LECTURE 13

Blueprints in Action 4

LECTURE GOALS AND OUTCOMES

Goals

The goals of this lecture are to

- Present more Blueprint functions
- Show how to use the Add Child Actor Component function
- Explain the AI MoveTo function
- Show how to execute a console command in Blueprint

Outcomes

By the end of this lecture you will be able to

- Identify an Actor by a tag
- Apply damage to an Actor
- Use the Add Child Actor Component function
- Create simple AI that moves in the Level

MAP RANGE CLAMPED

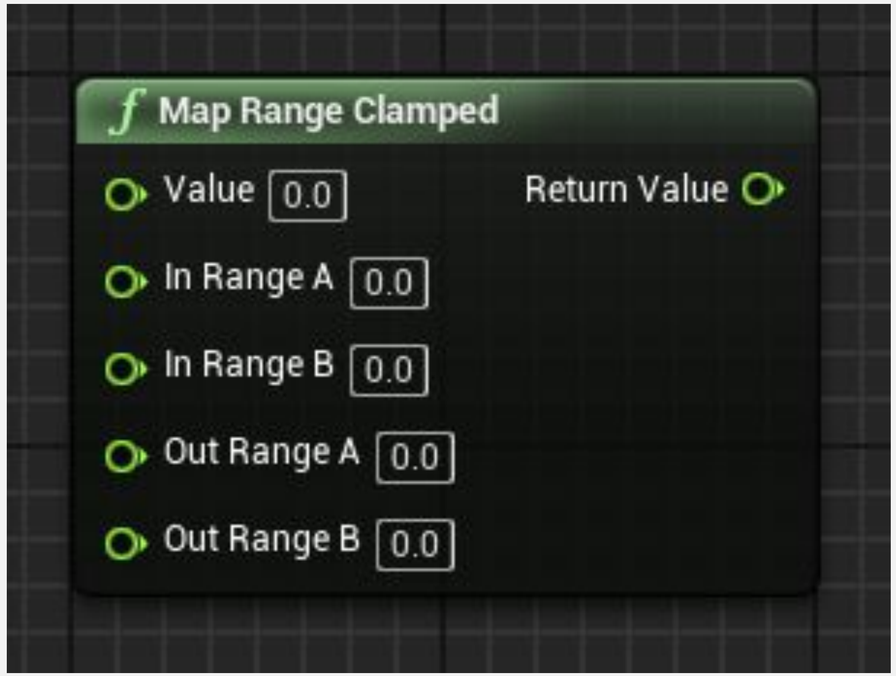
The **Map Range Clamped** function converts a value from one range of values to the corresponding value in another range of values. The end result will always be in the output value range.

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

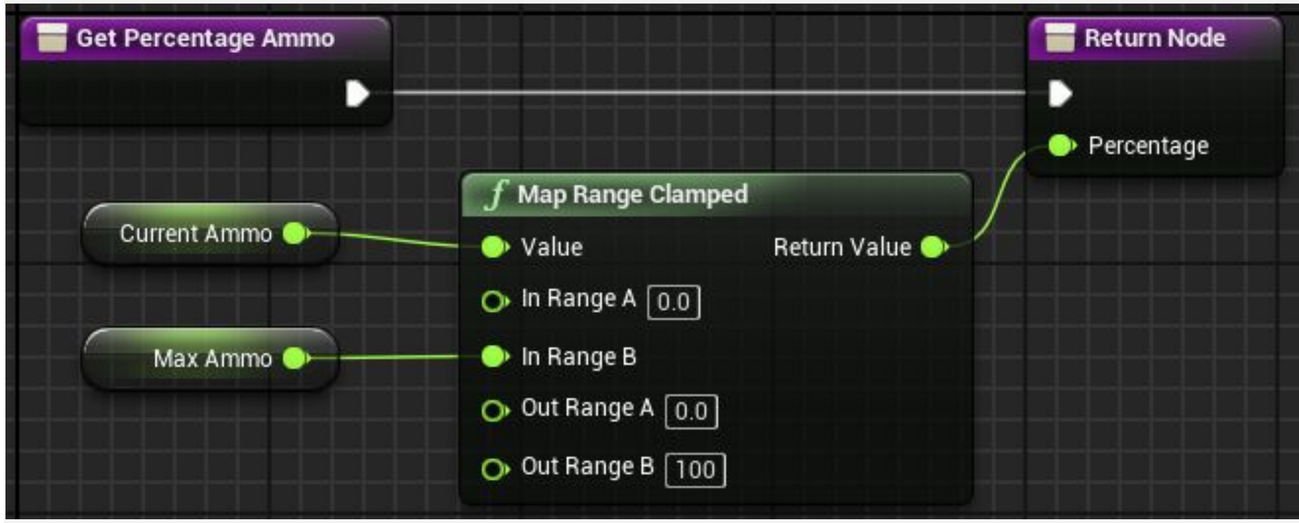
- **Return Value**: Converted value in the output value range.



MAP RANGE CLAMPED: EXAMPLE

In the example on the right, the **Get Percentage Ammo** function converts the current amount of the player's ammo into a percentage that 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.



