Community Experience Distilled

Discover all the secrets of Unreal Engine and create seven fully functional games with the help of step-by-step instructions

Unreal Engine Game Development Blueprints

Nicola Valcasara

We start with an introductory chapter to help you move fluidly inside the Blueprint user interface, recognize its different components, and understand any already written Blueprint script. Following this, you will learn how to modify generated Blueprint classes to produce a single-player tic-tac-toe game and personalize it.

Next, you will learn how to create simple user interfaces, and how to extend Blueprints through code. This will help you make an informed decision between choosing Blueprint or code. You will then see the real power of Unreal unleashed as you create a beautiful scene with moving, AI-controlled objects, particles, and lights.

Then, you will learn how to create AI using a behavior tree and a global level Blueprint, how to modify the camera, and how to shoot custom bullets. Finally, you will create a complex game using Blueprintable components complete with a menu, power-ups, dangerous objects, and different weapons.

Who this book is written for

This book is ideal for intermediate-level developers who know how to use Unreal Engine and want to go through a series of projects that will further develop their expertise. A working knowledge of C++ is a must.

What you will learn from this book

- Write clean and reusable Blueprint scripts
- Develop any kind of game you have in mind, following the rules used by experts
- Move through Unreal Engine 4, always knowing what you are doing and where to find the right tool for your needs
- Integrate C++ code into your projects using Visual Studio and the tools that Unreal provides
- Differentiate between classes, nodes, interfaces, macros, and functions
- Work with different types of assets, from 3D objects to audio sources, from UI buttons to animations
- Explore all the aspects of the game logic—collisions, navigation meshes, matinees, volumes, events, and states


Nicola Valcasara

$ 49.99 US
£ 31.99 UK

Prices do not include local sales tax or VAT

Unreal Engine Game Development Blueprints

Community Experience Distilled

Discover all the secrets of Unreal Engine and create seven fully functional games with the help of step-by-step instructions
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Getting Started with Unreal Blueprints'
- A synopsis of the book’s content
- More information on Unreal Engine Game Development Blueprints
Nicola Valcasara is a freelance game developer and cofounder of Deuxality Games Ltd. He is an expert programmer, specializing in mobile development, with a strong passion for games and technology. He started to work in the game industry in 2012, after winning the first prize at the Microsoft Rapid2D competition for young developers.

He has also been a reviewer of *Unreal Engine Android Game Development*, Packt Publishing.
Preface

This book will help you learn how to develop wonderful games using Unreal Engine 4 and its Blueprint Visual Scripting.

Discover all the secrets of this engine and create seven fully functional games with step-by-step instructions. In this book, you will learn the secrets of Blueprint; from the single node to the most complex function. Whether you are a beginner or an expert programmer, this guide will introduce you to this world and show you the infinite possibilities that this engine can offer by developing seven exciting and fully functional games.

What this book covers

Chapter 1, Getting Started with Unreal Blueprints, introduces you to the Unreal Engine editor and Blueprint graph. We will create the first project on both Unreal Engine and Visual Studio 2013.

Chapter 2, Tic-Tac-Toe, covers a simple game: a player versus player, classic board game. You will learn how to the create a Blueprint graph with nodes and wires.

Chapter 3, C++ Code – PAC-MAN, explains the creation of a classic coin-up game only using the C++ code. You will learn how to communicate between Visual Studio and UE4 in this chapter.

Chapter 4, UFO Run - Play with the Environment Effects, explains the particle effect system and user interface tools by creating an action game, starting from a template offered by the engine.

Chapter 5, Top-Down Shooter, helps you to play with animations and create an artificial intelligence by customizing the assets that are offered by the marketplace.
Chapter 6, *A Platform Maze*, explains how to use Matinee to create short cinematic clips or move objects around a level. Use the physics to handle ragdolls and destructible objects.

