LECTURE 11

Blueprints in Action 2

LECTURE GOALS AND OUTCOMES

Goals

The goals of this lecture are to

- Explain traces
- Show how to do different types of traces
- Show how to spawn and destroy sounds and particles
- Present various types of animations that can be created with actions

Outcomes

By the end of this lecture you will be able to

- Use traces to test collision
- Manage sounds and particles
- Create simple animations with actions

TRACES

TRACE TYPES

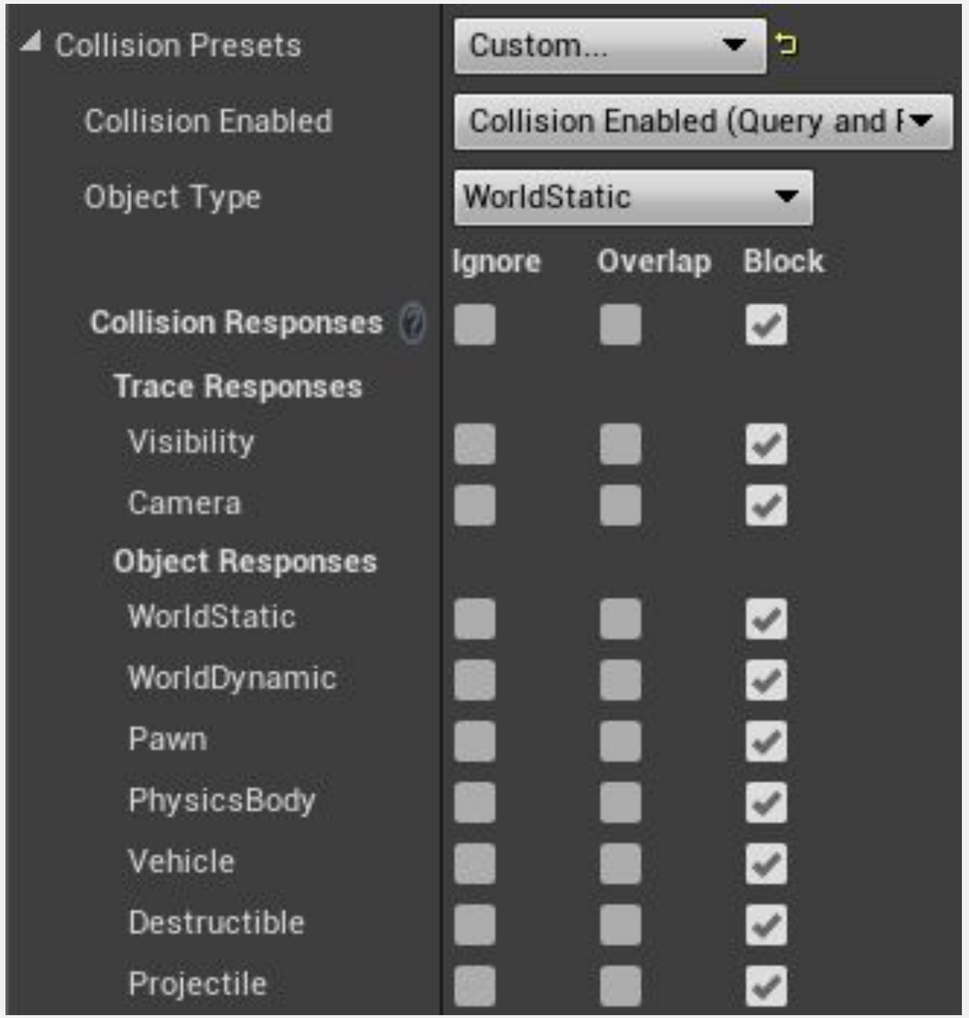
Traces are used to test if there are collisions along a defined line and can return the first object or multiple objects hit.

A trace can be done by **channel** or by **Object type**.

The channel can be “Visibility” or “Camera”. The Object type can be “WorldStatic”, “WorldDynamic”, “Pawn”, “PhysicsBody”, “Vehicle”, “Destructible”, or “Projectile”.

The image on the right shows the collision responses of a Static Mesh Actor.

The Object type is defined via the **Object Type** property drop-down, while the Visibility and Camera trace responses are defined in the **Trace Responses** section of the **Collision Responses** table.



HIT RESULT STRUCT

When a trace function collides with something, it returns one or more **Hit Result** structs. The **Break Hit Result** node can be used to access the Hit Result's elements, as seen in the image on the right.

Some elements of Hit Result are as follows:

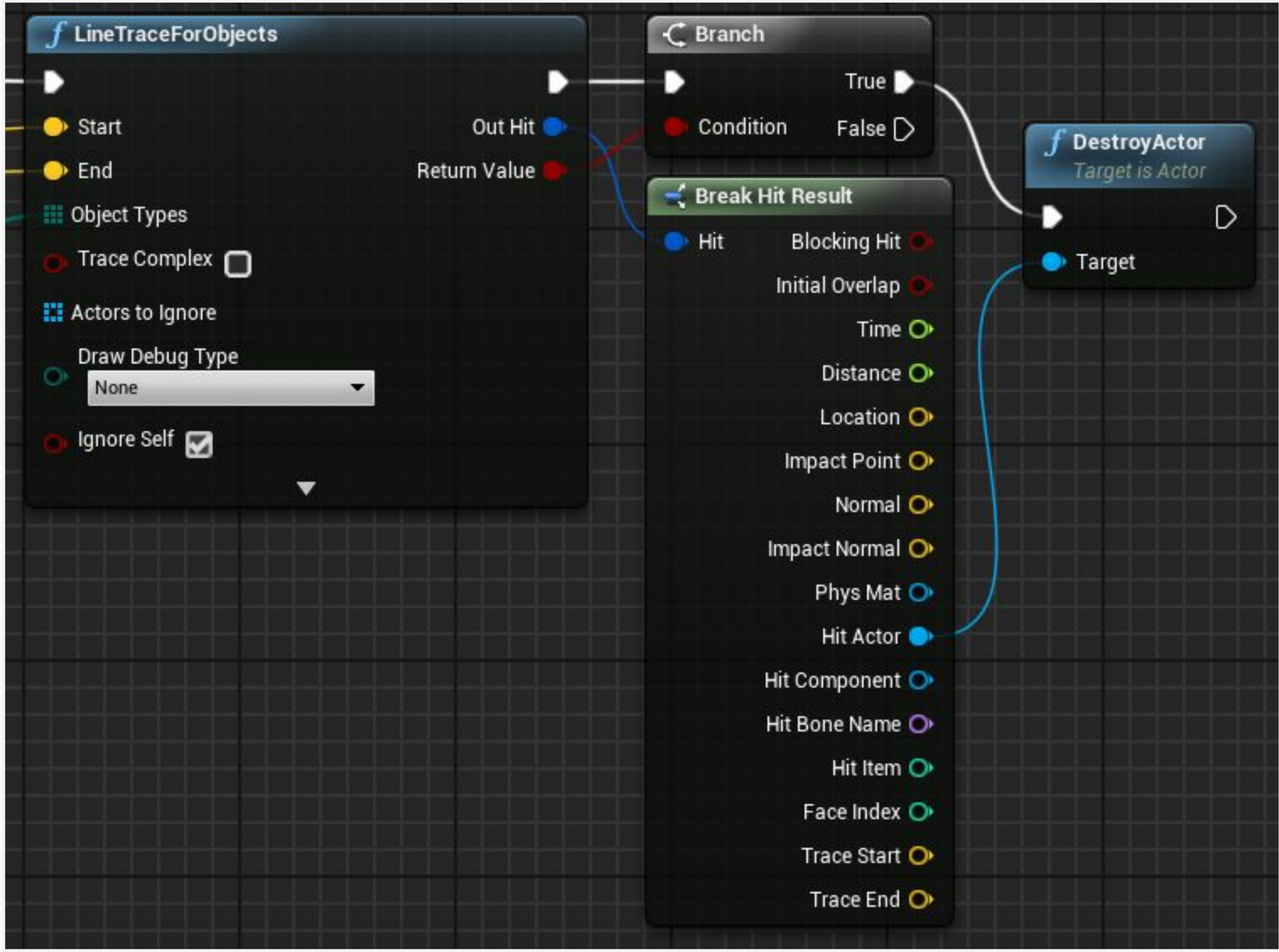
- **Blocking Hit**: Boolean value that indicates if there was a blocking hit.
- **Location**: Location of the hit.
- **Normal**: Normal vector of the hit in world space.
- **Hit Actor**: Reference to the Actor hit by the trace.

