Blueprint Best Practices

Tips and tricks to help you make decisions about when to use Blueprints and how best to set them up.

You can do many things with **Blueprints**. Everything from making a small game or procedural content tools, to prototyping new functionality, to tweaking and polishing things made by programmers, is possible with the Blueprint visual scripting system.

However, there are certain things that will impact your performance more if they are done in Blueprints. If you have a Blueprint that's doing a lot of operations and complex math every tick, you might want to consider using native C++ code. Blueprints are best suited to making event-driven functionality, such as handling damage taking, controls, and other things that don't get called every frame.

If you would like to read more about coding for Blueprints, or the technical details of Blueprint compiling, see the Blueprints Technical Guide.

Even if your functionality is well-suited for Blueprints, there are decisions you can make in setting up your Blueprints that will make things go more smoothly. This guide will go over some of the more common decisions you will have to make, as well as tips and tricks for Blueprint users.

Level Blueprints vs. Blueprint Classes

Level Blueprints may be very familiar to users of UE3's Kismet, because you can select objects directly in the level, and operate on them. They're great for one-off prototypes, and getting familiar with the Blueprints system, but they are specific to the level that they're used in. This does mean that Level Blueprints can be great places to set up functionality specific to the level, or Actors in it. Some examples would be kicking off a cinematic when a certain trigger is touched, or opening a particular door after you kill all the enemies.

In general, Blueprint Classes are the best way to get reusable behavior in your project. If you create a Blueprint Class, you can add it to any of your levels, and you can also add as many copies as you would like to the level, without needing to copy script around.

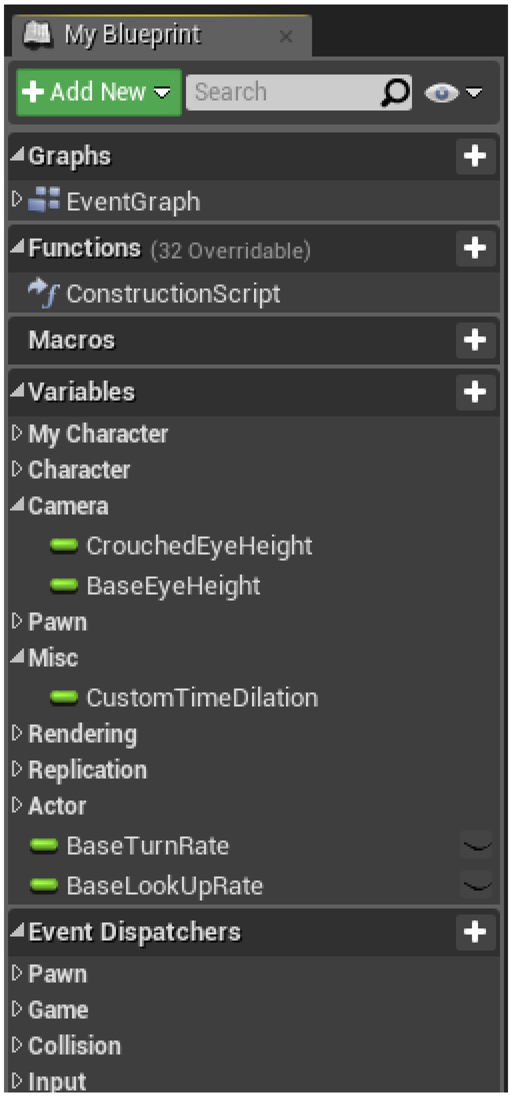
If you do start out in the Level Blueprint, and then decide to move behavior into a Blueprint Class, this should be a pretty simple process. When you copy functions from a Level Blueprint that operate on Actors (say, calling Set Brightness on a light), and paste them into a Blueprint Class based off of a Light Actor, the functions will update to be in the proper scope.

Read More:

- Level Blueprint
- Blueprint Class

Class Variables vs. Local Variables

When you're working in your Blueprint, you have the ability to add variables using the Blueprint Editor My Blueprint Panel tab. When you're inside a function though, you'll see an additional section of the **My Blueprint** tab for **Local Variables**.



Local variables are scoped, which means that they only exist where you define them. So, a local variable in a function is only visible to that function, and not to other functions, or the Event Graph. This helps to reduce clutter for things that are only relevant within the context of that function. You can consider local variables a "scratch pad" to work within a function. You can use them to help perform what the function needs to, but they get tossed away after the function has completed.

Class variables are for things that you are potentially going to be interested in accessing from multiple places in the Blueprint. So, things like references to meshes or other components, or variables that need to be public so you can access them from other Blueprints, should be class variables. They're the things you don't want to forget about after you use them.