Chapter 7, *An Open World Survival Game*, creates huge worlds with the terrain manipulation tools that are offered by the engine, populate them with object using the brush tools, and give them a life by creating an inventory system using the knowledge learned during the book.
Welcome! If you have arrived here, it is because you want to look at Blueprints in depth and learn all its secrets, from the simplest node to the most complex code extension. This is an introductory chapter. Here, you will take your first steps in Blueprint, you will create your first project, and start with the editor, learning its interfaces and its tools.

In this chapter, we will cover the following:

- What is Blueprint?
- Different types of data, nodes and Blueprint
- Knowing the environment
- Debugging your Blueprints
- Creating a visual studio solution
What is Blueprint?
Blueprint is a high level, visual scripting system that provides an intuitive, node-based interface that can be used to create any type of script events in the Unreal editor. The tools that are provided can be used by level designers, artists, and any non-programmer person, to quickly create and iterate gameplay (or even create entire games) without ever needing to write a line of the code:

For those of you coming from UE3, Blueprint is the evolution of Kismet. It inherits most of the strong keys of the Kismet system, adding the full range of concepts and tools that are generally only available to programmers.

Through the use of Blueprints, anyone can virtually prototype, implement, or modify any gameplay element. Here, we are going to discover how to create most of them. The following is a list of common uses that are covered by this guide:

- **Games:** Sets up game rules and tweaks gameplay conditions
- **Players:** Creates variants with different meshes and materials, or allows character customization
- **Cameras:** Changes the camera dynamically during play
- **Inputs:** Handles the inputs that are passed by the player
- **Items:** Includes weapons, pickups, triggers, and so on
- **Environment:** Creates randomized props or procedurally generated items
In order to understand Blueprint, we first need to understand its structure. The following image is an extremely simplistic graph that shows where Blueprint is collocated in a game and who are its parent and child:

![Diagram of Blueprint structure]

Each of these elements can have multiple children and each element has its different type and behavior.

**Types of Blueprints**

There are four main types of Blueprint, each one has a specific purpose and is useful in a specific situation. We will learn how to choose the correct one while studying the examples of this guide; however, in the meantime, let's take a look at them in order to understand their differences.

**Level Blueprints**

A Level Blueprint is a specialized type of Blueprint that, as the name suggests, acts as a level-wide event graph. A level Blueprint is created by default for each of your levels and can be edited only in the Level Blueprint Editor. This is the only type that cannot be created and there is only one Level Blueprint for each level.

In this Blueprint file we handle the level flow: we can control events, Matinee, and sequences of actions in the form of Function Calls or Flow Control operations.
To open the Level Blueprint for the purpose of editing, click on the Blueprints button in the Level Editor toolbar and select Open Level Blueprint, as follows:

Blueprint class
A Blueprint class, simply called Blueprint, is the most used type and you will become familiar with it during this guide. This type allows the content creator to easily add functionality on top of any existing gameplay classes. A Blueprint class extends a parent (either a code parent or another Blueprint class) and can be edited with a visual editor. Any Blueprint class that is created in the editor can be found in Content Browser and can be added to the map as an instance, like any other type of Actor.

The following are the most common Parent Classes that are used when creating a new Blueprint:

- **Actor**: It is an object that can be placed or spawned in the world
- **Pawn**: It is an Actor that can be possessed and it receives input from a Controller (which can be a user or an Artificial Intelligence)
- **Character**: It is a Pawn that includes the ability to walk, run, jump, and so on
- **PlayerController**: It is an Actor that is responsible for controlling a Pawn
- **Game Mode**: It defines the game rules, scores, and any aspect of a game type

Data-Only Blueprint
Data-only Blueprints are basically Blueprint classes without the node graph. They contain all the properties and components that are inherited from its parent and allow the user to tweak properties or set items with variations without needing to find these properties in a big node graph.
A data-only Blueprint doesn't allow you to add new elements; however, it can be converted in a Blueprint class with just one click, if required:

**Blueprint Interface**

A Blueprint Interface is similar to an interface in general programming. It allows different types of objects to share a common information setup. It is a collection of one or more functions (declarations only, no implementations) that can be added to other Blueprints. A Blueprint Interface needs to be added to a Blueprint class in order to work, and a Blueprint class that has implemented an interface can have and use all of its functions.