HIT RESULT STRUCT: EXAMPLE

The example on the right uses the **Break Hit Result** node. The **LineTraceForObjects** function's **Out Hit** output parameter returns a Hit Result struct if there was a hit.

The Actor hit is removed from the game by using the **DestroyActor** function, which uses the **Hit Actor** reference as the target.

The **LineTraceForObjects** function is explained on the next slide.

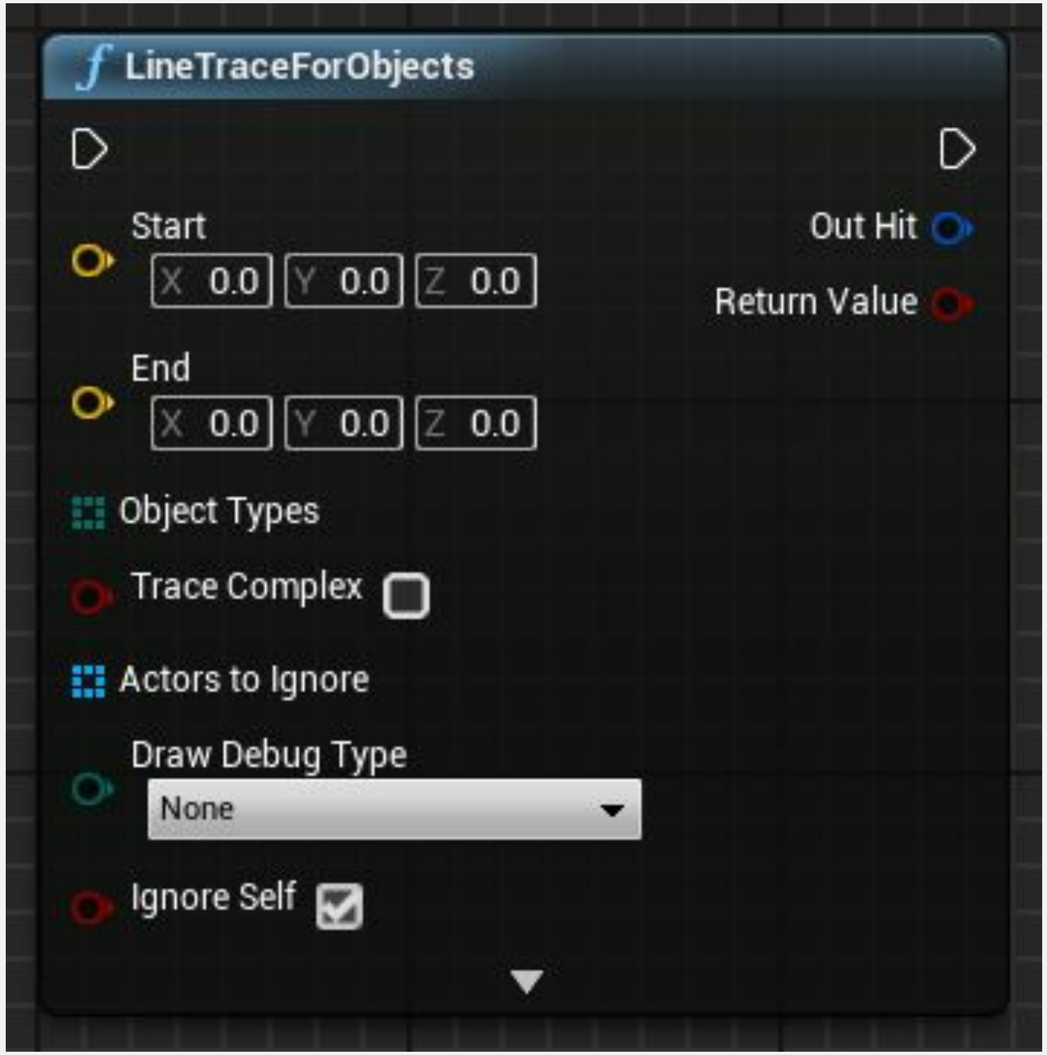


LINE TRACE FOR OBJECTS

The **LineTraceForObjects** function tests for collision along a defined line and returns a Hit Result struct with data for the first Actor hit that matches one of the Object types specified in the function call.

Input

- **Start** and **End**: Location vectors that define the start and the end of the line to be used for the collision test.
- **Object Types**: Array that contains the Object types that will be used in the collision test.
- **Trace Complex**: Boolean value indicating whether to use complex collisions.
- **Actors to Ignore**: Array of Level Actors that should be ignored in the collision test.
- **Draw Debug Type**: Allows the drawing of a 3D line representing the trace.



MULTI LINE TRACE FOR OBJECTS

The **MultiLineTraceForObjects** function has the same input parameters as the **LineTraceForObjects** function.

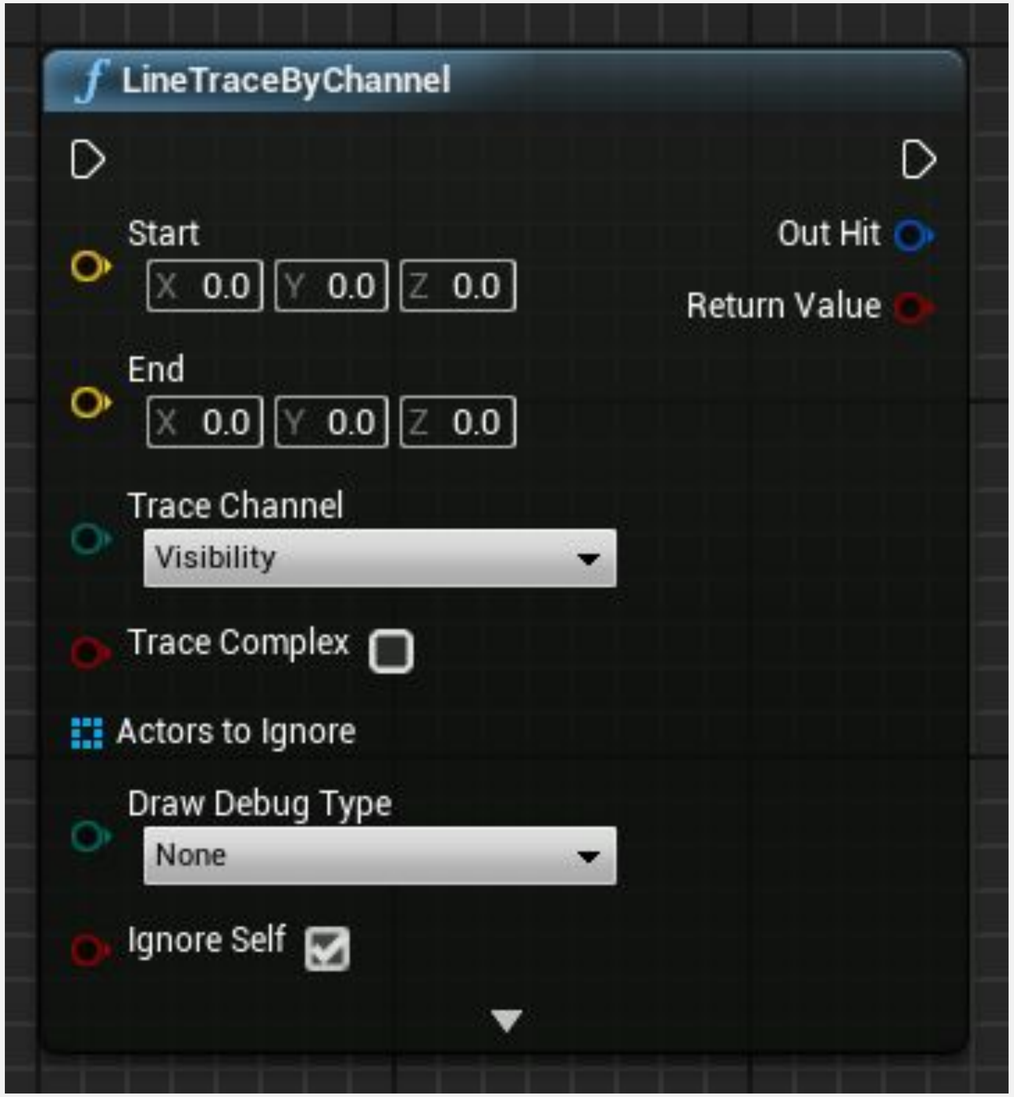
The difference between the functions is that the **MultiLineTraceForObjects** function returns an array of Hit Result structs, rather than a single one, making it more expensive to perform.