ACTOR HAS TAG

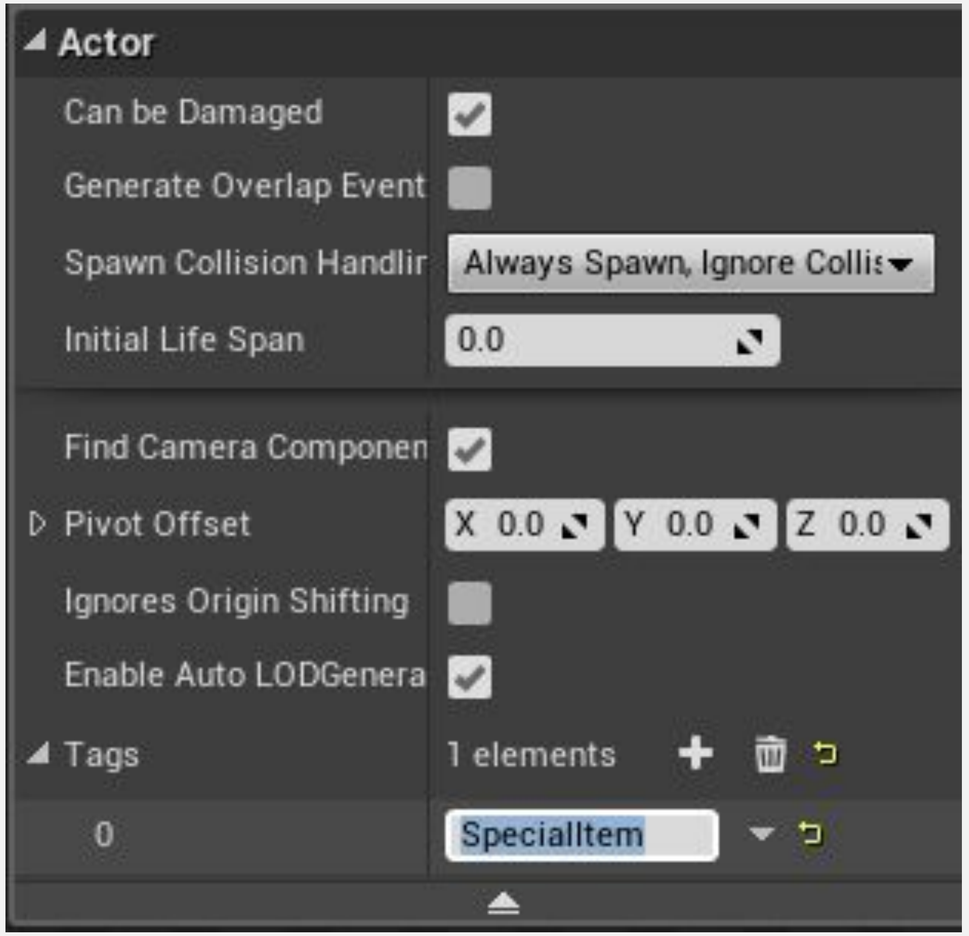
The **Actor Has Tag** function checks to see if a game Actor has a certain tag. The use of tags is a simple way to differentiate Actors in the Level. The image on the right shows that a tag called “**SpecialItem**” has been added to an Actor.

Input

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

Output

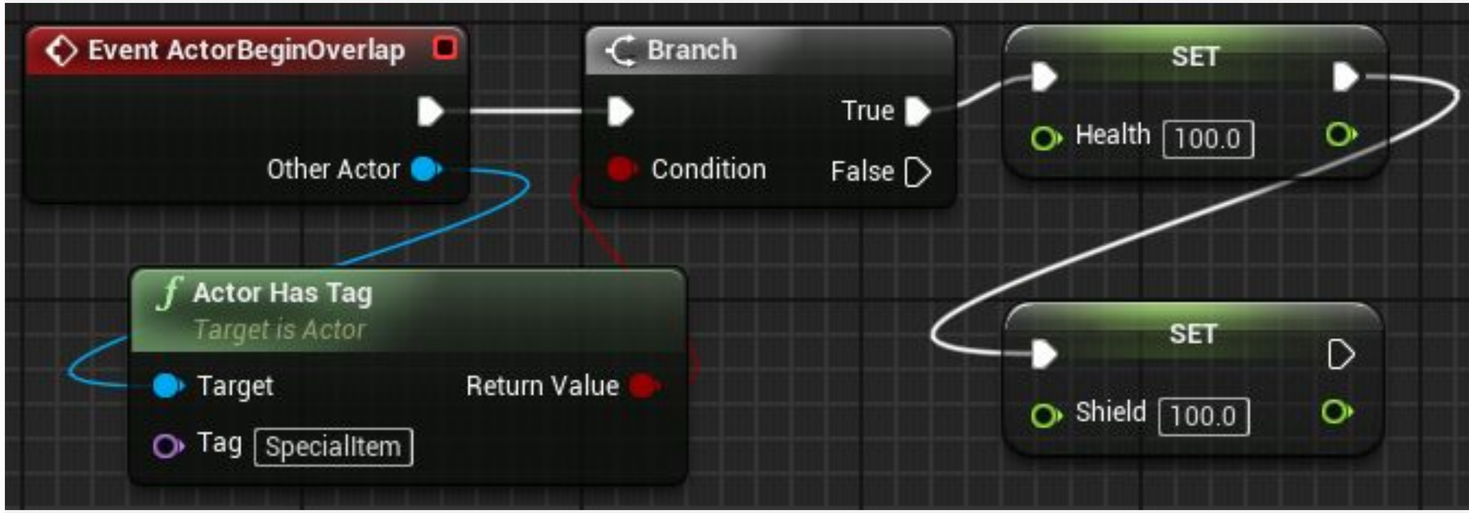
- **Return Value**: Boolean value. If the value is “**true**”, the Actor has the specified tag.