A Blueprint Interface can be made in the editor; however, it has limitations as it cannot do the following:

- Add new variables
- Edit graphs
- Add components
A good example to understand an interface is that a player, a tree, and a concrete wall are three completely different objects but all of them can receive a projectile shot by a weapon. Instead of creating a different function for all of them, an interface can help us by creating a function called onReceiveDamage that is shared (however, implemented differently) by all of them.

**Blueprint Macro Library**

A Blueprint Macro Library is a container that holds a collection of Macros or graphs that can be placed as nodes in other Blueprints. They are very handy as they can store the commonly used sequences of nodes with inputs and outputs for execution and data transfer.

**Knowing the environment**

Let’s take a look at **Unreal Engine 4** and its editor. I am assuming that you have already installed the engine and visual studio 2013 on your machine; therefore, I will skip the process of registering, downloading, and installing the engine. If this is not the case, you can go to the epic website (www.unrealengine.com), sign up for free and get your copy by following their instructions with a couple of easy steps.

**Creating a project**

Open the Unreal Engine Launcher. Under the **Library** section, choose the version of the engine that you prefer, and launch it, as follows:
The **Unreal Project** browser will open. By default, you will see the **Projects** screen. Here, you can see your projects and the samples that you downloaded from the **Marketplace**. For our purpose, we want to create a brand new and empty project. Under the **New Project** section, you can choose between the **Blueprint** or **C++** projects in a list of built-in templates:

![Unreal Project Browser](image)

Due to the nature of Blueprint, the code and Blueprint live happily together. These choices are different in only one way: the C++ project will also create the visual studio solution for your project but each of those choices will generate the same Uproject and the needed files to launch the editor.

Due to this harmony between Blueprint and code, if you choose to create a project from the **Blueprint** section you can, at any time, generate its C++ project: the engine will create the Visual Studio solution as soon as you add your first code class from the editor (**File** | **Add Code to Project**).
Choose a Blank Blueprint project, name it and choose a location (the default is C://Users/Your Name/Documents/Unreal Projects/). Before creating the project you can also set three main aspects of it: the general graphic quality, the device target (mobile, pc, console), and if you want to you can include the Unreal Engine Starter Content in it (the Starter Content contains some useful general purpose assets such as primitive meshes, particle effects, materials, and so on).

For our purpose, we can leave those settings as is and click Create Project.

**Creating your first Blueprint class**

Welcome to the Unreal Engine 4 editor. You will now see the example map opened and ready for your input in front of you. We are not creating anything fancy right now: we will only explore the user interface of Blueprint and start to learn the basic commands and shortcuts in Blueprint.

There are two ways to create a Blueprint class: from Content Browser or from the top tool bar. The toolbar Blueprint button gives you quick access to the existing modifiable Blueprint classes and you can access to the Level Blueprint only from here. Be aware that from here you can only create the Blueprint class. If you want to create, for example, Blueprint Macro Library, you need to use the Add New button from the Content Browser:
The Add New button and its equivalent mouse command (right-click in the Content Browser), will open a pop-up menu with all the assets that you can create in the engine, as follows:

Most of the asset needed for you projects can be created in the editor. We are now focusing on Blueprint; however, it is worth specifying what we can create from this menu and what needs to be created with an external software:

<table>
<thead>
<tr>
<th>Can be created in Unreal Editor:</th>
<th>Needs to be created using an external software:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game Levels</td>
<td>Static Meshes</td>
</tr>
<tr>
<td>Materials</td>
<td>Skeletal Meshes</td>
</tr>
<tr>
<td>Particle Systems</td>
<td>Skeletal Animations</td>
</tr>
<tr>
<td>Cinematic Sequences</td>
<td>Textures</td>
</tr>
<tr>
<td>Blueprint Scripts</td>
<td>Sounds (WAVs)</td>
</tr>
<tr>
<td>AI Navigation Meshes</td>
<td>IES Light Profiles</td>
</tr>
<tr>
<td>Pre-calculated Light Maps</td>
<td>Nvidia APEX files (APB and APX)</td>
</tr>
<tr>
<td>Level Lights</td>
<td></td>
</tr>
</tbody>
</table>

