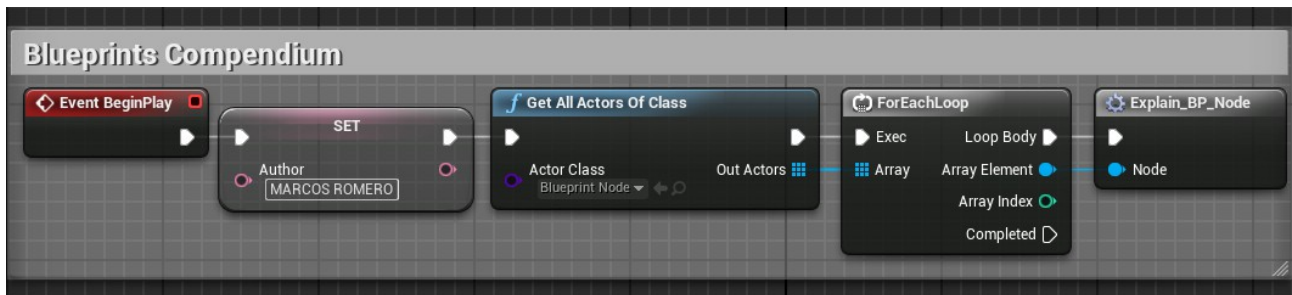


Blueprints Compendium



Author: Marcos Romero

Site: romeroblueprints.blogspot.com

About this document:

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

This document presents some key blueprint nodes with contextual examples of its use.

Blueprints Nodes:

• Branch	pg. 3
• Switch on Int	pg. 4
• For Each Loop	pg. 5
• Get All Actors of Class	pg. 6
• Spawn Actor from Class	pg. 7
• Random Integer in Range	pg. 8
• Open Level	pg. 9
• Quit Game	pg. 10
• Set Timer by Event	pg. 11
• Clear Timer by Handle	pg. 12
• Cast To	pg. 13
• Lerp	pg. 14
• Is Valid	pg. 15
• Gate	pg. 16
• Delay	pg. 17
• Dot product	pg. 18
• Event Dispatcher	pg. 19
• Bind Event	pg. 20
• Format text	pg. 21
• Attach actor to component	pg. 22
• Set view target with blend	pg. 23
• Timeline	pg. 24
• Math expression	pg. 25
• LineTraceForObjects	pg. 26
• MultiLineTraceByChannel	pg. 27

Branch

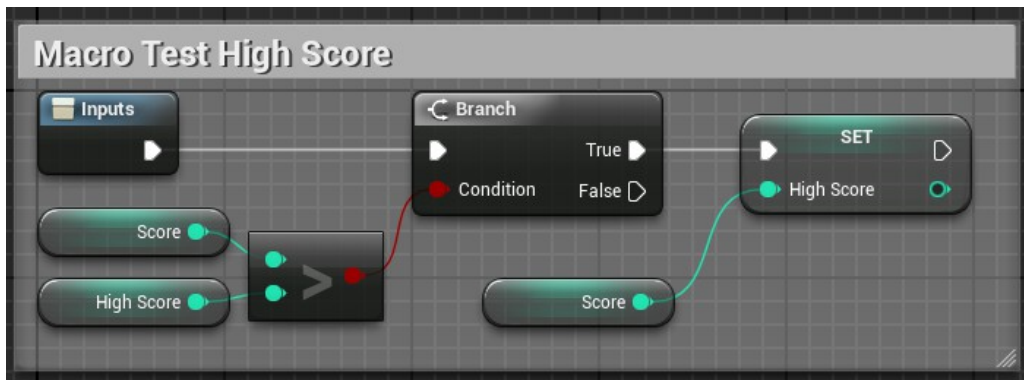
Directs the flow of execution of a Blueprint based on the Boolean value “Condition”, which can be true or false.

Input

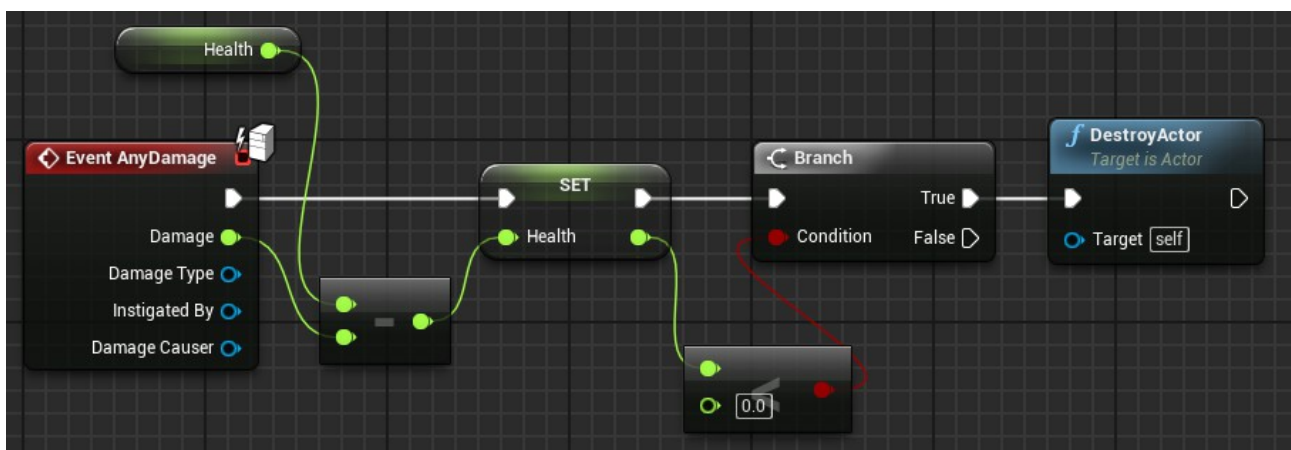
- **Condition:** This Boolean value can be obtained from a Boolean variable or may be the result of an expression.

Example Usage:

At the end of a match it is checked whether the player's score was higher than the highest score. If true, then this score is saved in the “High Score” variable.



In another example, when the event "AnyDamage" is triggered, the new value of "Health" variable becomes its old value minus the value of "damage" received. If the value of "Health" is less than zero, the Actor is destroyed.



Switch on Int

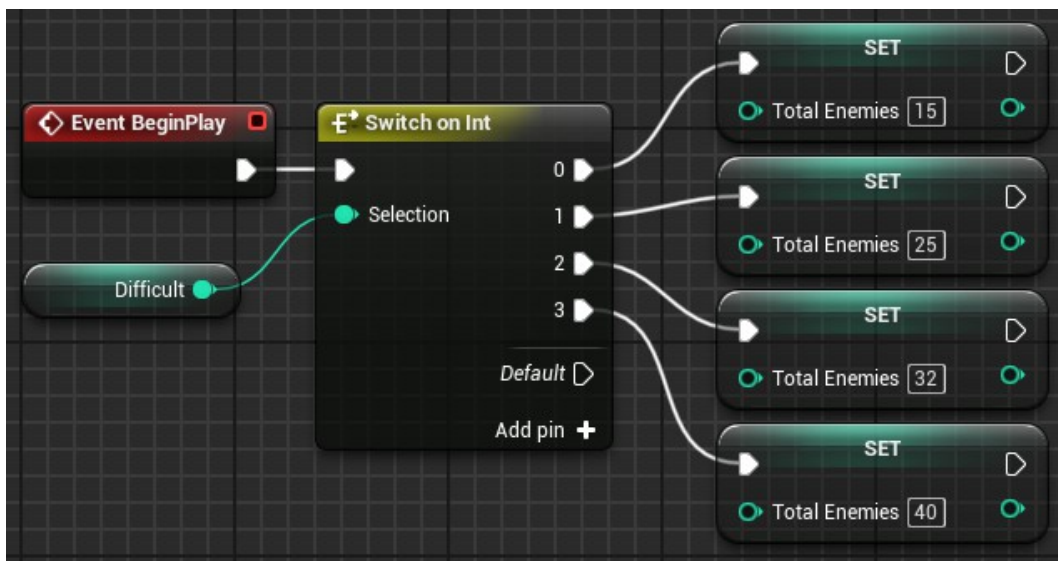
Determines the flow of execution according to the integer input value.

Input

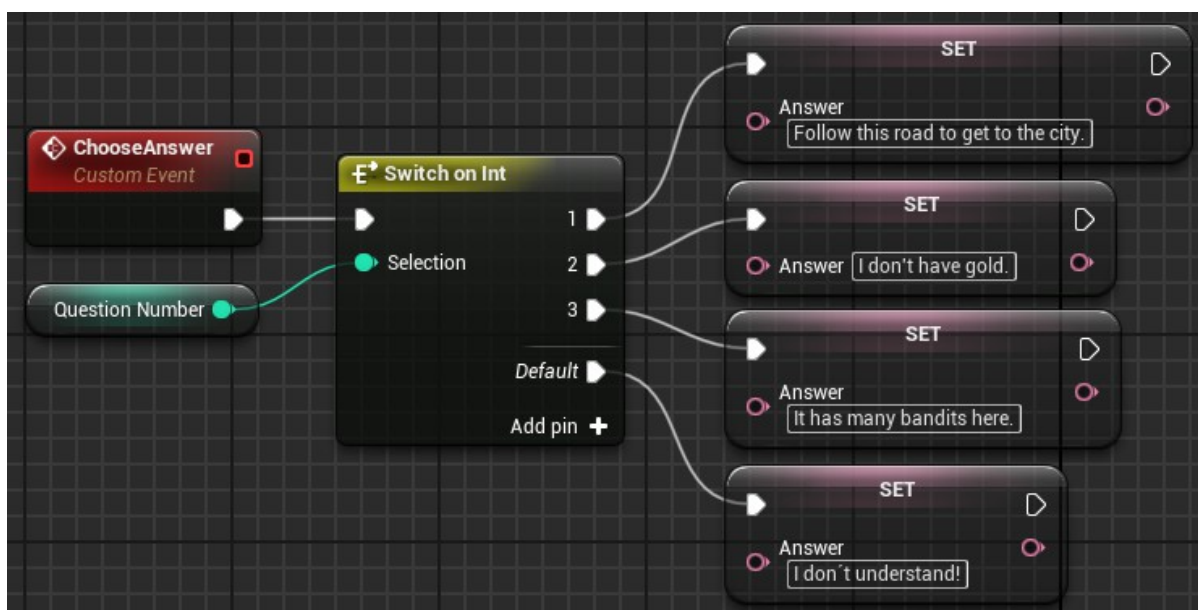
- **Selection:** Integer value that determines the output. If the value is not found, the output happens in the *default* pin.

Example Usage:

The difficulty of a game is stored in an integer variable named "Difficult" that can have the values from 0 to 3. The total number of enemies is set according to the difficulty.



In another example, the “Switch On Int” is used to choose a response in accordance with the number of the question that was asked.



For Each Loop

Takes as input parameter an Array and performs the set of actions that are associated with the output pin "Loop Body" for each element of the Array that can be obtained from the output pin "Array Element". After that the execution flow is directed to the output pin "Completed".

