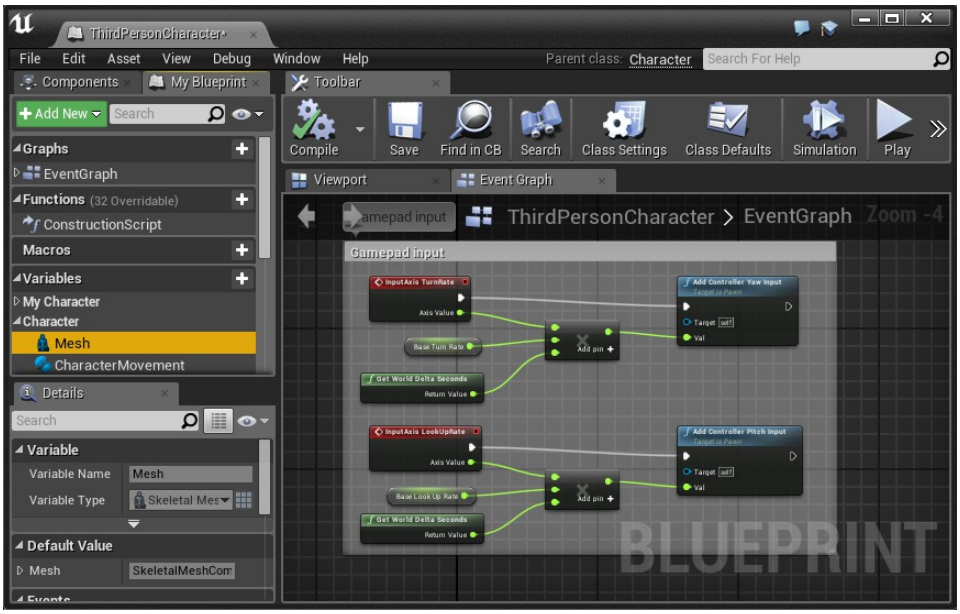
This second volume features 25 topics related to Blueprints that help to explore the true potential of the Blueprints.

Table of Contents:

Blueprint Editor

A game developed in Unreal Engine contains many Actors interacting with each other, for example players, enemies and items that can be collected are Actors. In Unreal Engine 4 the blueprint editor can be used to create new Actors for a game.

To open the blueprint editor you can double-click on a Blueprint or right-click and choose "Edit...". The blueprints editor has several tabs and can be used in various ways. The image below shows the blueprint "ThirdPersonCharacter" of the "Third Person" template, opened in the editor with the tabs "MyBlueprint" and "EventGraph" visible.



The "My Blueprint" tab shows the variables, macros, functions and graphs of the Blueprint. Events and Actions are defined in the "EventGraph" tab that determine the Blueprint behavior within the game.

In the "Components" tab may be added various types of components that will be part of the current Blueprint. As an example of Components we have mesh, light, sound, geometric shapes used in collision tests, and so on.

The "Viewport" tab contains the visual representation of the components that are part of this Blueprint. The "Details" tab displays the properties of the currently selected element so that they can be edited.

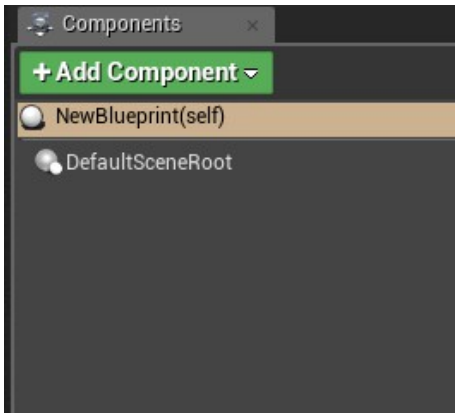
The toolbar which is on the top of the editor has a few essential buttons for editing Blueprints:

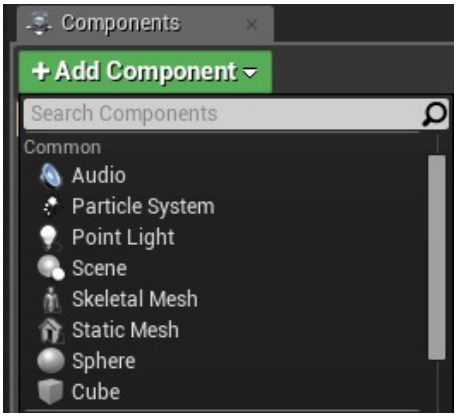
- Compile: It is necessary to "Compile" the Blueprint to validate the modifications.
- Save: Don't forget to save the Blueprint frequently.
- Class Settings: Blueprint properties such as description, category and "parent class".
- Class Defaults: It allows modification of the initial values of the Blueprint variables.

Components

Components are ready to use objects that can be used inside the Blueprints. Several features can be included in a Blueprint only using components.

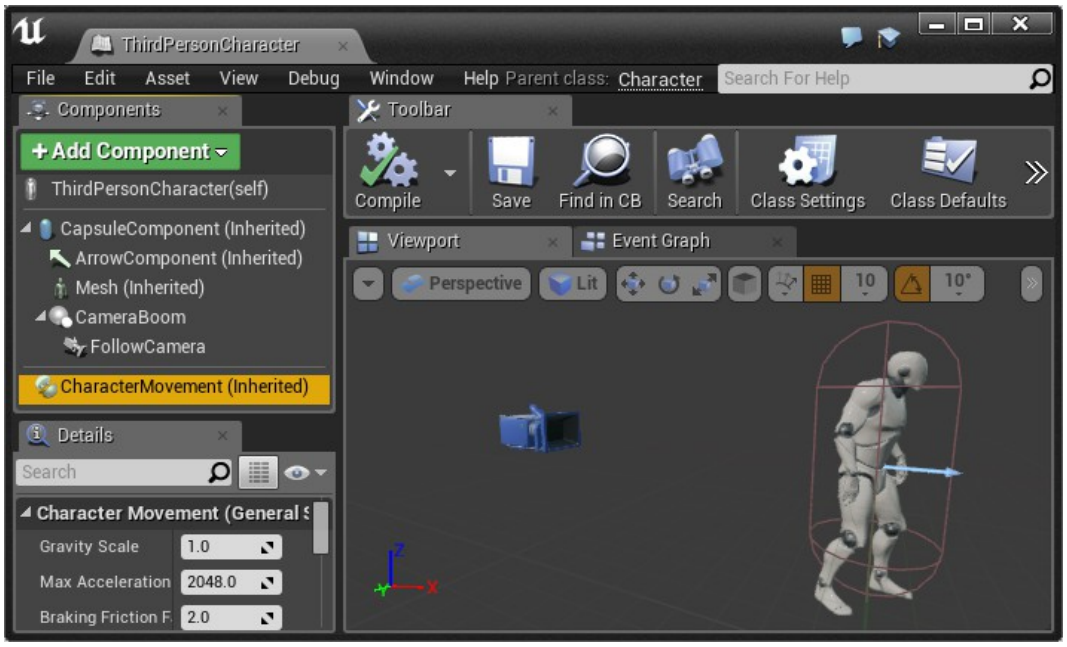
To add components use the "Components" tab of the Blueprint editor. The left image shows the "Components" tab of a new blueprint. The right image shows some components options that are displayed when the "Add Component" button is pressed.





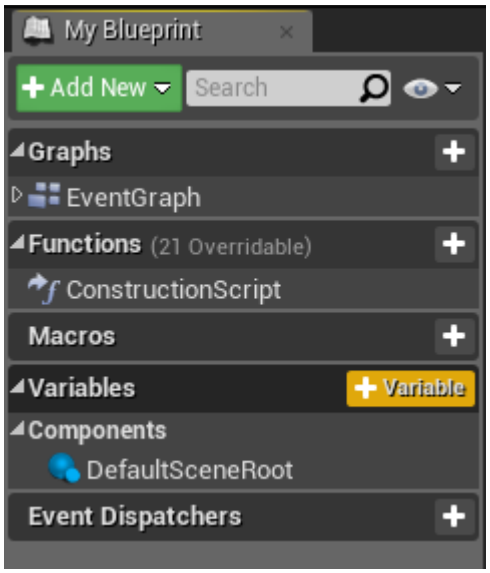
The properties of a component can be edited on the "Details" tab and the visual representation of the components can be seen on the "Viewport" tab.

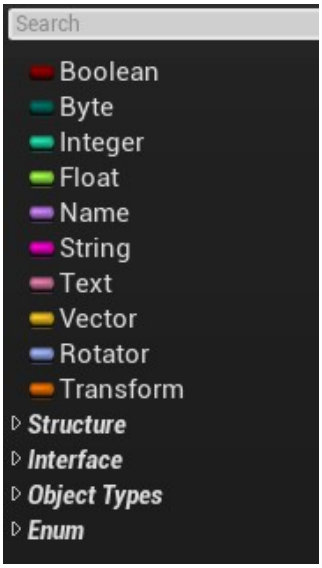
The image below shows the components that are part of the Blueprint "ThirdPersonCharacter" of the "Third Person" template. The components that have (*Inherited*) next to the name were inherited from the "Character" class. The *CapsuleComponent* is used for collision testing; The *Mesh* component is of the SkeletalMesh type that visually represents the character; the *FollowCamera* component is the camera that will be used to view the game; The *CharacterMovement* component contains various properties that are used to define the movement.