During studying the examples written in this book, we will see some of them, such as particle systems, navigation meshes, and materials, and some external assets, such as meshes and textures.
About the Static Meshes, it is possible to create them in the editor using the Binary Space Partitioning (BSP) brushes; however, it is a tedious process and worth only when talking about simple shapes such as walls or stairs. A dedicated software such as the freeware Blender or the more famous 3ds Max or Maya can surely do a better job in less time.

Navigate to Add New | Blueprints | Blueprint Class, as follows:

Here, we will choose the Parent Class of our Blueprint script. The editor shows us the Common Classes (we already saw them when previously talking about Blueprint classes); however, the list of parents that we can use is potentially unlimited. If you click on All Classes, in the left-hand side corner at the bottom, you can see a very long list containing all the objects that are available at that moment as a parent for your Blueprint.

Click on Actor and call it BP_Introduction.

It is very important, even for a small project, to name your assets/scripts in a smart manner from the very beginning using a suffix in order to recognize and immediately find the required file even between hundreds of files.
Double-click on the **BP_Introduction** file to open it and we will finally arrive at our Blueprint Editor:

As you can see from the preceding image, the Blueprint Editor is divided into several panels. Each panel is independent; this means that they can be moved, resized, deleted, and duplicated in order to have a workspace that fits your choice.

Let's take a closer look at all of these sections in the following:

**Menu bar**
Menu Bar has the following options:

- **File**: You can manage your Blueprint files from here. You can save and import other assets in the session, and manage source control. There is also a section dedicated to Blueprint, where you can compile, refresh, and compare your Blueprint revision in source control.

- **Edit**: This is a typical edit menu. It can undo, redo, and modify history. You can also search for something in your Blueprint or change the editor settings and preferences.

- **Asset**: Go here to open **Content Browser** or to check the references viewer of any of your assets.

- **View**: View preferences can be set by this menu. Change the pin visibility or set the zoom.
Getting Started with Unreal Blueprints

- **Debug**: Here, you can set the brake points and the watches for your Blueprint. We will go through the Blueprint debugging later in this chapter.

- **Window**: If you accidentally close one of these tabs or you want to open another tab, you can do this going in this menu. All the Blueprint Editor specific tabs are contained. It is also possible to save or load a custom layout here.

- **Help**: You can find useful information about Blueprint here or directly through the epic forum and Wiki.

**Toolbar**

The toolbar is displayed at the left-hand side top of the Blueprint Editor. Its buttons provide easy access to the common commands that are needed when editing Blueprints. This is a dynamic bar, which means that it provides different buttons, depending on which mode is active and which Blueprint type you are currently editing, as follows:

- **Compile**: Every time you modify the script and want to run it, you need to compile. This button changes, depending on the state of your script. It shows if there is an error or a warning and if the script need to be recompiled.

- **Save**: It saves the current Blueprint.

- **Find in CB**: It shows Content Brower and highlights the selected Blueprint.

- **Search**: It finds references to functions, events, variables, or pins in the current script.

- **Class Settings**: It opens the Blueprint properties Details panel. These settings usually belong to the parent class of Blueprint. You can add Blueprint Interfaces to the Blueprint class here.

- **Class Defaults**: It shows the default properties in the detail panel. Here, you can change the default properties of the new instances of this class.

- **Simulation**: It starts the game in simulation mode.

- **Play / Stop / Pause**: It manages the execution of the game in the selected environment such as mobile, standalone, and custom viewport.

- **Possess/Eject**: It switches from Simulate in editor to play in editor mode.

