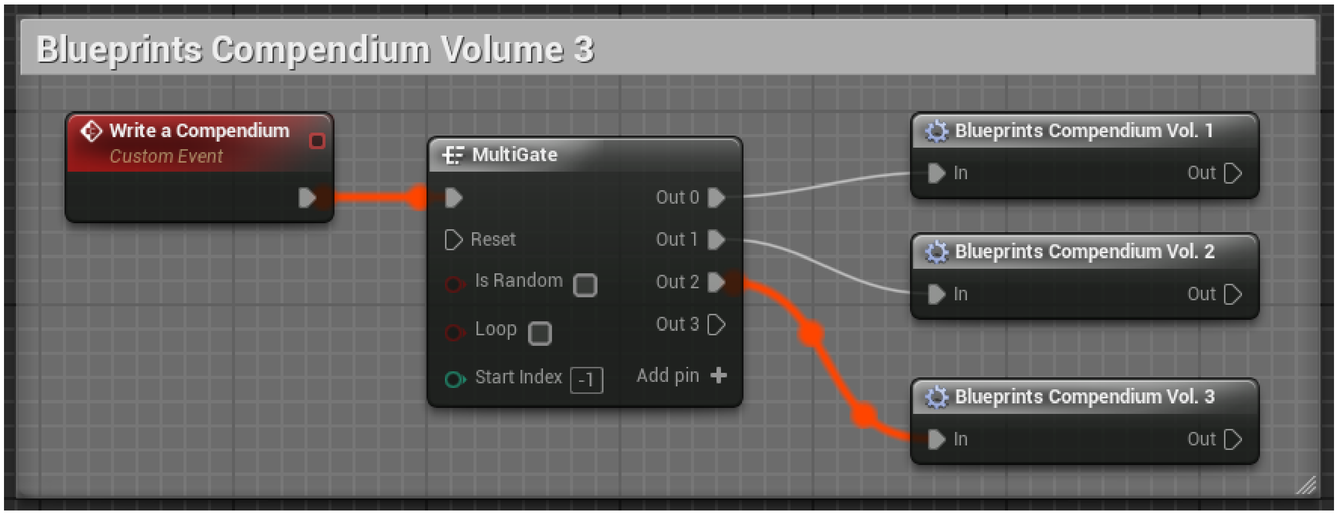
Blueprints Compendium

Volume 3


Blueprints Compendium Volume 3
Write a Compendium
Custom Event
MultiGate
Reset
Is Random
Loop
Start Index
-1
Out 0
Out 1
Out 2
Out 3
Add pin
Blueprints Compendium Vol. 1
Blueprints Compendium Vol. 2
Blueprints Compendium Vol. 3
In
Out

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 third volume of the Blueprints Compendium presents more 25 blueprints nodes with contextual examples of its use.

Table of Contents:

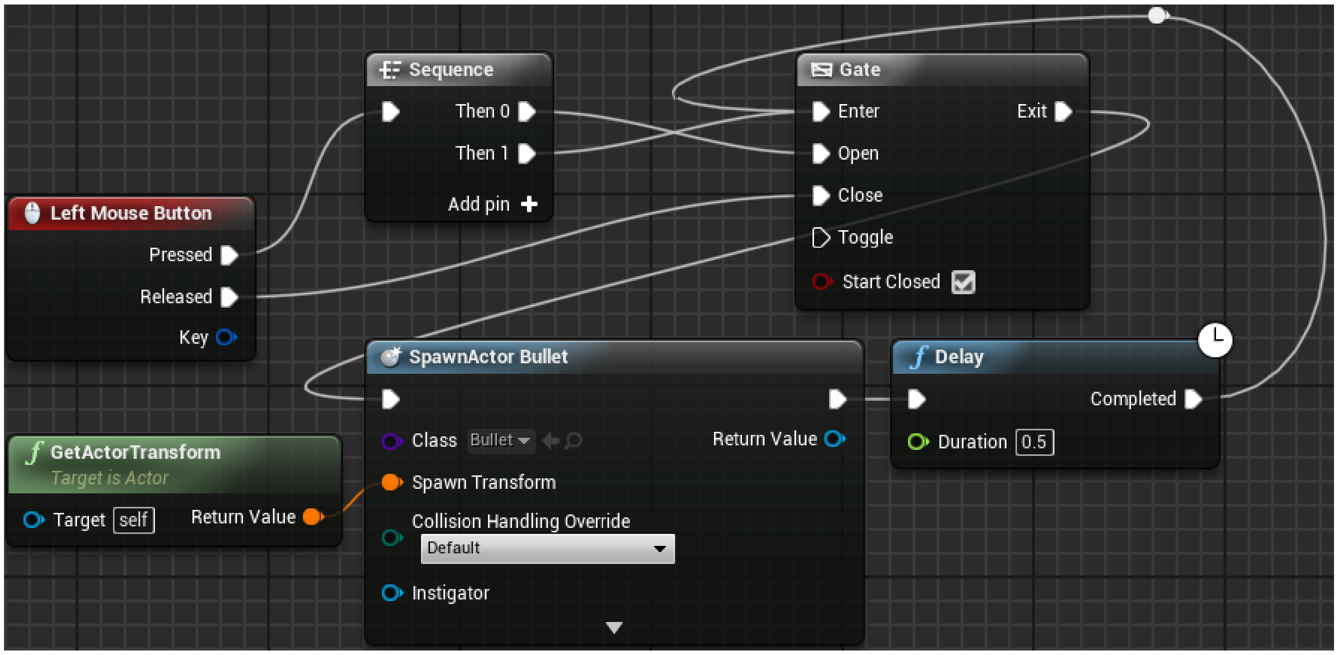
Sequence

A "Sequence" can be used to help organize other Blueprints Actions. When it is triggered it executes all the Actions connected to the output pins in order, that is, it executes all the actions of pin 0, then all the actions of pin 1 and so on.

Output pins can be added using the "Add pin +" option. To remove a pin, right-click the pin and choose the "Remove execution pin" option.

Example Usage:

The "Sequence" action is very simple to use, but if combined with other Actions it may present some interesting behaviors. The example below shows how to make a weapon that fires continuously when the left mouse button is pressed and only stops firing when the button is released.



The Sequence is being used in conjunction with the "Gate" Action. When the left mouse button is pressed, the pin 0 of the "Sequence" is used to open the "Gate" and allow the actions of the "Exit" pin to be executed when the "Enter" pin is triggered. After the "Gate" is open, the pin 1 of the "Sequence" continues execution through the "Enter" pin of the "Gate".

The Action "Spawn Actor Bullet" creates a bullet at the level based on a Blueprint called "Bullet". After this, the Action "Delay" waits half a second to enter again the "Enter" pin of the "Gate" and repeat the creation of the bullet.

This cycle will only be interrupted when the left mouse button is released causing the "Close" pin of the "Gate" to be triggered, preventing the actions of the "Exit" pin from being executed.

Flip Flop

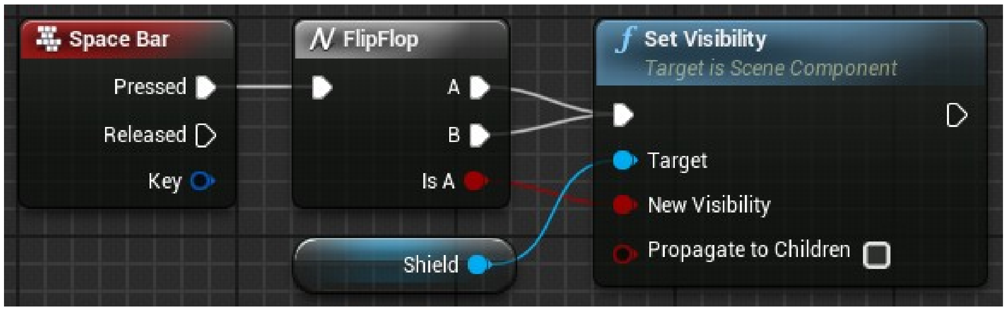
A Flip Flop has two output pins identified by A and B. When the Flip Flop is executed, only one of the output pins is executed. On the next run, only the other pin will be executed.

Output

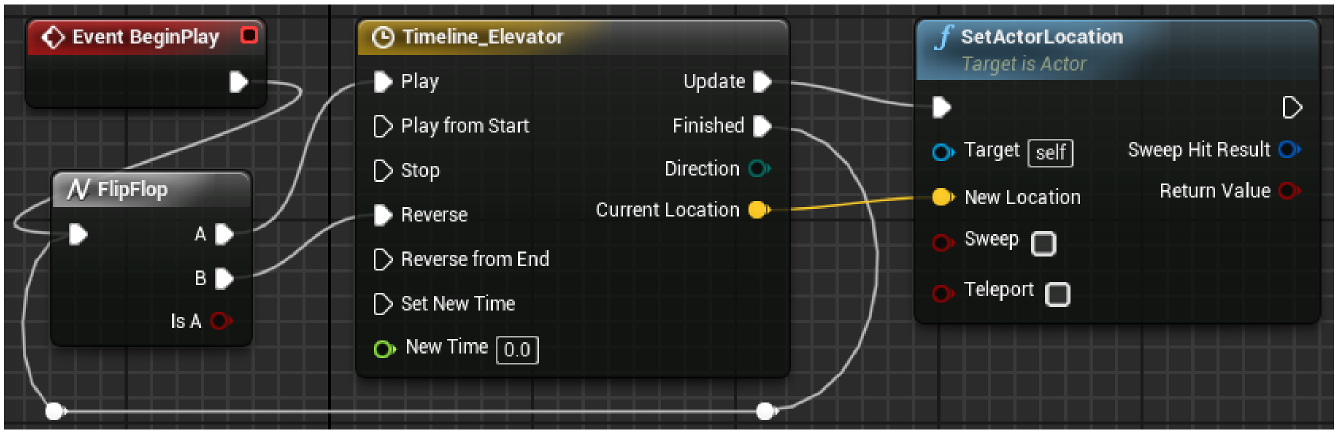
- **Is A**: Boolean value. If true, indicates that pin A is running. If false, pin B is running.

Example Usage:

In the image below the Flip Flop is being used to show or hide a shield when the space bar is pressed. The value of the variable "Is A" is being used to determine the visibility of the shield.



In another example, the Flip Flop is being used along with a Timeline to simulate the movement of an elevator. The first run of the Flip Flop uses the "Play" pin of the Timeline that moves the elevator up. At the end of the Timeline, the "Finished" pin executes the Flip Flop that this time will use the "Reverse" pin of the Timeline to move the elevator down. When the Timeline finishes the reverse movement, it starts all over again.



Do Once

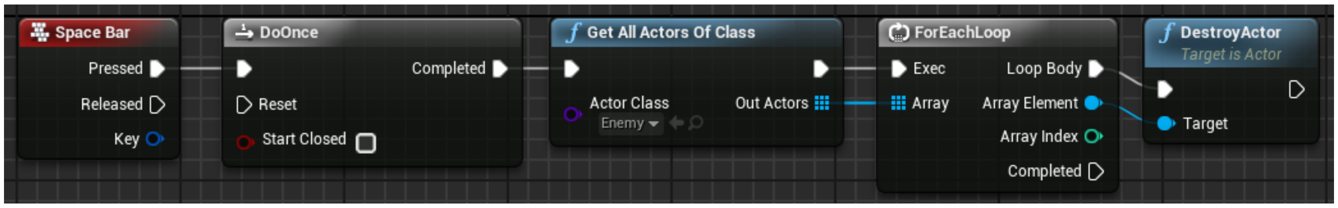
The "Do Once" action performs only once the actions attached to the output pin. After its first run, if the "Do Once" action is performed again, its output pin will not run. To allow the "Do Once" action to execute the output pin again, the "Reset" pin need to be triggered.