Variables

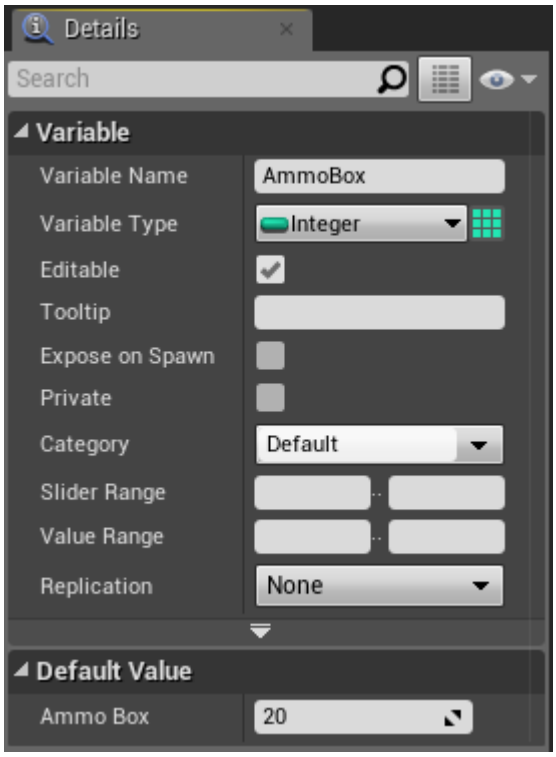
Variables are used to store values of attributes of Blueprints that can be modified during the execution of the game. To create variables use the "MyBlueprint" tab and click the button with the "+" symbol next to the "Variables" category. The images below show the "MyBlueprint" tab and the variable types that exist in Unreal Engine.





The simplest types of variables are:

- **Boolean**: A boolean variable can only hold the values "true" or "false".
- **Integer**: Type of variable used to store integer values.
- **Float**: Type of variable used to store fractional values.
- **String/Text**: These types are used to store text. Prefer the **Text** type since it supports internationalization.

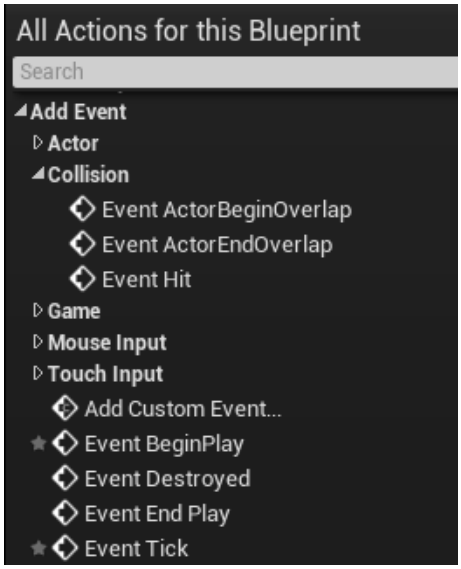


In the image an integer variable was created with the name "AmmoBox". This variable can be used in a Blueprint that represents an ammo box. Its value indicates the amount of ammunition that the player will receive.

The "Editable" option selected means that the variable can be changed in the level editor. The default value of this variable is 20, but the level designer can easily change the amount of ammunition of each box in the level.

Events

Unreal Engine uses *Events* to send messages to Actors, which are objects that can be added to the level. The scripts that we created in Blueprints are Actions that are executed in response to events that are generated during the game. To see the events available, right click on the "EventGraph" and expand the "Add Event" category. In this category there are subcategories that group other related events as shown in the image below:



Below is a brief description of the main events:

- **Collision Events**: They are triggered when two Actors collide or overlap.
- **Mouse Events**: They are triggered by clicking and releasing the mouse button or when the mouse cursor is over the Actor.
- **Touch Events**: They are triggered by Touch Screens.
- **Custom Events**: They are new events that can be created in the blueprints.
- **Event Begin Play**: It is triggered when the game starts for the Actor.
- **Event Destroyed**: It is triggered when the Actor is about to be removed from the game.
- **Event Tick**: It is called every frame of the game. For example, if a game runs at 60 frames per second, this event will be called 60 times in a second.

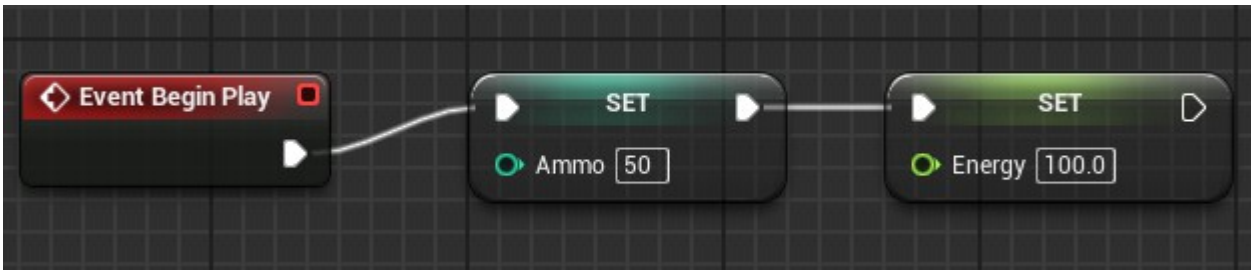
You can add many different events in a "EventGraph" but each event can only be added once in the "EventGraph". The image below shows some examples of *Events* that have been added to the "EventGraph" but without any *Action* associated with them.

Actions

Actions are used to define the behavior of a Blueprint. Actions always start from an Event. The main types of Actions are changes in the variables of the Blueprint or function calls that modify the state of Actors.

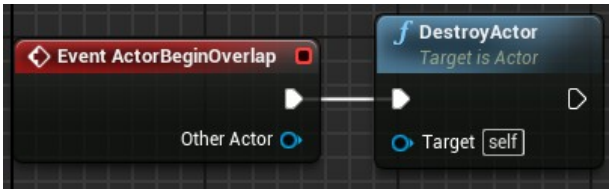
For example, imagine that a Blueprint has an integer variable called "Ammo". To initialize this variable with the value "50" when the game starts use the Event "Begin Play". Drag the "Ammo" variable to the EventGraph using the "Set" option:

Each event can be added only once in the EventGraph. To use more than one Action for a specific Event it's necessary to put the Actions in sequence. The image below shows the initialization of two variables.



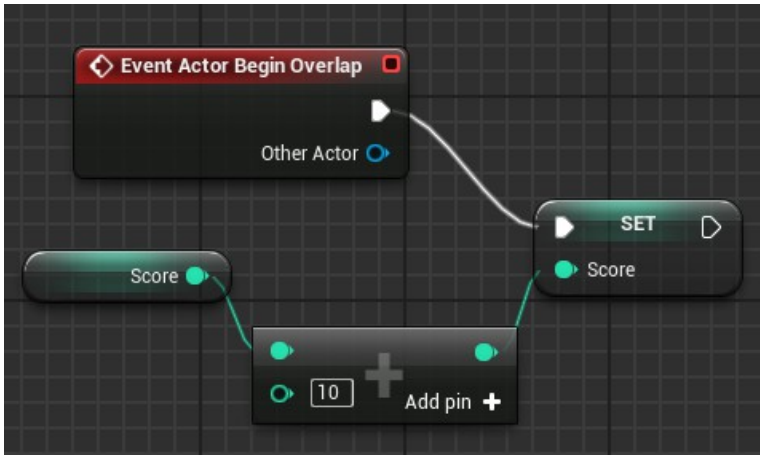
The functions allow values to be passed through the use of parameters. Most functions has a parameter called "Target". This parameter indicates the Actor to be modified by the function. The default value for this parameter is "**self**" which is a special reference to the Actor that owns the script being executed.

The images below show different ways to use the "Target" parameter of the "DestroyActor" function when an overlap event happens. In the example that uses "self", the Actor that has the script will be destroyed. To destroy the other actor who was part of the collision use the reference "Other Actor" of the event. In the last example, the actor who will be destroyed is the one referenced by the variable "Item BP."



Arithmetic Operators and Expressions

The arithmetic operators (+, -, *, /) can be used to create mathematical expressions in Blueprints. The image below shows a simple expression that add 10 points to the current player's score.

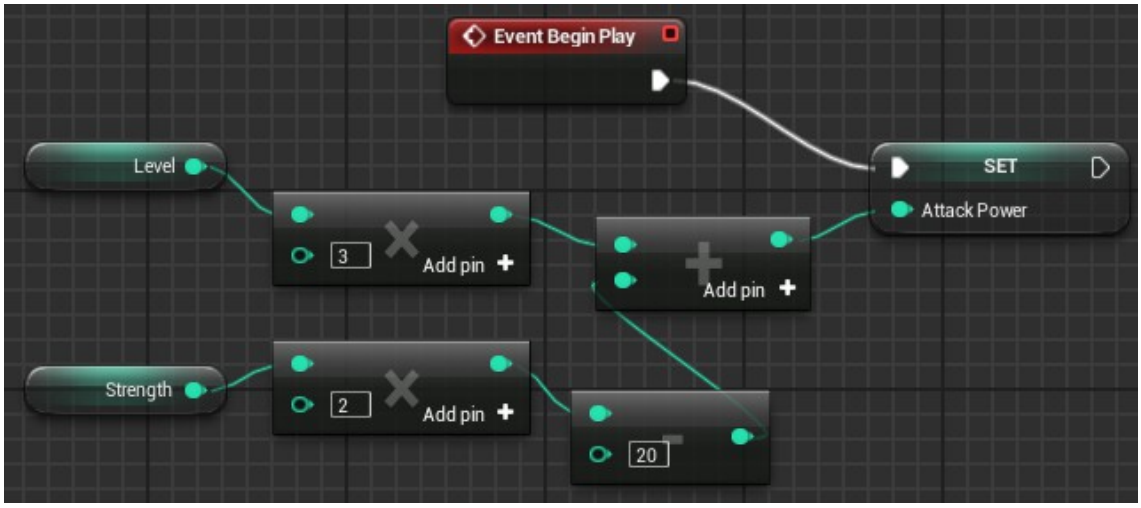


The "+" operator receives two input values on the left and gives the operation result on the right. To use more than two input values just click on the "Add pin" option. The input values can be entered directly or can be obtained from variables.