- **Debug Filter**: If you have two or more instances of this class in the game, you can choose which one to debug here.
Viewport

In Viewport, you can view and manipulate your Blueprint’s components:

By default, you have a three-dimensional perspective view of your object. You can manipulate the settings of the Viewport using the buttons on the top-left corner. The first button allows you to switch between Perspective and the orthographic view, the second one sets how you see the object if it is Lit (rendered with light), Unlit (rendered without light) or in simple Wireframe instead.

The right-hand top series of buttons gives you some useful tools in order to manipulate your object:

- **Select and translate / rotate / scale object**: If one of these is selected, the corresponding three axis images appear on the pivot point of the object and you are allowed to move, rotate, or scale the object in one or all its axis.
- **Toggle Coordinate System**: This button toggles the coordinate system between world and the local (object-related) system.
- **Surface snapping**: This button toggles surface snapping, it enables an object to snap in a surface when possible.
- **Snap to the grid**: This button toggles whenever the object snap to the grid or not.
- **Snap size**: This button sets the accuracy of the snapping.
- **Rotation snapping**: This button toggles the snap through a rotation grid.
- **Rotation size**: This button sets the rotation-snap angle.
- **Scale snapping**: This button toggles snapping object through a scale grid.
- **Scale size**: This button sets the scale snap value.
- **Camera Speed**: This button sets the speed of the camera when it is moving in the viewport with values between 1 to 8.
Component panel

In the **Components** panel, you can find all the components of your Blueprint that are shown in a hierarchy form. A component is a piece of functionality that can be added to an Actor. Components cannot exist by themselves; however, when added to an Actor, the Actor will have access to the component and use the functionality provided by it:

![Component panel](image)

In this panel, you can add/remove and manage your components. Each component has its own specific purpose and combining them allows you to create almost anything that you need.

**CapsuleComponent**, for example, provides collision geometry to the Actor. **MovementComponent** controls the movement, **AudioComponent** enables the Actor to emit sound, and so on.

**Components** added in the component list can also be assigned to instance variables, providing them access in the graphs editor.
In order to add a component to Blueprint, you can click on the **Add Component** button and select the component from its menu, as shown in the following image:

Components can also be added by dragging and dropping them from **Content Browser** in the **Components** panel.

Each component is placed at the location of the instance by default. However, they can be transformed, rotated, and scaled if necessary in either the **Details** panel or the Viewport, as we saw earlier.
Detail panel

The **Details** panel contains information, utilities, and functions that are specific to the current selection in the Viewport or the content panel.

It contains all the editable properties of the selected object (such as the **Transform** parameters to move, rotate, and scale it):

![Detail panel](image)

At the very top, you find the search filter. This allows you to quickly find the property that you need (very handy when you have a long list of properties).
The Property Matrix button will open the Property Matrix grid. It is a special tool that allows easy bulk editing and value comparison for a large number of objects or Actors. It displays a configurable set of properties for a collection of objects as columns in a table view that can be sorted on any column. The Property Matrix grid also provides a standard property editor that displays all the properties for the current selection set in the table view.

The display filter icon allows you to filter the properties according to your need.

Some properties have three buttons. They allow you to open the selected property in Content Browser, attach the property from the selected one in Content Browser, or revert the property to default:

My Blueprint panel
The My Blueprint panel shows all the Graphs, Functions, Macros, Variables, and EventDispatchers contained in your Blueprint, including component instance variables that are added in the component list or variables that are created by promoting a value to a variable in the graph editor.

By default, your Blueprint contains one EventGraph and one ConstructionScript for your Functions but you can add any Graph, Variable, or function you might need by the Add New button:
Graph editor

The graph editor panel is the heart of the Blueprint system. It is here that you will create your network of nodes and thanks to their wires, your game lives:

First of all, this table gives you some handy shortcuts for your movements in the graph.

A smart usage of these shortcuts can save a lot of time when developing your projects. Try to memorize this table and always use the shortcuts when possible.

