Blueprints Visual Scripting Overview

The Blueprint Overview page breaks down the anatomy of a Blueprint and the different types of Blueprints available.

The **Blueprint Visual Scripting** system in Unreal Engine is a visual programming language that uses a node-based interface to create gameplay elements. The node-based workflow provides designers with a wide range of scripting concepts and tools that are generally only available to programmers. In addition, Blueprint-specific markup available in Unreal Engine's C++ implementation provides programmers with a way to create baseline systems that designers can extend.

As with many common scripting languages, you can use the system to define object-oriented (OO) classes or objects in the engine. The system, along with the objects you define, are often referred to as just "Blueprints".

Does that mean Blueprints are a replacement for UnrealScript? Yes and no. Gameplay programming and everything that UnrealScript was used for in the past can still be handled through code using C++. At the same time, while Blueprints are not meant as a replacement for UnrealScript, they do serve many of the same purposes that UnrealScript handled, such as:

- Extending classes
- Storing and modifying default properties

- Managing subobject (e.g. Components) instancing for classes

The expectation is that gameplay programmers will set up base classes which expose a useful set of functions and properties that Blueprints made from those base classes can use and extend upon.

The table below provides a comparison of how various aspects would be handled in UnrealScript (from Unreal Engine 3), C++, and Blueprints to help those transitioning from previous versions of the engine as well as show how native code and Blueprints compare.

UnrealScript (UE3)	Blueprints (UE5)	C++ (UE5)
.uc file	Blueprint Asset	.h/.cpp files
UClass	UBlueprintGeneratedClass	UClass
extends [ClassName]	ParentClass	: [ClassName]
Variable	Variable	UProperty()
Function	Graphs/Events	UFunction()
defaultproperties{}	Class Defaults	native constructor
Default Components	Components List	native constructor

Types of Blueprints

Blueprints can be one of several types that each have their own specific use from creating new types to scripting level events to defining interfaces or macros to be used by other Blueprints.

Blueprint Class

A **Blueprint Class**, often shortened as **Blueprint**, is an asset that allows content creators to easily add functionality on top of existing gameplay classes. Blueprints are created inside of Unreal Editor visually, instead of by typing code, and saved as assets in a content package. These essentially define a new class or type of Actor which can then be placed into maps as instances that behave like any other type of Actor.

Data-Only Blueprint

A **Data-Only Blueprint** is a Blueprint Class that contains only the code (in the form of node graphs), variables, and components inherited from its parent. These allow those inherited properties to be tweaked and modified, but no new elements can be added. These are essentially a replacement for archetypes and can be used to allow designers to tweak properties or set items with variations.

Data-Only Blueprint are edited in a compact property editor, but can also be "converted" to full Blueprints by simply adding code, variables, or components using the full **Blueprint Editor**.

Refer to Class Blueprint for additional documentation.

A **Level Blueprint** is a specialized type of **Blueprint** that acts as a level-wide global event graph. Each level in your project has its own Level Blueprint created by default that can be edited within the Unreal Editor, however new Level Blueprints cannot be created through the editor interface.

Events pertaining to the level as a whole, or specific instances of Actors within the level, are used to fire off sequences of actions in the form of Function Calls or Flow Control operations. Those familiar with Unreal Engine 3 should be very familiar with this concept as this is very similar to how Kismet worked in Unreal Engine 3.

Level Blueprints also provide a control mechanism for level streaming and Sequencer as well as for binding events to Actors placed within the level.

Refer to Level Blueprint for additional documentation.

Blueprint Interface

A **Blueprint Interface** is a collection of one or more functions - name only, no implementation - that can be added to other Blueprints. Any Blueprint that has the Interface added is guaranteed to have those functions. The functions of the Interface can be given functionality in each of the Blueprints that added it. This is essentially like the concept of an interface in general programming, which allows multiple different types of Objects to all share and be accessed through a common interface. Put simply, Blueprint Interfaces allow different Blueprints to share with and send data to one another.

Blueprint Interfaces can be made by content creators through the editor in a similar fashion to other Blueprints, but they come with certain limitations in that they cannot:

- Add new variables
- Edit graphs
- Add Components

Refer to Blueprint Interface and Inteface QuickStart Guide for additional documentation.

Blueprint Macro Library

A **Blueprint Macro Library** is a container that holds a collection of **Macros** or self-contained graphs that can be placed as nodes in other Blueprints. These can be time-savers as they can store commonly used sequences of nodes complete with inputs and outputs for both execution and data transfer.

Macros are shared among all graphs that reference them, but they are auto-expanded into graphs as if they were a collapsed node during compiling. This means that Blueprint Macro Libraries do not need to be compiled. However, changes to a Macro are only reflected in graphs that reference that Macro when the Blueprint containing those graphs is recompiled.

Refer Macro Library and Making Macrosfor additional documentation.

Blueprint Utilities

A **Blueprint Utility** (or **Blutility** for short), is an editor-only Blueprint that can be used to perform editor actions or extend editor functionality. These can expose Events with no parameters as buttons in the UI and have the ability to execute any functions exposed to *Blueprints* and act on the current set of selected Actors in the viewport.

Blueprint Anatomy

The functionality of Blueprints is defined using various elements; some of which are present by default, while others can be added on an as-needed basis. These provide the ability to define Components, perform initialization and setup operations, respond to events, organize and modularize operations, define properties, and more.

Components Window

With an understanding of Components, the **Components** window inside the **Blueprint Editor** allows you to add Components to your Blueprint. This provides a means of adding collision geometry via CapsuleComponents, BoxComponents, or SphereComponents, adding rendered geometry in the form of StaticMeshComponents or SkeletalMeshComponents, controlling movement using MovementComponents, etc. The Components added in the Components list can also be assigned to instance variables providing access to them in the graphs of this or other Blueprints.

Construction Script

The **Construction Script** runs following the Components list when an instance of a Blueprint Class is created. It contains a node graph that is executed allowing the Blueprint Class instance to perform initialization operations. This can be extremely powerful as actions like performing traces into the world, setting meshes and materials, and so on can be used to achieve context-specific setup. For instance, a light Blueprint could determine what type of ground it is placed upon and choose the correct mesh to use from a set of meshes or a fence Blueprint could perform traces extending out in each direction to determine how long of a fence is needed to span the distance.

Event Graph

The EventGraph of a Blueprint contains a node graph that uses events and function calls to perform actions in response to gameplay events associated with the Blueprint. This is used to add functionality that is common to all instances of a Blueprint. This is where interactivity and dynamic responses are setup. For example, a light Blueprint could respond to a damage event by turning off its `LightComponent` and changing the material used by its mesh. This would automatically provide this behavior to all instances of the light Blueprint.

Refer to EventGraph for additional documentation.

Functions

Functions are node graphs belonging to a particular **Blueprint** that can be executed, or called, from another graph within the Blueprint. Functions have a single entry point designated by a node with the name of the Function containing a single exec output pin. When the Function is called from another graph, the output exec pin is activated causing the connected network to execute.

Variables

Variables are properties that hold a value or reference an Object or Actor in the world. These properties can be accessible internally to the **Blueprint** containing them, or they can be made accessible externally so that their values can be modified by designers working with instances of the Blueprint placed in a level.