Functions vs. Macros

Functions and **Macros** both accomplish the same goal externally: they let you send input into a node, something happens to it, and then outputs come out. They accomplish these in two different ways, but they have some similarities:

- Both have a central place where you can edit the functionality of what the node does (macros in their macro graph, and functions in their function graph), and every node that calls the function or macro will be updated whenever that central place is changed.
- Both are reusable.
- Both allow for local variables. Macros use "anonymous" local variables, which don't have a specific name, but just hold a value.
- Both are awesome ways to encapsulate functionality for reusability and clarity.

There are some key differences that will become more apparent as you get into more advanced use cases.

- Functions are actually called when you place nodes to call the function. That means you can target them (i.e. "call a function on another object"), and they allow for communication between Blueprints.
- Macros take the nodes from the macro graph, and actually replace the macro node with a copy of all those nodes. Basically, when the Blueprint is compiled, the macro copies all the graph nodes, and pastes them in where the macro node is.

Because of how they work under the hood, that means there are a few differences in what you can do:

- Macros can use any node for the class they're scoped to (the class you pick when you create a new Macro Blueprint, or if it's a local macro, the class the macro is defined in). This means that you have a little more versatility in the nodes you can put in a macro.
- One of the biggest differences between a function and a macro is that you can place latent nodes in a macro, but not in a function.
- Functions allow you to override their functionality in child Blueprints. For instance, suppose you had a car Blueprint that had a function "PlayerInteractedWithMe". In that function, you might play a horn honk. Now, suppose you had two children of the car Blueprint, for a police car and a fire truck. In the police car, you could override the function to play a siren noise and turn on the lights. In the fire truck, you could have it squirt water. That sort of overriding functionality isn't possible with macros.
- Since macros are just copy-and-pasted into the graph at compile-time, you can have multiple execution wires in and out of a macro. You can't do that in a function.

In general, a good rule of thumb might be to use a macro if there's a quick bit of functionality that you want to reuse everywhere. But, if you might want to tweak that behavior in children of the Blueprint, or you want to access it directly from another Blueprint, it's better to use a function!

Read More:

- Functions
- Macros

Blueprint Communication Types

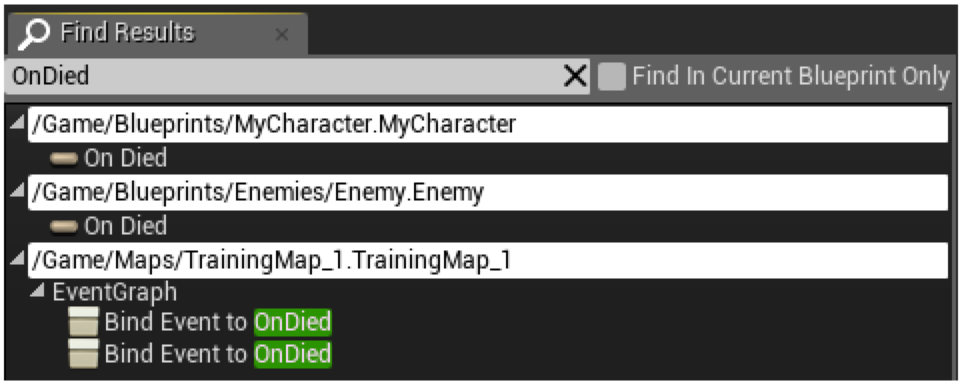
There are several different methods for communicating between two Blueprints. While the most common use case is direct Blueprint communication, sometimes the functionality you are going for means that you will use Event Dispatchers or Blueprint Interfaces instead. For an overview of each type of communication, some example use cases, and tutorials to help you get started, see Blueprint Communication Usage.