<table>
<thead>
<tr>
<th>Control</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right-click + Drag</td>
<td>Pans the graph</td>
</tr>
<tr>
<td>Mouse Scroll</td>
<td>Zooms the graph</td>
</tr>
<tr>
<td>Right-click</td>
<td>Opens context menu</td>
</tr>
<tr>
<td>Click on node</td>
<td>Selects the node</td>
</tr>
<tr>
<td>Click + Drag in the empty space</td>
<td>Selects the nodes in the marquee select box</td>
</tr>
<tr>
<td>Ctrl + Click + Drag in the empty space</td>
<td>Toggles selection of the nodes in the marquee select box</td>
</tr>
<tr>
<td>Control</td>
<td>Action</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Shift + Click + Drag in the empty space</td>
<td>Adds the nodes in the marquee select box to the current selection</td>
</tr>
<tr>
<td>Click + Drag on node</td>
<td>Moves node</td>
</tr>
<tr>
<td>Click + Drag from pin to pin</td>
<td>Wires the pins together</td>
</tr>
<tr>
<td>Ctrl + Click + Drag from pin to pin</td>
<td>Moves the wires from the origin pin to the destination pin</td>
</tr>
<tr>
<td>Click + Drag from pin to the empty space</td>
<td>Brings up the context menu, showing only relevant nodes. Wires the original pin to a compatible pin on the created node</td>
</tr>
<tr>
<td>Click + Drag + C on the empty space</td>
<td>Adds a comment box containing the selected nodes</td>
</tr>
</tbody>
</table>

To add a new node to the graph, you can use the two methods explained in the table (right-click on the empty space or drag any pin from an existing node) and you can also drag and drop any asset from **Content Browser** to the graph editor. It will automatically add the corresponding node to the graph, as follows:
Getting Started with Unreal Blueprints

You can also drag and drop any Variables from the My Blueprint panel to the graph in order to automatically add its correspondent getter or setter (by selecting the desired node from the pop-up window that appears or by holding control for a getter or Alt for a setter) as shown in the following image:

![Blueprint graph editor](image)

You can find the same behavior seen in the Blueprint graph editor exhibited in the Construction Script Editor and in the Macros Graph Editor.

Let's now check the graph editor in deep: which variables are accepted and what are the nodes and pins that we just introduced.

Types of variables and data

Under Unreal, there are different types of variables: typical data types, such as Boolean, Integer, Float, and so on, and more complex reference types, such as objects, Actors, and custom classes. Each type has a unique color for easy identification, as shown in the following table:

<table>
<thead>
<tr>
<th>Variable type</th>
<th>Color</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Red</td>
<td><img src="image" alt="Is Alive" /> Spawn Pawn</td>
<td>Boolean represents true/false data.</td>
</tr>
<tr>
<td>Variable type</td>
<td>Color</td>
<td>Example</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
<td>--------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Byte</td>
<td>Dark Green</td>
<td><img src="image" alt="Enum Choice" /></td>
<td>Numbers from 0 to 255. This is the smallest data type in terms of spaces; only 1 byte of memory.</td>
</tr>
<tr>
<td>Integer</td>
<td>Cyan</td>
<td><img src="image" alt="Socket" /></td>
<td>Integer values (number without decimals). Ranges from -32,768 to 32,767. Used to store values such as ammo, lives, and collected items.</td>
</tr>
<tr>
<td>Float</td>
<td>Light Green</td>
<td><img src="image" alt="Radius" /></td>
<td>Float values (numbers with decimals). More accurate than integers as it has a precision of seven digits and is used, for example, to store the radius of a sphere, or the damage taken by an enemy, or any value that should contain decimal numbers.</td>
</tr>
<tr>
<td>Name</td>
<td>Violet</td>
<td><img src="image" alt="Game Name" /></td>
<td>Name is the lightweight system for using string. It is case-insensitive and cannot be manipulated. Similar to the byte, it is the smallest data type when talking about text and is used to store keywords and indices.</td>
</tr>
</tbody>
</table>
### Variable type