Input

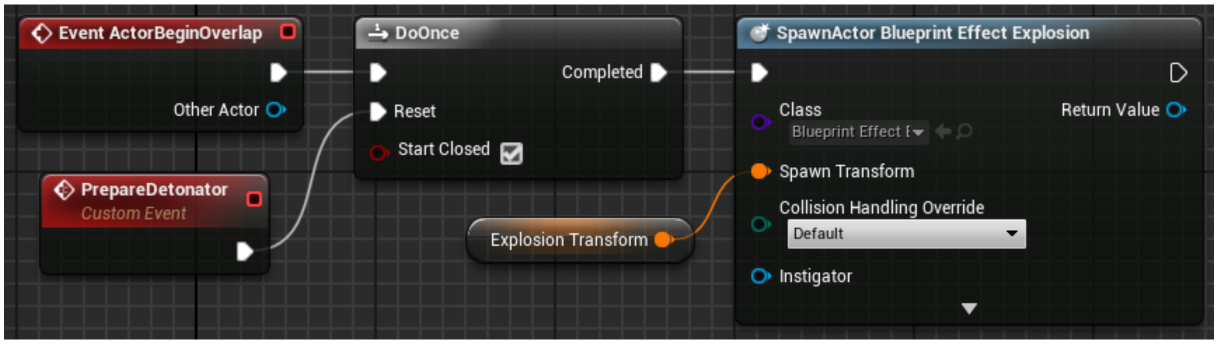
- **Reset**: Execution pin used to allow "Do Once" to run the output pin.
- **Start Closed:** Boolean value. If true indicates that "Do Once" needs to be reset to allow the first run.

Example Usage:

Imagine a special weapon that can only be used once per level. This weapon destroys all objects of the "Enemy" class that are currently at the level. The weapon is triggered by pressing the spacebar. After the first execution, even if the space bar is pressed several times, the "Do Once" action prevents the other actions that destroy the enemies from being executed.



In another example there is a detonator which generates an explosion when colliding with it. This detonator is controlled by the action "Do Once". The "Start Closed" property is marked indicating that the detonator starts disarmed. The "PrepareDetonator" event needs to be executed to trigger the Reset pin of the "Do Once" action. After this, if there is a collision with the detonator the explosion will be created. To allow a new explosion, the "PrepareDetonator" event needs to be executed again.



Do N

The "Do N" action is similar to the "Do Once" action, but instead of executing only once it is possible to inform how many times the actions connected to the output pin can be executed. After these executions, the actions of the output pin will only be executed again if the "Reset" pin is triggered.

Input

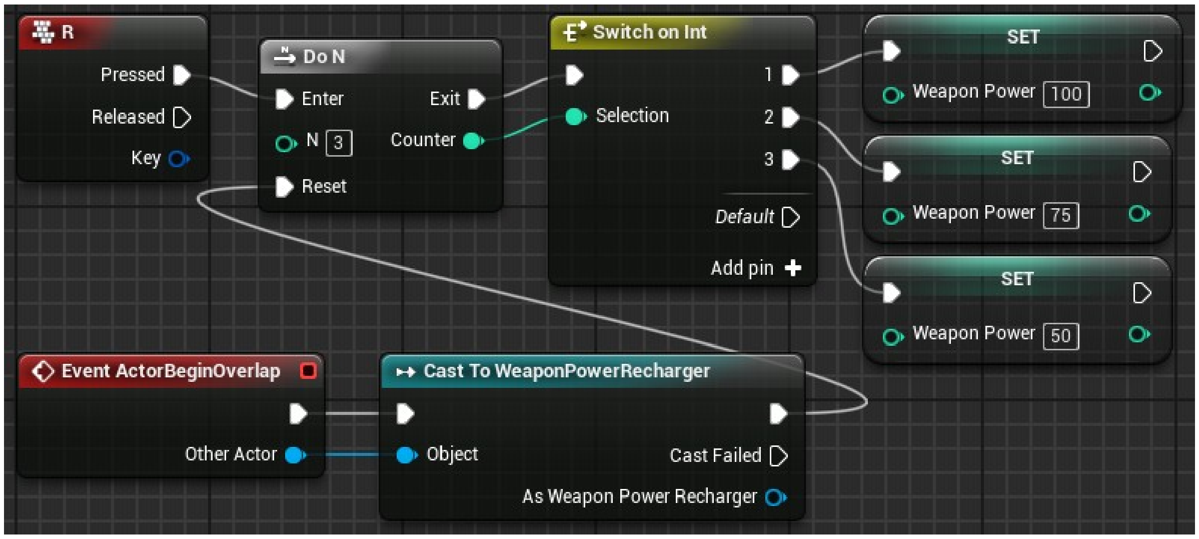
- **N**: Number of times the output pin actions can be executed.
- **Reset**: Execution pin used to restart the "Do N" count and allow new executions of the output pin.

Output

- **Counter**: Number that indicates the current execution.

Example Usage:

As an example, a player has a special weapon. When the weapon is fully charged, its "Power" is 100. The player can press the "R" key to recharge this weapon. This recharge can be done 3 times. The first recharge is complete, the second recharge leaves the "Power" of the weapon at 75 and the third recharge recovers only half the weapon's capacity. To recharge again, the player needs to collect an item of type "WeaponPowerRecharger".



The "R" event is triggered when the R key is pressed. It executes the "Do N" action that was created with N = 3, allowing up to 3 executions. After "Do N", the "Switch on Int" action is used with the "Counter" value of "Do N" to perform different actions according to the execution number. After the third execution, the "Do N" action will no longer execute the output pin when the R key is pressed. When a collision event occurs with an item of type "WeaponPowerRecharger", the "Reset" pin of "Do N" will be triggered allowing the weapon to be recharged again.

MultiGate

A "MultiGate" can have multiple output pins. At each execution of the "MultiGate" only one of the output pins is executed. The order that the output pins are executed may be in sequence or random. When all the output pins are executed and if the "Loop" option is not selected, "MultiGate" will stop executing the output pins. In order for the "Multigate" to run the output pins again, the "Reset" pin must be triggered.

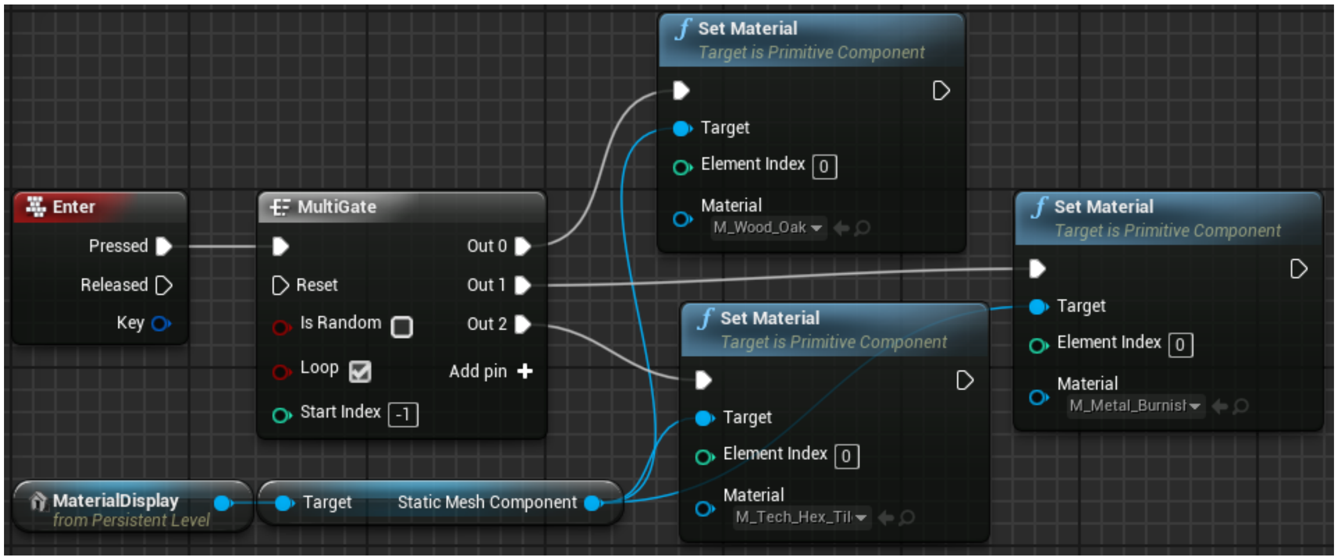
Output pins can be added using the "Add pin +" option. To remove a pin, right-click the pin and choose the "Remove execution pin" option.

Input

- **Reset**: Execution pin used to restart the MultiGate and allow new executions of the output pins.
- **Is Random:** Boolean value. If true it indicates that the order of execution of the output pins is random.
- **Loop:** Boolean value. If true, the "MultiGate" continues to execute the output pins after the last output pin is executed.
- **Start Index:** Integer value indicating the first output pin to be executed.

Example Usage:

At the level there is an object called "MaterialDisplay" whose function is to display various materials for the user. When the "Enter" key is pressed, a "MultiGate" is used to define a different material at each execution. As the "Loop" option is selected, after the last output pin is executed, "MultiGate" will return to the first output pin.



Important: "MultiGate" and other actions as "Gate", "Do Once", "Do N" and "Flip Flop", have internal variables to store their current state. Because of this, these actions do not work correctly within **Functions** because every time a function is executed, the internal variables used in this function are restarted.

BuildString

Converts a variable to a **String**. There are different versions of BuildString for the various types of variables like *integer, float, boolean* and *vector*. It also receives as parameters other strings to be used in the formation of the final string.

Input

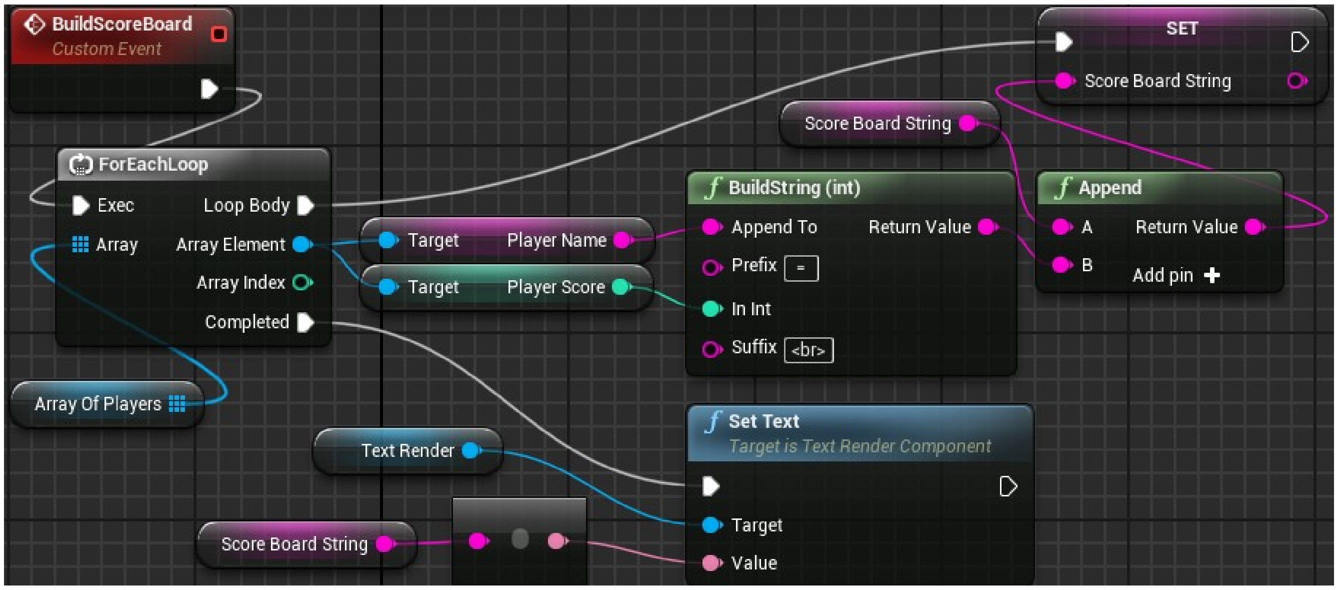
- **Append To**: String that will be used at the beginning of the final string.
- **Prefix**: String that will be used after the "Append To" in the final string.
- **In** *type*: Variable that will be converted to string and inserted after "Prefix".
- **Suffix**: String that will be inserted after the value of the variable.