ACTOR HAS TAG: EXAMPLE

In the example on the right, the **Actor Has Tag** function is used to check if the Actor that was overlapped by the player has the **SpecialItem** tag.

If it does, then the health and shield of the player are restored by setting their values to “**100.0**”.



APPLY DAMAGE

The **Apply Damage** function is used to apply damage to an Actor. Information related to the damage can be passed through the function. The hit Actor responds to the damage using the **AnyDamage** event, but the event only fires if the value of the **Apply Damage** node's **Base Damage** parameter is different from "**0.0**".

Input

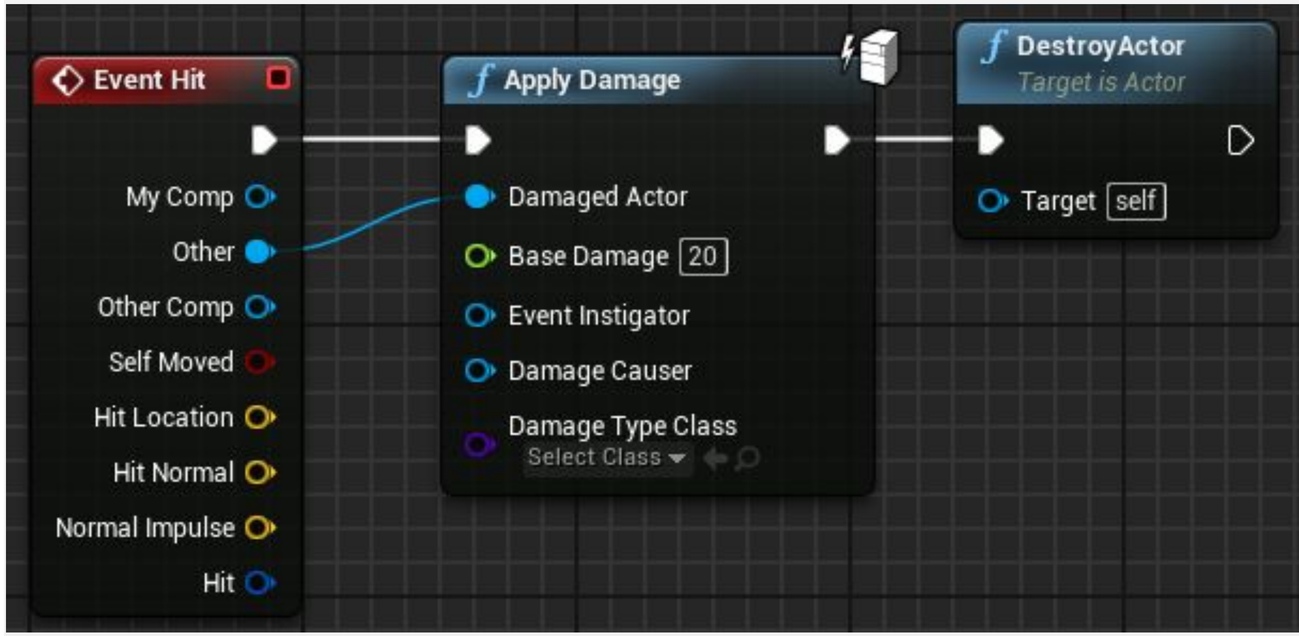
- **Damaged Actor**: Reference to the Actor who will suffer the damage.
- **Base Damage**: Float value that represents the damage.
- **Event Instigator**: Reference to the Controller responsible for causing the damage. Use of this parameter is optional.
- **Damage Causer**: Reference to the Actor that caused the damage. Use of this parameter is optional.
- **Damage Type Class**: Class that represents the type of damage done. Use of this parameter is optional.