Input

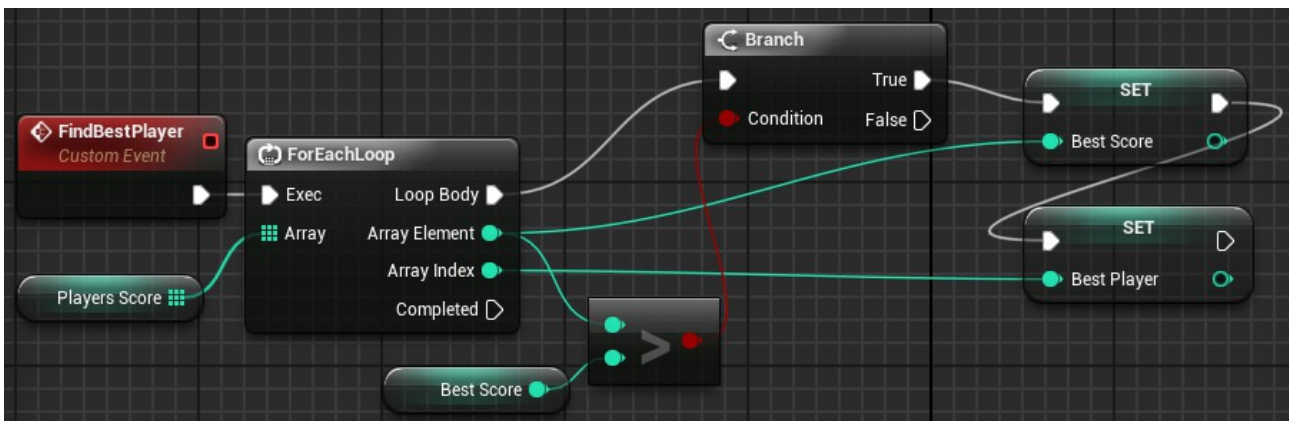
- **Array:** Array containing the elements that will be used in the loop.

Output

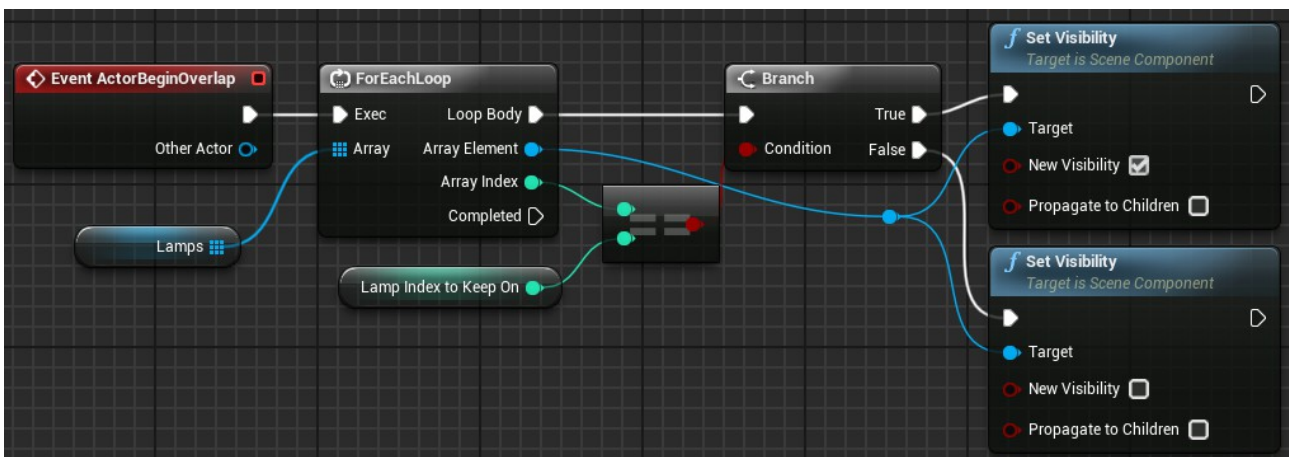
- **Array Element:** Reference to the current array element.
- **Array Index:** Index of the current array element.

Example Usage:

A ForEach loop is used to iterate through an array that contains the scores of the players. For each value a test is done to check whether it is the highest score, if true the value is stored in the variable "Best Score" and the player's index is stored in the variable "Best Player".



In another example, when the player overlap a Blueprint, all lamps in the Array "Lamps" will turn off except the lamp whose index is stored in the "Lamp Index to Keep On" variable.



Get All Actors of Class

Get the references of all the actors who belong to the specified class or subclass.

Input

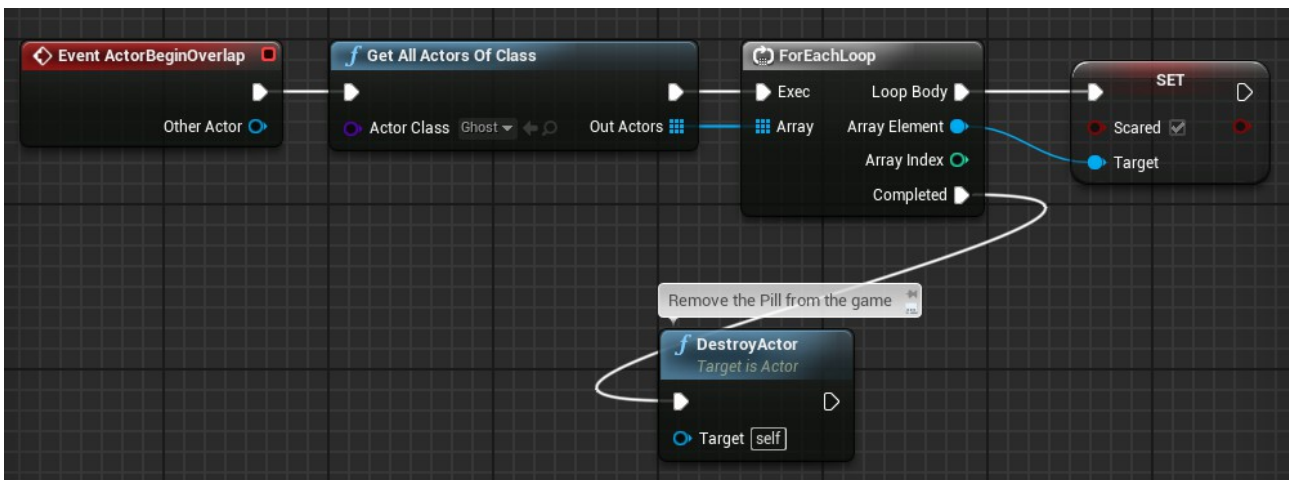
- **Actor Class:** The class that will be used in the search.

Output

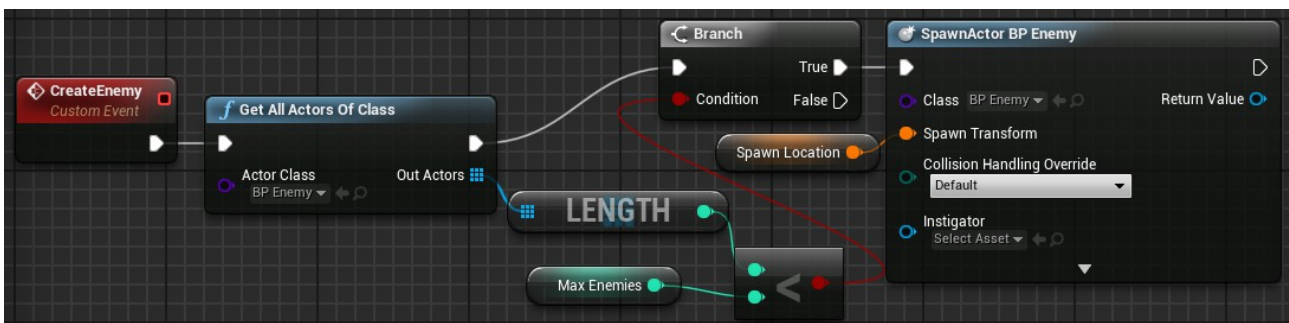
- **Out Actors:** Array containing references to the actors of the specified class.

Example Usage:

Imagine a Pac Man game with a "Pill" blueprint and a "Ghost" blueprint that has a Boolean variable "Scared". When the pill is caught, it has to inform all the ghosts to become scared.



In a second example, before creating a new enemy, the "Get All Actors of Class" function is used to see how many enemies are on the level. If there are less than the maximum number, then a new enemy is created.



Spawn Actor from Class

Creates an actor at the level according to the class and transformation specified.

Input

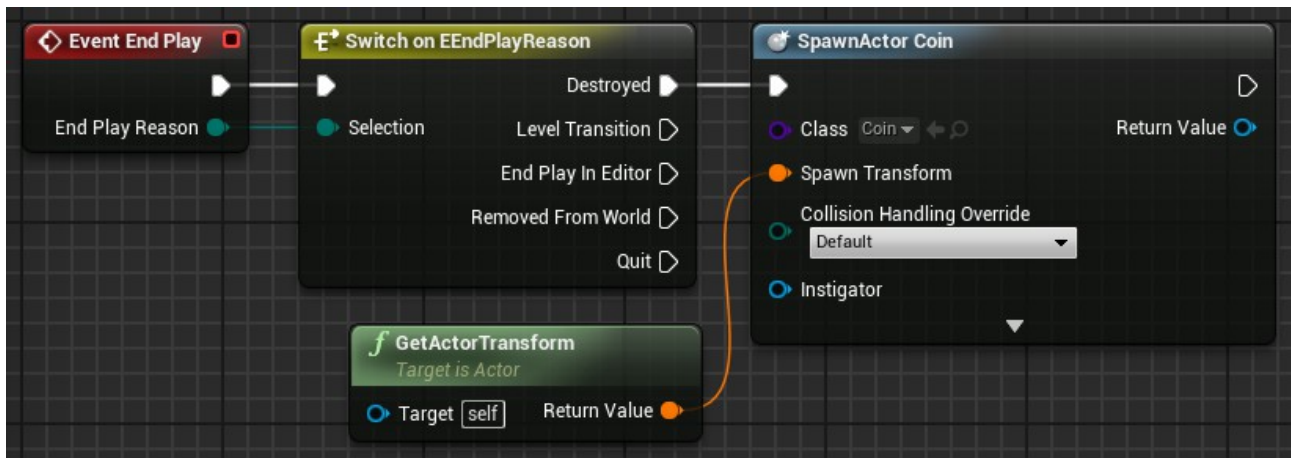
- **Class:** The class of the actor that will be created.
- **Spawn Transform:** The transformation that will be used by the actor.
- **Collision Handling Override:** Defines how to handle the collision at the time of creation.

Output

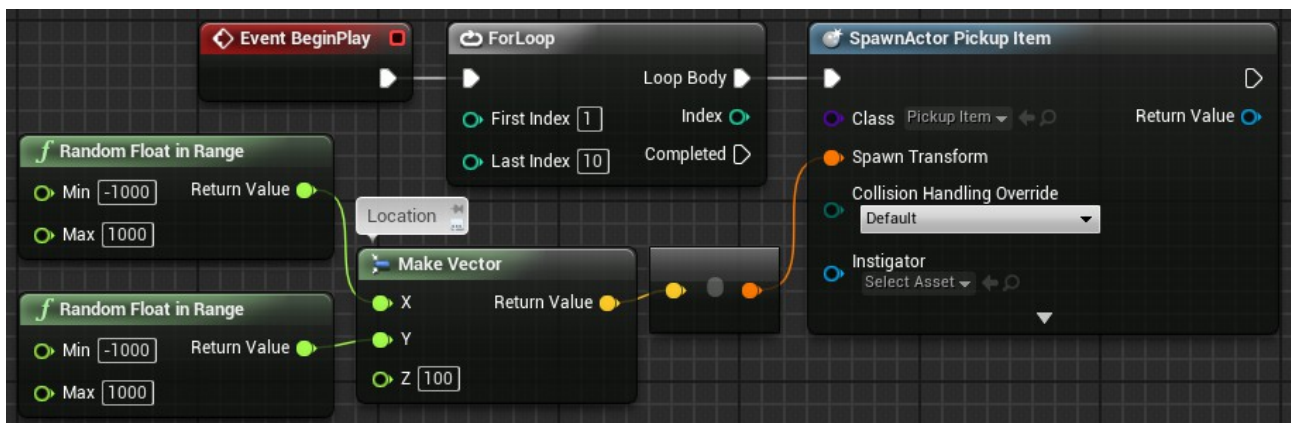
- **Return Value:** Reference to the actor created.