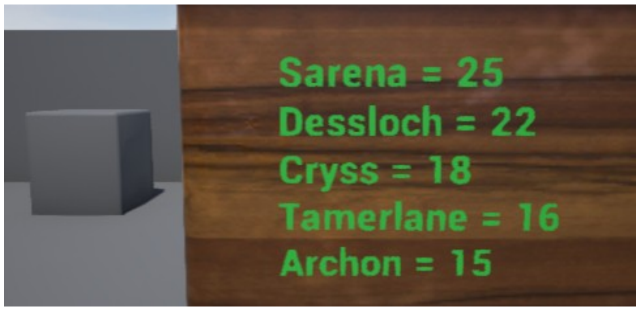
Output

- **Return Value**: String resulting from joining the input parameters: AppendTo + Prefix + In*type* + Suffix.

Example Usage:

In the example below, the "BuildScoreBoard" event prepares a string called "ScoreBoardString" that contains the name and score of each player in a match to be displayed on a scoreboard. This scoreboard is represented by a "Text Render" component. The "ForEachLoop" action is used to traverse the Array that holds the players' data. After completing the "ForEachLoop", the "Text Render" component defines its contents using the value of the "ScoreBoardString" variable.





The "BuildString" action creates a scoreboard line. The "Text Render" component interprets the <br> tag as line break.

Each line created is appended to the content of the "ScoreBoardString" variable using the "Append" action. These actions are repeated for each player.

Draw Text

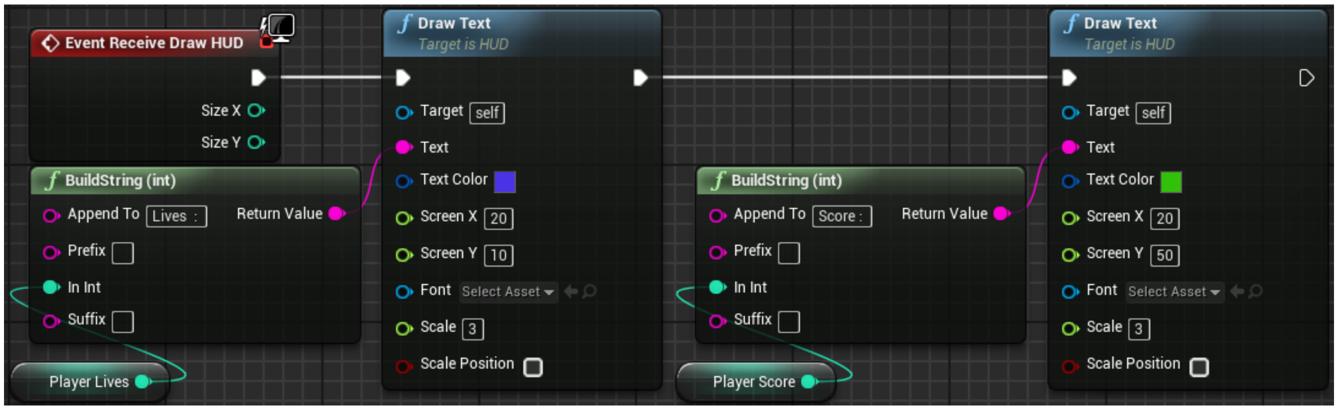
"Draw Text" is a function of the HUD class that is used to draw text on the screen.

Input

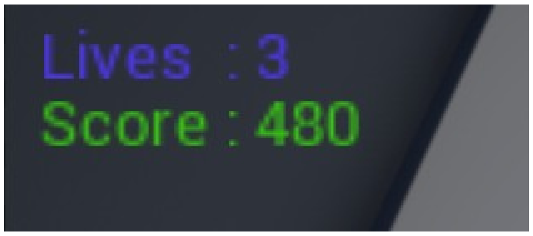
- **Text**: Variable of type String containing the text that will be drawn.
- **Text Color**: Color that will be used when drawing the text.
- **Screen X**: X position of the screen where the text will be drawn. The value of X starts at the left of the screen and increases to the right.
- **Screen Y**: Y position of the screen where the text will be drawn. The Y value starts at the top of the screen and increases from top to bottom.
- **Font**: Font used to draw the text. If none is chosen, the default font will be used.
- **Scale**: Used to change the size of the text.
- **Scale Position**: Indicates whether the scale should also affect the position of the text.

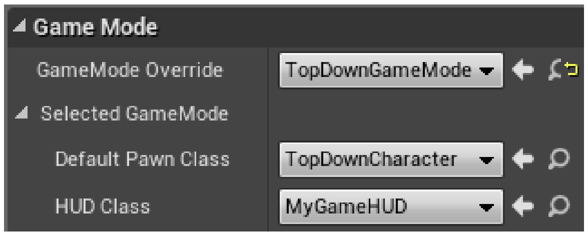
Example Usage:

A Blueprint called "MyGameHUD" was created using the HUD class as Parent. The "Draw Text" function must be used in the "Receive Draw HUD" event that belongs to the HUD class. Two "Draw Text" functions were used, one to display the number of lives and the other to display the score.



The number of lives is drawn in blue in the screen coordinate (X = 20, Y = 10) and the score is drawn in green in the screen coordinate (X = 20, Y = 50). To use this HUD class you must select it in the GameMode blueprints that are being used at the level or it can be configured in the level editor in the option "Settings → World Settings → Game Mode".


```
Lives : 3
Score : 480
```



Game Mode

GameMode Override	TopDownGameMode
Selected GameMode	
Default Pawn Class	TopDownCharacter
HUD Class	MyGameHUD

Enable Input

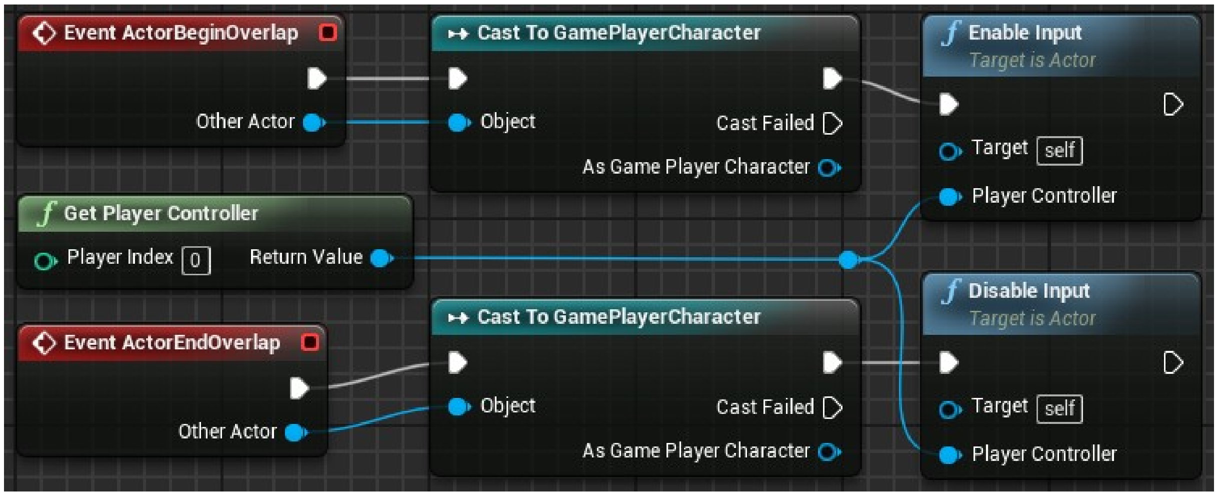
The "Enable Input" action is used to allow a Blueprint to respond to input events such as keyboard, mouse, and gamepad.

Input

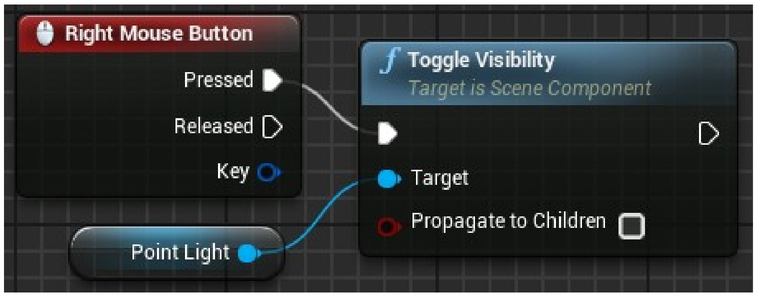
- **Player Controller**: A reference to the PlayerController in use.

Example Usage:

The image below represents a Blueprint that only responds to input events when the player is near it. This blueprint uses a "Box Collision" type component to define the area the player needs to be in order to interact. The player is represented by the Blueprint "GamePlayerCharacter", when the player is overlapping the Blueprint, the action "Enable Input" is used. When the player walks away, the "Disable Input" action is used so that Blueprint no longer responds to input events.



As an example of interaction, imagine that the Blueprint above represents a lamp. This Blueprint has a "Point Light" component. When the player is near the Blueprint and press the right mouse button, the light will be turned on or off.



Modulo (%)

The "Modulo" operation that is represented by the symbol "%" returns the remainder of a division.

Input

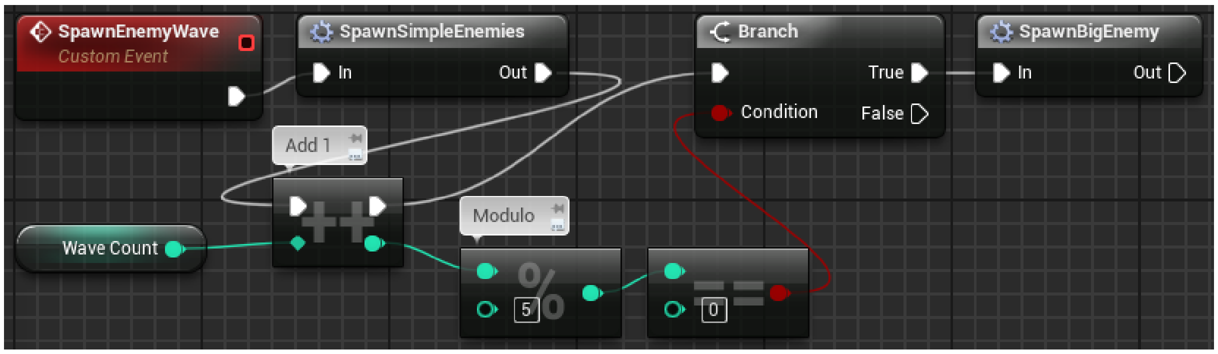
- **A**: Value to be divided.
- **B**: Value that will be used as the divisor.

Output

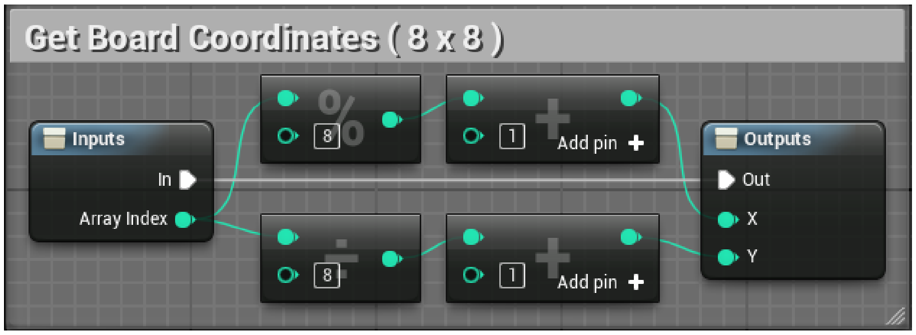
- **Return Value**: Remainder of the division.