As another example will be created an expression to calculate the "Attack Power" of a character. The expression uses the variables "Level" and "Strength" of the character. The expression is as follows:

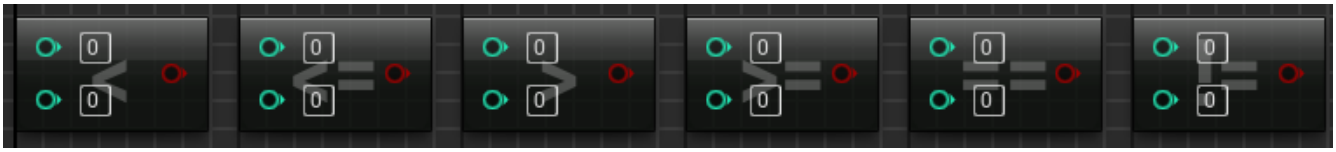
Attack Power = (Level x 3) + (Strength x 2 – 20)

To create an expression of this type in Blueprints is easier to identify the latest operations to be done and create the blueprint in the reverse order that resolves an expression, in other words, starting with the operation of lower precedence. In this example start with the "+" operator, whose input values are the results of the small expressions that are inside the parentheses.

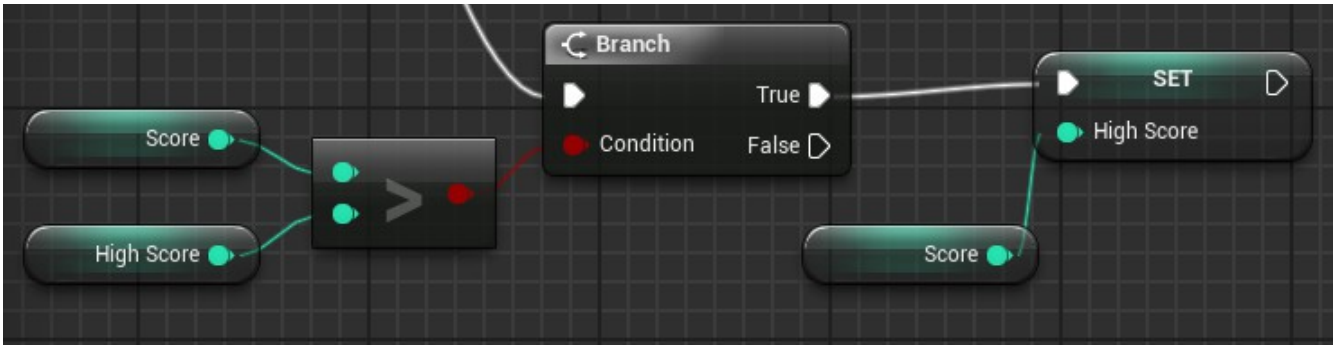


Relational and Logical Operators

The relational operators perform a comparison between two values and return a Boolean value (true or false) as a result of the comparison. The relational operators are as follows:



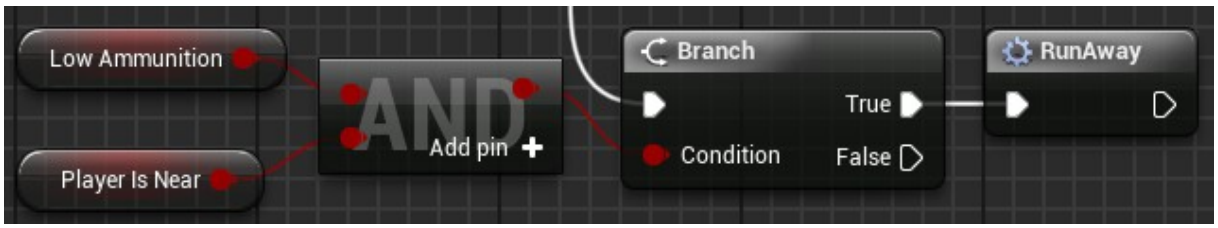
The example below shows the use of a relational operator and a "Branch". At the end of a game will be compared the current player's score (Score variable) with the highest game score (High Score variable). If the player's score is higher, the value of the "Score" variable will be stored in the "High Score" variable.



Logical operators perform an operation between boolean values and return a boolean value (true or false) as a result of the operation. The main logical operators are:

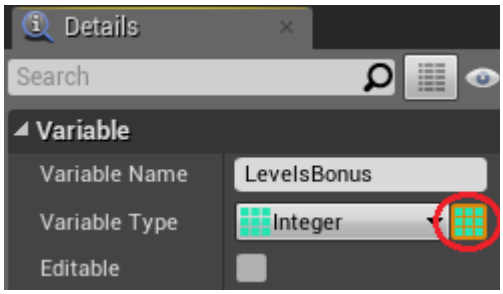
The logical OR operator returns "true" if any of the input values are "true". The logical AND operator returns "true" only if all inputs are "true". The logical NOT operator receives only one input value and the result will be the reverse value.

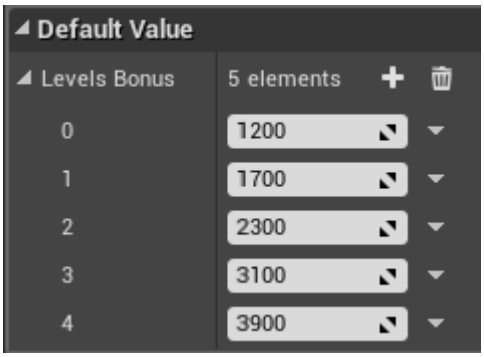
The following example simulates a simple decision of an Enemy in a game. If the Enemy is low on ammo (LowAmmunition variable) and the player is nearby (PlayerIsNear variable) then the Enemy decides to run away.



Arrays

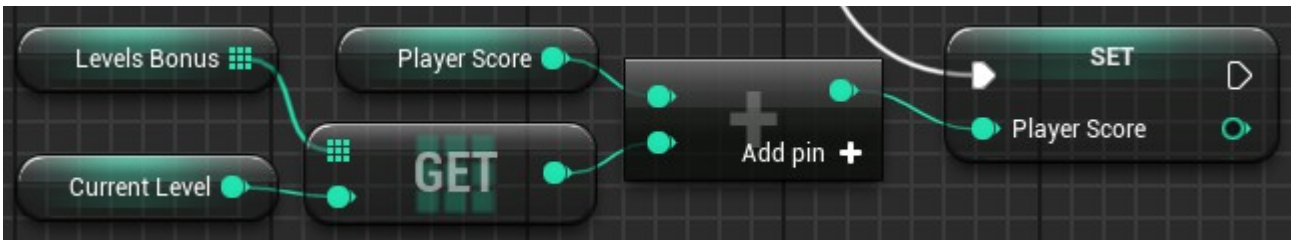
The use of Arrays allows grouping of variables of the same type. Creating an Array in Blueprints is very simple. Create a new variable and select the desired type. Beside the type of the variable there is an icon that must be clicked to make the variable an Array. After compiling the Blueprint it is possible to add in the Array elements with default values, as in the images below:





In this example was created the Array "LevelsBonus" that keeps the extra scores that the player will receive when completing the levels. Each element of the Array represents a level. Note that the first element of an Array has the index zero.

The actions below show how to add the extra score on the player's score. The integer variable "CurrentLevel" represents the current level of the game and is used as the Array index to get the extra score equivalent to the current level.

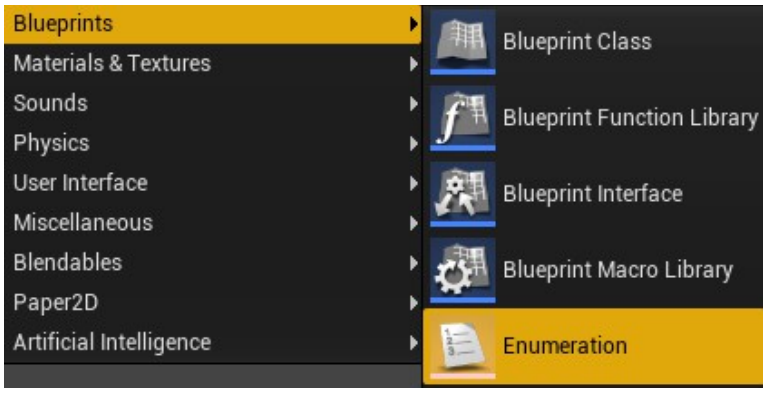


The main actions related to an Array are:

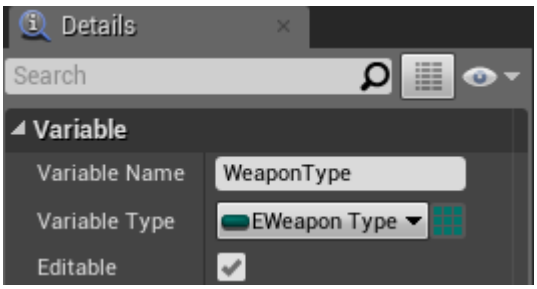
- Get: Returns the element that is in the position specified by the index used.
- Length: Returns the number of Array elements.
- Add: Add a new element at the end of the Array.
- Insert: Inserts a new element at the position specified by the index used.

Enumerations

An enumeration is a set of constants with meaningful names that specifies all possible values a variable can have. To create a new Enumeration click the "Add New" button and in the "Blueprints" submenu select "Enumeration". Double-click the enumeration that was created to edit it. Click the "New" button to add the names that are part of the enumeration.



Now create a new variable in a Blueprint using the type of the enumeration that was created. Check the "Editable" option so that the value of the variable can be chosen in the level editor. The images below show an enumeration example created to represent the type of weapon.

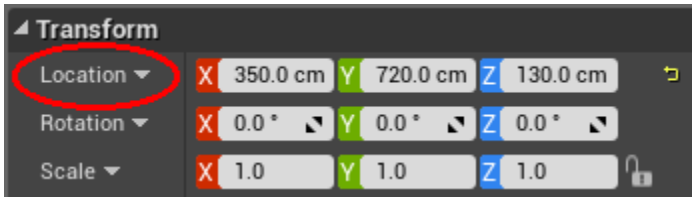


There is a type of "Switch" action that determines the flow of execution in accordance with the value of the enumeration. In the example below "Weapon Type" is a variable of the enumeration type and "Weapon Mesh" is a Static Mesh component. The Static Mesh will be set according to the type of weapon.

Vectors

A vector is a structure that contains the values X, Y and Z. The use of vectors simplifies many things in 3D game programming. Vectors can be used to represent position, velocity, acceleration, distance, direction...