General Tips

Working in the Blueprint Editor

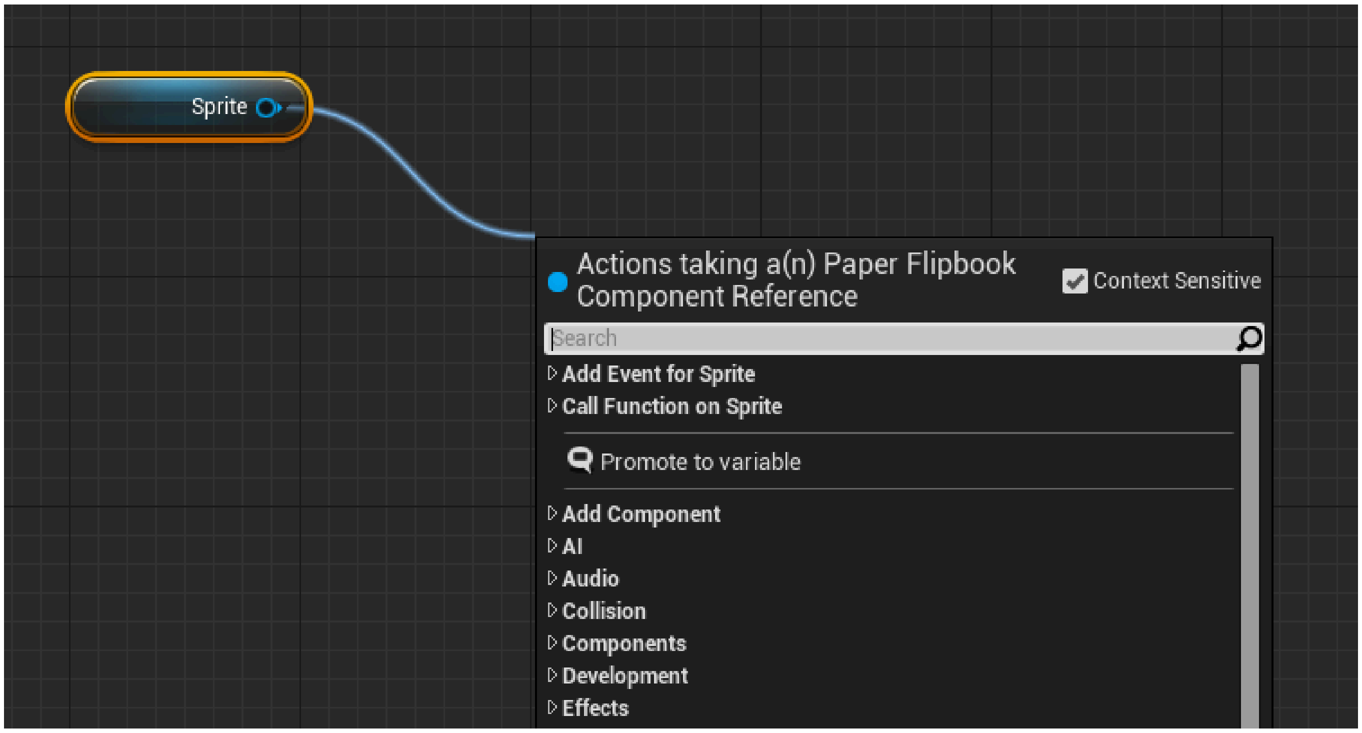
When working in the Blueprint Editor, there are a few tricks you can use to find the nodes that you want to use, as well as nodes and comments you've already created.

- Use the **Search** button in the Blueprint Editor when you're looking for something in an existing Blueprint, whether it's a variable, function, or comment. It searches all Blueprints, even unloaded ones, if you uncheck the **Find in Current Blueprint** only box, so it can help you track down just where you implemented something.

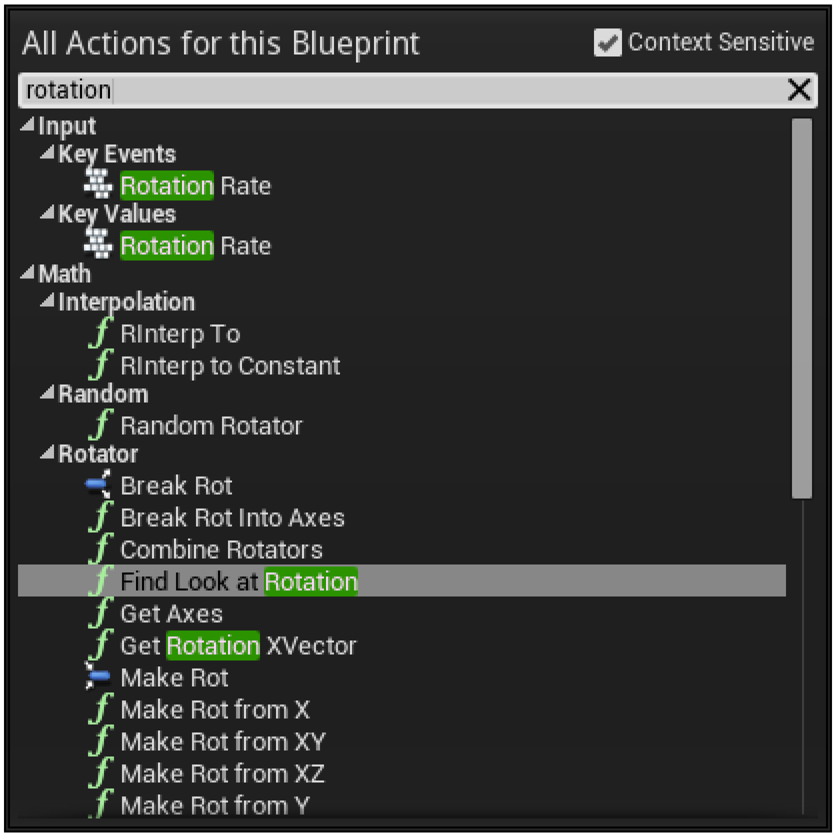


```
OnDied
/Game/Blueprints/MyCharacter.MyCharacter
    On Died
/Game/Blueprints/Enemies/Enemy.Enemy
    On Died
/Game/Maps/TrainingMap_1.TrainingMap_1
  EventGraph
      Bind Event to OnDied
      Bind Event to OnDied
```