Example Usage:

When this Blueprint is destroyed in the game, it creates an actor of the "Coin" type in the same place.



In this example, a "For Loop" is used to create 10 "Pickup Item" at random positions.



Random Integer in Range

Function that returns a random integer within a range of specified values.

Input

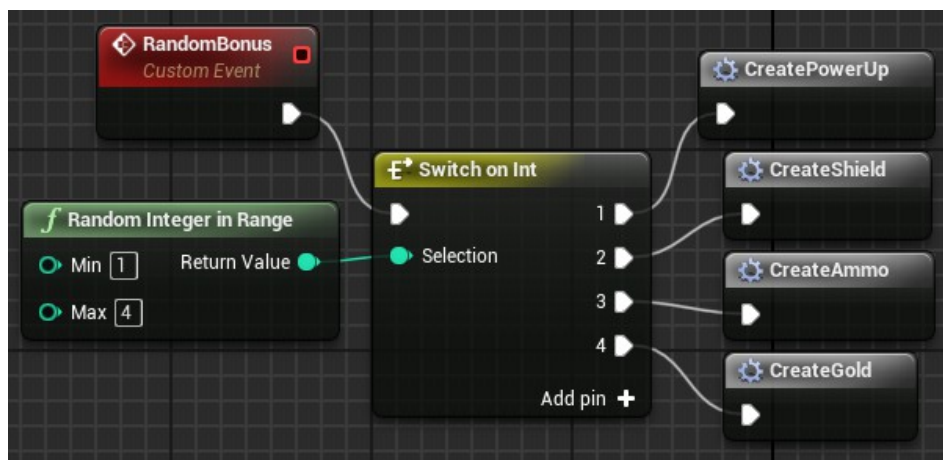
- **Min:** The lowest value that can be generated.
- **Max:** The highest value that can be generated.

Output

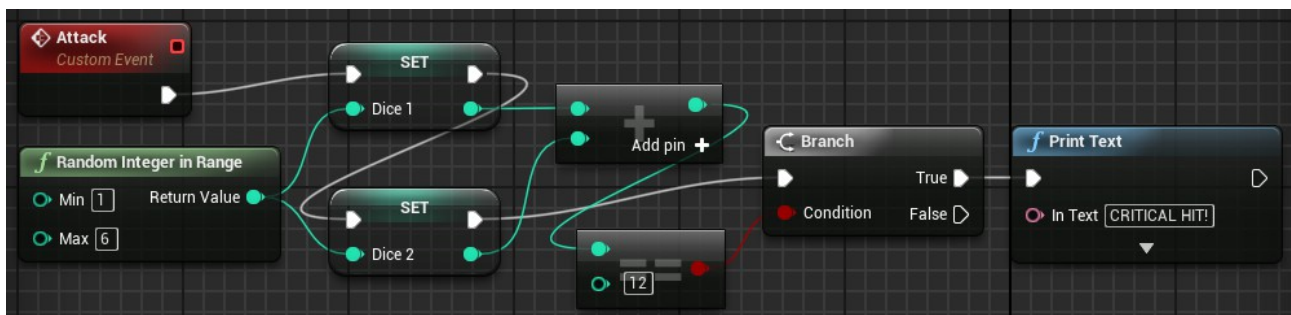
- **Return Value:** Random integer value within Min and Max.

Example Usage:

The custom event “Random Bonus” create a random type of bonus based on the result of the function “Random Integer in Range”.



The custom event below simulates an attack on a simple RPG based on two six-sided dice. If the sum of two dice is 12, the player does a critical hit. Note that each time the Random function is accessed, it generates a new random value.



Open Level

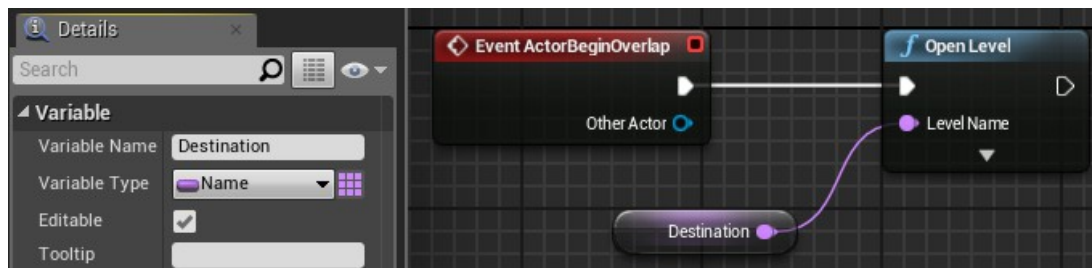
Loads a new level of the game.

Input

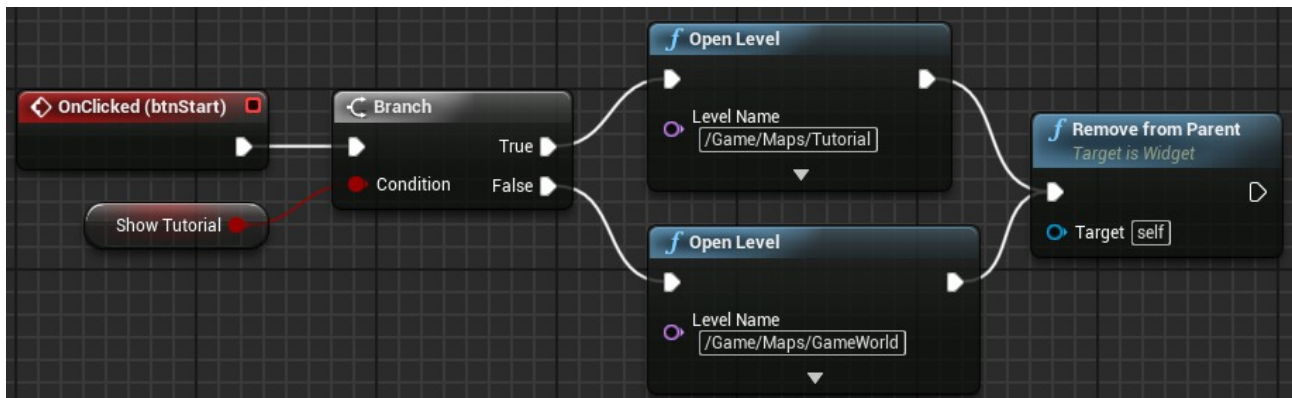
- **Level Name:** Full name of the level. (Path + Name)

Example Usage:

Imagine a blueprint “Teleporter” which when overlap by the player, will load a new level. The name of the level that should be loaded is informed in editable variable "Destination".



In another example, when a UMG button is clicked, a test is done to check whether to load the tutorial level or the level of the game world. After that the UMG widget is removed from the screen.



Quit Game

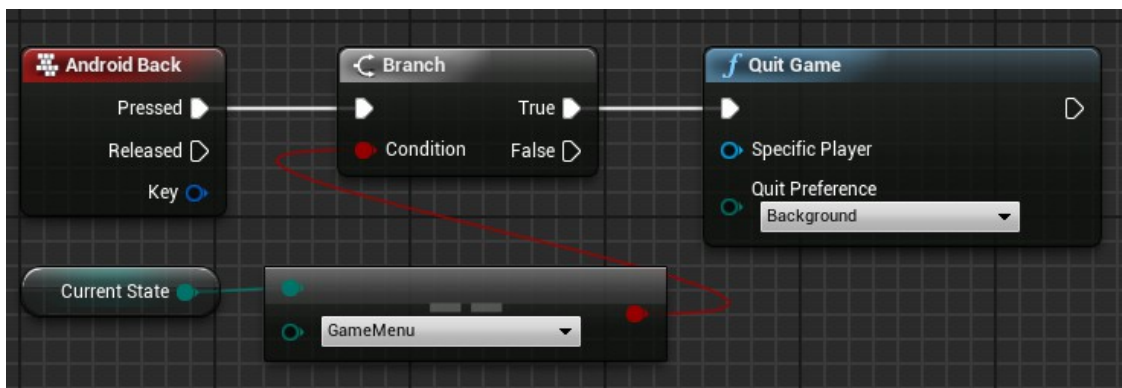
Exit the game or move the application to the background.

Input

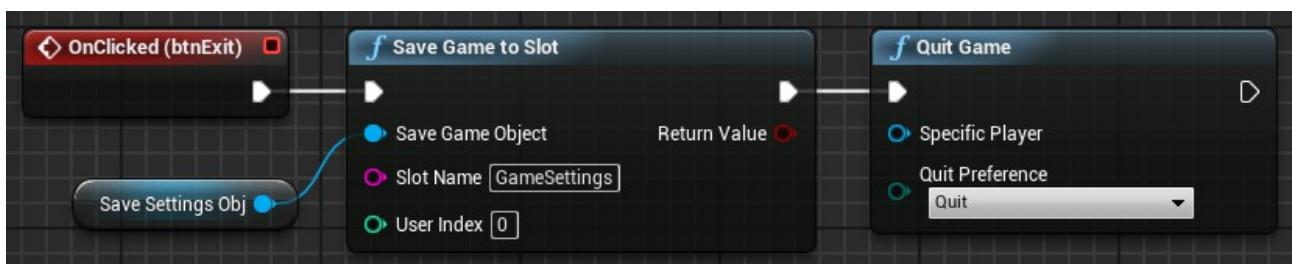
- **Quit Preference:** It can be "Quit" or "Background".

Example Usage:

When the Android back button is pressed, a test is done to check if the game is on the Menu. If true, the application is moved to the background.



In another example, after clicking on a UMG button to exit the game, the game settings is saved and then the application is closed.



Set Timer by Event

A timer is programmed to perform a given event after finishing the specified time.

Input

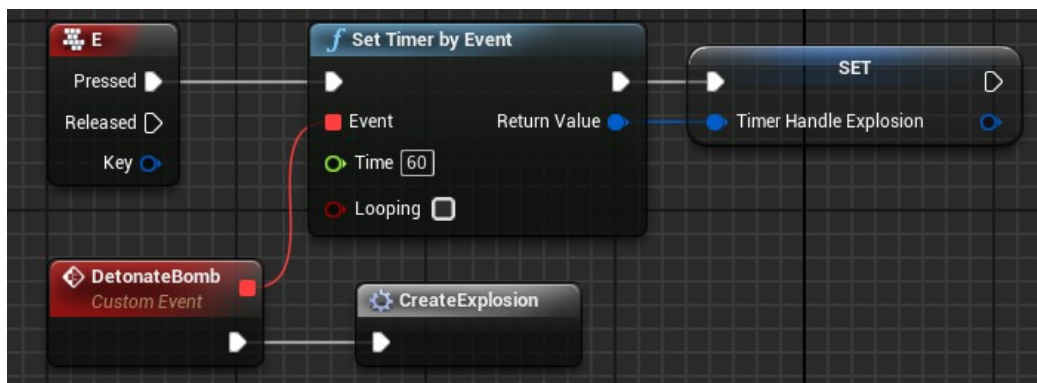
- **Event:** A reference to the event.
- **Time:** Time in seconds.
- **Looping:** Indicates whether the Timer should always repeat or if it's just one execution.