An Actor has a vector called **Location** that represents its current position in the 3D world.

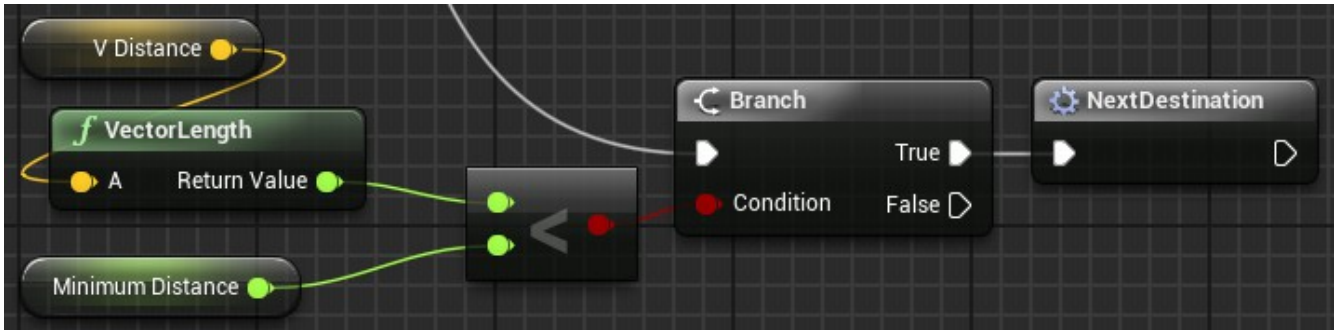


Blueprints has some utility functions for vectors. The two most used are:

- **Length**: The length of a vector can be used to represent the distance between two points.
- **Normalize**: Calculate the unit vector. The unit vector has a length equal to 1 and it is often used to indicate a direction.

In the example below an Actor is following a number of destination points. The vector "vDistance" is obtained from the subtraction between the current destination point and the current position of the Actor.

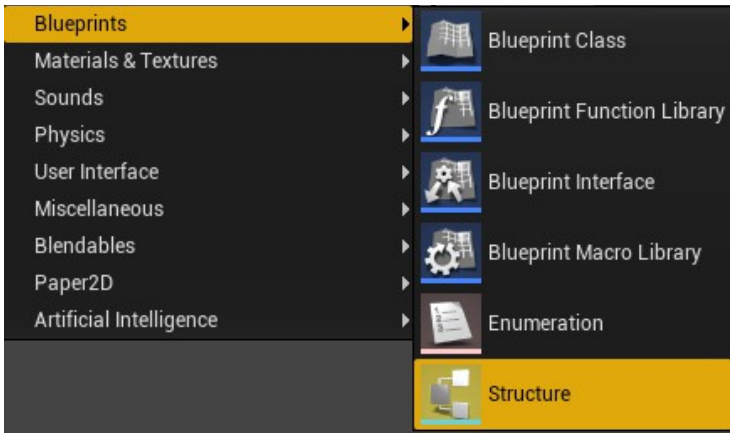
The length of the "vDistance" vector represents the distance remaining to the Actor to get to the current destination point. If this value is less than the value of the "MinimumDistance" variable then it means that the Actor arrived at the destination point and can now go to the next destination point.

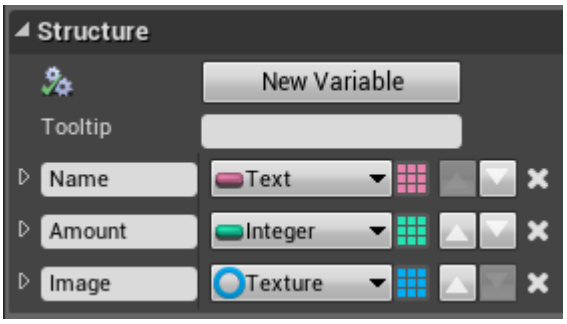


Structures

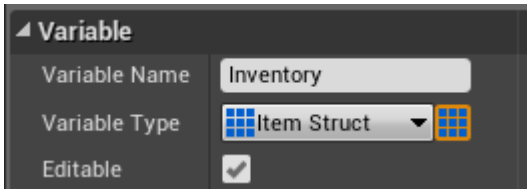
A structure can be used to gather in one place several related variables. The variables that belong to a structure can be of different types. You can also have structures that contain other structures and *arrays*.

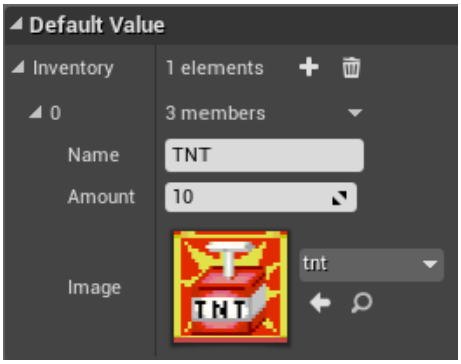
To create a new structure, click the "Add New" button and in the submenu "Blueprints" select "Structure". Double-click the structure that was created to edit it. Click the "New Variable" button to add variables to the structure.



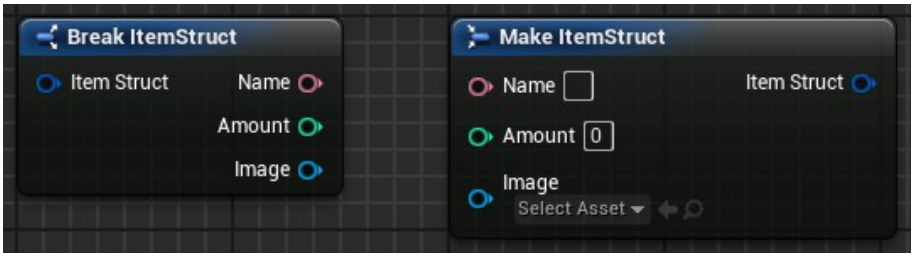


As an example will be created a variable to represent the Inventory of the player. Create a new variable in a Blueprint named "Inventory" using the the structure type that was created. Click the icon next to the type to make this an Array variable. Compile the Blueprint. In the "Default Value" of the variable you can add elements in the inventory. Each element contains the variables defined in the structure.





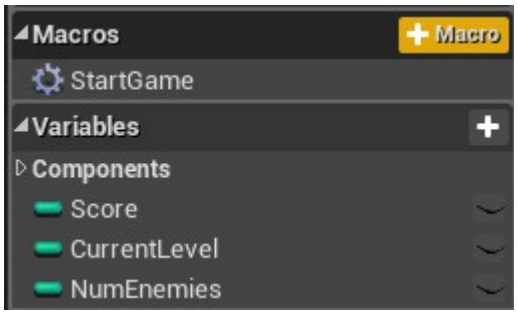
When a structure is created, the Actions "Break" and "Make" of this structure are added for use in the Blueprint. The Action "Break" receives a structure as input and separates its elements. The Action "Make" receives as input the separate elements and creates a new structure. The image below shows the Actions "Break" and "Make" of the structure "ItemStruct" that was created.

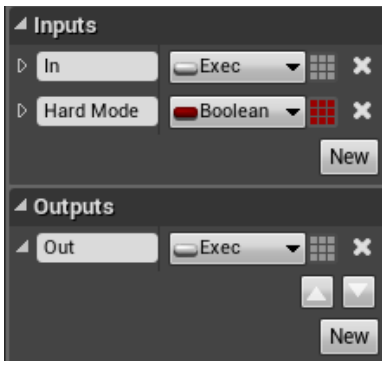


Macros

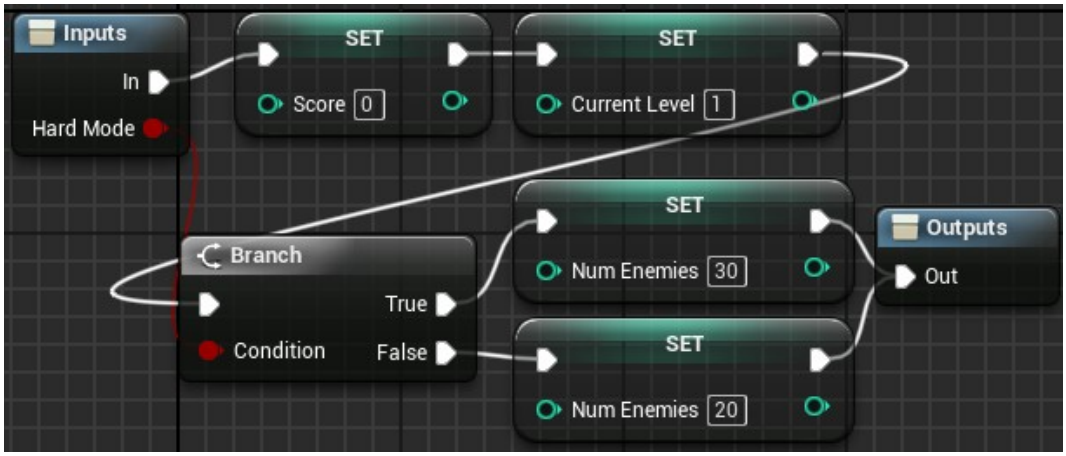
The use of a Macro allows the definition of a set of actions that can be called in other parts of the EventGraph. That way we avoid repeating this set of actions in the EventGraph simplifying its modification by being in one place.

To create Macros use the "MyBlueprint" tab and click the button with the "+" symbol next to the "Macros" category. The input and output parameters are specified in the "Details" tab. The type of parameter "Exec" is an Execution pin.

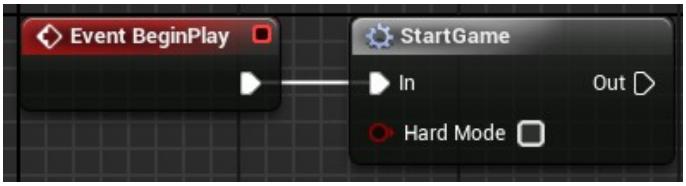




The image below shows a Macro called "StartGame". It is responsible for initializing the variable to a new match. It has a Boolean parameter called "Hard Mode" which increases the number of enemies if true.