LINE TRACE BY CHANNEL

The **LineTraceByChannel** function tests for collision along a defined line using the trace channel, which can be “Visibility” or “Camera”, and returns a Hit Result struct with data for the first Actor hit in the collision test.

Input

- **Start** and **End**: Location vectors that define the start and the end of the line to be used for the collision test.
- **Trace Channel**: Channel used for the collision test. It can be “Visibility” or “Camera”.
- **Trace Complex**, **Actors to Ignore**, and **Draw Debug Type**: The same parameters used in the **LineTraceForObjects** function (see slide 7).



MULTI LINE TRACE BY CHANNEL

The **MultiLineTraceByChannel** function has the same input parameters as the **LineTraceByChannel** function.

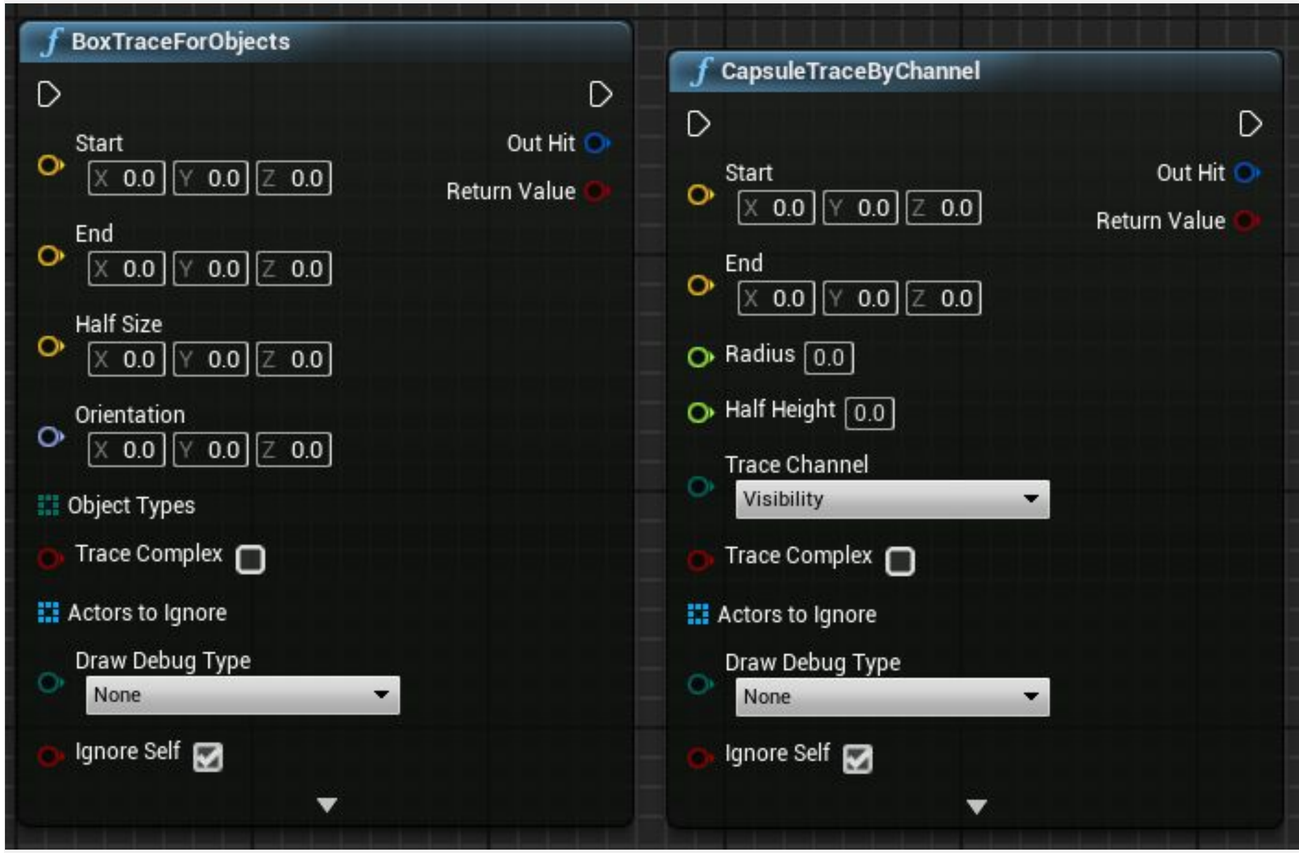
The difference between the functions is that the **MultiLineTraceByChannel** function returns an array of Hit Result structs, rather than a single one, making it more expensive to perform.

SHAPE TRACES

Traces can also be done using shapes. There are trace functions for the box, capsule, and sphere shapes, but these functions are more expensive to perform than line traces.

For all of these shapes, there are functions to trace by channel and by Object type. There are also functions for single hits or multiple hits.

The image on the right shows the **BoxTraceForObjects** and **CapsuleTraceByChannel** functions.