Output

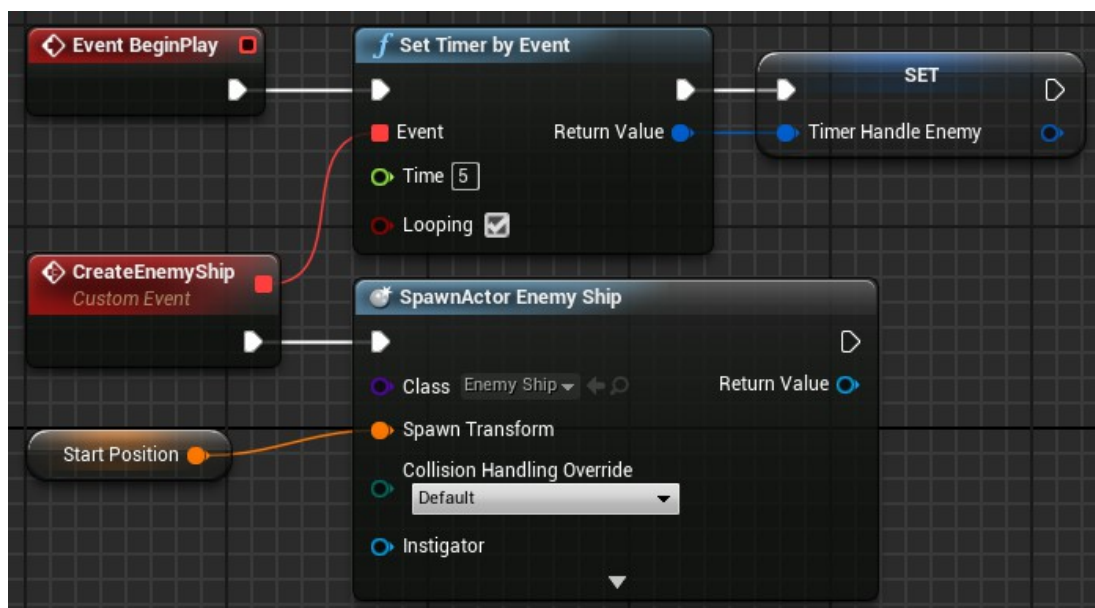
- **Return Value:** A reference to the Timer known as "Timer Handle" which can be stored in a variable of the type "Timer Handle" so that it can be handled later.

Example Usage:

When the "E" key is pressed, a timer is activated to run the custom event "DetonateBomb" after 60 seconds.



Another timer is programmed to call every 5 seconds an event that creates an enemy ship.



Clear Timer by Handle

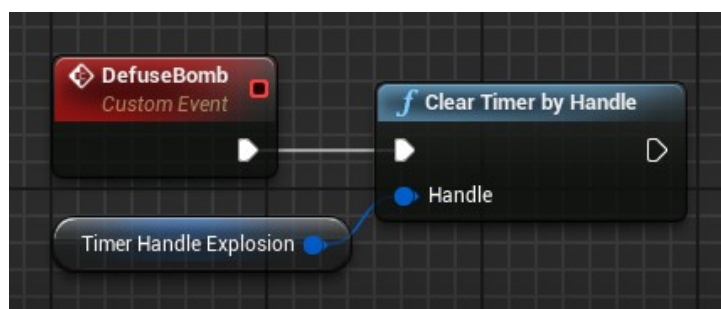
Stops the Timer that is referenced by the Timer Handle informed.

Input

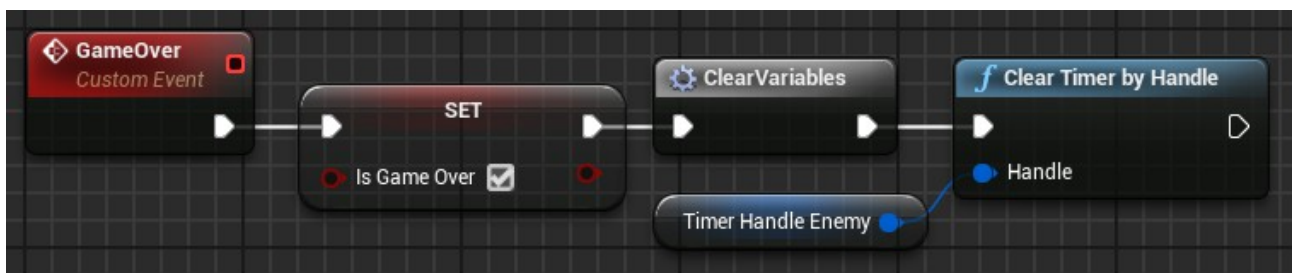
- **Handle:** "Timer Handle" that references a Timer.

Example Usage:

When the event "DefuseBomb" is called it will stop the timer that was being used for the countdown of the explosion.



At a "GameOver" custom event, a macro is called to clear the values of variables and the "Clear Timer by Handle" is called so that the timer which creates enemies does not continue to run.



Cast To

Convert the type of the reference variable to a new type specified. This is necessary in some situations to be able to access the variables and functions of a class or blueprint.

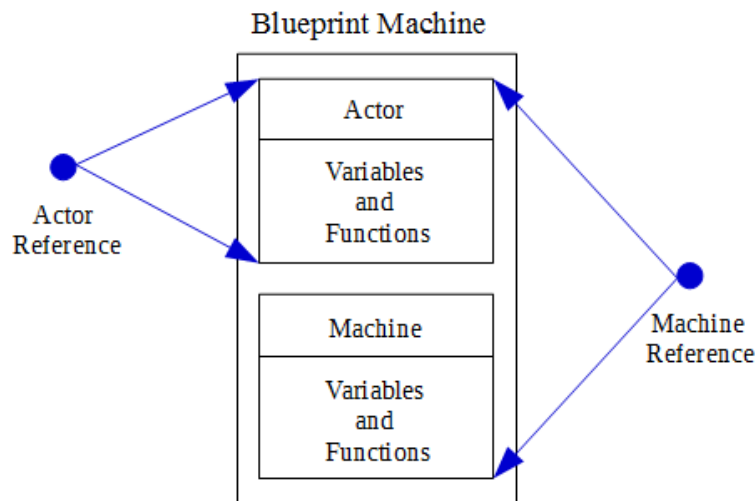
Input

- **Object:** A reference to an object.

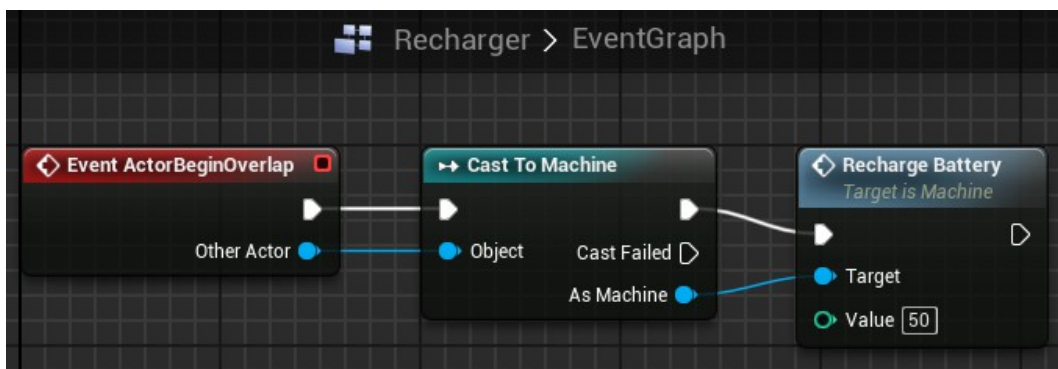
Output

- **Cast Failed:** Execution pin that is used if the referenced object is not of the type used in the Cast.
- **As (new type):** A reference using the new type specified in the Cast.

Example Usage: A new blueprint was created called "Machine". It is of the "Actor" type. This blueprint has a function called "Recharge Battery". A reference of the "Actor" type can be used to manipulate this new blueprint, but it would not have access to the function "Recharge Battery" because it can only access the variables and functions that were created in the "Actor" type. To access the "Recharge Battery" function we need a reference of the "Machine" type. See the illustration below:



In another blueprint called "Recharger", there is an overlap event. If the actor who overlaps is of the "Machine" type, then it will be called the "Recharge Battery" function using the reference of the "Machine" type that was provided in the "Cast To".



Lerp

Lerp is an abbreviation of "Linear Interpolation". The function generates a value within a range of two specified values, based on the value of the "Alpha" parameter.

Input

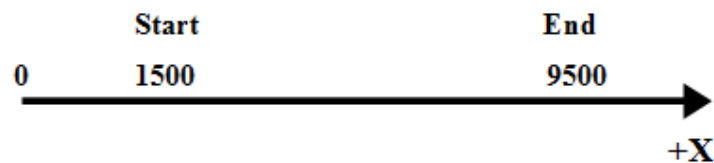
- **A:** Float value that represents the lowest value that can be returned.
- **B:** Float value that represents the highest value that can be returned.
- **Alpha:** Float value between 0 and 1. If Alpha is 0 return the lowest value and if alpha is 1 return the highest value.

Output

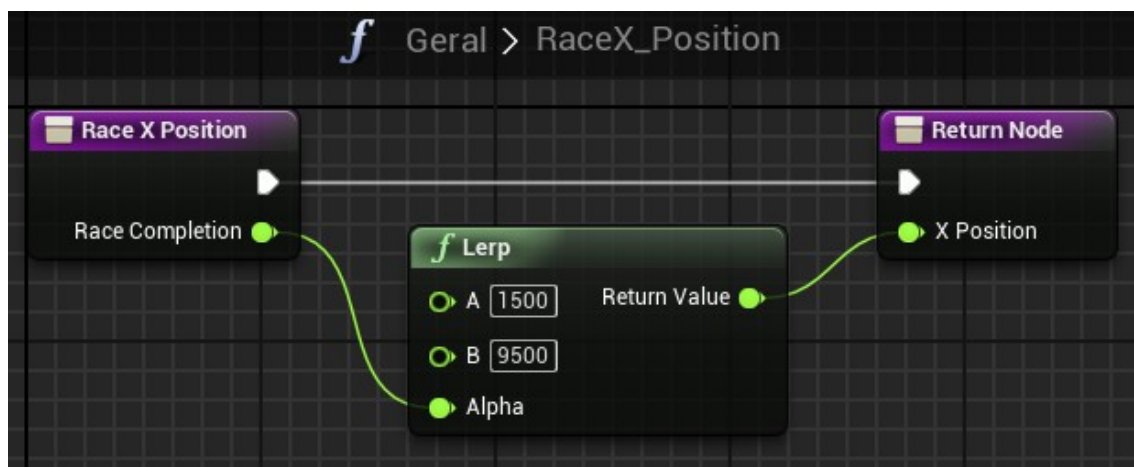
- **Return Value:** Float value between A and B depending on the value of alpha.