To add the Macro to the EventGraph simply drag the name of the Macro and drop it in the EventGraph. When the Macro is executed, the actions that are in it will be executed. Because there are several ways to start a game, just use the Macro "StartGame" to initialize the variables instead of repeating these actions in different parts of the EventGraph.



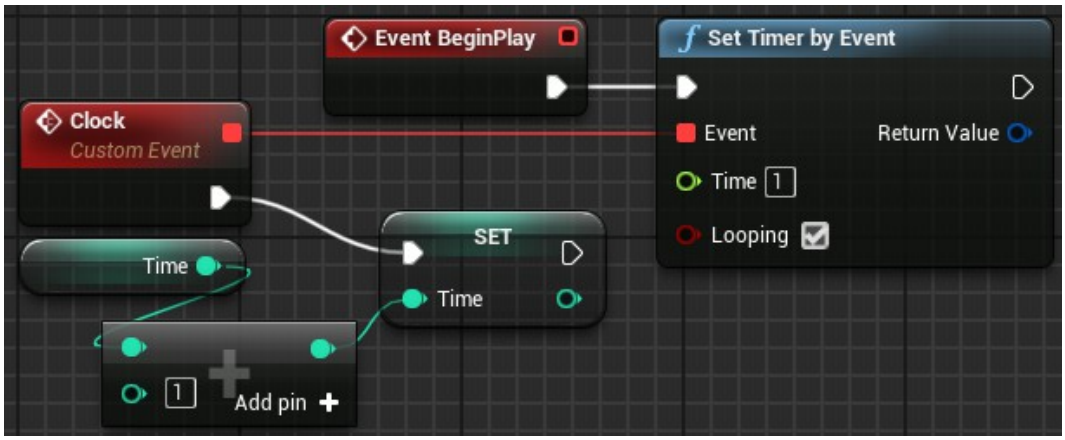
Custom Events

In addition to the events provided by the Unreal Engine, you can create new events that can be used in the Blueprint where the event was created or be called from other Blueprints.

To create a Custom Event, right-click on the EventGraph, expand the "AddEvent" category and select "Add Custom Event...". In the "Details" tab can be set the event name and the input parameters. Events do not have output parameters. The images below show an event called "AddScore" which receives a number of points that should be added to the current score.

Assuming that this event was created in a Blueprint called "BP_Player", the image below shows the event being called in another Blueprint that represents an item that gives points when it is collected. The "Cast To" action is used to ensure that was the "BP_Player" that collided with the item and also to get a reference of the type "BP_Player" to access the "AddScore" event.

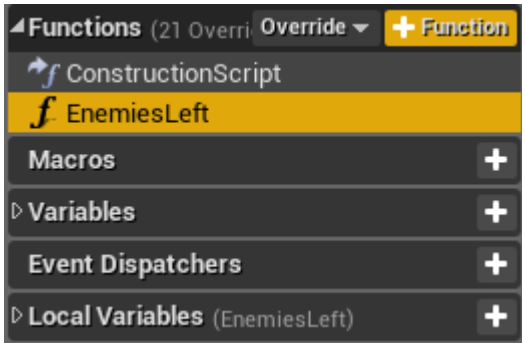
An event has a small red square in the right corner that is known as "**Delegate**". This is just a reference to the event. Some actions receive an event as a parameter and is using the Delegate that makes this possible. In the example below the Delegate of the event "Clock" was used in the action "Set Timer by Event", so the "Clock" event will be called every second.

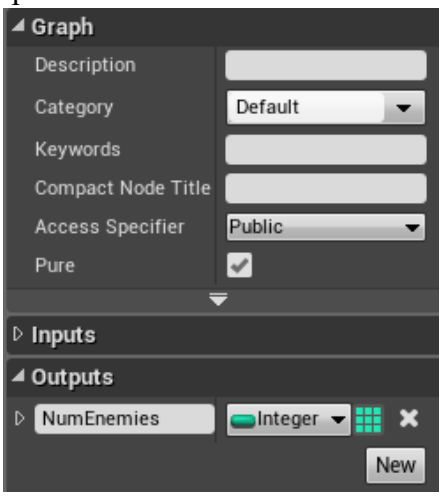


Functions

Just as Macros and Custom Events, Functions allow a set of actions that are executed in various parts of the Blueprint to be gathered in one place for easy organization and maintenance of the script. Functions can be called from other Blueprints and allow the use of output parameters. A disadvantage is that the Functions need to be completed immediately when executed, so in the Functions is not allowed the use of *Latent Actions* such as Delay and Timeline.

The following example shows the creation of a function named "EnemiesLeft" that has an output parameter called "NumEnemies". This function is marked "Pure", so it will not have execution pins. Pure functions should not modify the variables of its Blueprint.

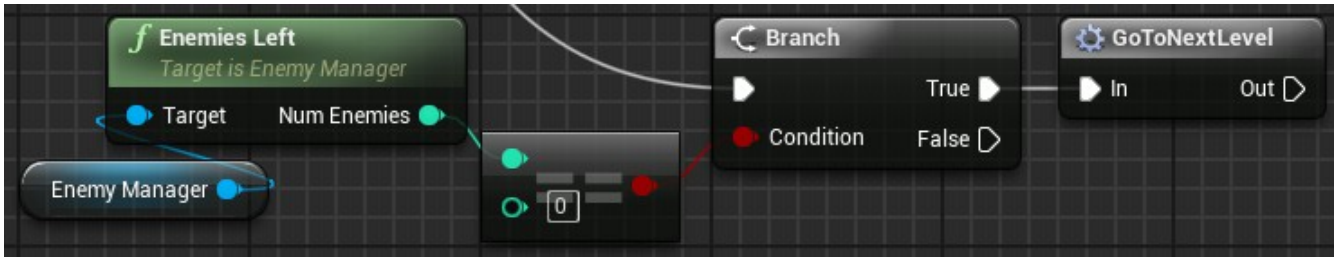




The functions allow the use of local variables that are only visible inside the function. They are very useful to assist in complex functions and do not mix with the other variables of the Blueprint.

The "EnemiesLeft" function was created on a Blueprint named "Enemy Manager" that is responsible for managing the enemies of the game. It will return the number of enemies that still exists based on the enemy class that is in the variable "Current Enemy Class".

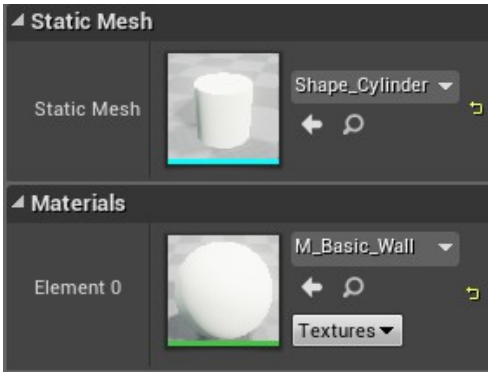
This image shows the function being called from another Blueprint.



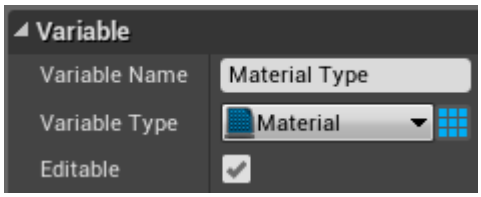
Construction Script

"Construction Script" is a special function that all Blueprints have that will be performed when the Blueprint is added to the level and when there is any change in their properties. The "Construction Script" has a separate tab where the actions to be performed can be placed.

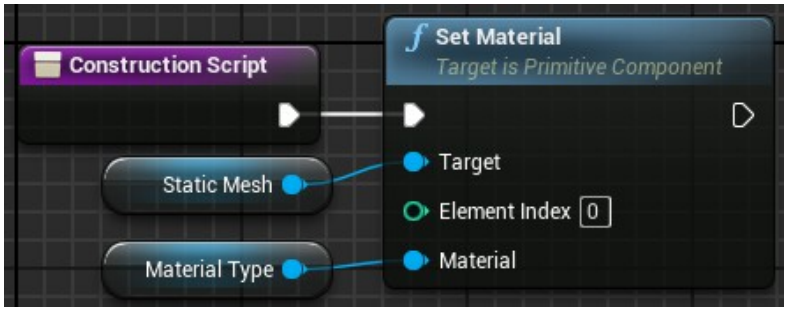
As an example, create a new blueprint, add a component of the type "Static Mesh" and choose the "Shape_Cylinder" as "Static Mesh". See the image below that the "Static Mesh" already has a default material. The "Construction Script" will be used to allow the material to be set in the level editing and independently for each object of this blueprint.

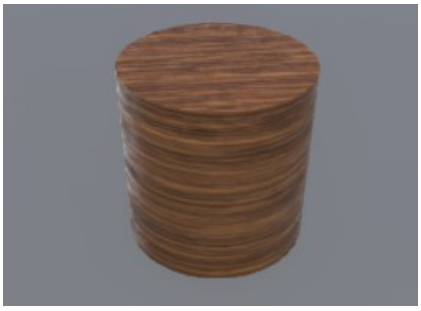


Create a variable named "Material Type" using the type "Material", select the "Editable" option. This will allow the value of this variable to be changed on an object that has been placed in the level.



The "Construction Script" uses the "Set material" function to define the type of material of the "Static Mesh" according to the material selected in the variable "Material Type". Whenever the "Material Type" is modified, the "Construction Script" runs again updating the object with the new material.