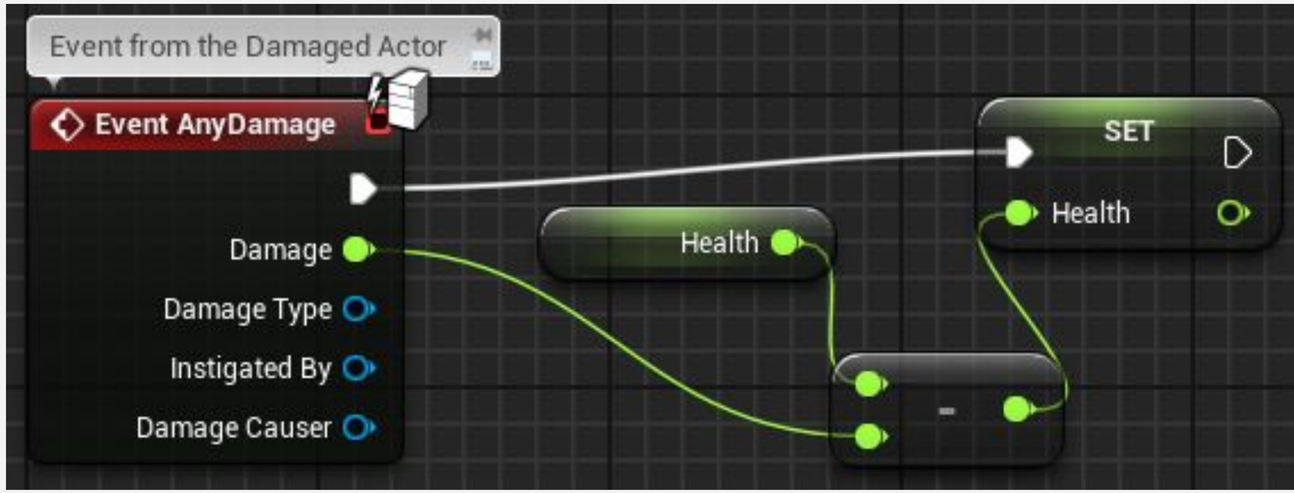
APPLY DAMAGE: EXAMPLE

The script seen in the top image on the right can be used in a Blueprint that represents a bullet.

When the bullet hits an Actor, the **Apply Damage** function is used to apply damage with a value of “**20**”. After that, the bullet is destroyed.

The bottom image shows the **AnyDamage** event for the damaged Actor. It will decrease the value of the **Health** variable by the **Damage** value.





GET OVERLAPPING ACTORS

The **Get Overlapping Actors** 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 the target and the other that uses a component as the target.

Input

- **Target**: Reference to the Actor/component that will be used in the overlapping test.
- **Class Filter**: Indicates which class or subclasses will be used in the test. Use of this parameter is optional.

Output

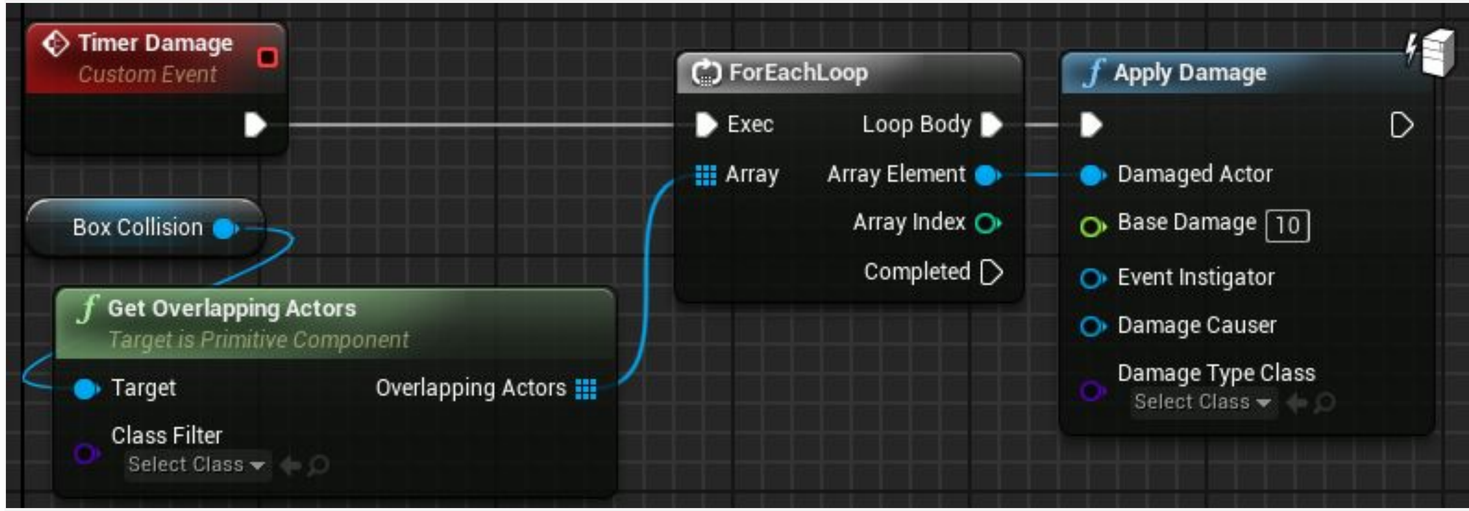
- **Overlapping Actors**: Array containing the Actors that are overlapping the Actor/component.

GET OVERLAPPING ACTORS: EXAMPLE

Imagine an area that deals damage to all the Actors in it.