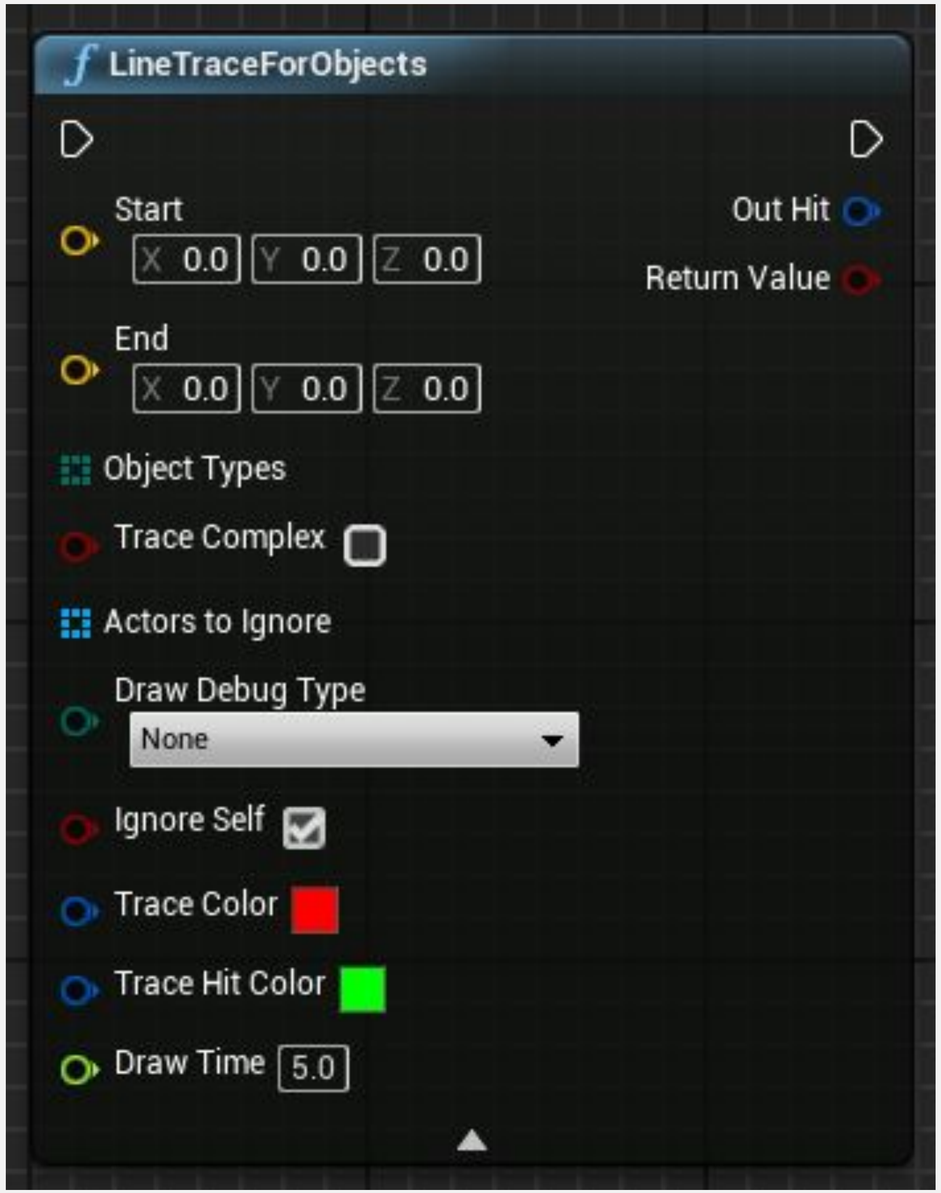
DEBUGGING

The trace functions have an option to draw debug lines that help when testing the traces.

The **Draw Debug Type** parameter can be set to one of the following values:

- **None**: Don’t draw the line.
- **For One Frame**: The line appears only for one frame.
- **For Duration**: The line stays for the amount of time specified in the **Draw Time** parameter.
- **Persistent**: The line does not disappear.

To display the **Trace Color**, **Trace Hit Color**, and **Draw Time** parameters, click on the small arrow at the bottom of the function.



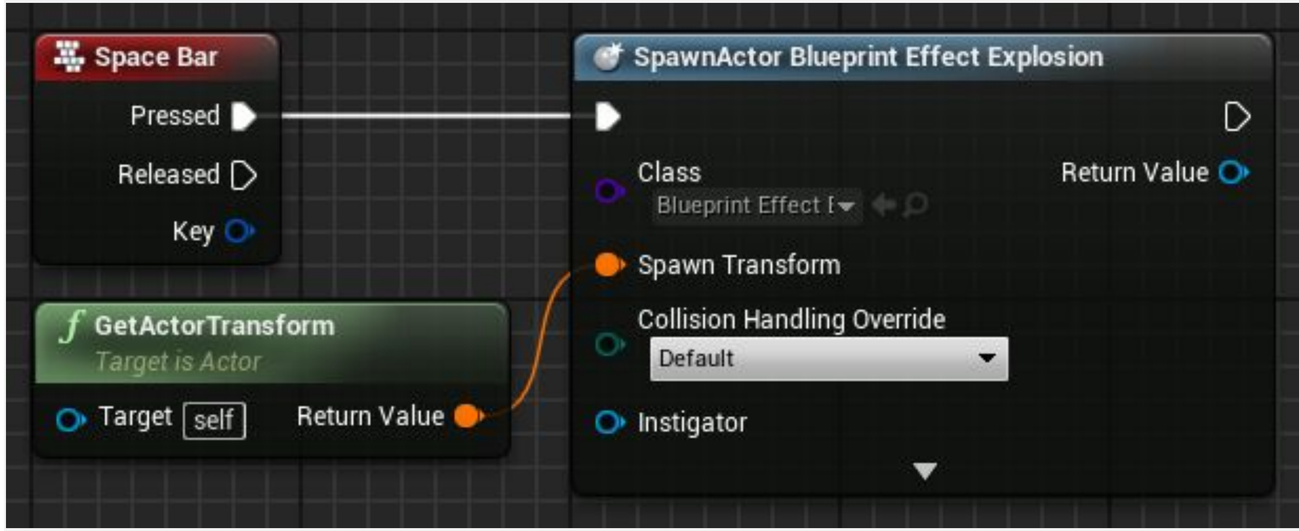
SPAWNING AND DESTROYING

SPAWNING ACTORS

Spawn Actor from Class is a function that creates an Actor instance using the class and transform specified.

The **Collision Handling Override** input defines how to handle the collision at the time of creation. The output parameter **Return Value** is a reference to the newly created instance.

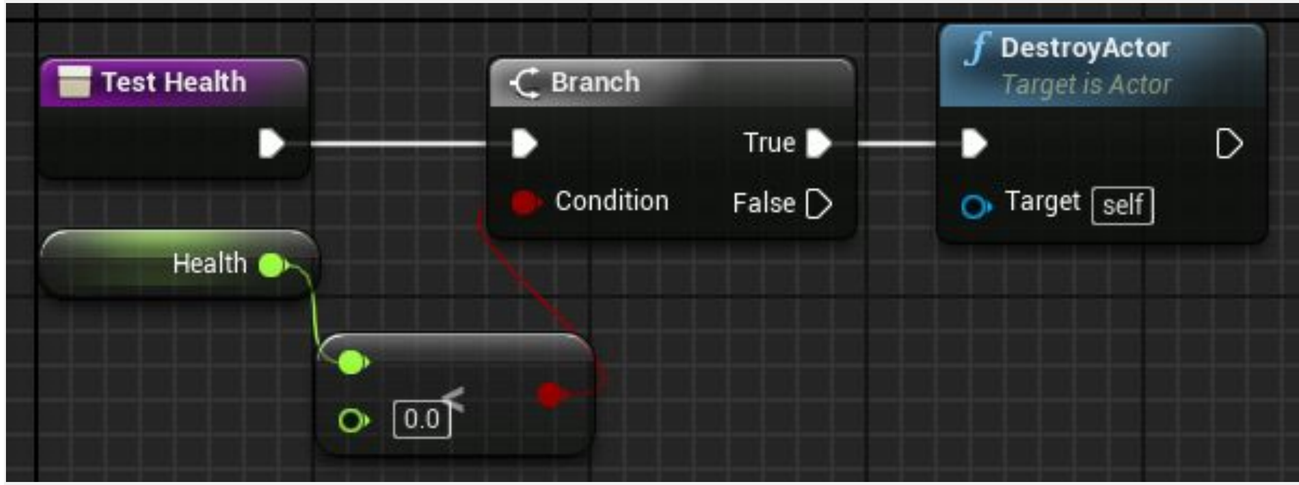
In the example on the right, when the **space bar** is pressed, an instance of the **Blueprint Effect Explosion** class is created at the same location (transform) of the current Blueprint.



DESTROYING ACTORS

The **DestroyActor** function removes an Actor instance from the Level at runtime. The instance to be removed must be specified in the **Target** parameter.

The image on the right shows a function named “**Test Health**” that will check if the value of the **Health** variable is less than zero. If “**true**”, the current instance of this Blueprint, which is represented by “**self**”, will be destroyed.