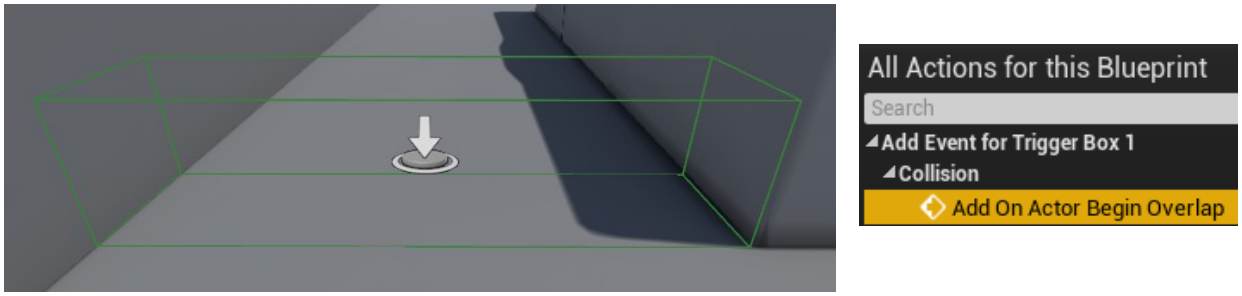




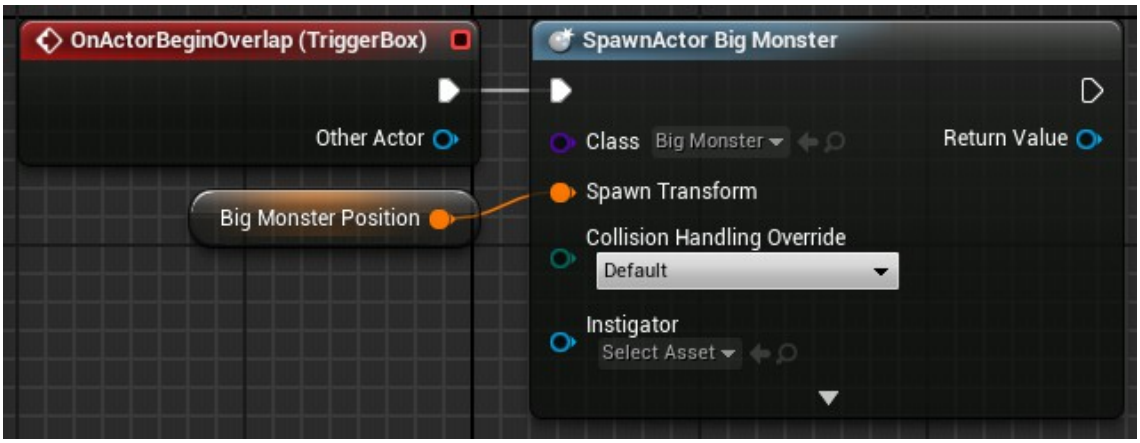
Level Blueprints

Unreal Engine 4 has a special type of Blueprint known as "Level Blueprints". Each game level has its "Level Blueprint". To open it click the "Blueprint" button on the top of the editor and choose the "Open Level Blueprint" option. A common use of "Level Blueprints" is to define actions to be performed when the player activate Triggers that are placed on the Level. To add a Trigger right click somewhere in the level and choose "Place Actor → Trigger → Box Trigger".

As an example will be placed a "Box Trigger" on a path of the game. The Box Trigger is invisible during the game. When the player touches the "Box Trigger" will be created an Actor of the "Big Monster" type in a near location to face the player. To add an event related to the "Trigger Box" that is on the level, select it and then open the "Level Blueprint". Right click on the "EventGraph" and choose the option shown in the image below.

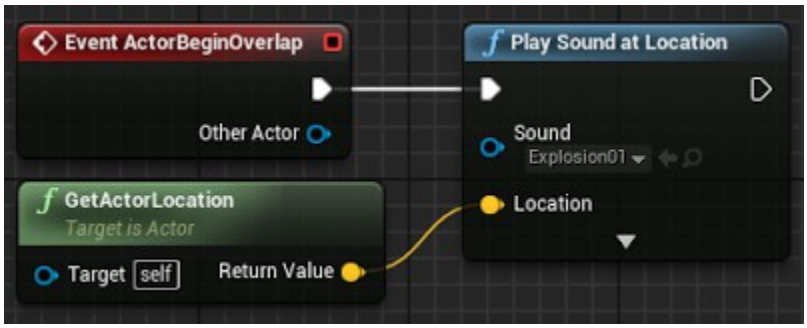


This is the script of the Level Blueprint:



Sounds

There is a simple way to play a sound in Blueprints. Just use the action "Play Sound at Location":

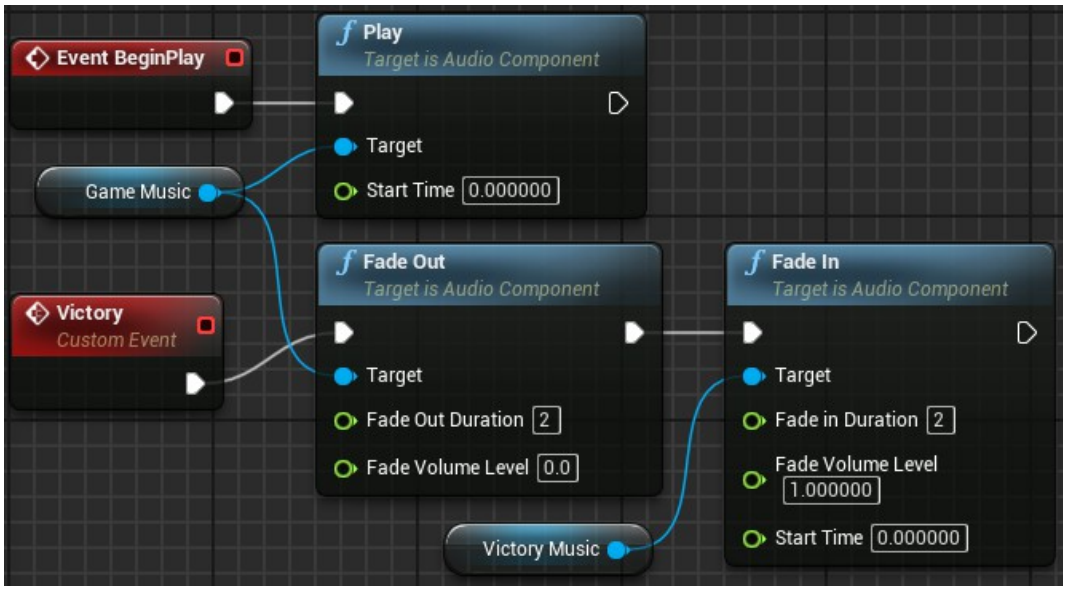


In this action choose in the combobox the sound to play. In parameter "Location" can be used the position of the Actor related to the sound. Imagine that the above example is a Blueprint of some type of explosive. When the player touches the explosive, the sound is played in the location where the Blueprint is.

The best way to have control over sounds in the Blueprints is using "**Audio Component**". As an example, imagine that there are two songs. One is the music that plays during the game and the other is a victory music that plays when the goal of the game is reached.

To add a song click on the "Add Component" button of a Blueprint and choose "Audio". Create two "Audio Component" with the name "GameMusic" and "VictoryMusic". For each of them, in the "Details" tab, select in the combobox "sound" the song that will be used and uncheck the property "AutoActivate" so that the song does not start playing automatically.

A Custom Event was created with the name "Victory" that must be called when the goal of the game is reached. The actions "Fade Out" and "Fade In" were used to gradually change the music for 2 seconds. The image below shows the transition between the songs:

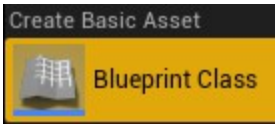


Object-oriented programming

Since its first version, the Unreal Engine was based on the principles of object-oriented programming. An object contains data that is represented by its variables and also has behaviors that are represented by Actions. Several objects interact to accomplish the goal of a program.

Classes x Objects:

A class is the definition of data and behavior that will be used by a particular type of object. A Blueprint represents a class:



An object is an **instance** of a class. For example, there is only one "Actor" class but a level has several actors. Each of these actors is an instance of the "Actor" class.

Encapsulation:

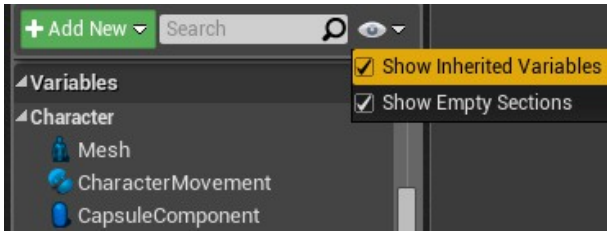
Encapsulation consists in hide the complexity of a class. For this, each class must be as independent as possible by avoiding interaction with many classes. It should be available to other classes only the data and functions actually needed for the use of this class to facilitate understanding.

As an example, when a function is created in Blueprint, you can specify the level of access to this function. *Private* means that access can only be done in the Blueprint class that has the function. In *Protected* the function can be accessed in subclasses. In *Public* any other class can call the function.