Example Usage:

Imagine a race that occurs along the X axis. The race starts at the position $X = 1500$ and ends at the position $X = 9500$. If we use the "Lerp" function passing as parameters $A = 1500$, $B = 9500$ and $\text{Alpha} = 0.5$, the result is the position 5500 representing the middle of the race.



The image below shows a function that takes a value between 0 and 1 that indicates how much has been completed of the race and returns the equivalent X position.



Is Valid

Macro that verifies the validity of the input object. For an object to be valid it can not be null or be marked to be destroyed.

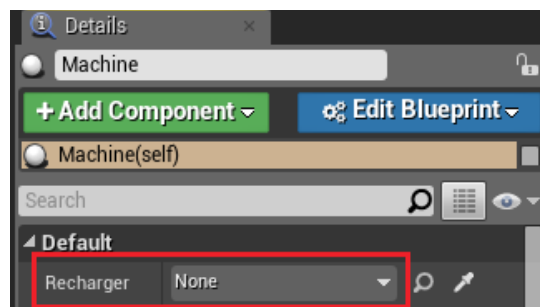
Input

- **Input Object:** A reference to an object.

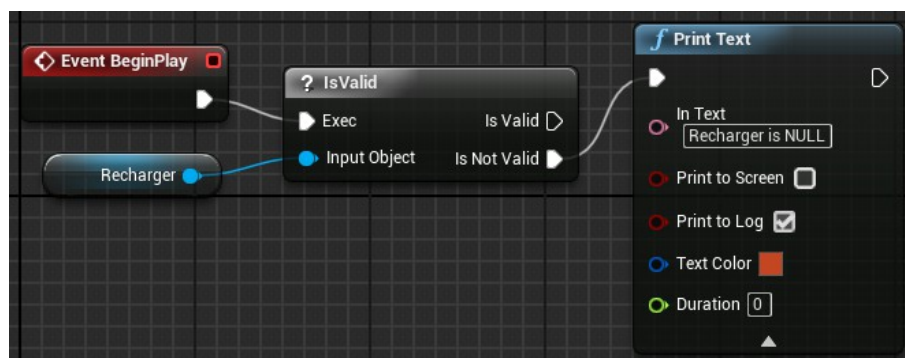
Output

- **Is Valid:** Execution pin that is used if the referenced object is valid.
- **Is Not Valid:** Execution pin that is used if the referenced object is invalid.

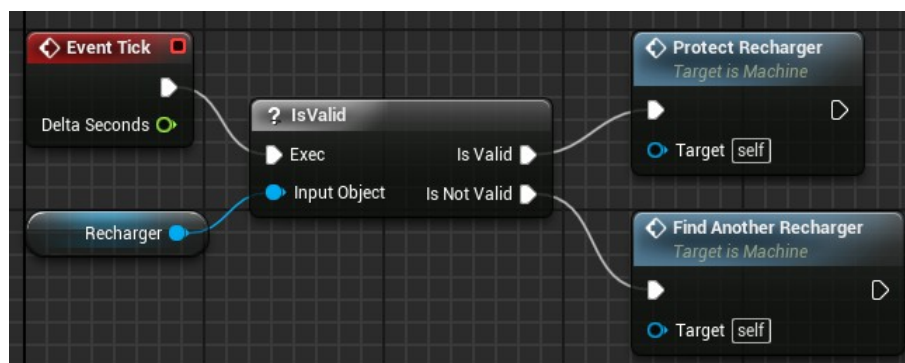
Example Usage: On a blueprint called "Machine" an editable variable was created that will be used to reference another blueprint called "Recharger". By adding a blueprint "Machine" at the level we can see that the contents of the variable "Recharger" is "None" as below:



This means that the variable "Recharger" is not referencing any object, so it is not valid. This variable should receive a reference to an object "Recharge" that is on the level. You can see when the game starts if the variable "Recharger" received this reference, if it has not received then it will be written a message in the log, as the image below.



In another example, imagine that the Blueprint "Machine" is responsible for protecting a Blueprint "Recharger" that is on the level. If the Blueprint recharger that it was protecting is destroyed, it needs to find another "Recharger" to protect:



Gate

Gate is a flow control node that can be open or closed. If it is open it allows the execution of the actions related to the Exit pin.

Input

- **Enter:** Execution pin that receives the current flow of execution.
- **Open:** Execution pin that changes the state of the gate to Open.
- **Close:** Execution pin that changes the state of the gate to Close.
- **Toggle:** Execution pin that toggle the current state of the gate.
- **Start Closed:** Boolean variable that determined if the "Gate" should start running in the "Closed" state.

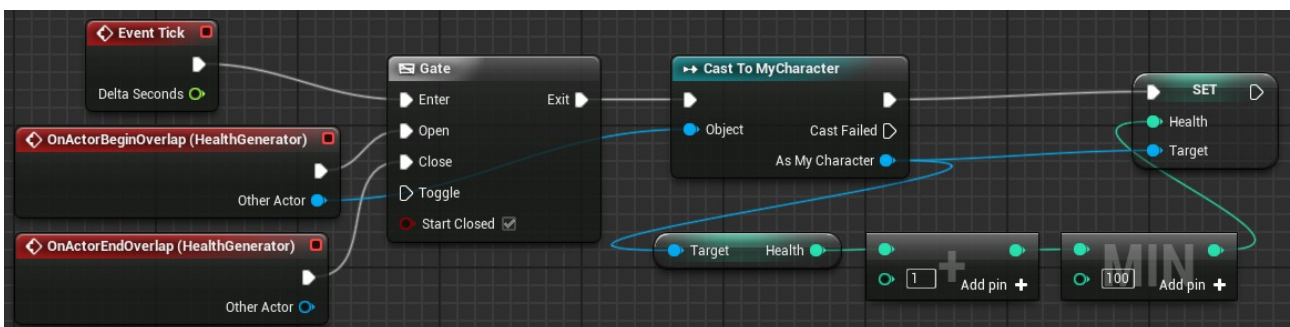
Output

- **Exit:** This pin will execute if the gate is open.

Example Usage:

In the example below we have an Actor called "HealthGenerator". When the player is colliding with this actor, his health will be restored slowly every "Tick" event.

If the player stops colliding with the "HealthGenerator", the "Gate" will be closed and the actions that restore health are no longer performed. The action "Min" is used so that the value of the variable "Health" is never greater than 100.



Delay

Delay is a *latent function* that performs the actions connected to the pin "Completed" only after the time specified in the parameter "Duration" has elapsed. The latent functions do not follow the normal flow of execution of the Blueprints. They run in parallel and can take several ticks until its completion.

Input

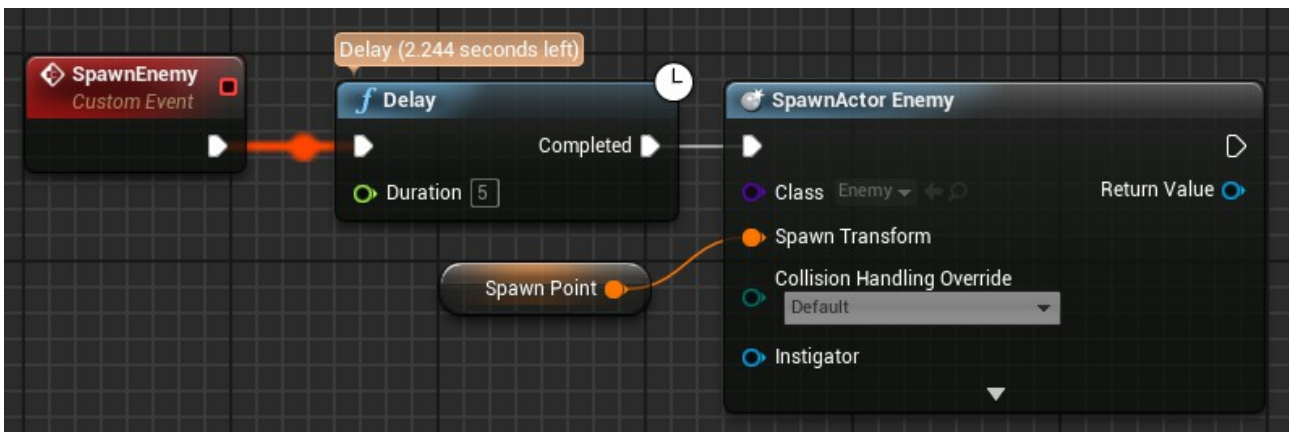
- **Duration:** Time in seconds the Delay function must wait.

Output

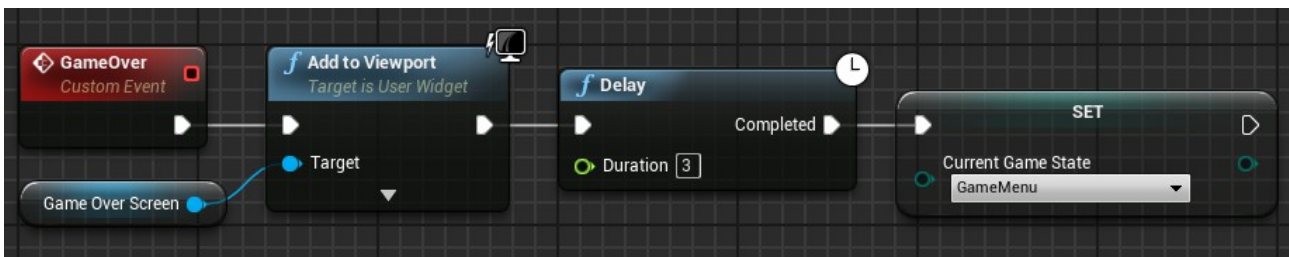
- **Completed:** Execution pin that will be activated after the time of "Duration" has elapsed.

Example Usage: Suppose in a game that there are several factors that can lead to the spawn of an enemy. The custom event "SpawnEnemy" was created with a delay function to ensure that it has elapsed at least 5 seconds before creating a new enemy. Even if the "Spawn Enemy" event is called again in less than 5 seconds, the delay function does not allow the creation of a new enemy.

The image below shows that the event "Spawn Enemy" is running and also shows that there is still "2.244" seconds for the creation of a new enemy.



In another example there is a custom event "Game Over" which displays a game over screen and after 3 seconds changes the value of a variable that keeps the game state so that the game return to the menu.



Dot product

Dot product can be used to verify the relationship between two vectors, for example, if they are perpendicular or parallel.

Input

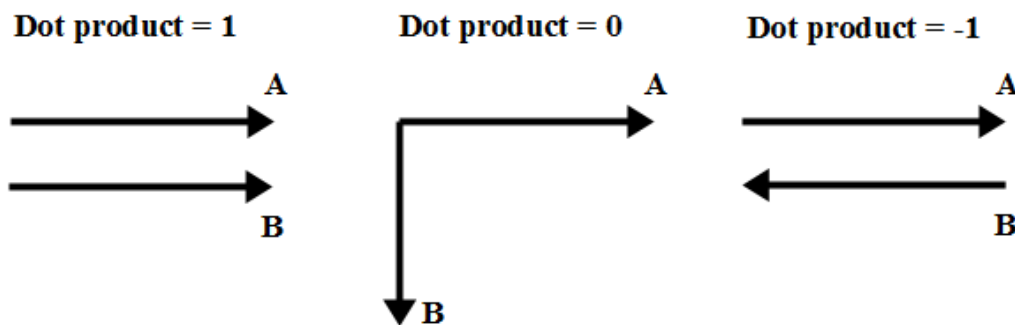
- A: Vector A.
- B: Vector B.