<table>
<thead>
<tr>
<th>Variable type</th>
<th>Color</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>Magenta</td>
<td><img src="example.png" alt="Game State" /></td>
<td>String is the only string class that allows manipulation. It is more expensive than the other two text data; however, strings can be searched, modified, and compared against other strings.</td>
</tr>
<tr>
<td>Text</td>
<td>Pink</td>
<td><img src="example.png" alt="Description" /></td>
<td>Text represents a display string. It is used to store object descriptions, times, numbers, and any formatted text. It is typically used in a table for the localization system and cannot be manipulated.</td>
</tr>
<tr>
<td>Vector</td>
<td>Yellow</td>
<td><img src="example.png" alt="Position" /></td>
<td>Vector contains an array of three float values and is typically used to store positions on three-dimensional space (XYZ) or color information (RGB).</td>
</tr>
<tr>
<td>Rotator</td>
<td>Purple</td>
<td><img src="example.png" alt="Rotate To" /></td>
<td>This is similar to Vector; it stores an array of three float values that contains the rotation of an object in a three-dimensional space (in the order: Roll, Pitch, and Yaw).</td>
</tr>
<tr>
<td>Transform</td>
<td>Orange</td>
<td><img src="example.png" alt="Transform" /></td>
<td>Transform combines translation, rotation, and scale of a three-dimensional object.</td>
</tr>
</tbody>
</table>
Apart from these default data types, there are tons of other custom data types and we will see how to create our custom ones further in this book. These types can be regrouped in five categories, as follows:

- **Structure**: Struct (value) types. A structure is a container of custom variables. It is used to group related variables in a single entity in order to simplify data combining and data management.

- **References to objects or Actors**: As the name suggests, these data types are references of any object/actor in the game. They are useful when we want to communicate between two different Blueprint classes.

- **References to interfaces**: They are the same as object pointers; however, they are referred to as interfaces objects.

- **References to classes**: Similar to object references, this type of variable contains references to a class. The main difference is that this type points to the default class, while the object reference points to a single instance of this particular class in the game.

- **Enumeration**: An enumerator is basically a byte variable that, instead of numbers, has a human-readable list of names. It can be used to store any kind of object state or type (Game States, tree types, weapon types, player states, and so on).

**Nodes**

A node is an object that can perform a unique function, such as variable holder, event, math calculation, flow control operation, and so on. However, the way in which nodes are created and used is common to all nodes. This helps the user during the process of the graph creation.
A node has a common layout that we can find in any kind of node that we create, as follows:

In the preceding image, we can notice the following: on the top we find his name and a symbol. Name, symbol, and color are self-explanatory and help the user to identify the node's behavior quickly, even if it is the first time that he is using it. In the preceding image, \( f \) means function, typically with a blue background, and the title suggests that this node is a function that will **Add Camera Component** to a target when called.

On the left-hand side of the node, we find the **INPUT** pins, and on the right-hand side, we find the **OUTPUT** pins. We can find nodes with only input (or output) pins; however, their position is unequivocal, as follows:
Pins

There are two main types of pins, execution pins and data pins, as follows:

Execution pins are used to connect nodes together in order to create a flow of execution. A node is executed when its input execution pin is activated by another node. Once execution of the node completes, it activates its output execution pin to continue the flow of execution. Usually, there is only a single input and output execution pin (as functions only have one entry point and one exit point); however, other types of nodes can have multiple input or output execution pins, allowing different behavior depending on which pin is activated. For example, Timeline has multiple input execution pins to call Play, Stop, Reverse, and so on, and multiple output pins in order to call a custom function each time when each time loop is finished.

Data pins are used to put the data in a node or receive data from a node. Data pins are type-specific and can be wired to variables or other data pins of the same type. Unreal helps us to recognize the different types of variable, not only with the name, but also with its color. Their color is unique and common in all the tools of Unreal, not only in Blueprint. As execution pins, data pins are also displayed as an outline when not wired to anything and solid when wired.
Blueprint debugging

When developing your Blueprints, you will soon find that at times something is not working as you expected. To diagnose these problems, Unreal Engine 4 gives you a powerful debugger system that allows you to see your Blueprint script flow in real time, as follows:

When you play or simulate in the editor, you can see the pulsating active wires as your script gets executed in your graph editor.