ATTACHING ACTOR TO COMPONENT

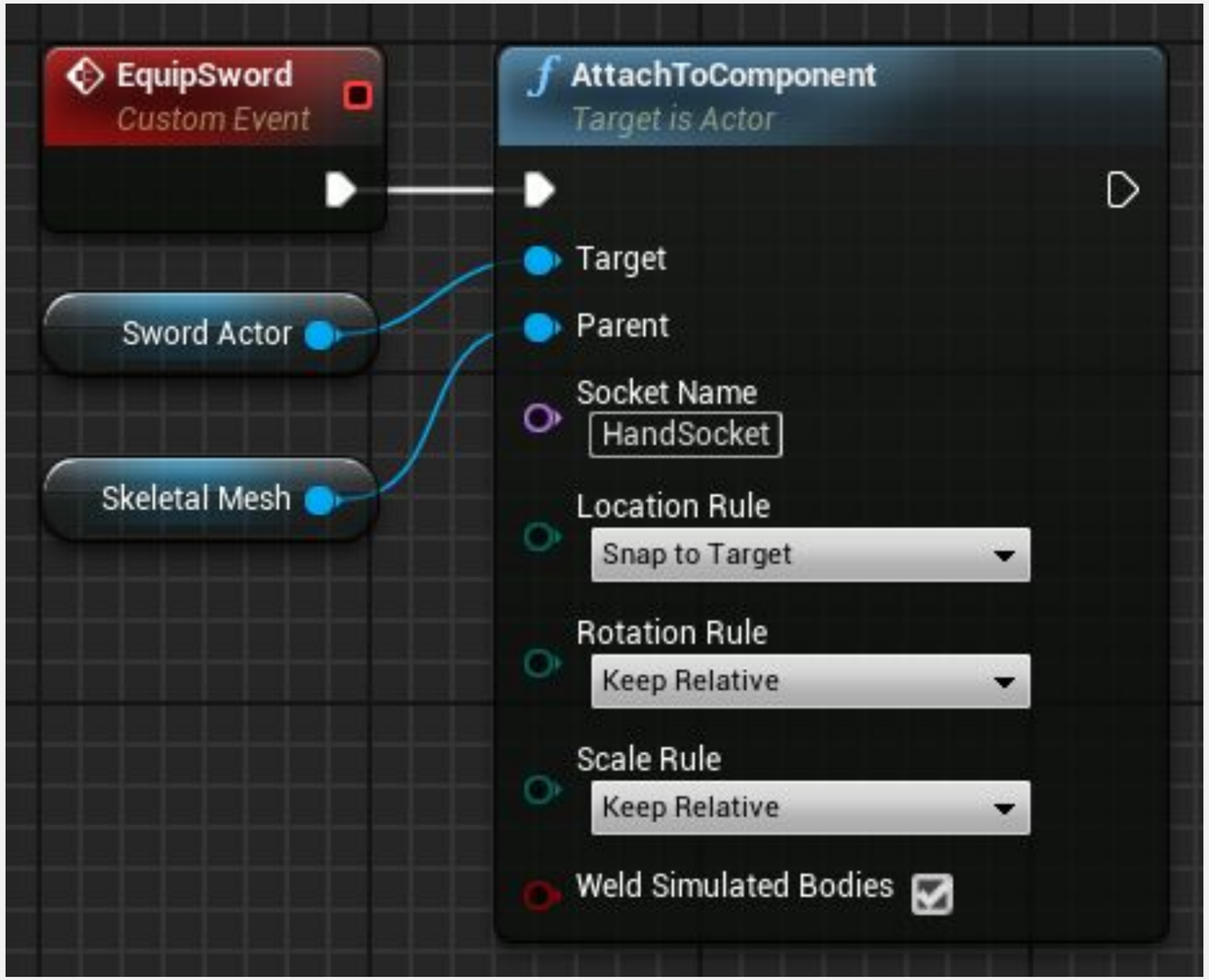
The **AttachToComponent** function attaches an Actor to the component referenced in the **Parent** input parameter. The Actor attached respects the transformations of the parent component.

Input

- **Target**: Actor to be attached.
- **Parent**: Component that receives the Actor.
- **Socket Name**: Name of the socket where the Actor will be attached. Use of this parameter is optional.

Example

In the custom event on the right, the Skeletal Mesh of the player is equipped with a sword using a socket labeled “**HandSocket**” that indicates where the sword must stay in the Skeletal Mesh.



SOUNDS:
PLAY SOUND AT LOCATION

There is a simple way to play a sound in Blueprints. Just use the **Play Sound at Location** function.

This function spawns an Ambient Sound Actor into your Level.

Use the combo box to define the Sound Cue or Sound Wave asset to play. The **Location** parameter is used to define the world position where the sound will play from.

Use of the others parameters is optional.

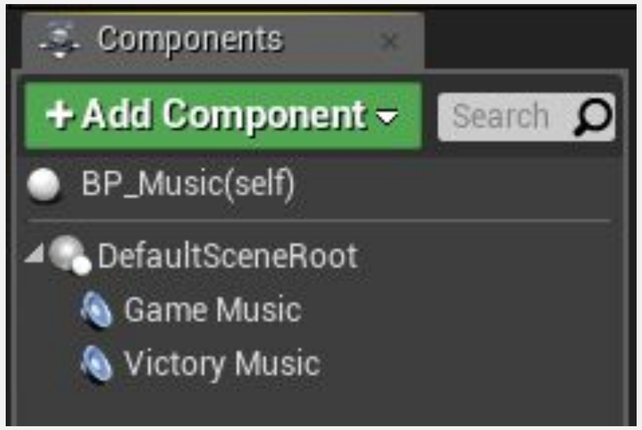
When the Actor finishes playing, it is destroyed.

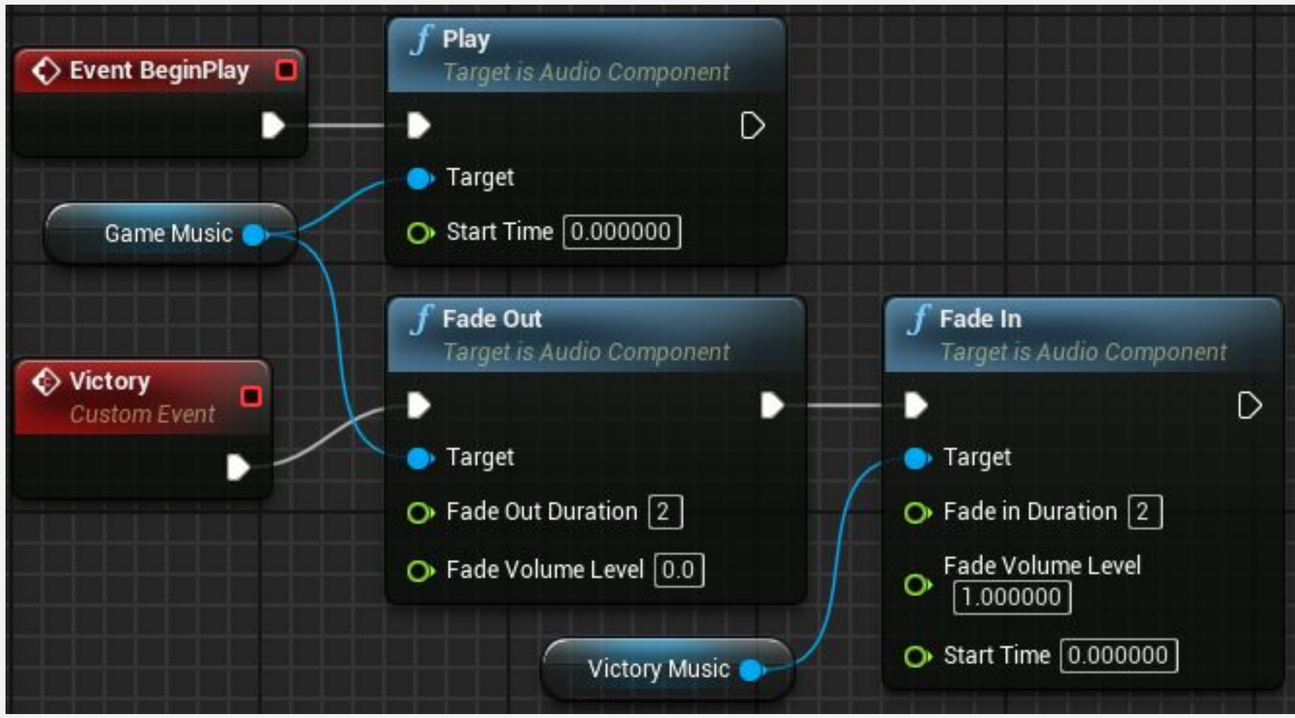
SOUNDS:
AUDIO COMPONENT

A good way to have control over sounds in Blueprints is to use an **Audio component**.