In the example on the right, this area is represented by a Blueprint that has a Box Collision component that defines the location where the Actors suffer damage.

The **Timer Damage** event is periodically called to apply damage to all Actors who are overlapping the Box Collision.



ADD CHILD ACTOR COMPONENT

The **Add Child Actor Component** function 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 is also destroyed. The new Actor's class must be specified in the Details panel of the **Add Child Actor Component** function.

Input

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

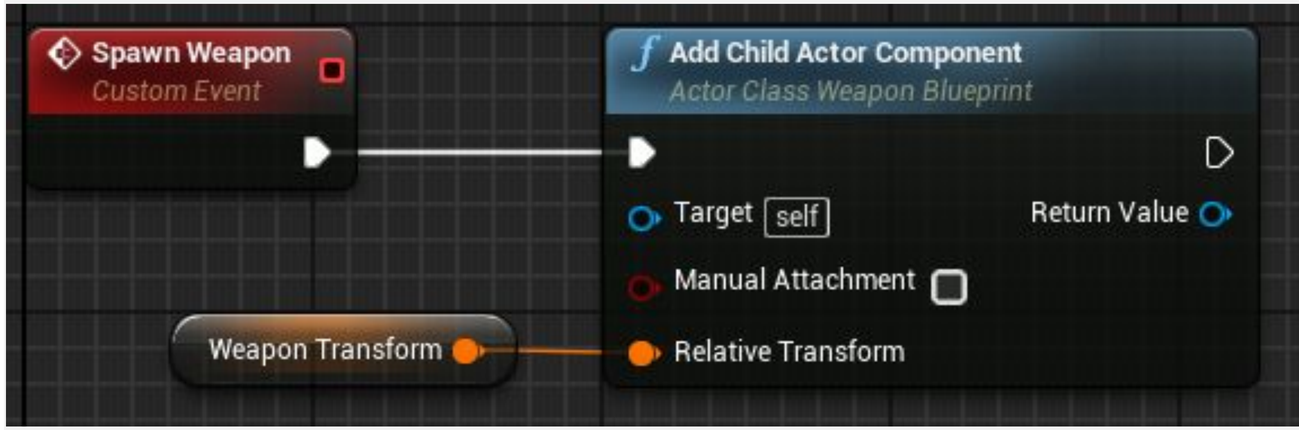
Output

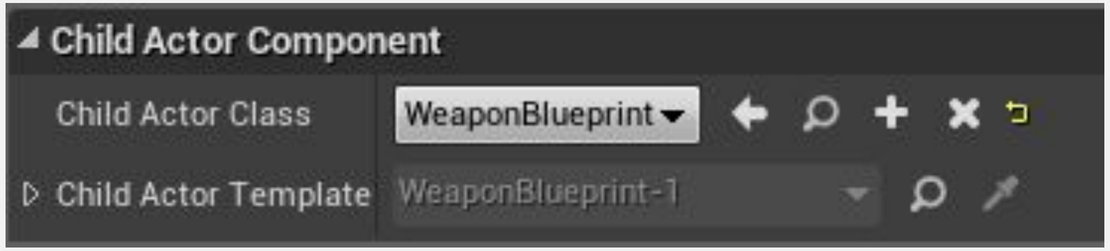
- **Return Value**: Reference to the created Actor.

ADD CHILD ACTOR COMPONENT: EXAMPLE

In the example on the right, there is a Blueprint named “**WeaponBlueprint**” that represents a weapon. In another Blueprint that represents a character in the game, there is an event that will be called during the game to add a weapon of type “**WeaponBlueprint**” to the character.

The bottom image is from the Details panel of the **Add Child Actor Component** function. It is displayed when the function is selected. The class that will be used by the new Actor must be specified in the **Child Actor Class** property field.