- The context menu only shows nodes that you can connect to the pin you dragged from (e.g. when you drag from a light, it shows you functions relevant to the light). When you're trying to discover all the possible functionality available to act on a variable or component, just browse through the context menu and see what is available.



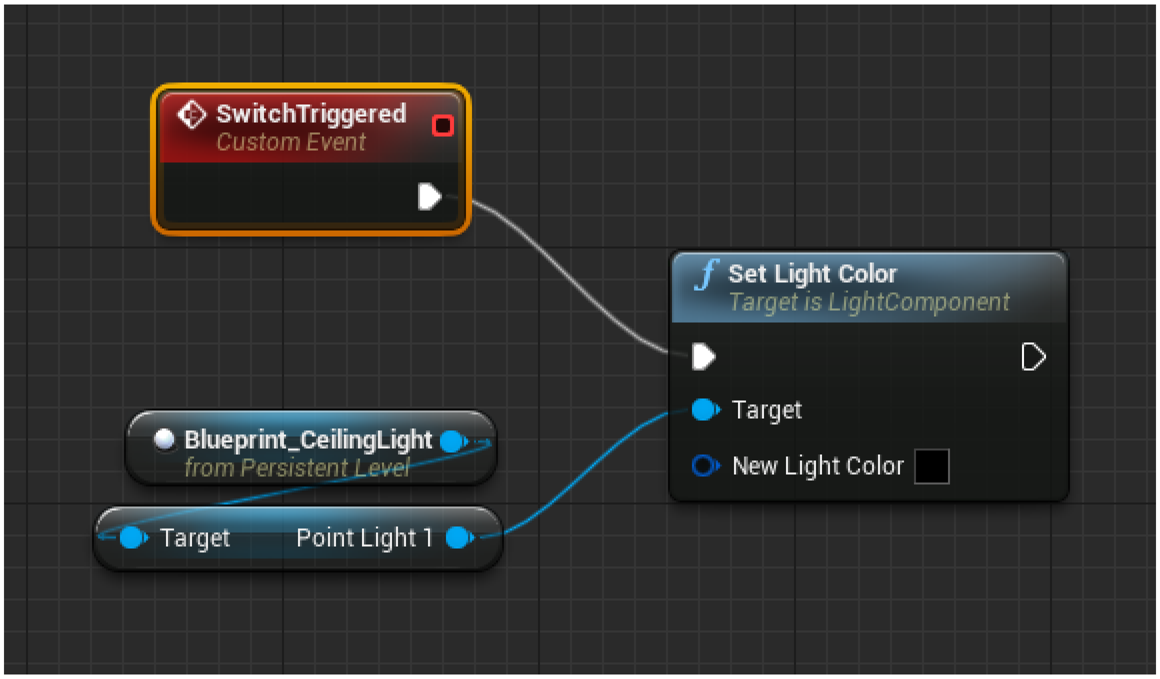
- Whether you're in the context menu or the palette, the search bar can be very helpful. We tag functions with lots of keywords to help you find what you're looking for, even if you don't know what it's called!



Setting Up Your Graph

While everyone has their own graph setup preferences, here are a few tips that can help you keep things organized and easy to follow:

- Keep things tidy from the get go! It's much harder to clean up after you've made a bunch of code than to work clean as you go.
- If you find yourself using the same set of nodes more than twice in a graph, consider making it a function or a macro so you can reuse it.
- You can stack nodes with their context, because it keeps things more spatially compact. For example, if you have a reference to a light, then you access its point light component, you can stack those two nodes one on top of the other, and think of them as a single block.



- Use comments often! Also, don't forget that you can change the color of comment boxes, for extra help identifying sections of your graph.