To add an Audio component, click the **Add Component** button in the **My Blueprint** panel and choose “**Audio**”. In the **Details** panel, select in the **Sound** combo box the audio/song that will be used and uncheck the **Auto Activate** property so that the audio does not start playing automatically.

In the example on the right, there are two Audio components, one named “**Game Music**” and the other named “**Victory Music**”. The **Fade Out** and **Fade In** nodes are used to gradually change the music for two seconds.





PARTICLES:
SPAWN EMITTER AT LOCATION

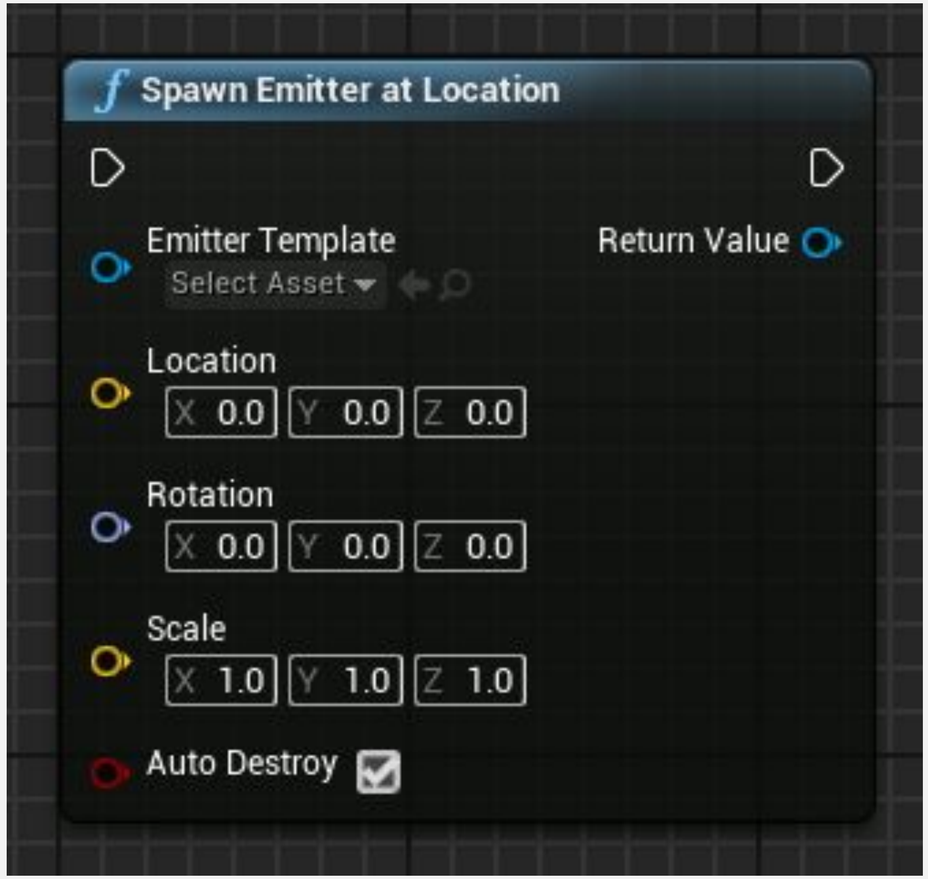
The **Spawn Emitter at Location** function spawns and plays a Particle System component at the specified location.

Input

- **Emitter Template**: Particle System template asset that will be used.
- **Location:** Location where the Particle System will be placed.
- **Rotation:** Rotation that will be applied to the Particle System.
- **Scale**: Scale that will be applied to the Particle System.
- **Auto Destroy**: Boolean variable. If the value is “**true**”, the Particle System will automatically be destroyed when execution is complete.

Output

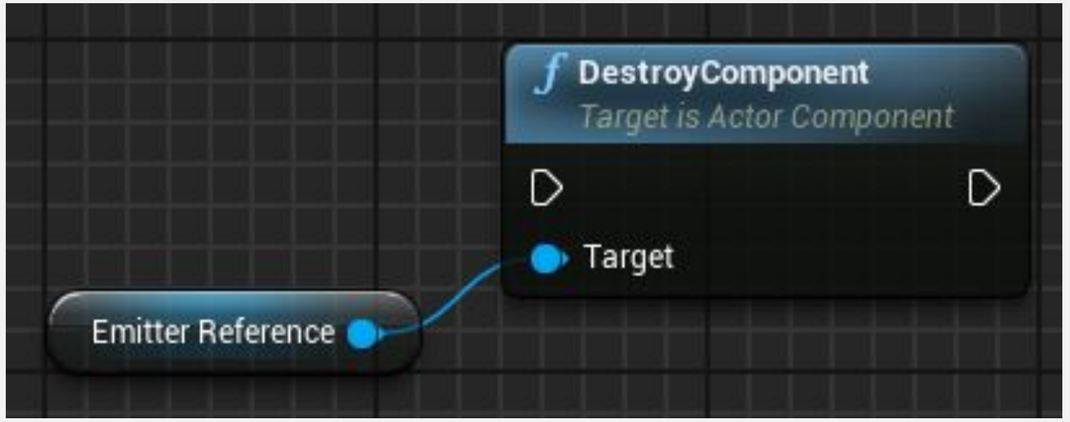
- **Return Value**: Reference to the Particle System component that was created.



PARTICLES: DESTROYING

Particle Systems can stay in a loop or be deactivated and kept in memory to be reactivated later.

If it is necessary to destroy a Particle System, the **DestroyComponent** function can be used, since a reference to a Particle System is a Particle System component.



ANIMATIONS

SET VIEW TARGET WITH BLEND

The **Set View Target with Blend** function from the Player Controller class is very useful for switching 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 to use for blending.
- **Blend Exp**: Exponent value that controls the shape of the curve. It is used by some blend functions.
- **Lock Outgoing**: Boolean variable. If the value is “**true**”, the outgoing view target will be locked to the last frame’s camera position.