Example Usage:

The "Modulo" operation with the integer type can be used to control the execution of certain actions that should only occur after an event is executed N times. In the example below there is an event that represents a new wave of enemies. There is an integer variable called "WaveCount" that keeps the current number of the wave of enemies. The "Modulo" is used to check when this variable is a multiple of 5. This is done by verifying if the remainder of the division of this variable with the value 5 is equal to zero. If this happens, a new type of enemy is created in this wave.



Another situation where it is common to use the "Modulo" is to convert the index of a one-dimensional array into a two-dimensional coordinate. For example, an array of integers is being used to represent a chessboard. The array contains 64 positions, so its indexes range from 0 to 63. Each position of the board is represented by a coordinate (X, Y). The first position has coordinates (X = 1, Y = 1) and the last position is (X = 8, Y = 8). The macro below converts an array index to the (X, Y) coordinate of the board.



Clamp

It receives an input value and the minimum and maximum values. If the input value is between minimum and maximum, it is returned without modification. If it is below the minimum, the minimum value is returned. If it is above the maximum, the maximum value is returned.

Input

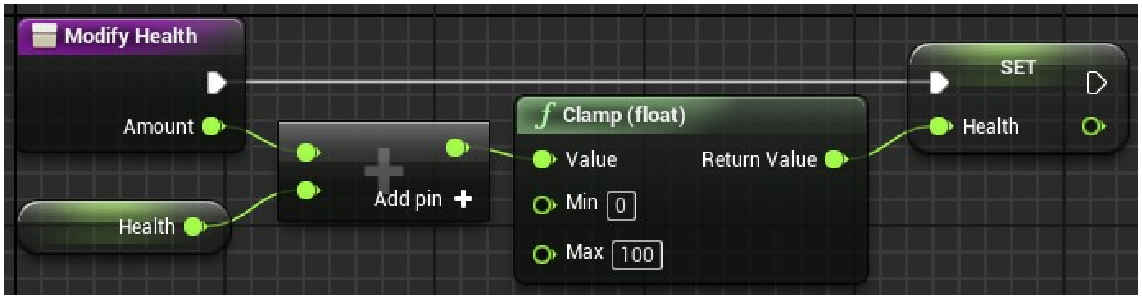
- **Value**: Input value that will be compared with the "Min" and "Max".
- **Min**: The lowest value that can be returned.
- **Max**: The highest value that can be returned.

Output

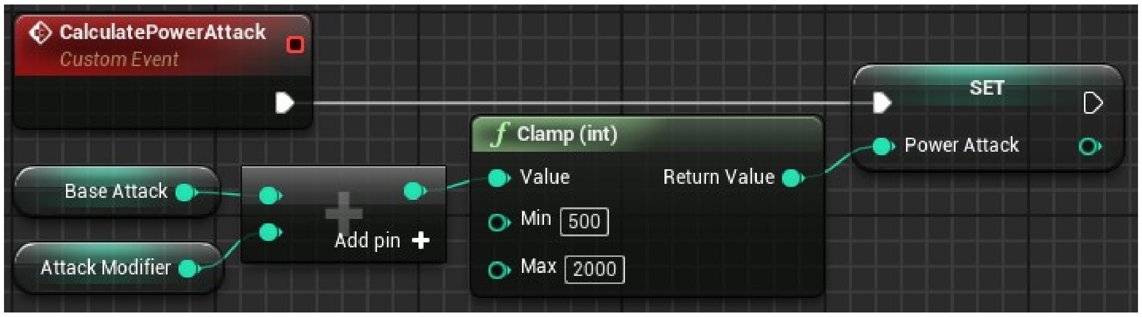
- **Return Value**: Value between "Min" and "Max".

Example Usage:

In this example there is a variable called "Health". The value of this variable must be between 0 and 100. A function was created that should be used to modifier the value of the "Health" variable. This function has the parameter "Amount". The value of "Amount" can be negative indicating damage or can be positive when the player picks up a health pack. The "Clamp" is used to ensure that the "Health" variable is between 0 and 100.



In another example there is an event that calculates the power attack of a character. This power attack is based on the sum of the variables "Base Attack" and "Attack Modifier". The variable "Attack Modifier" can indicate attack bonuses or it can have negative value indicating some bad state of the character. But no matter how bad the character's attack penalties are, the minimum power attack is 500. Similarly, the maximum value of the power attack is 2000, even if the sum of the base attack with the attack bonus was higher.



Interp To

Set of functions used to change a value smoothly until it reaches the specified target value. Some examples: "FInterp To" for values of type "float", "VInterp To" for vectors and "RInterp To" for "rotators".

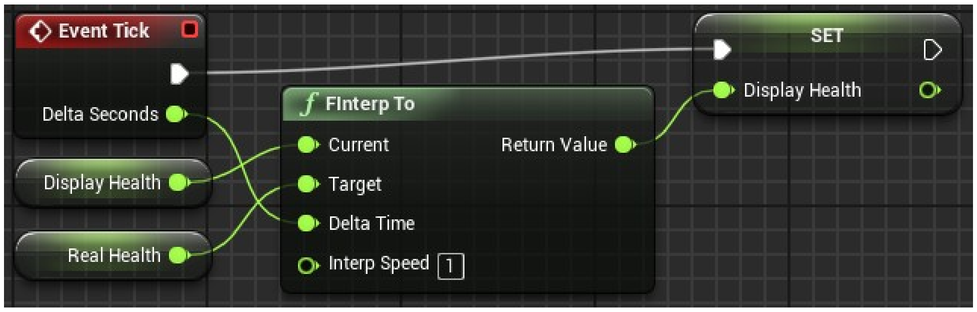
Input

- **Current**: Current value that will be modified.
- **Target**: Target value to be pursued.
- **Delta Time**: The time interval that has elapsed since the last execution.
- **Interp Speed**: Interpolation speed.

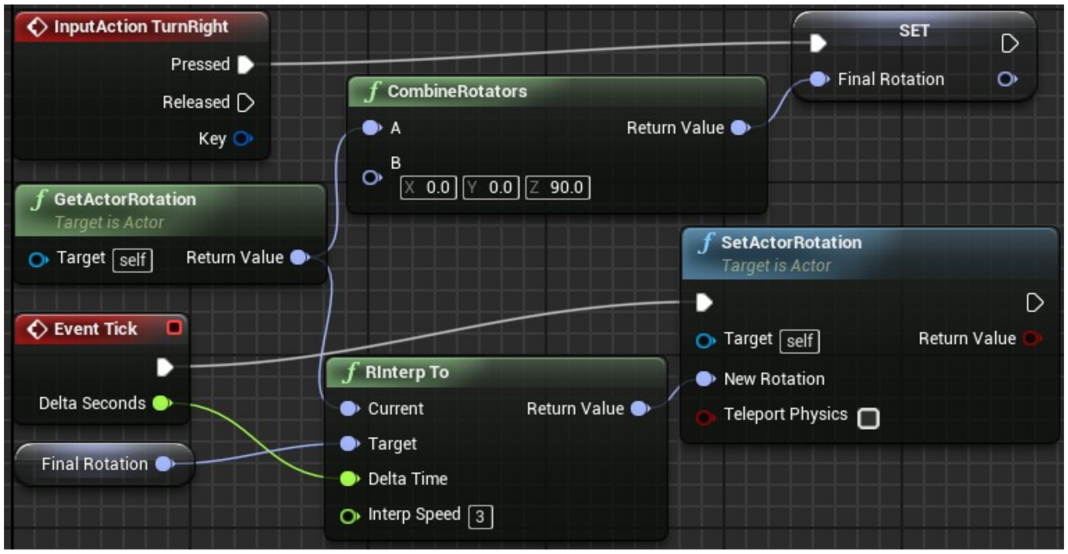
Output

- **Return Value**: New value closer to the target value.

Example Usage:
The image below has two variables. The "Real Health" variable contains the player's current health. The "Display Health" variable is used to display a health bar on the screen. When the player suffers damage, the value of "Real Health" is changed immediately, but the value of "Display Health" is modified using "FInterp To" so that the health bar smoothly decreases until "Display Health" is equal to "Real Health".



In another example, the "TurnRight" event commands a robot to turn 90 degrees to the right. This rotation is done smoothly using the "RInterp To".



Get Actor Forward Vector

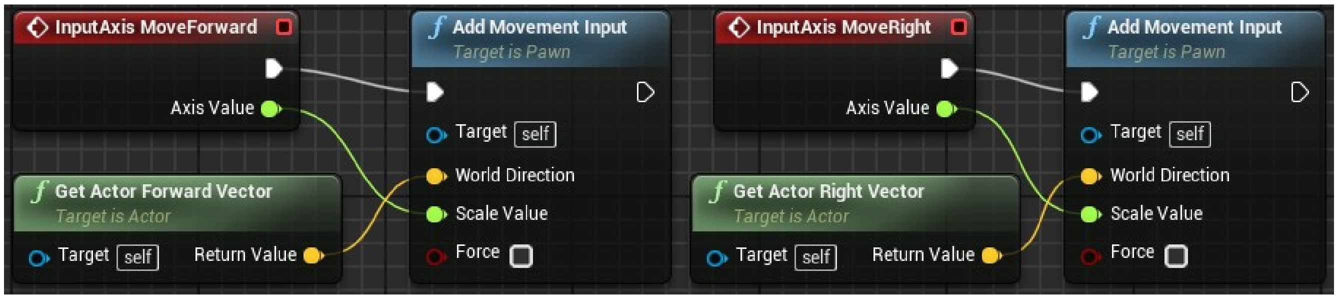
Function that returns a normalized vector (length = 1) that represents the direction an actor is pointing to.

Output

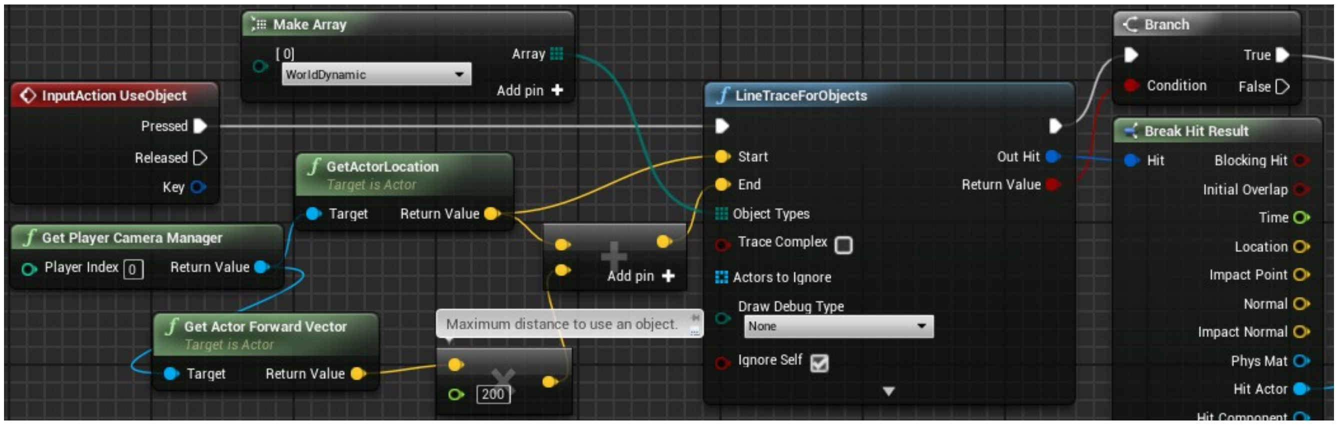
- **Return Value**: The vector indicating the direction of the actor.

Example Usage:

This image shows how the player moves in the "First Person Template". The "InputAxis MoveForward" event uses the "Get Actor Forward Vector" to move the player forward or backward based on the value of "Axis Value". The "InputAxis MoveRight" event uses the "Get Actor Right Vector" to move right or left. For example, using the standard WASD keys for movement, the W key activates the "MoveForward" event using Axis Value = 1 and the S key uses the "MoveForward" with Axis Value = -1 to reverse the direction. The "InputAxis MoveRight" event is triggered by the D key (Axis Value = 1) and the A key (Axis Value = -1).