Output

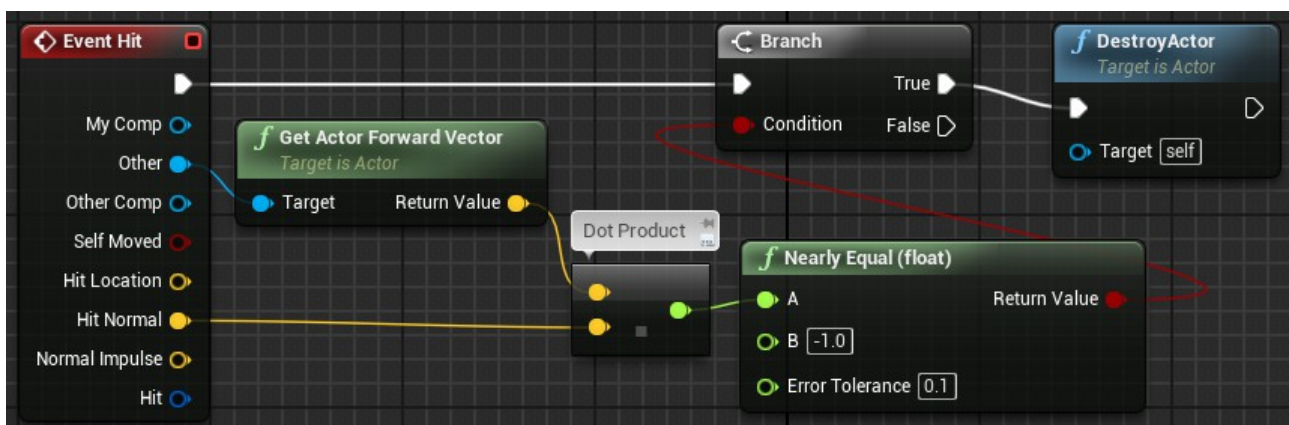
- **Return Value:** Float value representing the dot product between vectors A and B.

Example Usage:

If the two vectors are normalized, the dot product is equal to the cosine of the angle formed between the vectors and can range from -1 to 1. This image shows some examples of dot product between two vectors.



In the script below, an enemy can only be destroyed if hit by a straight shoot. This test is done based on the dot product between the "Forward Vector" of the projectile and the vector "Hit Normal" of the enemy. They need to be opposite, so the dot product should be close to -1.

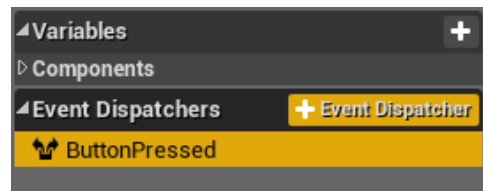


Event Dispatcher

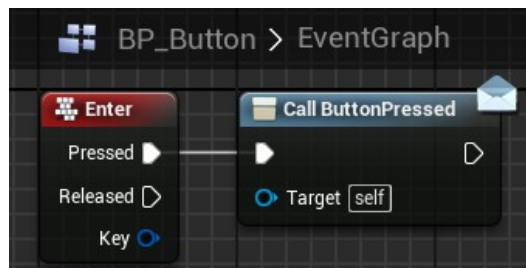
"Event Dispatcher" allows a type of communication between Class Blueprints and Level Blueprint. It is created in a Blueprint class and can be implemented in the Level Blueprint.

Example Usage: Imagine that we are creating a blueprint "BP Button" to represent a button that can be pressed. The purpose of this blueprint is just to inform when the button is pressed. In this way we can use this button for several different situations. The actions that will occur when the button is pressed will be set at the Level Blueprint.

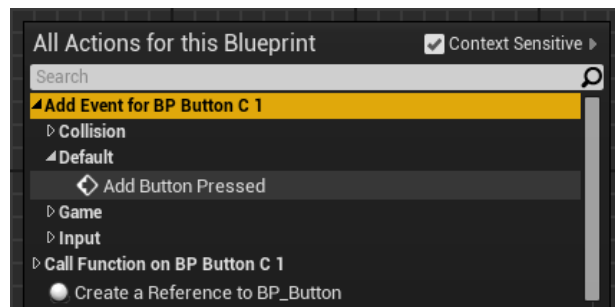
In the "MyBlueprint" tab of the "BP_Button", create an EventDispatcher named "Button Pressed":



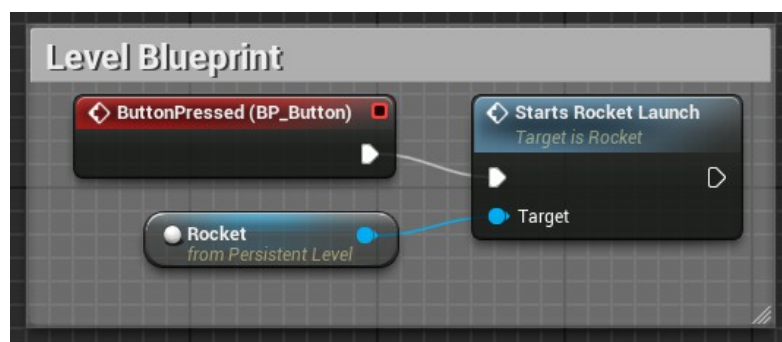
When the "Enter" key is pressed, the event dispatcher "Button Pressed" will be called:



Add a "BP Button" to the level and select it. Open the Level Blueprint and right click in the EventGraph to add the event "Button Pressed" related to the EventDispatcher that was created.



Assuming there is a blueprint called "Rocket" in the level and that it has a function called "Starts Rocket Launch". The Level Blueprint is responsible for calling the "Starts Rocket Launch" function when the button is pressed.



Bind Event

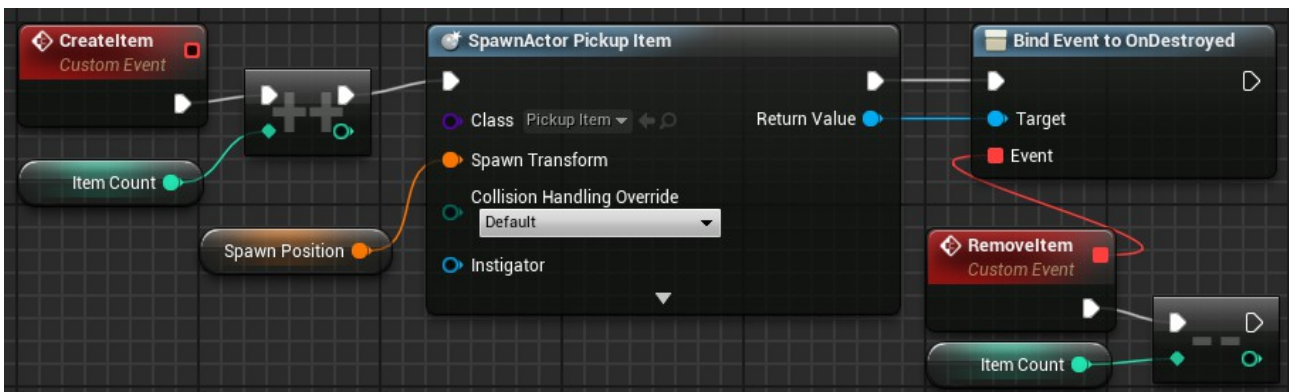
Bind one event to another event or *event dispatcher*, which can be in another Blueprint. When an event is called, all the other events that are bound to it are also called.

Input

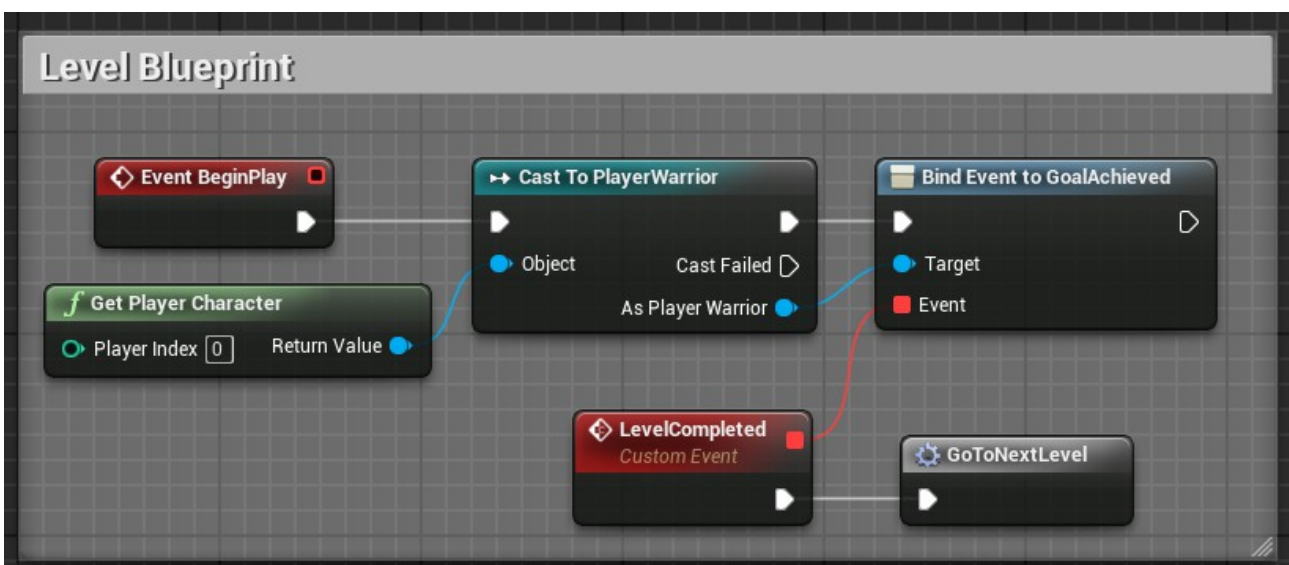
- **Target:** Object that has the event that receives the bind.
- **Event:** Reference to an event that will be bind to another event, to be called later.

Example Usage: There is a blueprint "PickupManager" that is responsible for creating "PickupItem" and keeps the variable "ItemCount" with the amount of "PickupItem" that exists in the level. For this, a custom event "RemoveItem" was created which decreases by 1 the value of the variable "ItemCount".

The event "RemoveItem" is bound to the event "OnDestroyed" of the "PickupItem" that was created. Thus, when the "PickupItem" is destroyed in the game, the event "RemoveItem" will be called.



In another example there is a blueprint called "PlayWarrior" which is used by the player. This blueprint has an *event dispatcher* named "GoalAchieved". When starting the game, the Level Blueprint does a bind with the event dispatcher "GoalAchieved". When the event dispatcher "GoalAchieved" is triggered by the player, the Level Blueprint will change to the next level.



Format text

Node that builds a text based on parameters that can be specified in the "format" property.

Input

- **Format:** The text that will be part of the final result. To set parameters, simply put a name between the delimiters {} for each parameter.
- **Parameters defined in "Format":** For each pair of {} is generated a new input parameter with the name that is between {}.

Output

- **Result:** Final text build with the values of the format parameter and the other parameters.