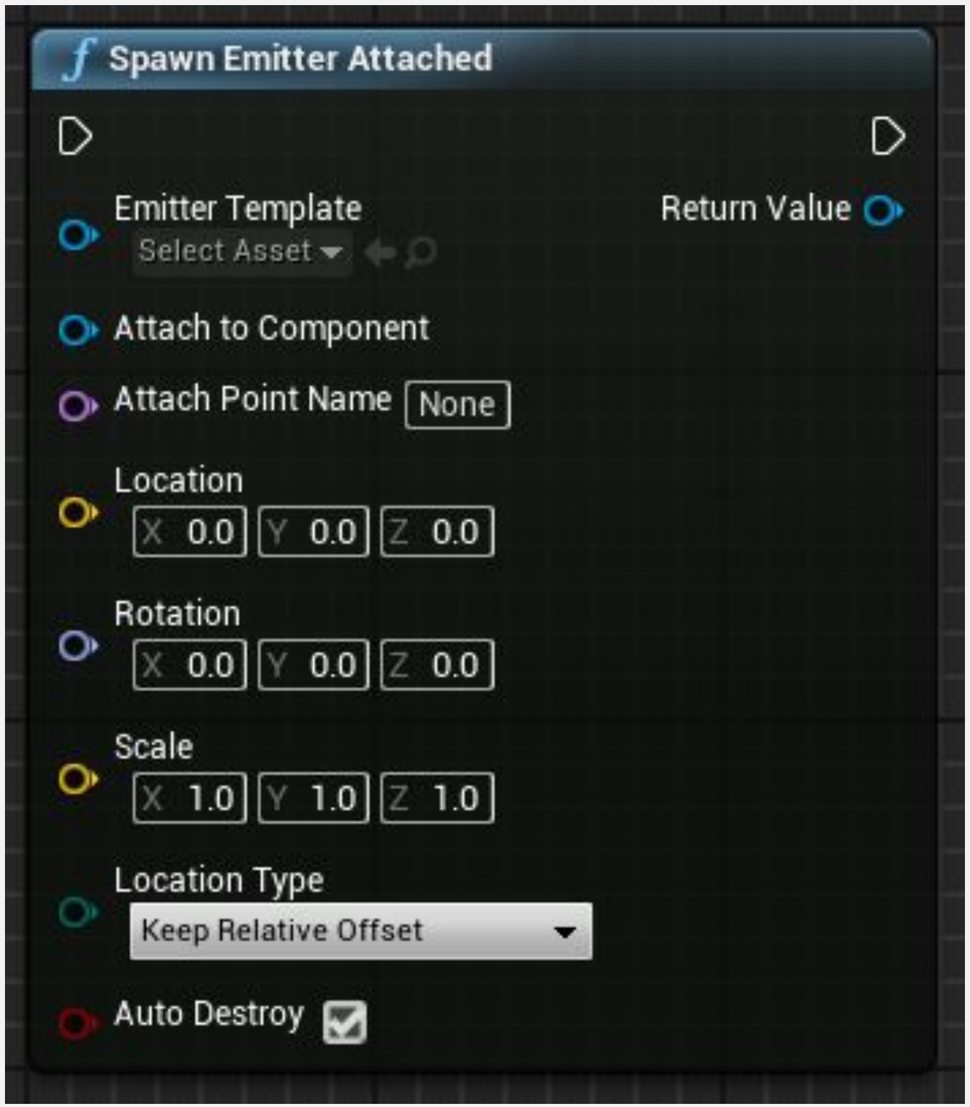


SPAWN EMITTER ATTACHED

The **Spawn Emitter Attached** function plays a Particle System and attaches it to a component.

Input

- **Emitter Template**: Particle System template that will be used.
- **Attach to Component**: Reference to a component.
- **Attach Point Name**: Name of the socket where the emitter will be attached. Use of this parameter is optional.
- **Location, Rotation, Scale**: World or relative transform depending on the value of **Location Type**.
- **Location Type**: World or relative location.
- **Auto Destroy**: Boolean value. If the value is “**true**”, the Particle System will be destroyed when execution is complete.