"Get Actor Forward Vector" is also widely used to perform "Line Traces" in order to check if a shot has hit a target. Another type of "Line Trace" that is being used below is to check if it has an object nearby (2 meters) and in front of the player to interact.



If the "LineTraceForObjects" function find an object, it is returned in a "Hit Result" structure. A reference to the found object can be obtained in the "Hit Actor" variable of the "Hit Result" structure.

Set Material

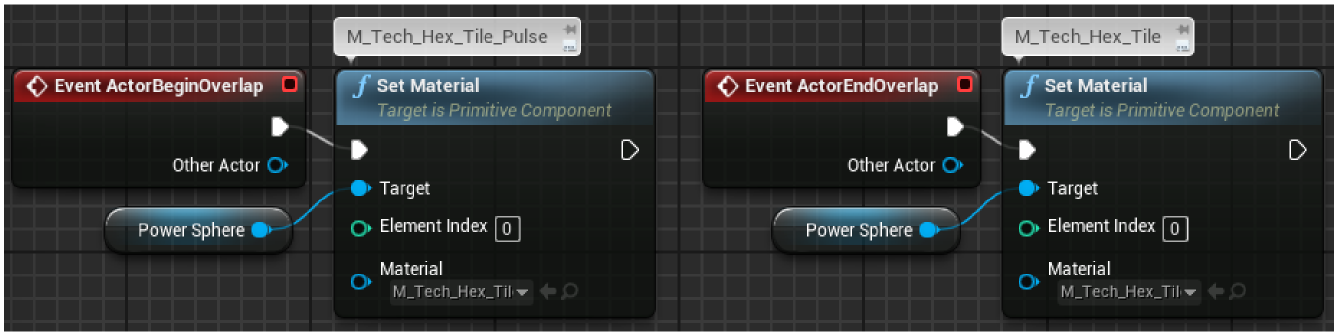
Changes the material being used by a component.

Input

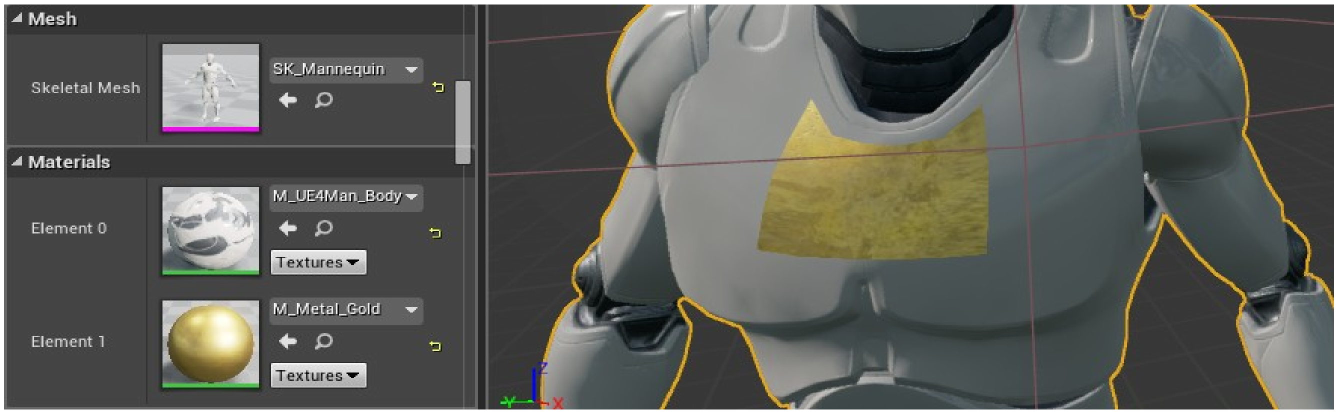
- **Element Index**: This index indicates which material will be modified in the component.
- **Material**: Reference to the new material that will be used.

Example Usage:

A Blueprint contains a "Static Mesh" called "Power Sphere". When an actor approaches the Blueprint, the "Power Sphere" material is modified to indicate that it is active. When the actor moves away, the material is modified to its initial value indicating that it is disabled.



A component may have several materials associated with it. In this case it is necessary to inform in the parameter "Element Index" which material will be modified. The image below shows the Skeletal Mesh used in the Unreal Engine templates. The material with Element Index = 0 is used throughout the body of the character. The material with Element Index = 1 had the Unreal logo, but was modified to a gold-like material to illustrate the position where it is applied.



Set Input Mode Game and UI

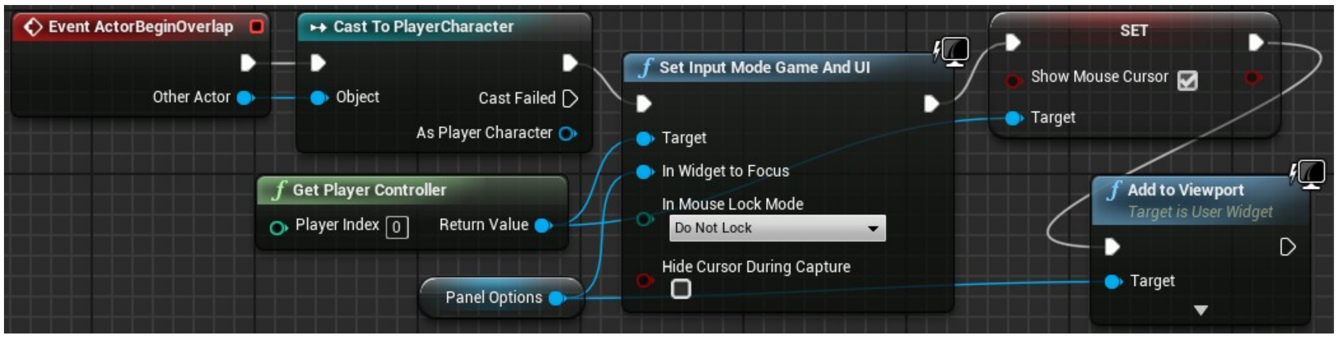
Defines an input mode in which the User Interface (UI) has priority in handling user input events. If the UI does not know how to handle a particular event, it is passed on to the Player Controller to be handled.

Input

- **Target:** A reference to the PlayerController.
- **In Widget to Focus**: A reference to an UMG Widget.
- **In Mouse Lock Mode**: Indicates how the mouse will be handled.
- **Hide Cursor During Capture**: Indicates whether the mouse cursor should be hidden.

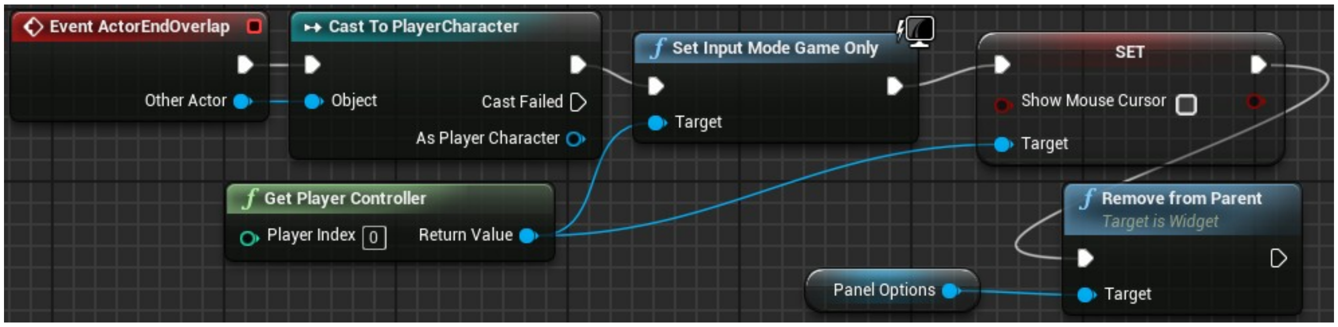
Example Usage:

In a "First Person" game where the mouse is used to modify the player's vision, there is a blueprint that interacts with the player. When the player approaches this blueprint, a panel is displayed with options for the player to choose using the mouse. The "Set Input Mode Game and UI" function causes the mouse to control the cursor so that it can interact with the panel instead of changing the player's view. To exit the panel, simply move away from the blueprint using the WASD movement keys.



The "Panel Options" variable represents a UMG Widget that was previously created in the "Begin Play" event. As the player approaches the blueprint, the "Panel Options" is displayed on the screen via the "Add to Viewport" function.

The event below shows the actions required to return to normal input game mode and hide the panel when the player moves away from the blueprint.



Teleport

Moves an actor to the specified location. If there is an obstacle in the location, the actor is moved to a nearby place where there is no collision.

Input

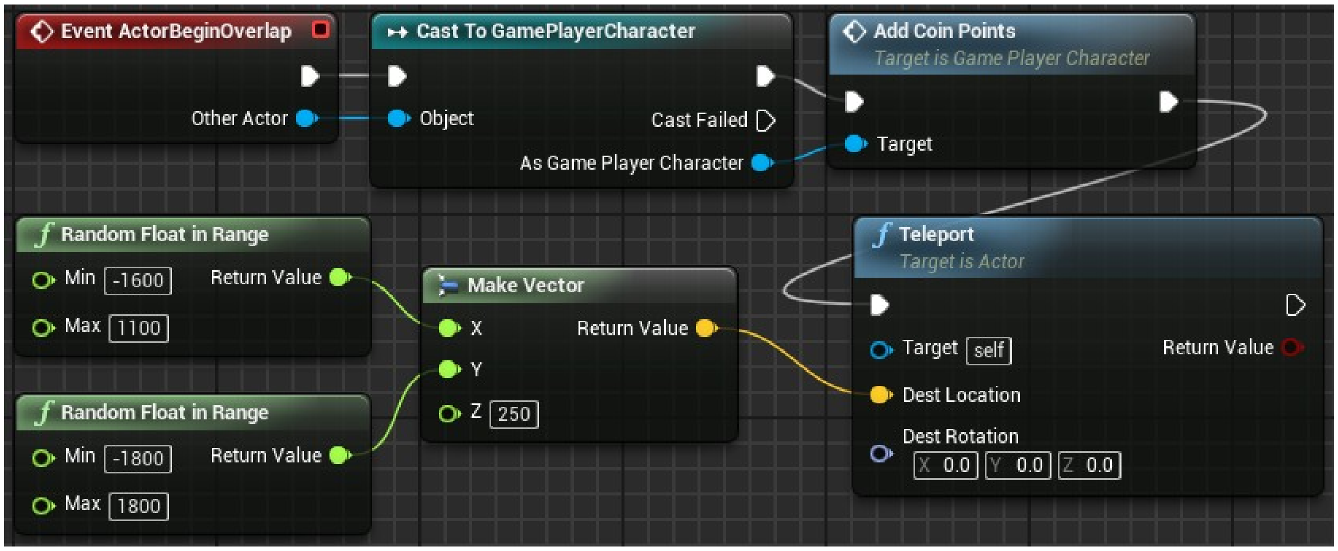
- **Dest Location**: Destination location.
- **Dest Rotation**: Rotation applied to the actor.

Output

- **Return Value**: Boolean value. If "false" indicates that the actor could not be moved.

Example Usage:

In a game, the player collects coins. The image below shows the "ActorBeginOverlap" event of the blueprint that represents the coins. When a coin is collected the "AddCoinPoints" function of the player is called to register the points and after that the coin is teleported to another location in the playing area. The playing area has several obstacles, but the "Teleport" function will place the coin in a free place.



The playing area is defined by the values of X > -1600 and X < 1100, Y > -1800 and Y < 1800. The "Random Float in Range" function returns a random float value that is between the "Min" And "Max" that are passed as parameter.

The "Make Vector" creates a vector using the random values that were found for X and Y. This vector is used as the destination in the "Teleport" function.

Random Point in Bounding Box

A function that returns a vector representing a random point located within a volume specified by a origin point and a "Box Extent".