Inheritance:

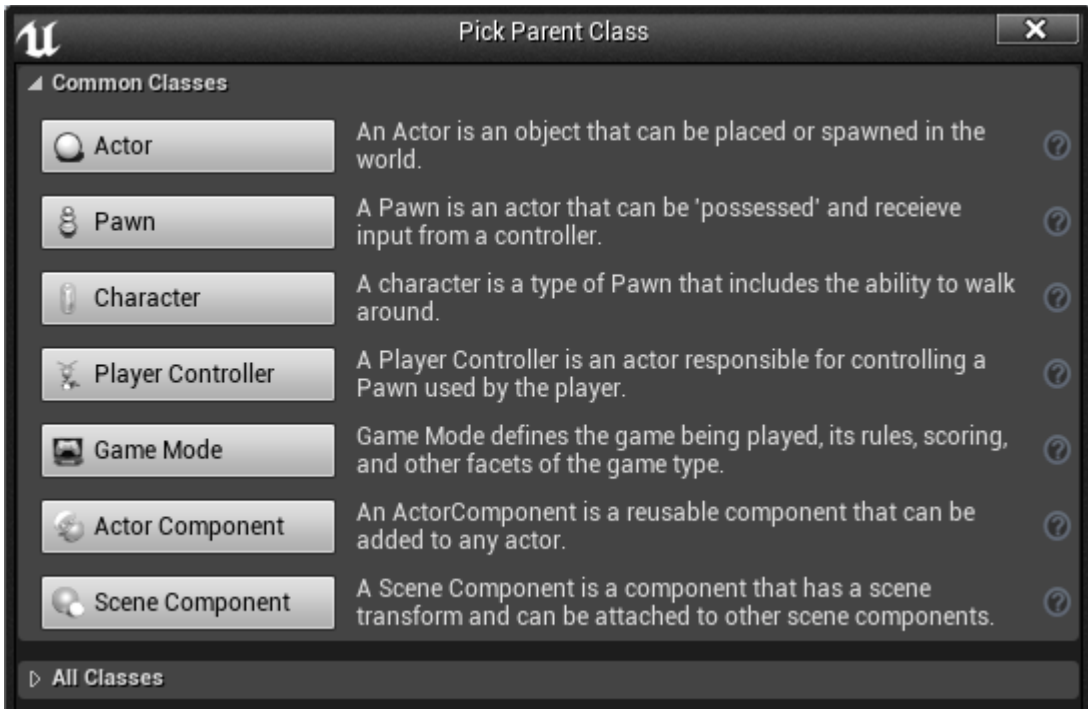
When a new Blueprint is created, you need to define which is the Parent class. All variables and Actions of the Parent class will be part of the child class, which is also known as *subclass*. This concept is called **Inheritance**. Inheritance is also used to define the type of an object because the object accumulates all types related to the parent class. For example, the "Pawn" class is a subclass of "Actor" class. If a Blueprint is created using "Pawn" as parent class, it can be said that this new Blueprint is of the "Pawn" type and is also of the "Actor" type.

In "MyBlueprint" tab you can see the variables that were inherited from the parent class. Click the eye icon and select the "Show Inherited Variables" option.



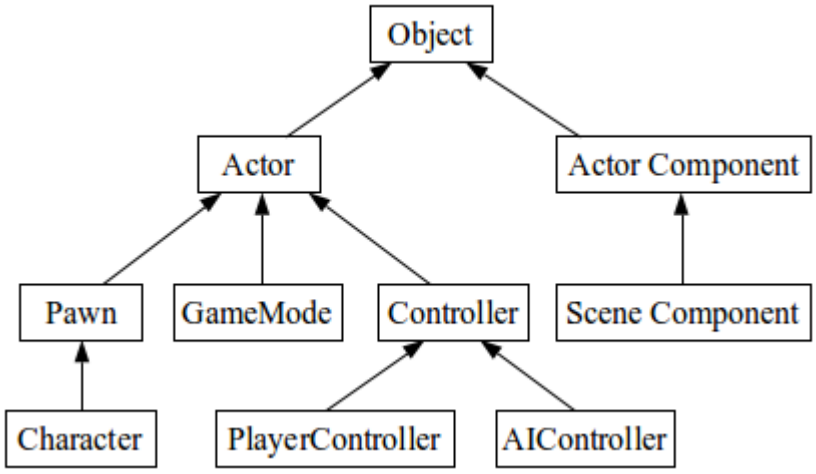
Common Classes

When a Blueprint is being created there are some suggestions for common classes to be used as the Blueprint Parent class. These common classes are known as "**Gameplay Framework**" and are used to represent players, characters, controllers and game rules.



To use another class as Parent class, just expand the category "All Classes" and search for the desired class. The classes "Actor Component" and "Scene Component" allow the creation of components using Blueprints.

The image below shows the hierarchy of the common classes. The arrow indicates that the class below is inheriting from the class above. There is a base class in Unreal called "Object".

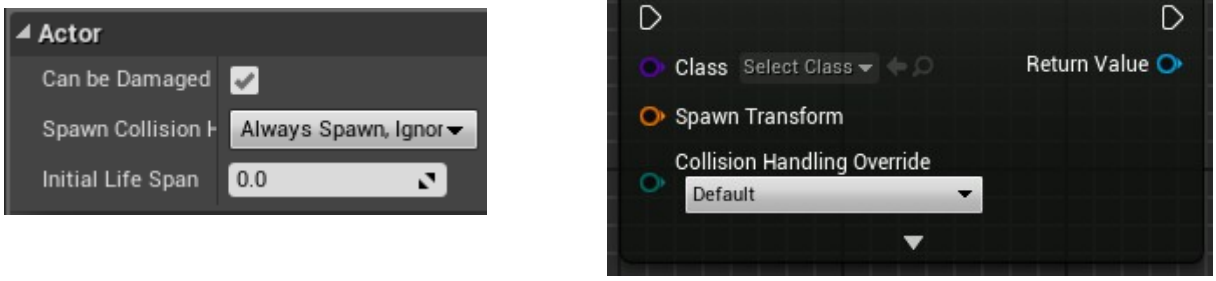


Actor

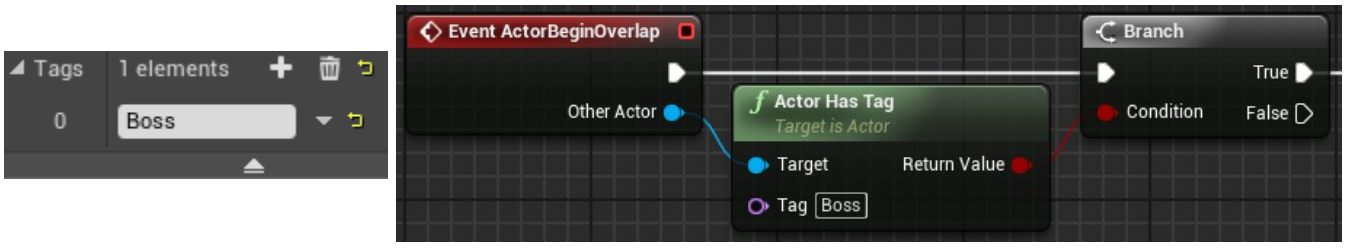
Base class for objects that can be added to the Level. The actors may contain components and have functionality for replication that are used in network games. Most of the time the Blueprint will be based on "Actor" or a subclass of "Actor". It is important to know the attributes and actions that are part of the "Actor" class because they are available to all subclasses of "Actor".

For example, all "Actor" has a "Transform" which is a structure that defines the position, rotation and scale of the "Actor". There are functions to change the "Transform" or part of it as the "Location".

Click the "Class Defaults" button to see other attributes that are part of the "Actor" class. Some attributes indicate whether the actor receives damage, how to deal with the collision when the actor is created in the level and also specifies the actor's life span before being automatically removed. There is an action named "Spawn Actor from Class" which is used to create actors in the Level during the execution of the game. Just inform the "Transform" and the "Actor" subclass that will be used.



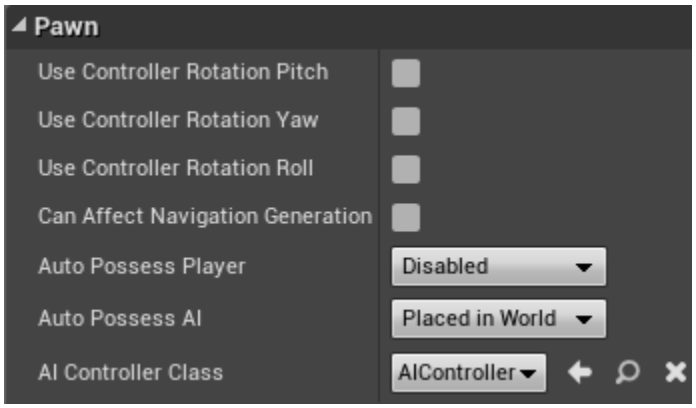
In "Class Defaults" you can specify a set of Tags to identify the actors. The example below shows that was created in an actor one tag with the name "Boss". In another blueprint a collision test is done and the actions will only be executed if the actor who is colliding has the Tag "Boss".



Pawn

The "Pawn" class represents the actors that can be controlled by players or by artificial intelligence. For each object of the "Pawn" class there is an object of the "Controller" class. The "Pawn" class is the physical body and the "Controller" class is the brain.

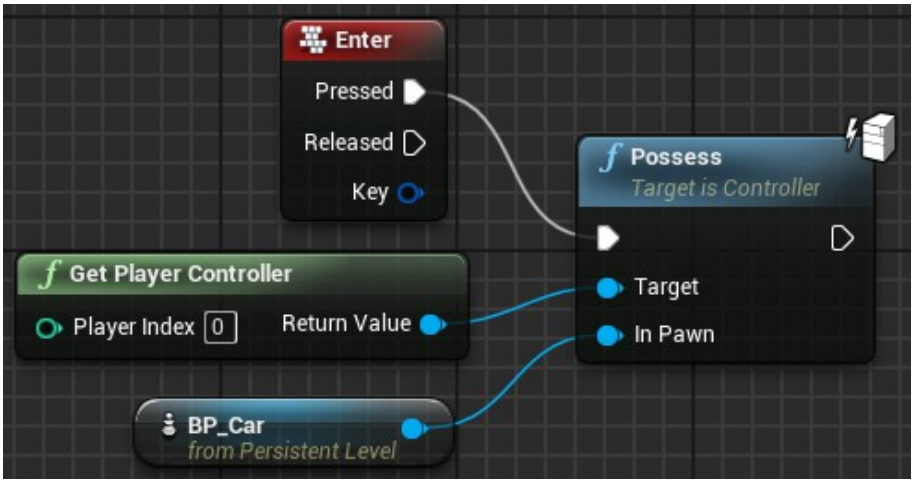
Create a Blueprint based on "Pawn" class and click the "Class Defaults" button to display the attributes that have been added in the "Pawn" class.



The "Pawn" class can use the rotation values of the "Controller" that is associated with it. Other attributes indicate how the "Pawn" is owned by a "Controller".

For a "Controller" to control an "Pawn" object it must possess this "Pawn". A "Controller" can change the "Pawn" being possessed during the execution of the game. For example, in a game that the player controls a person who can get into a car, there is the "Pawn" representing the person and there is the "Pawn" that represents the car. When the player get into the car, the "Controller" will possess the car "Pawn" and the movement commands will be answered by the car.