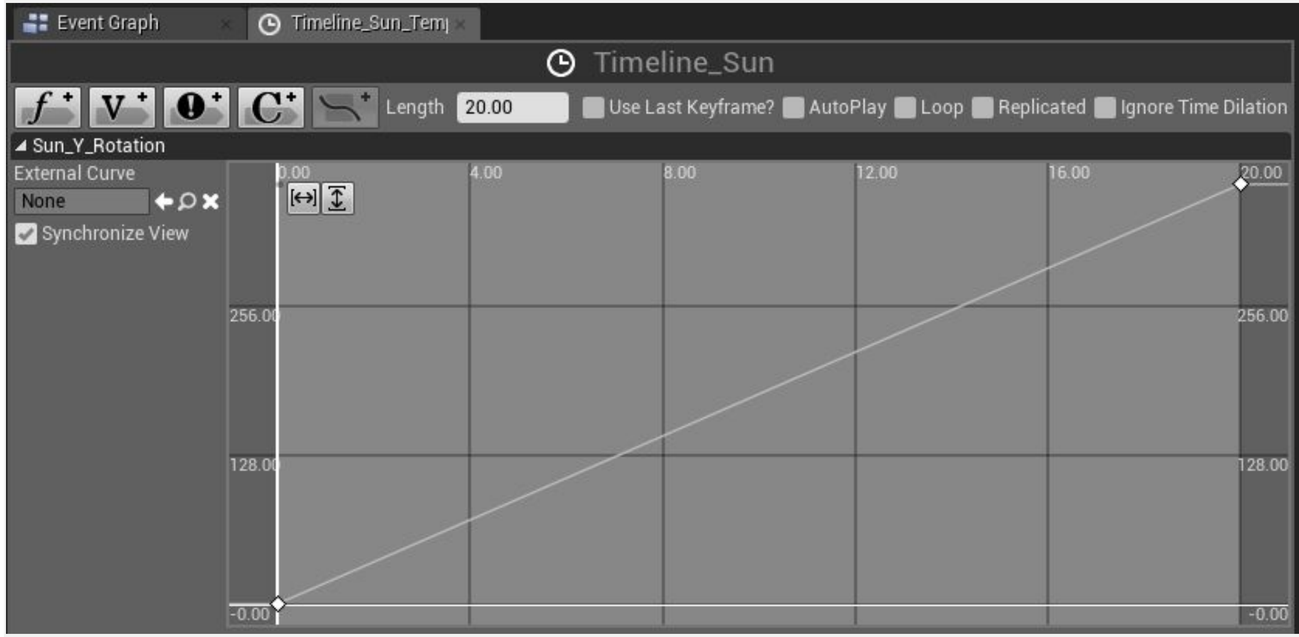
TIMELINES

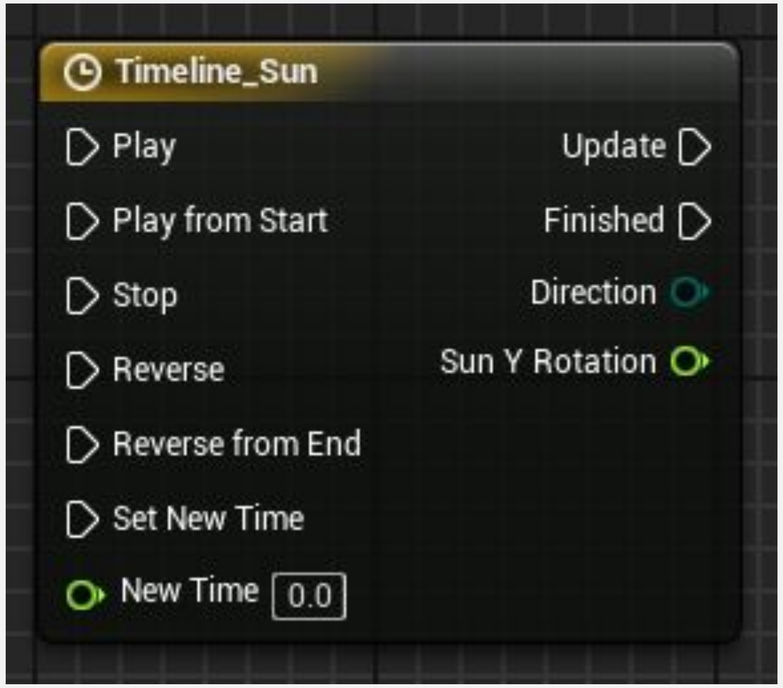
Timelines allow for the creation of simple time-based animations inside Blueprints. After a Timeline has been added to the Event Graph, it can be edited in the Blueprint Editor by double-clicking it.

The variables that are added to the Timeline are shown as output parameters so that their values can be accessed. The top image on the right shows the Timeline Editor with a track named “**Sun_Y_Rotation**”.

The **Update** pin is called constantly while the Timeline is running.

A Timeline can be played forward or backward.

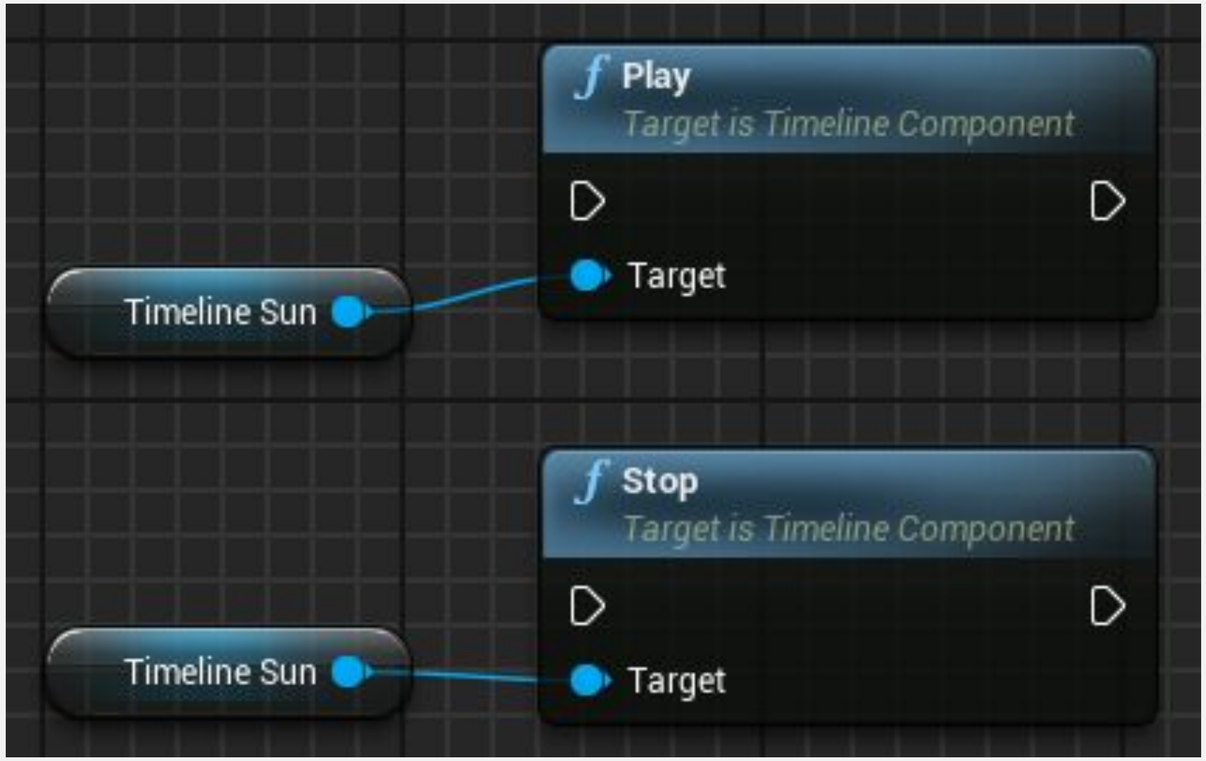




TIMELINES AS VARIABLES

Once created, Timelines can also be accessed as variables, so it is possible to call Timeline functions from anywhere in the graph.

A **Get** node of a Timeline can be obtained from the **My Blueprint** panel by going to **Variables > Components**.