Input

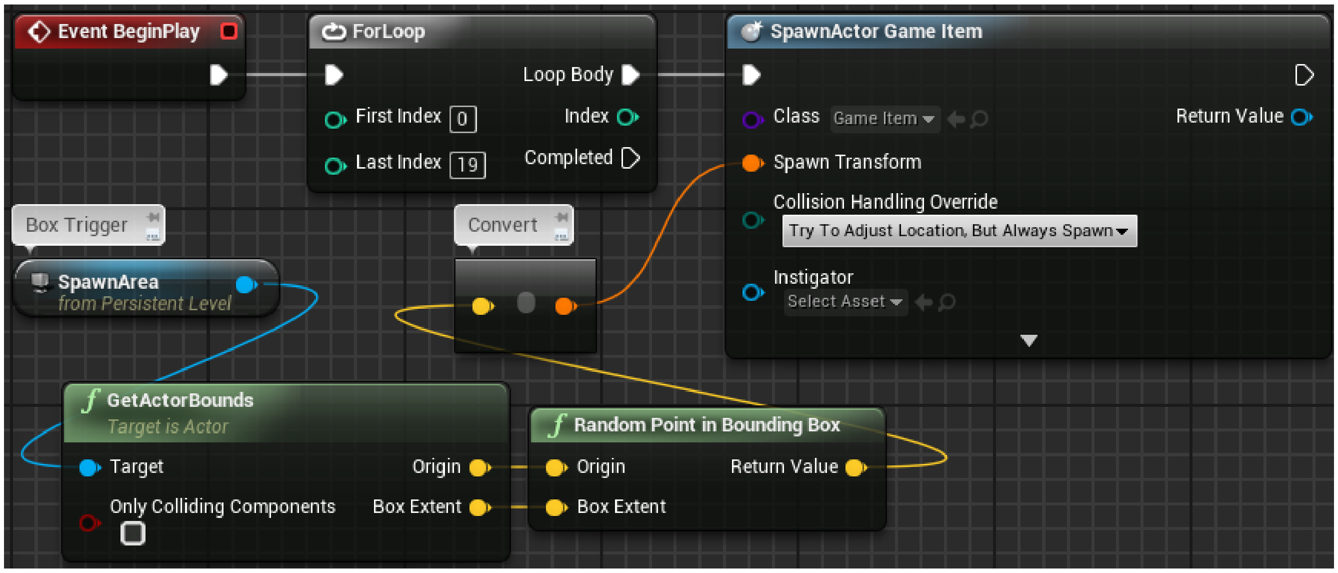
- **Origin**: Vector representing the central position of the "Box Extent".
- **Box Extent**: Vector that contains the dimensions of the box that defines the 3D volume.

Output

- **Return Value**: Vector representing a point that was generated randomly inside the "Box Extent" informed.

Example Usage:

The image below is from a "Level Blueprint". A "Box Trigger" named "SpawnArea" has been placed on the level to define an area where items will be created at random positions within this area. When the game starts, the "ForLoop" action repeats the "SpawnActor" action 20 times to create 20 items in random positions. The position of each item is obtained with the "Random Point in Bounding Box" function using the "Origin" and "Box Extent" of the "SpawnArea".


First Index 0
Last Index 19
Class Game Item
Collision Handling Override
Try To Adjust Location, But Always Spawn
Instigator Select Asset

The "SpawnActor" action needs a "Transform" type structure to apply to the actor that will be created. When trying to connect a vector to a "Transform" the editor automatically creates a converter using the vector as "Location". The other attributes of the "Transform" are set to their default values.

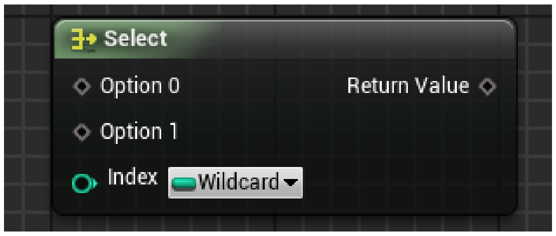
In the "Collision Handling Override" parameter of "SpawnActor" the "Try to Adjust Location, But Always Spawn" option was used. This causes the "SpawnActor" to check for any obstacles in the position where the actor will be created. If there is an obstacle, the actor will be moved to a nearby free location.

Select

The "Select" action returns one of the values associated with the options that corresponds to the index that is passed as input. This "Select" is generic and can work with several types of variables for the index and for the values of the options. To change the type of the options or the index, right click on the parameter and choose the option "Change Pin Type". To add more options, right-click on the node and choose "Add Option Pin".

Input

- **Option 0, 1…** : Values that can be returned by the Select. The options can be of any type.
- **Index**: Value that will determine the option to be returned. The "Index" can be of the type Boolean, Byte, Integer or Enum.

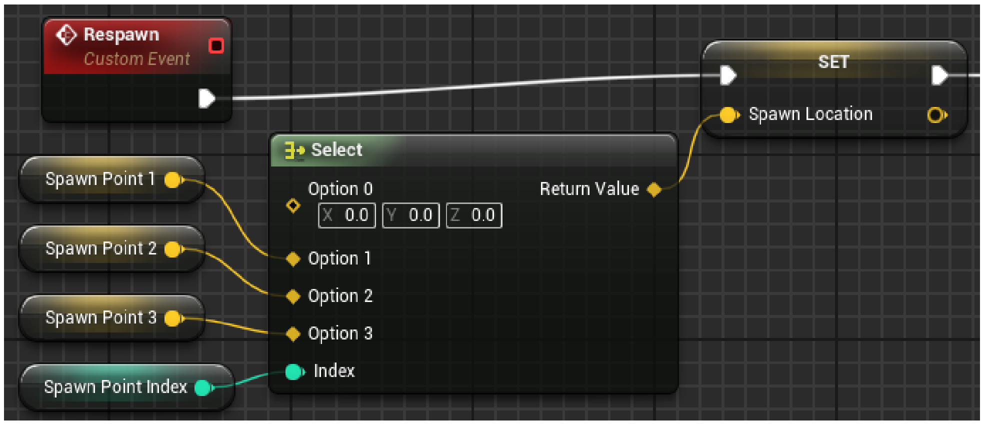


Output

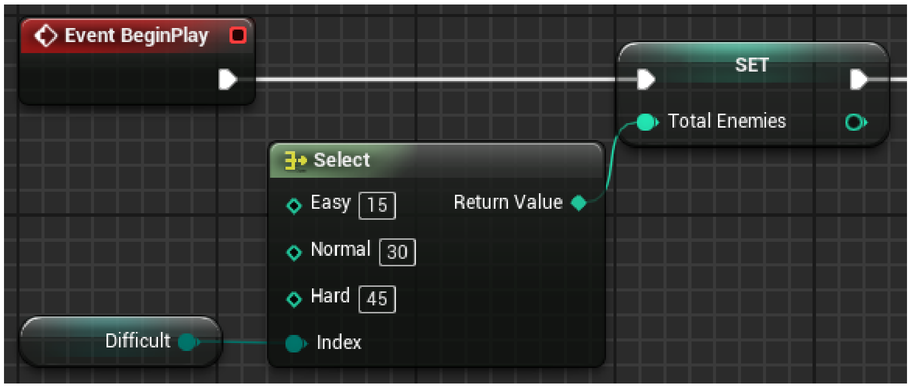
- **Return Value**: The option indicated by the "Index".

Example Usage:

In a "Respawn" event, the "Select" is being used to get the location of a spawn point based on the value of the integer variable "Spawn Point Index".



In another example the "Index" is an Enum called "Difficult" that has the Easy, Normal and Hard values. The type of options is "Integer" and represents the number of enemies according to the difficulty.



Map Range Clamped

Converts a value that is in a range of values to the corresponding value in another range of values. The end result will always be in the range of output values even if the original value is outside the range of input values.

Input

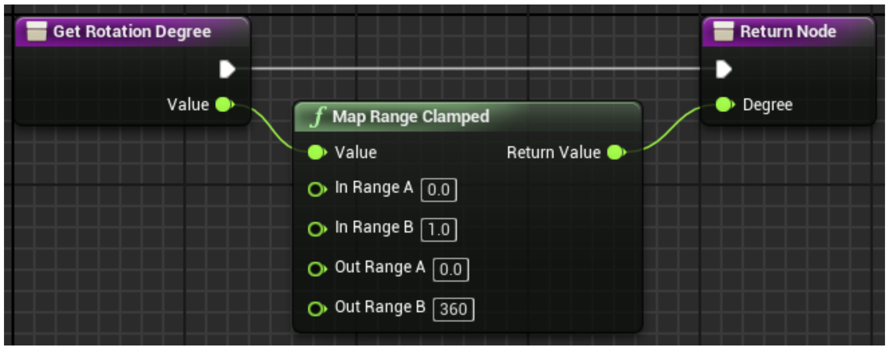
- **Value**: Original value to be converted.
- **In Range A**: Minimum value of the input value range.
- **In Range B**: Maximum value of the input value range.
- **Out Range A**: Minimum value of the output value range.
- **Out Range B**: Maximum value of the output value range.

Output

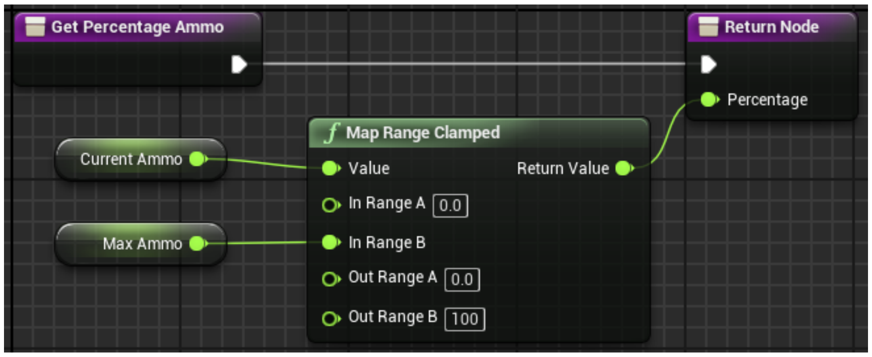
- **Return Value**: Converted value to the output value range.

Example Usage:

In the example below, a float variable is being used to represent the rotation of an object. This variable can contain values between 0.0 and 1.0. A function was created to convert the value of this variable to the equivalent value in degrees. If the value is 0.0 then the equivalent in degrees is 0. If it is 1.0 the equivalent in degrees is 360. If the value is 0.5 then the result will be 180 degrees.



In another example, the "Get Percentage Ammo" function converts the current amount of the player's ammo into a percentage so it is displayed on the screen. The calculation is based on variables that hold the maximum ammunition that the player can have and the current amount of ammunition.



There is a variation of this function called "**Map Range Unclamped**", the only difference being that the end result is not limited to the range of output values.

Actor Has Tag

This function checks if a game Actor has a certain Tag. The use of tags is a simple way to differentiate some actors in the level.

Input

- **Target**: Actor reference that will be used in the tag verification.
- **Tag**: Tag that will be used in the test.

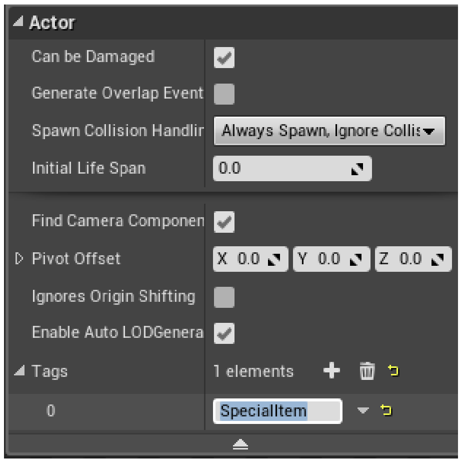
Output

- **Return Value**: Boolean value. If "true" indicates that the actor has the tag informed.