Example Usage:

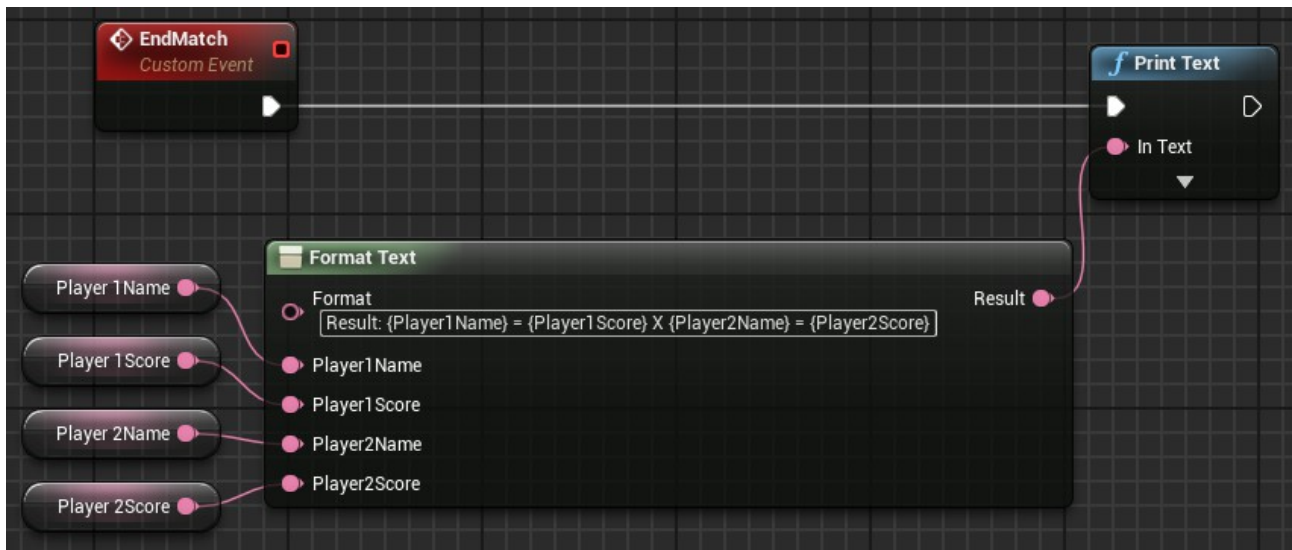
At the end of a match a text will appear with the result. This text contains the values of 4 variables related to the names and scores of two players.

The text used in the "format" parameter was:

Result: {Player1Name} = {Player1Score} X {Player2Name} = {Player2Score}

After placing the value of "Format", the Blueprint editor generates the other input parameters.

The image below shows the Format Text node.



This is an example of the generated text:

Result: Romero = 17 X Luke = 14

Attach actor to component

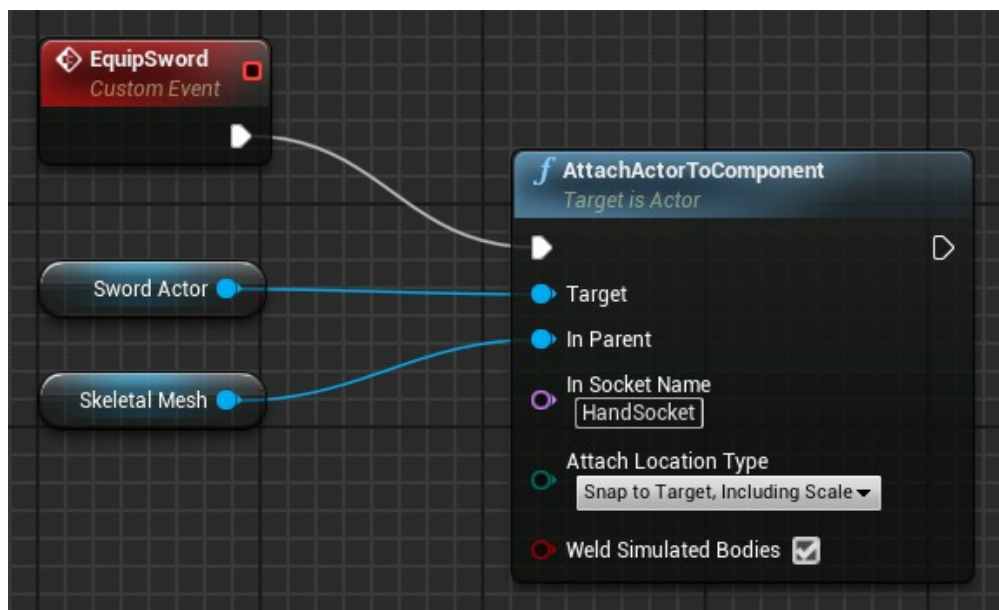
Attach an actor to the component of another actor. The actor attached suffers the transformations of the parent component.

Input

- **Target:** Actor to be attached.
- **In Parent:** Component that receives the actor.
- **In Socket Name:** (Optional) Name of the Socket to attach the actor.
- **Attach Location Type:** Can be AbsoluteWorld, RelativeOffset or SnapTo.

Example Usage:

The custom event below equips the Skeletal Mesh of the player with a sword using a socket labeled "HandSocket" that indicates where the sword must stay in the Skeletal Mesh.



Set view target with blend

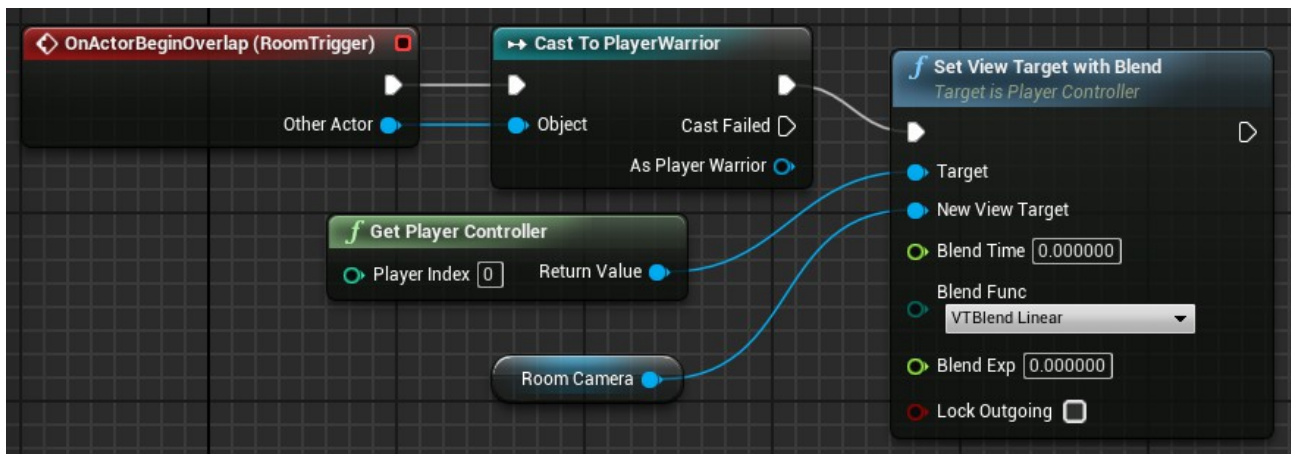
This feature is found in the PlayerController class. Very useful to switch the game view between different cameras.

Input

- **Target:** Reference to a Player Controller.
- **New View Target:** Actor to set as view target. Usually a camera.
- **Blend Time:** Time it takes to complete the blending.
- **Blend Func:** Type of function for blending.

Example Usage:

The event below belongs to a Level Blueprint and is triggered when the player enters a room. In this event the "Set View Target with Blend" function is used to change the game view for the camera that is in the room.



Timeline

Timeline allows the creation of simple time-based animations inside the Blueprints. After adding to the editor, the Timelines can be edited in the Blueprint editor by double-clicking them.

Input

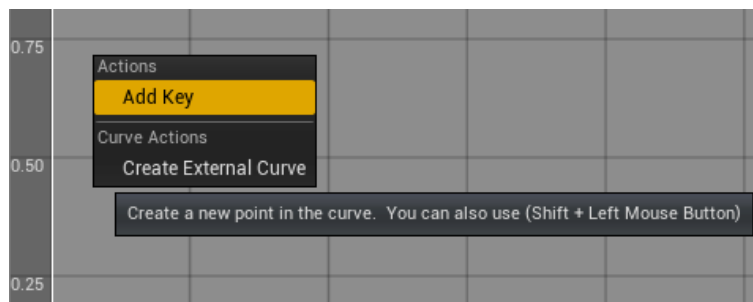
- **Play**: Play forward from its current position.
- **Play from Start**: Play forward from start.
- **Stop**: Stops the Timeline at the current time.
- **Reverse**: Reverse the Timeline from its current position.
- **Reverse from End**: Reverse the Timeline from end.
- **Set New Time**: Sets the current time with the value of parameter "New Time".
- **New Time**: The value in seconds used by "Set New Time".

Output

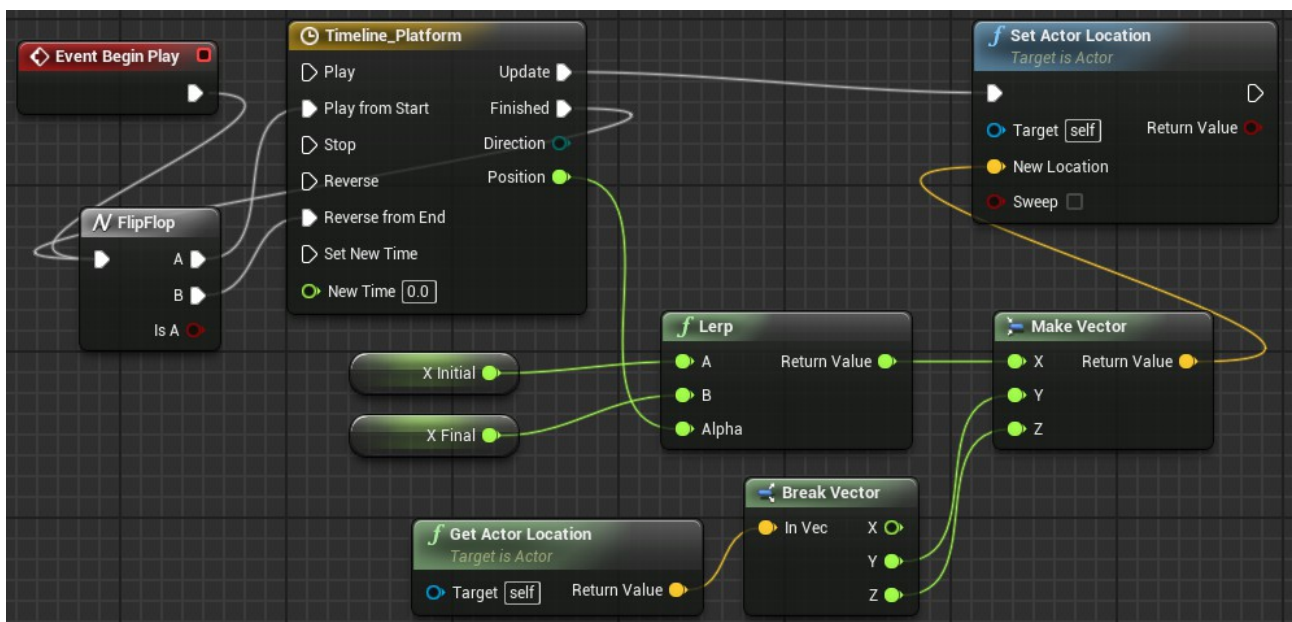
- **Update**: Execution pin called constantly while the timeline is running.
- **Finished**: Execution pin called when the playback ends.
- **Direction**: Indicate the direction of the timeline (forward or backward).

