

Developer

/ Documentation

/ Unreal Engine ▾

/ Unreal Engine 5.4 Documentation

/ Programming and Scripting

/ Blueprints Visual Scripting

/ Blueprints Technical Guide

/ Exposing C++ to Blueprints

# Exposing C++ to Blueprints

Tips and tricks for how best to write a Blueprint-friendly API



There are two driving considerations when deciding to use C++ or Blueprints:

- Speed
- Complexity of expression

Beyond those two factors, a lot of it comes down to the complexity of the game, and the composition of the team. If you have many more artists than programmers, you'll probably have significantly more Blueprints than C++ code. In contrast, if you have a lot of programmers, they're probably keen to keep things in C++. We expect people to fall somewhere in the middle. At Epic, a lot of the workflow is that content creators will make a very complex Blueprint, and a programmer can see how they can compress a lot of that work into C++ by coding a new Blueprint node, so they move that chunk of functionality into a new C++ function. A good practice to follow would be to use Blueprints extensively, and then push things into C++ as they reach a complexity where that would enable a more concise expression of the functionality (or it becomes too complex for a non-programmer), or speed of execution dictates a move to C++.

## Speed

For speed, the fact is that Blueprint execution is slower than C++ execution. That's not to say performance is bad, but if you're doing something that requires a lot of calculations, or operates at a high frequency, it might be better to use C++ instead of Blueprints. However, it is possible to combine the two in the way that works best for your team and your project's performance. If you have a Blueprint that has a lot of functionality, you can push some of that functionality into C++ to speed it up, but keep the rest in Blueprints to retain the flexibility. If your profiling shows that one operation is taking a lot of time in Blueprints, consider moving just that section into C++, while keeping the rest in Blueprints.

An example of a system that would take a lot of time to execute if done with Blueprint visual scripting is a crowd system that controls a thousand Actors. In this case, it would be better for performance to handle decision making, pathing, and other crowd functionality in C++, and then maybe expose some tweaking parameters and controlling functions to Blueprints.

## Complexity

For complexity of expression, there are just some things that are easier to do in C++ than in Blueprints. Blueprints do a lot of things well, but some things are just not very easy to express in nodes. Operating on large sets of data, doing string manipulation, complex math over large data sets, etc. are all very complex, and aren't easy to follow in a visual system. Those things are better kept in C++ than in Blueprints, just because they're easier to look at and figure out what's going on. Complexity of expression is another reason a crowd system would be better to implement in C++ code than in Blueprints.

## Examples

Since there are different pieces of functionality that are best handled in C++ or in Blueprints, here are some examples of how C++ programmers and Blueprint authors can work together while creating a game.

- The programmer could create a Character class in C++ that defines some custom events, and then Blueprints could be used to extend that Character class and actually assign meshes and set defaults. Check out the player character and the enemy bots in the ShooterGame sample project to see implementations like this.
- A ability system where the base class is implemented in C++ but then designers can create Blueprints which actually do something. In the StrategyGame sample, there is a

base turret defined in C++, but the behavior of the flamethrower, cannon, and arrow turrets is all defined in those Blueprints.

- A pickup where the "Collect" or "Respawn" functions are Blueprint Implementable Events that can be overridden for designers to spawn different particle emitters and sound effects. Both ShooterGame and StrategyGame have examples of pickups created this way.

## Creating a Blueprint API : Tips and Tricks

Here are some points to consider as a programmer creating a Blueprint-exposed API:

- Optional parameters are handled well in Blueprints:

```
1  /**
2   * Prints a string to the log, and optionally, to the screen
3   * If Print To Log is true, it will be visible in the Output Log window.
   * Otherwise it will be logged only as 'Verbose', so it generally won't
   show up.
4   *
5   * @param InString The string to log out
6   * @param bPrintToScreen Whether or not to print the output to the screen
7   * @param bPrintToLog Whether or not to print the output to the log
8   * @param bPrintToConsole Whether or not to print the output to the
   console
9   * @param TextColor Whether or not to print the output to the console
10  */
11  UFUNCTION(BlueprintCallable, meta=(WorldContext="WorldContextObject",
   CallableWithoutWorldContext, Keywords = "log print", AdvancedDisplay =
   "2"), Category="Utilities|String")
12  static void PrintString(UObject* WorldContextObject, const FString&
   InString = FString(TEXT("Hello")), bool bPrintToScreen = true, bool
   bPrintToLog = true, FLinearColor TextColor =
   FLinearColor(0.0,0.66,1.0));
13
```


 Copy full snippet

- Favor functions with lots of return parameters over functions that return structs. Here's a snippet showing how to create multiple output pins on your node:

```

1 UFUNCTION(BlueprintCallable, Category = "Example Nodes")
2 static void MultipleOutputs(int32& OutputInteger, FVector& OutputVector);
3

```

 Copy full snippet

- Adding new parameters to an existing function is fine, but if you need to remove or change them, you should deprecate the original and add a new function. Make sure to use the deprecation metadata so the information about the new function will show up in Blueprints:

```

1 UFUNCTION(BlueprintCallable, Category="Collision", meta=
    (DeprecatedFunction, DeprecationMessage = "Use new CapsuleOverlapActors",
    WorldContext="WorldContextObject", AutoCreateRefTerm="ActorsToIgnore"))
2 static ENGINE_API bool CapsuleOverlapActors_DEPRECATED(UObject*
    WorldContextObject, const FVector CapsulePos, float Radius, float
    HalfHeight, EOverlapFilterOption Filter, UClass* ActorClassFilter, const
    TArray<AActor*>& ActorsToIgnore, TArray<class AActor*>& OutActors);
3

```

 Copy full snippet

- If a function needs to take an enum, consider using the 'expand enum as execs' metadata, as this can make the node easier to use.

```

1 UFUNCTION(BlueprintCallable, Category = "DataTable", meta =
    (ExpandEnumAsExecs="OutResult", DataTablePin="CurveTable"))
2 static void EvaluateCurveTableRow(UCurveTable* CurveTable, FName RowName,
    float InXY, TEnumAsByte<EEvaluateCurveTableResult::Type>& OutResult,
    float& OutXY);
3

```

 Copy full snippet

- Many operations that take time to complete (e.g. move here) should be latent functions.

```


1 /**
2  * Perform a latent action with a delay.
3  *
4  * @param WorldContext World context.

```

```

5 * @param Duration length of delay.
6 * @param LatentInfo The latent action.
7 */
8 UFUNCTION(BlueprintCallable, Category="Utilities|FlowControl", meta=
    (Latent, WorldContext="WorldContextObject", LatentInfo="LatentInfo",
    Duration="0.2"))
9 static void Delay(UObject* WorldContextObject, float Duration, struct
    FLatentActionInfo LatentInfo );
10

```

 Copy full snippet

- Consider putting functions in a shared library if possible. This makes them easy to use across multiple classes, and avoids a 'target' pin.

```

1 class DOCUMENTATIONCODE_API UTestBlueprintFunctionLibrary : public
    UBlueprintFunctionLibrary
2

```

 Copy full snippet

- Remember to mark nodes as pure if possible, as this will avoid needing an execution pin on a node that requires wiring.

```

1 /* Returns a uniformly distributed random number between 0 and Max - 1 */
2 UFUNCTION(BlueprintPure, Category="Math|Random")
3 static int32 RandomInteger(int32 Max);
4

```

 Copy full snippet

- Marking a function as `const` will also cause the Blueprint node to not have execution pins:

```

1 /**
2 * Get the actor-to-world transform.
3 * @return The transform that transforms from actor space to world space.
4 */
5 UFUNCTION(BlueprintCallable, meta=(DisplayName = "GetActorTransform"),
    Category="Utilities|Transformation")
6 FTransform GetTransform() const;

```

 Copy full snippet