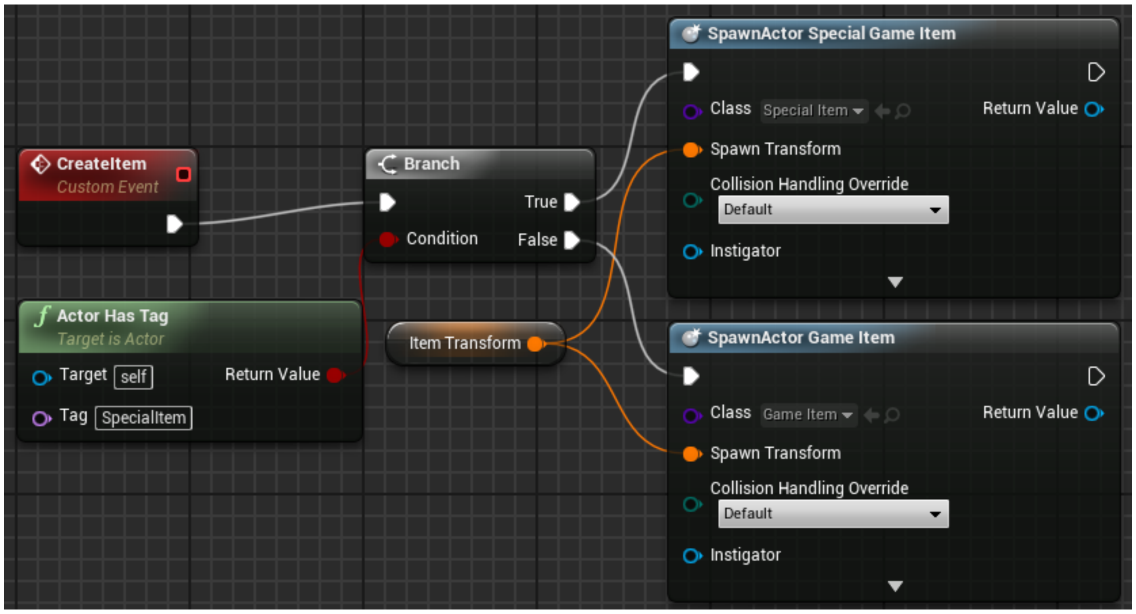
Example Usage:

An Actor can have multiple **Tags**. To add Tags to an Actor in the Blueprints Editor, click the "Class Defaults" button and in the "Details" tab look for the "Actor" category.

This image shows that a tag called "SpecialItem" has been added. Tags can also be added in the level editor, just click on an Actor that is on the level and modify the Tags in the "Details" tab.


Actor
Can be Damaged ✓
Generate Overlap Event
Spawn Collision Handlir Always Spawn, Ignore Collis
Initial Life Span 0.0
Find Camera Componen ✓
Pivot Offset X 0.0 Y 0.0 Z 0.0
Ignores Origin Shifting
Enable Auto LODGenera ✓
Tags 1 elements
0 SpecialItem

In this example, the "Actor Has Tag" function is used to check if the actor has the "SpecialItem" tag. If it has then a new actor of "Special Game Item" type will be created, otherwise the actor created is a "Game Item" type.



Apply Damage

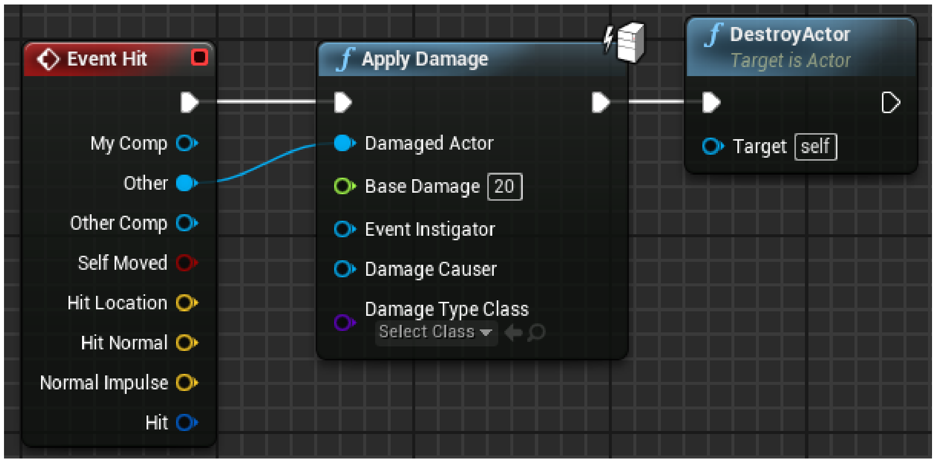
This function is used to apply damage to an Actor. Various information related to the damage can be passed through the function. The hit Actor responds to this damage using the "AnyDamage" event.

Input

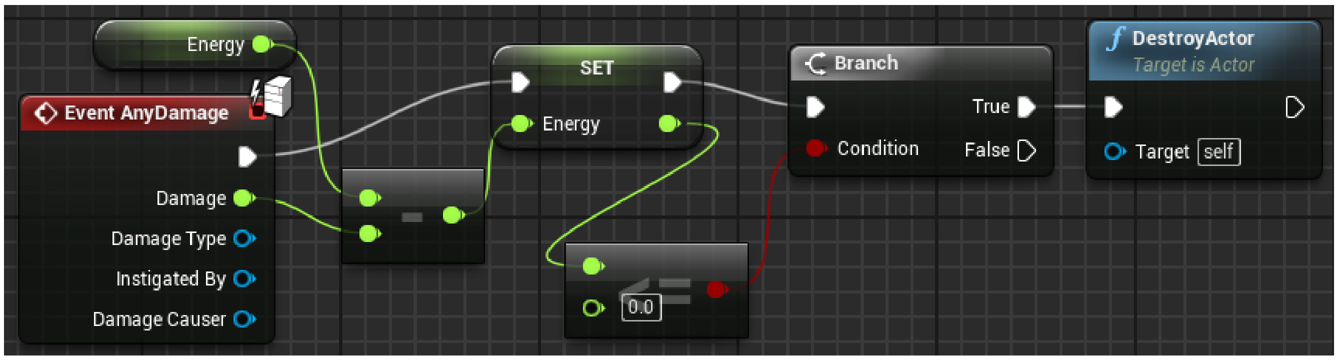
- **Damaged Actor**: Reference to the Actor who will suffer the damage.
- **Base Damage**: Value of float type that represents the damage.
- **Event Instigator**: Optional. Reference to the Controller responsible for the damage.
- **Damage Causer**: Optional. Reference to the Actor which caused the damage.
- **Damage Type Class**: Optional. Class that represents the type of damage. E.g. Fire, Electricity.

Example Usage:

This simple example can be used on a blueprint that represents a bullet. When the bullet hits an Actor, the "Apply Damage" function is used to deal a damage with a value of 20. After that the bullet is destroyed.



In the Actor Blueprint that was hit, a float variable with the name "Energy" was created. The "AnyDamage" event will be activated by the "Apply Damage" function. The damage is subtracted from the "Energy" variable. If "Energy" is less than or equal to zero, the Actor will be destroyed.



There are more specific versions such as the "Apply Point Damage" and "Apply Radial Damage" functions. For each of them there is an associated event.

Get Overlapping Actors

This function returns a list of actors that are overlapping the Actor/Component. There are two versions of this function, one that uses an Actor as Target and the other that uses a Component as Target.

Input

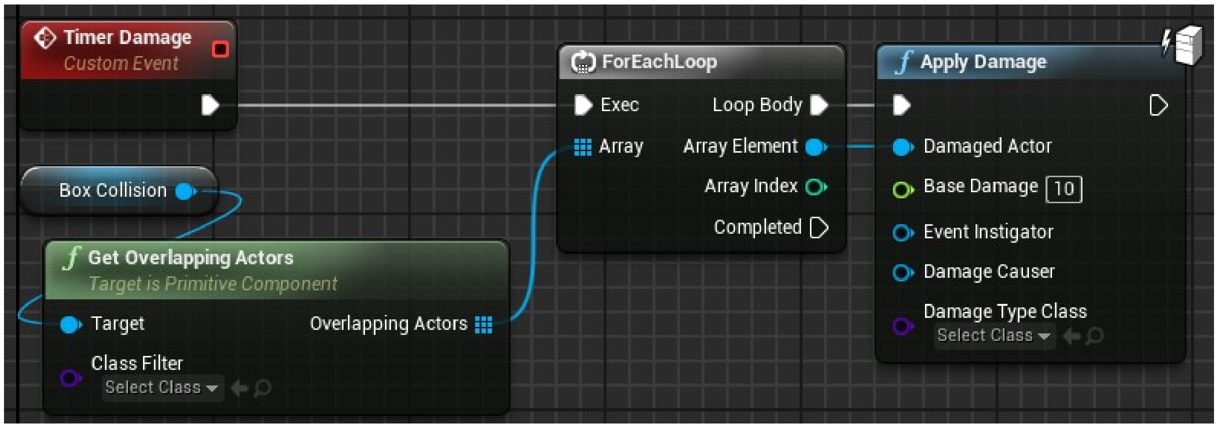
- **Target**: Reference to the Actor/Component that will be used in the overlapping test.
- **Class Filter**: Optional. Indicates which class/subclasses will be used in the test.

Output

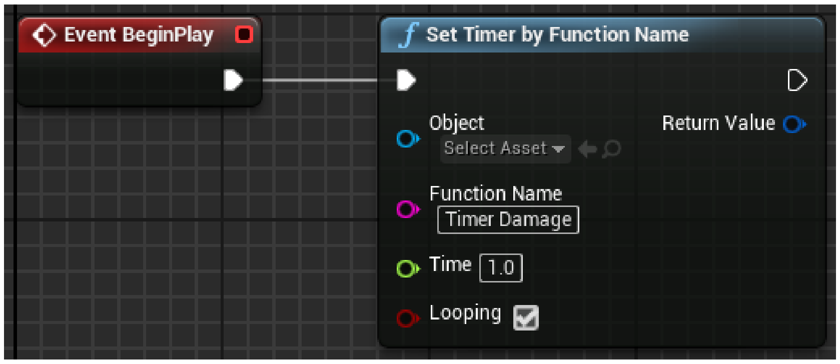
- **Overlapping Actors**: Array containing the Actors that are overlapping the Actor/Component.

Example Usage:

Imagine an area that deals damage to all the actors in it. This area is represented by a Blueprint that has a Box Collision component that defines the location where the actors suffer damage. The event below is periodically called to apply damage to all actors who are overlapping the Box Collision.



The image below shows the Timer that was set to call the "Timer Damage" event once per second.



Add Child Actor Component

Adds an Actor as a component of another Actor. Thus the component actor follows the transformations of the parent Actor. When the parent Actor is destroyed, the Actor component will also be destroyed. The new actor's class must be specified in the "Details" tab of the "Add Child Actor Component" function.

Input

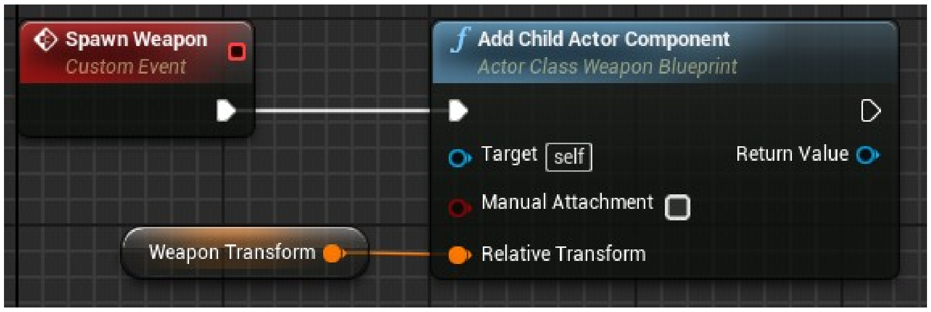
- **Target**: Reference to the Actor who will own the new component.
- **Manual Attachment**: Boolean value. If false, the new Actor will be automatically attached.
- **Relative Transform**: Transformation used by the new component.