Example Usage:

After opening a timeline by double-clicking it, the keyframes of animation can be added by right-click in the center of the timeline and choose "Add Key".



This Blueprint is from a platform that moves between two positions along the X axis. The variables "X_Initial" and "X_Final" keep the positions where the platform will move in the world. The Flip Flop action is used to switch the execution of the Timeline between the normal direction and the reverse.



Math expression

This is a special type of action that generates a sub-graph from a mathematical expression informed.

Input

- **Expression:** Expression that will be analyzed.
- **Parameters defined in “Expression”:** For each variable name in the “Expression” is generated a new input parameter.

Output

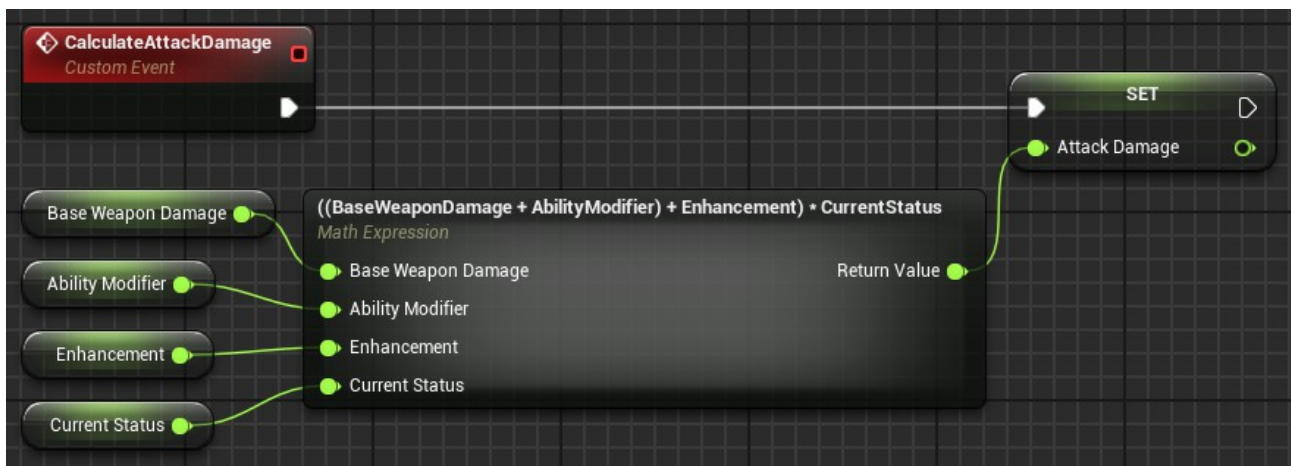
- **Return Value:** Result of the expression.

Example Usage:

The event below calculates the attack damage and stores the result in a variable. It used the "Math Expression" with the following expression:

$(\text{BaseWeaponDamage} + \text{AbilityModifier} + \text{Enhancement}) * \text{CurrentStatus}$

Based on this expression the input parameters were generated.



LineTraceForObjects

The `LineTraceForObjects` function tests the collision along a defined line and returns the first found object that belongs to one of the object types specified in the function call.

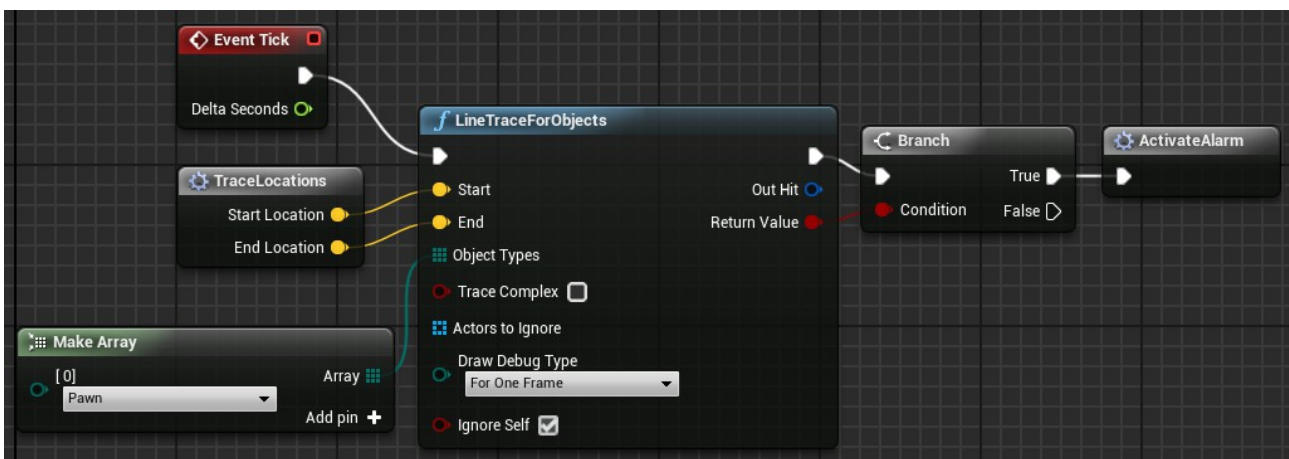
Input

- **Start:** Vector that defines the beginning of the line to be used for the collision test.
- **End:** Vector that defines the end of the collision test line.
- **Object Types:** Array that contains the types of objects that will be tested in the collision.
- **Trace Complex:** Boolean value indicating whether to use complex collisions.
- **Actors to Ignore:** Array with the actors that should be ignored in the collision tests.
- **Draw Debug Type:** It allows the drawing of the line that is being used in the collision test.
- **Ignore Self:** Boolean value that indicates whether the Blueprint that is calling the function should be ignored in the collision test.

Output

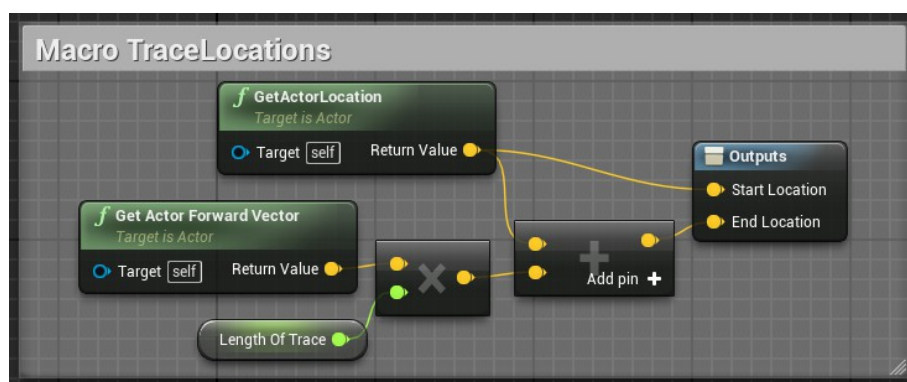
- **Out Hit:** Structure "HitResult" that contains various information about the collision found.
- **Return Value:** Boolean value. If true, it means that a collision was found.

Example Usage: On a blueprint called "AlarmDevice", the `LineTraceForObjects` function is used to activate an alarm if an object of the Pawn type is detected. The image below shows this example. A macro was used to get the start and end locations of the line.



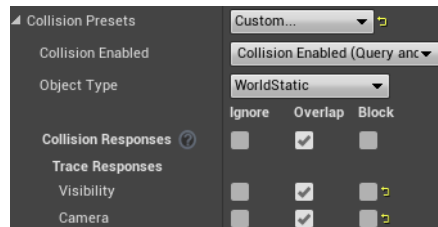
To create the array with the type of objects, drag off the pin "Object Types" and choose the "Make Array" option.

The `TraceLocations` macro looked like this:



MultiLineTraceByChannel

The *MultiLineTraceByChannel* function has two differences from the *LineTraceForObjects* function. It provides as output an array that can contain several elements as a result of the collision test. The collision test is based on the "Trace Channel". It can be seen in the details tab of a StaticMeshActor, in "Trace Responses" is defined how an object will respond to the trace test on the channels "Visibility" and "Camera". For example, a transparent material, will set the "Visibility" as "Overlap" because it can be viewed and does not block the line of sight. The function *MultiLineTraceByChannel* immediately ends if it finds an object with "Block" in the Trace Channel used.



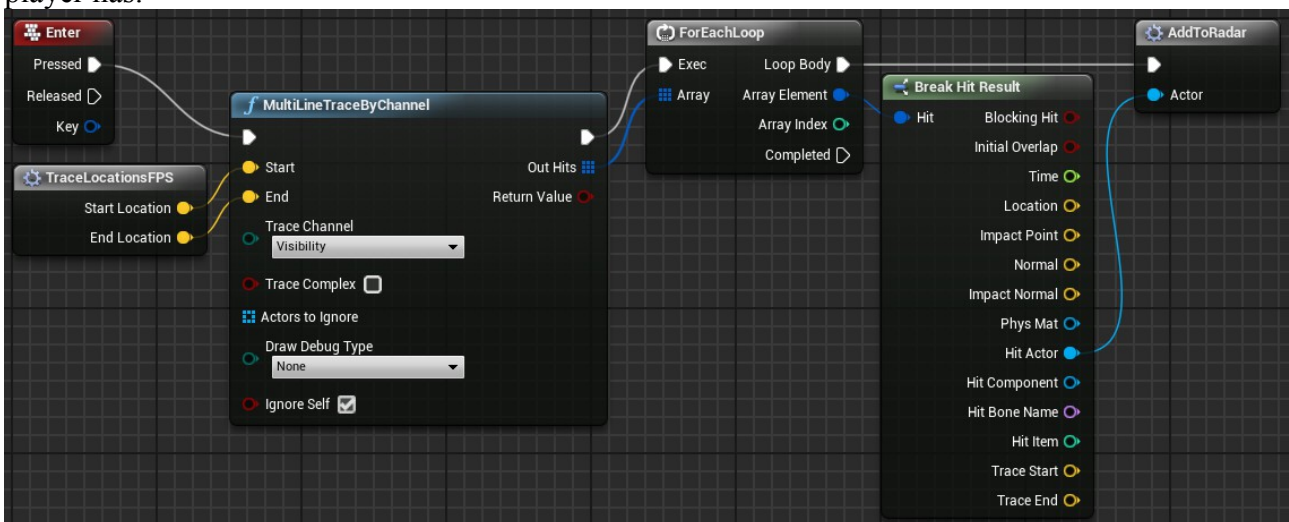
Input

- **Start:** Vector that defines the beginning of the line to be used for the collision test.
- **End:** Vector that defines the end of the collision test line.
- **Trace Channel:** Channel used for the collision test. It can be "Visibility" or "Camera".
- **Trace Complex, Actors to Ignore, Draw Debug Type and Ignore Self:** Are the same parameters used in the *LineTraceForObjects* function.

Output

- **Out Hits:** Array of structure "HitResult" containing the collisions that were found.
- **Return Value:** Boolean value. If true, it means that a collision was found.

Example Usage: The player has a device that is activated using the "Enter" key. This device uses the *MultiLineTraceByChannel* function to get objects visible and add to a radar system that the player has.



The macro *TraceLocationFPS* calculates the positions used in Trace based on the camera used by the player.