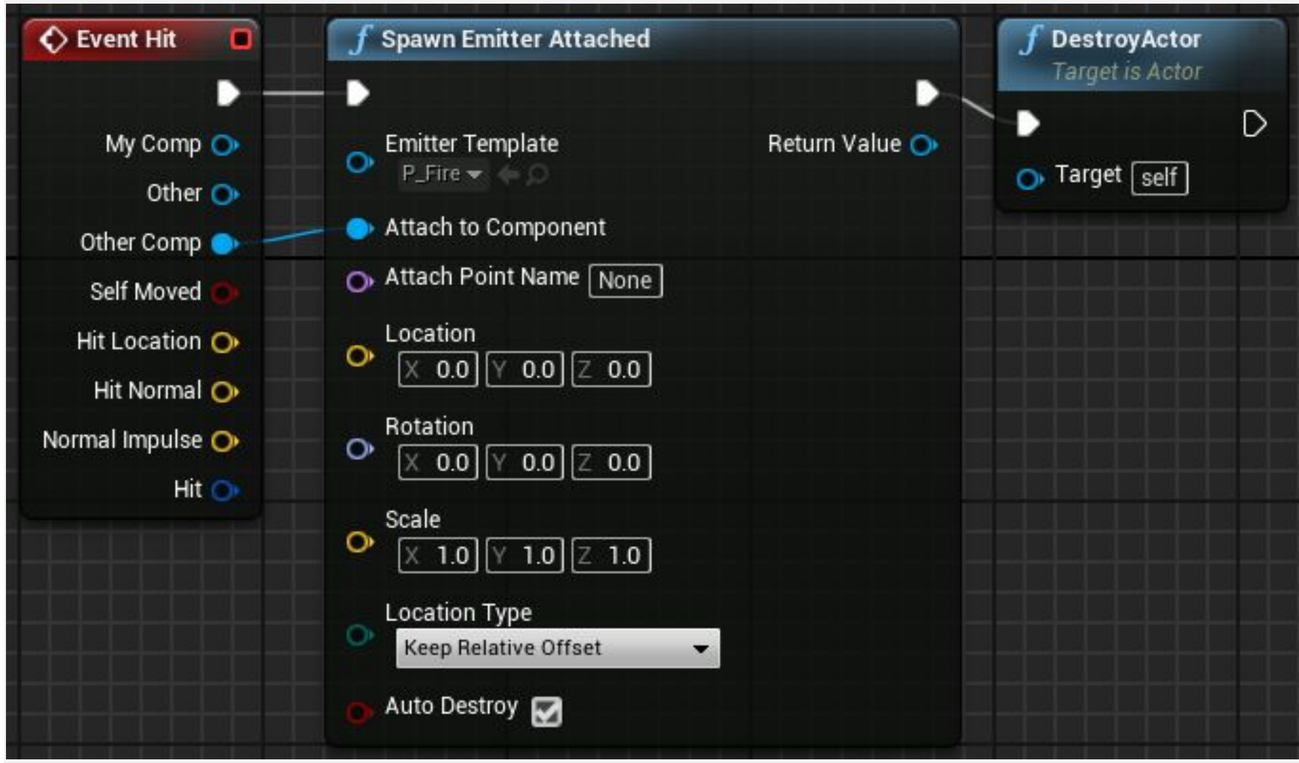


SPAWN EMITTER ATTACHED: EXAMPLE

The image on the right is from a Blueprint that represents a bullet.

When the bullet collides with something, a Particle System will be created using the **P_Fire** template and will be attached to the component that was hit.

After that, the bullet will be destroyed.



AI MOVE TO

The **AI MoveTo** node is used to move a Pawn. 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 MoveTo** action can use. It is a latent action, so it runs in parallel to the normal flow of execution of the Blueprints.

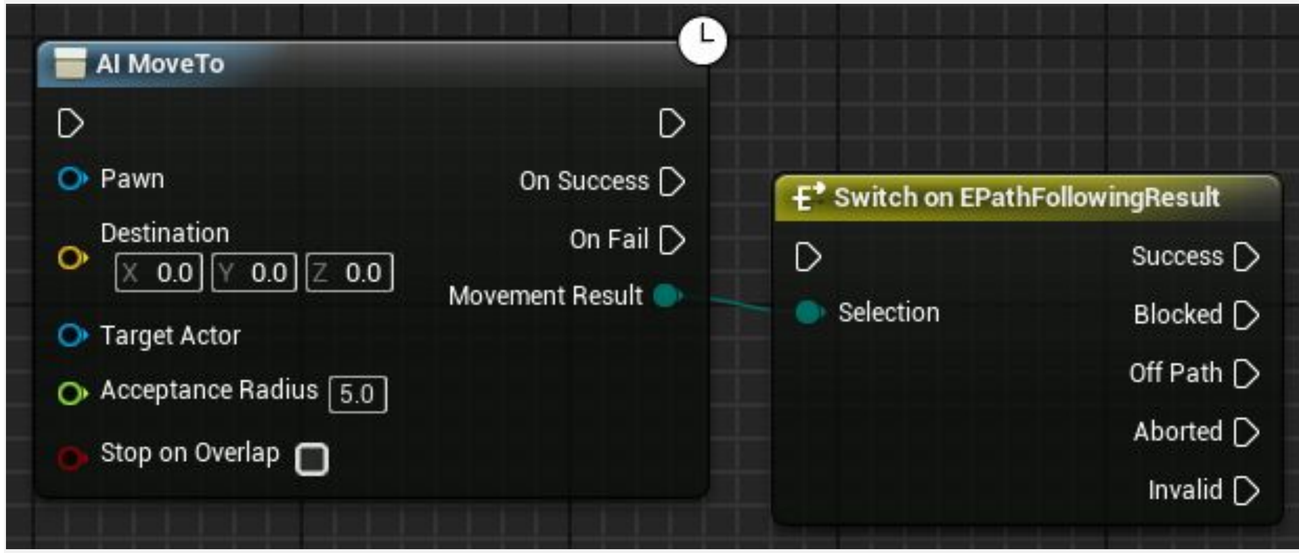
AI MOVE TO:
INPUT AND OUTPUT

Input

- **Pawn**: Reference to the Pawn that will be moved.
- **Destination**: Vector indicating the destination.
- **Target Actor**: Reference to an Actor that will be continuously tracked.
- **Acceptance Radius**: Maximum destination distance to complete movement.
- **Stop on Overlap**: Boolean value. If the value is “**true**”, the move will be complete as soon as the Pawn begins to overlap the acceptance radius.