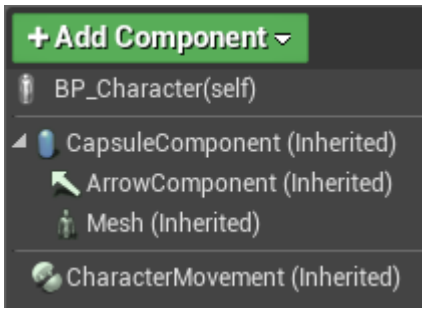
The image below is from a *Level Blueprint* showing the use of the "Possess" function. This function belongs to the "Controller" class. It requires a "Controller" and a "Pawn" to be possessed. In this example it is being used the "Player Controller" of the first player who will possess a "Pawn" called "BP_Car" that is on the level.



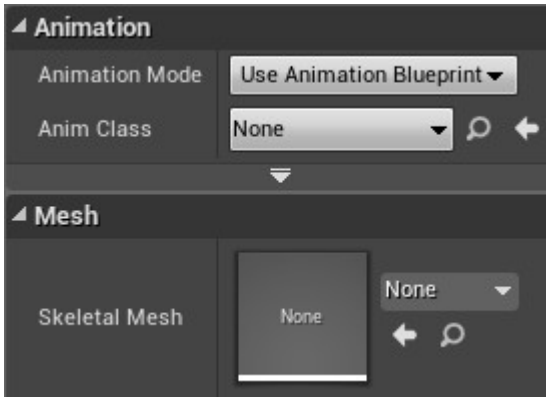
Character

The "Character" class is a subclass of the "Pawn" class, so it can also be possessed by a "Controller". This class was created to represent characters that can walk, run, jump, swim and fly. This class already has a set of components to assist in this goal.

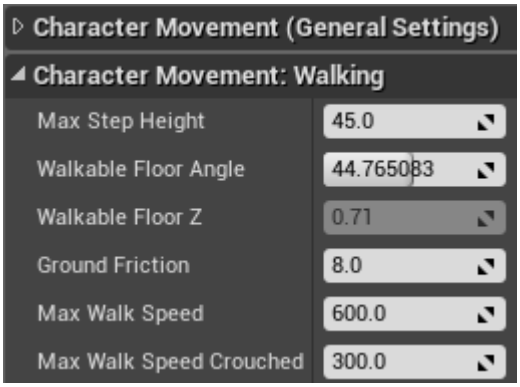
Create a Blueprint based on the "Character" class and see in the "Components" tab the components that were inherited from the "Character" class.



The "CapsuleComponent" is used for collision tests. The "ArrowComponent" indicates the current direction of the "Character". The Mesh component is the visual representation of the "Character", is of "Skeletal Mesh" type and can be controlled by an **Animation Blueprint**.



The "CharacterMovement" component has a large number of properties to define various types of movements of the "Character" such as walk, run, jump, swim and fly. The image below shows some properties that are used when the "Character" is walking.



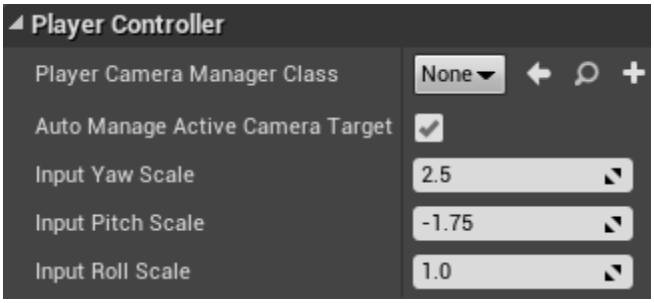
Player Controller

The "Controller" class has two main subclasses. The "PlayerController" class is used by human players and the "AIController" class uses artificial intelligence to control the "Pawn". In a simple game it is not necessary to create a Blueprint based on "PlayerController" as Unreal Engine already uses a default instance of "PlayerController" which is sufficient for most cases.

Objects of "Pawn" and "Character" classes only receive Input events if they are being possessed by a "PlayerController". So the use of a "PlayerController" is required even if it is the default. Input events can be placed on the "Player Controller" or on the "Pawn", but events and actions related to the movement of a "Pawn" should be in the class that represents the "Pawn".

For example, if the player controls a character who can use several different types of vehicles such as car, helicopter and speedboat. Each vehicle is represented by a subclass of "Pawn" and must contain the equivalent move actions. But an Input event that pause the game should not be in the "Pawn" class, because it would have to be repeated in all subclasses of "Pawn" used by the game. The ideal place for this type of input event that is not related to "Pawn" is in the "PlayerController" class.

Create a Blueprint based on "PlayerController" class and click the "Class Defaults" button to view the specific attributes of the "PlayerController" class.



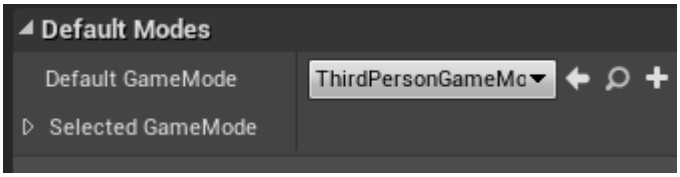
You can specify a new "CameraManager" class. If none is specified, a default class is used that controls the active camera used by the Pawn being controlled. Other attributes allow you to adjust the scale used in the inputs related to rotation.

The image below shows a simple way to pause/unpause the game when the "P" key is pressed. These actions are in a "PlayerController". It used the "SetGamePaused" function that receives a Boolean value (true or false) as input. Each call to the FlipFlop macro toggles the value of the variable "Is A" which is used to determine whether the game should pause or unpause. A very important detail, in the input event of the "P" key is necessary to check the "Execute when Paused" option.

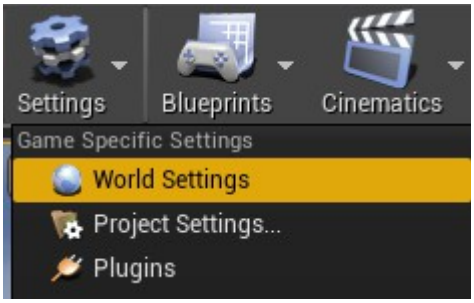
Game Mode

The "GameMode" class is used to define the rules of the game. The "GameMode" also specifies the classes that will be used for the creation of "Pawn", "PlayerController", "HUD" and other classes.

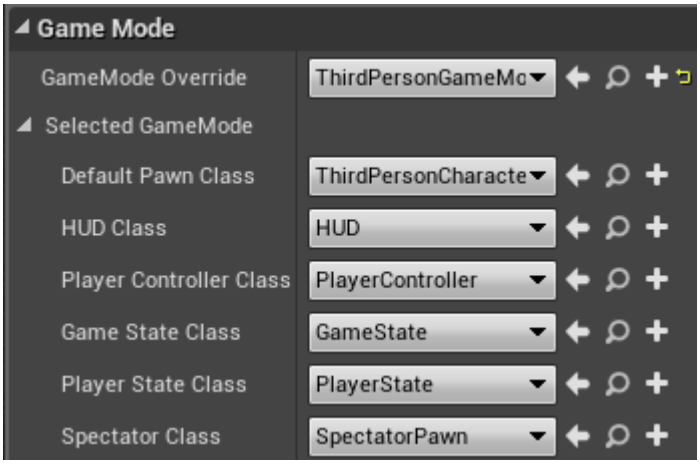
After you create a blueprint of the "GameMode" type it is possible to set it as default "GameMode" for the project by accessing in the main editor the menu option: *Edit->Project Settings->Maps & Modes*. Then just choose in the combobox of the option "Default GameMode" the Blueprint that was created.



Each level can have a different "GameMode". If a "GameMode" is not specified for the level, it will be used the "GameMode" that has been set for the project. To specify the "GameMode" of a level click on the "Settings" button of the level editor and choose the "World Settings" option.



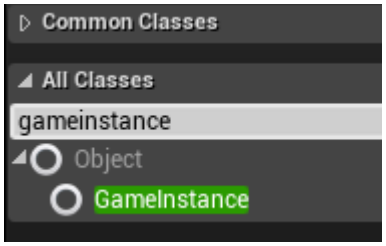
Choose the "GameMode" of the level in the "GameMode Override" option. The editor also allows other classes to be selected to replace some attributes of the "GameMode". This way it is possible to make small changes in the "GameMode" of each level without creating another "GameMode".



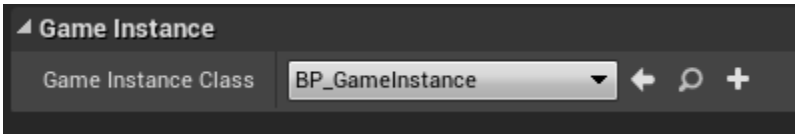
Game Instance

Before loading a new level, the current level is unloaded and the variables used by the current level are removed from memory. When the new level is loaded, the variables related to the level such as GameMode, PlayerController and Pawn are created with default values. This means that if there are values in the variables that need to be preserved for the next level, they must be saved before leaving the current level and then loaded into the new level.

Another option to preserve variable values between the levels is by using the "**GameInstance**" class. An instance of this class is created at the beginning of the game and is only removed when the game is closed. To create a Blueprint of this type search for "gameinstance" in the "AllClasses" category.



After creating the Blueprint of the type "GameInstance" you must configure the project to use this new "GameInstance". Access the main editor menu option: *Edit-> ProjectSettings -> Maps & Modes*. A Blueprint called "BP_GameInstance" was created to be used in this project.



As an example it was created in "BP_GameInstance" a variable named "Time Spent in Game" to save the time a person has spent in the game. The image below is a custom event that shows how to access this variable from another Blueprint.

Use the "Get Game Instance" function to get the reference to the "GameInstance" being used by the project. But this reference is of the type "GameInstance" and to access the variable "Time Spent in Game" you must use a reference of the type "BP_GameInstance" which is obtained using the "Cast to BP_GameInstance" action.