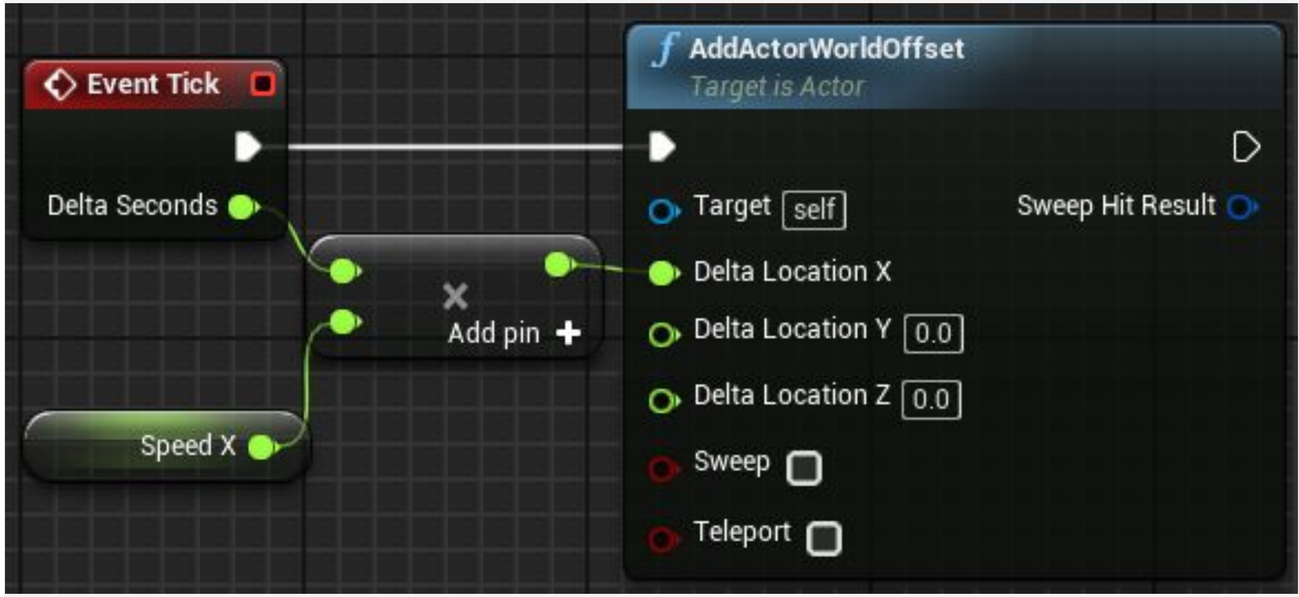
TICK EVENT AND DELTA TIME

There is an event named “**Tick**” that is called every frame of the game. For example, in a game that is running at 60 frames per second, the Tick event is called 60 times per second.

The Tick event has a parameter known as **Delta Seconds**, which contains the amount of time that has elapsed since the last frame.

In the Tick event illustrated on the right, the value of **Delta Seconds** is multiplied by the speed specified in the **Speed X** variable to determine the speed (in cm/s) at which the Actor needs to move along the X axis for each frame.

Tick actions can get very expensive, so alternatives should be considered if possible, such as Timelines or Timers.



INTERP TO

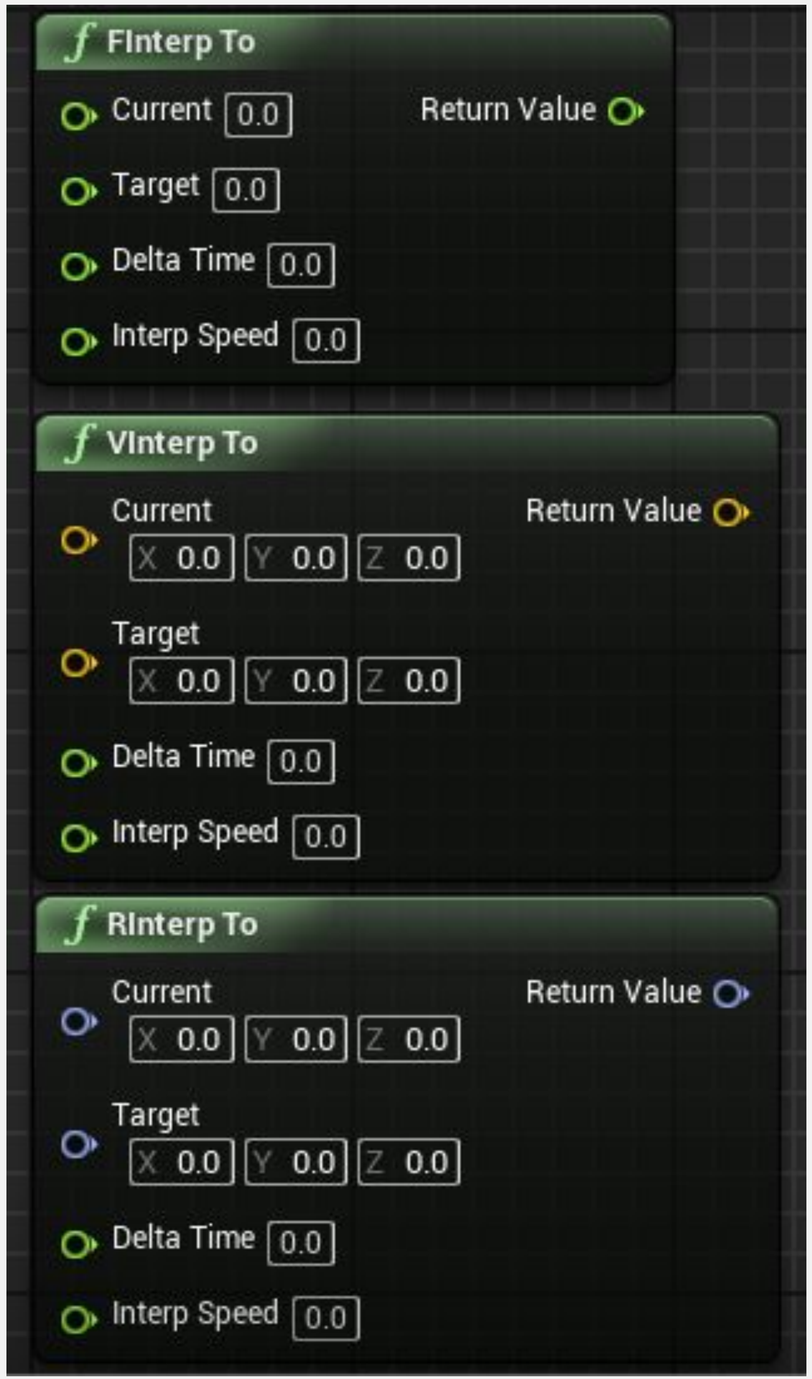
Interp To functions are used to change a value smoothly until it reaches the specified target value. Some examples include the **FInterp To** function for float values, the **VInterp To** for vectors, and the **RInterp To** function for rotators.

Input

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

Output

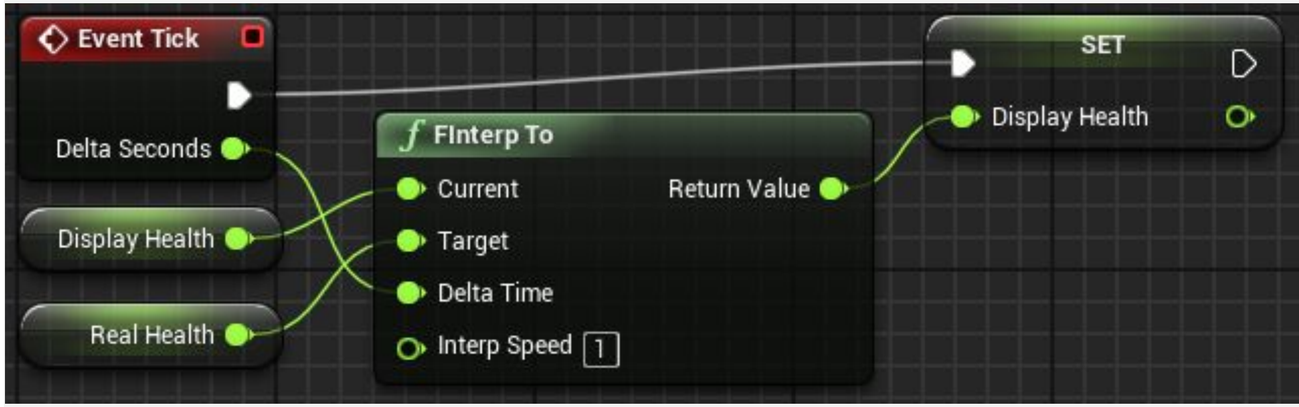
- **Return Value**: New value closer to the target value.



INTERP TO: EXAMPLE

The example on the right contains two variables. The **Real Health** variable stores 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 the **FInterp To** function so that the health bar smoothly decreases until the value of **Display Health** is equal to that of **Real Health**.



SUMMARY

This lecture explained the concept of traces. It showed how to spawn and destroy sounds and particles and how to create different types of animations using actions.