Output

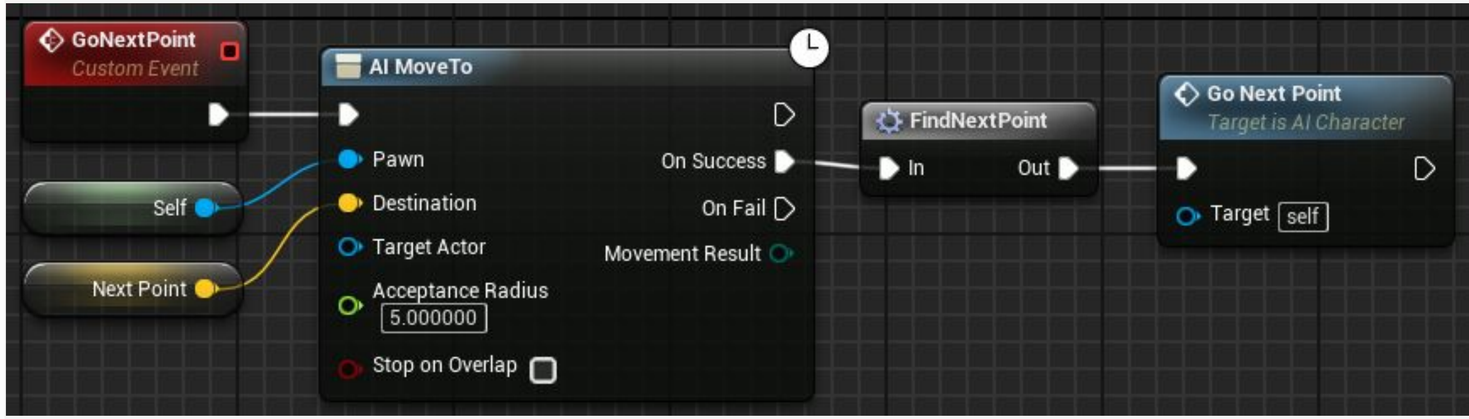
- **On Success**: Exec pin that will be executed when the **AI MoveTo** function reaches its goal.
- **On Fail**: Exec pin that will be executed if the **AI MoveTo** function cannot reach its goal.
- **Movement Result**: Enumeration with the following possible result: “Success”, “Blocked”, “Off Path”, “Aborted”, or “Invalid”.



AI MOVE TO: EXAMPLE

In the example on the right, the **AI MoveTo** node is being used to direct a Pawn to follow a path represented by a set of predefined points in the Level. These points representing the path may be stored in an array. The **FindNextPoint** macro is responsible for picking the next point on the path and storing it in the **Next Point** variable.

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



EXECUTE CONSOLE COMMAND

The **Execute Console Command** function allows the execution of a console command.

Console commands are text-based commands that can be executed in the Editor or in the game.

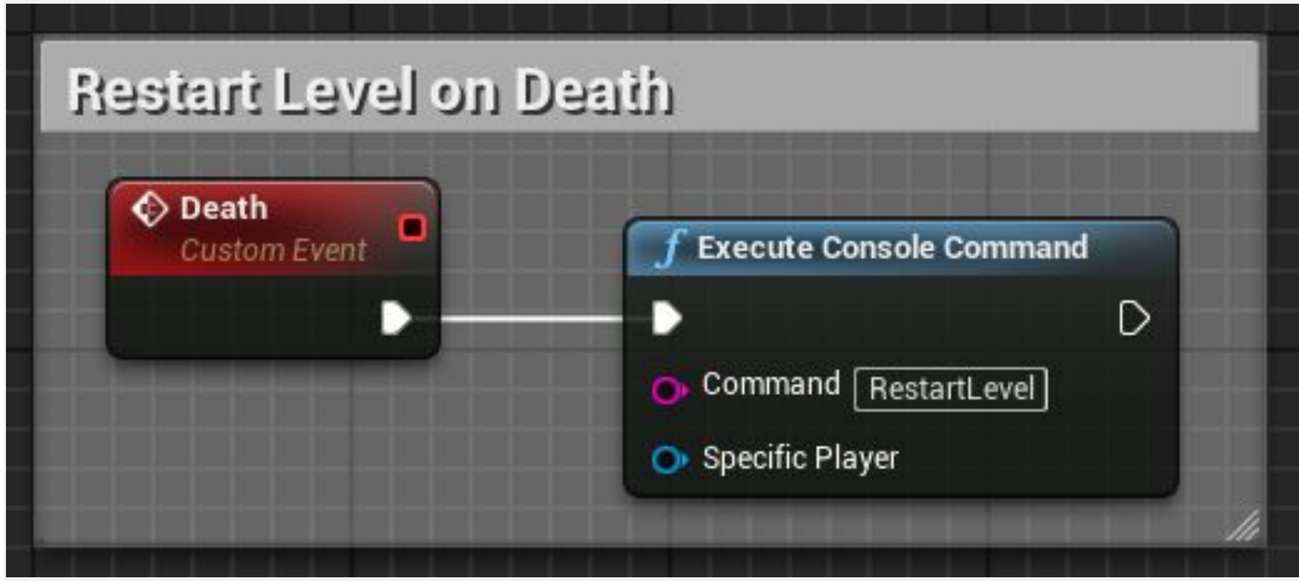
Input

- **Command**: Console command that will be executed.
- **Specific Player**: Reference to a Player Controller that will receive the command. Use of this parameter is optional.

EXECUTE CONSOLE COMMAND: EXAMPLE

In a simple game, the Level will be restarted when the player dies. In the example on the right, 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 the **Enter** key to get the list of commands.



SUMMARY

This lecture presented more Blueprint functions. It also showed how to use the Add Child Actor Component and the AI MoveTo functions and how to execute a console command in Blueprint.