Output

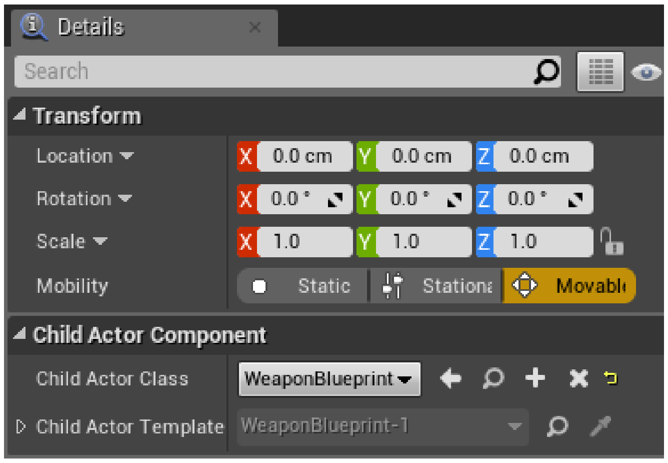
- **Return Value**: Reference to the created Actor.

Example Usage:

In this example we have a blueprint that represents a weapon. The name of this blueprint is "WeaponBlueprint". In another blueprint that represents a character of the game there is an event that will be called during the game to add a weapon of type "WeaponBlueprint" to the character.



The image below shows the "Details" tab of the "Add Child Actor Component" function that is displayed when the function is selected. The class that will be used by the new actor must be informed in the "Child Actor Class" property.



Location X 0.0 cm Y 0.0 cm Z 0.0 cm
Rotation X 0.0 ° Y 0.0 ° Z 0.0 °
Scale X 1.0 Y 1.0 Z 1.0
Mobility Static Stationa Movabl

Child Actor Class WeaponBlueprint
Child Actor Template WeaponBlueprint-1

Spawn Emitter at Location

Plays a particle system at the specified location.

Input

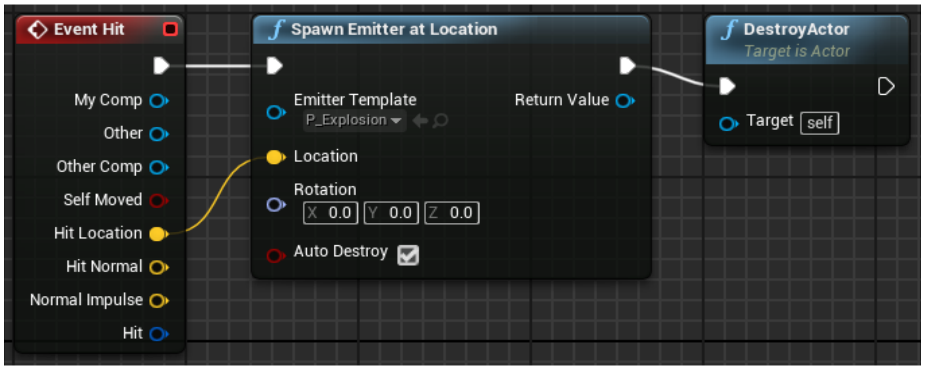
- **Emitter Template**: Particle system template that will be used.
- **Location**: Location where the particle system will be created.
- **Rotation**: Rotation used by the particle system.
- **Auto Destroy**: Boolean value. If true, it destroys the particle system when execution is complete.

Output

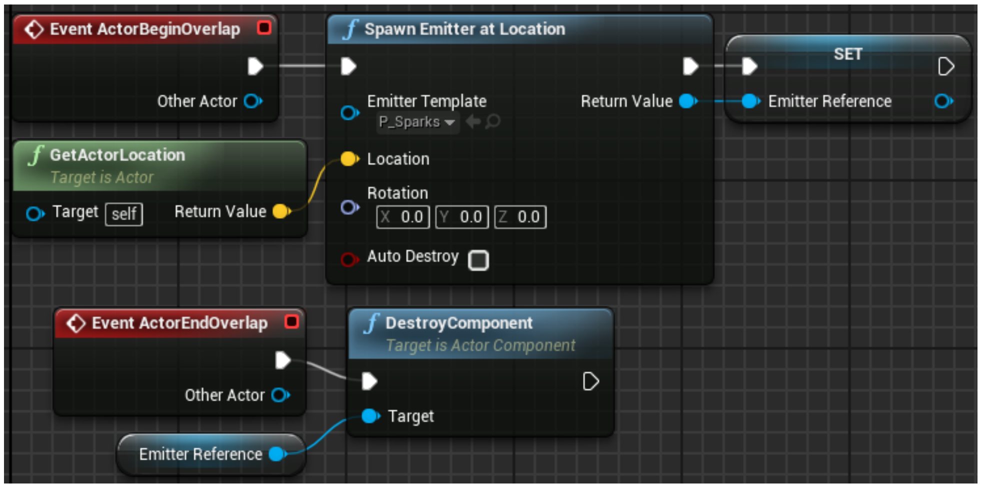
- **Return Value**: Reference to the particle system component that was created.

Example Usage:

The image below is from a blueprint that represents a "Bullet". When this bullet collides with something a particle system will be created at the collision location using the "P_Explosion" template. After this the bullet is destroyed.



In another example, a particle system is created when an actor overlaps a blueprint and is destroyed when an actor ends the overlap. The reference to the particle system created is saved in the "Emitter Reference" variable. The particle system template "P_Sparks" that was used is looped and does not stop running automatically.



AI Move To

The "AI Move To" action is used to move a Pawn/Character. The destination can be a location or another actor. The level must have a "Nav Mesh Bounds Volume" to define the area that the "AI Move To" action can use. It is a **latent** action, so it runs in parallel to the normal flow of execution of the Blueprints.

Input

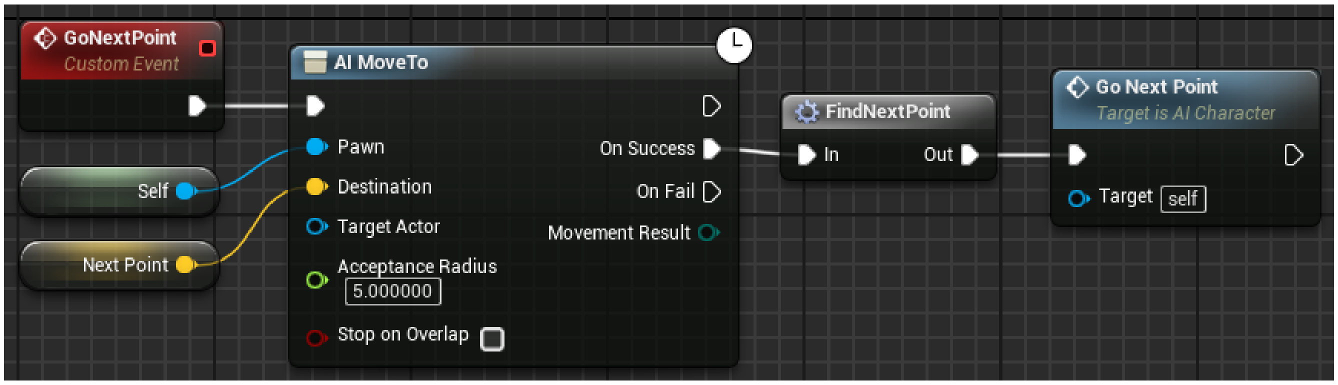
- **Pawn**: Reference to the Pawn/Character that will be moved.
- **Destination**: Vector indicating the destination.
- **Target Actor**: Reference to an actor to be used as destination.
- **Acceptance Radius**: Maximum destination distance to complete movement.
- **Stop on Overlap**: Boolean value. If true, complete the move as soon as the Pawn begins to overlap the "Acceptance Radius".

Output

- **On Success**: Execution pin that will be activated when and if the "AI Move To" manages to reach the target.
- **On Fail**: Execution pin that will be activated if the destination can not be reached.
- **Movement Result**: Enumeration that indicates the result of the movement.

Example Usage:

The image shows the "AI Move To" action being used for a Pawn to follow a path represented by a set of predefined points on the level. These points representing the path may be stored in an array. The "FindNextPoint" macro is responsible for picking the next point in the path and storing it in the "Next Point" variable.



The clock icon indicates that "AI Move To" is a **latent** action. The event "GoNextPoint" starts the "AI Move To" action, but only when the Pawn reach the destination is that the actions connected to the pin "On Success" will be performed. After the "FindNextPoint" macro updates the "Next Point" variable, the "GoNextPoint" event is called again.

Execute Console Command

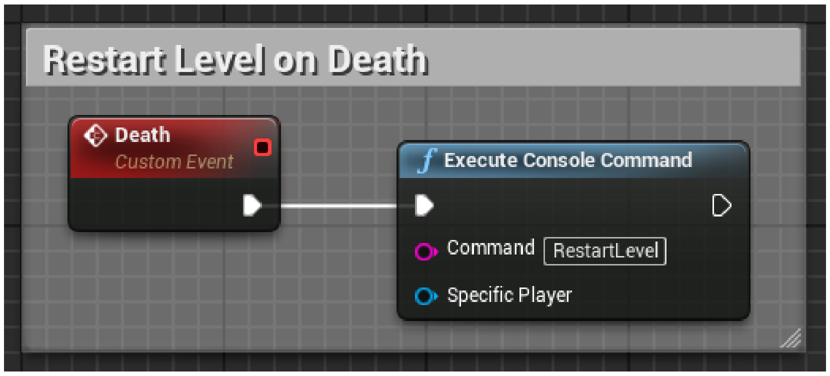
Blueprint function that allows the execution of a console command. Console commands are text-based commands that can be executed in the editor or in the game.

Input

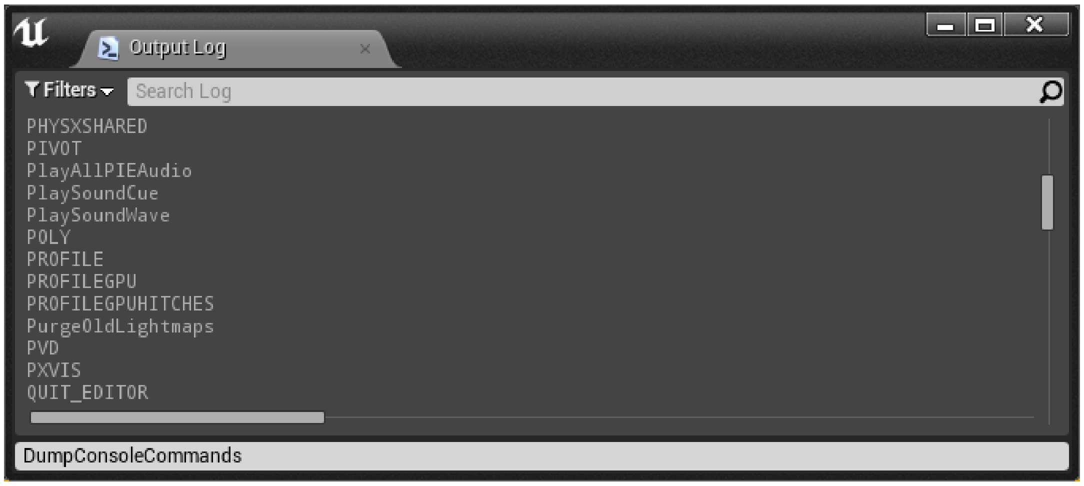
- **Command**: Console command that will be executed.
- **Specific Player**: Optional. Reference to a Player Controller that will receive the command.

Example Usage:

In a simple game, the level will be restarted when the player dies. This is done using a custom event named "Death" that executes the console command "RestartLevel".



For a complete list of console commands, open the "**Output Log**" window (Window → Developer Tools → Output Log). The line at the bottom of the window is for entering console commands. Enter the "*DumpConsoleCommands*" command and press Enter to get the list of commands.



```
PHYSXSHARED
PIVOT
PlayAllPIEAudio
PlaySoundCue
PlaySoundWave
POLY
PROFILE
PROFILEGPU
PROFILEGPUHITCHES
PurgeOldLightmaps
PVD
PXVIS
QUIT_EDITOR
```

```
DumpConsoleCommands
```