The debugger system is attached to the first instance of your Blueprint class that the editor finds in your level (alphabetic order) as soon as you play or simulate your game. If you have more than one instance and you want to specify which one to debug, you can select it from the toolbar.

You can set a Breakpoint in a node: when added you can play your game and when the simulation reaches that node, the game will pause and jump to that node in your graph so that you can step through your script to see where the issues are occurring.
To add a Breakpoint to your Blueprint, right-click on any execution node and choose **Add breakpoint**. You can also toggle the Breakpoint of a selected node by pressing **F9**:

![Breakpoint Image]

When a Breakpoint has been added to a node, a red circle will appear in the left-hand top corner of the node. This means that, as soon as the gameplay reaches that node, the game will pause and focus on this node.

Another debugging feature is Watch Values. You can set any variable in your Blueprint to be able to see any variation of it in real time while the game is running. This is an important tool that helps you to find any logical error due to wrong calculations and human mistakes.

To set a value to be watched, right-click on a variable in your graph and select **Watch this value**. A floating text bubble will appear above of the variable, showing the value of this variable being changed while the game is executed, as follows:

![Watch Value Image]
Getting Started with Unreal Blueprints

Blueprint debugger tab
From the **Window** tab, in the menu bar, you can open **Blueprint Debugger**. This panel shows all the watched variables or Breakpoint assigned. You can add multiple Blueprint debugger tabs by holding **Shift** and clicking on **Actor** in your scene:

![Blueprint Debugger Panel]

The lower section of the debugger is called **Execution Trace**, this will become populated as soon as you play or simulate in the editor and show all the executed commands in the order in which they were issued (the top one as the most recent).

Compiler result
The last panel in the debug section is the compiler result. This will show all the compiler errors or warnings that occur when compiled with your script. Each line contains a message about the issue and a direct link to the node that causes the problem.

Visual Studio
You learned how to use the Unreal Engine 4 (UE4) editor and the basics of Blueprints. Now is time to go through the core of the engine. Code! Let’s take a look at Visual Studio and get ready to comprehend lines of code together while we create our Blueprint scripts. In the examples provided in this book, you will often see different approaches to the same simulation. The goal of this guide is to teach you to not only be able to decide when Blueprint is useful, but also be able to write some lines of code.
Creating the project solution

We created our project as a Blueprint empty project, now we need to create our Visual Studio solution for it. Open your project folder through Explorer. You should have a situation similar to the following screenshot:

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Config</td>
<td>19/05/2015 14:47</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>Content</td>
<td>19/05/2015 14:47</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>Intermediate</td>
<td>19/05/2015 15:05</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>Saved</td>
<td>19/05/2015 15:05</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>myProject.uproject</td>
<td>19/05/2015 14:47</td>
<td>Unreal Engine Proj...</td>
<td>1 KB</td>
</tr>
</tbody>
</table>

Unreal Engine provides you a C++ wizard that helps you in this process. Locate .uproject (usually the name of your project.uproject). Right-click on Generate Visual Studio project files. The UnrealBuildTool file should start and you should see your folder slightly changed at the end of the process, as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Binaries</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Config</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Content</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intermediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Saved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myProject.opensdf</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myProject.sdf</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myProject.sln</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myProject.uproject</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myProject.v12.suo</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If this solution doesn't work, you can generate the project solution directly from the editor. Under the menu bar, navigate to File | Generate Visual Studio Project. If it still doesn't work, remember that the engine will generate a project solution as soon as you add a C++ class to the project in case there isn't any Visual Studio project.
Let's have a brief look at these folders, as shown in the following:

- **Binaries**: It contains executable or other files that are created during compiling.
- **Config**: Configuration files are used to set values that control engine and game default behavior.
- **Content**: It holds all the content of the game, including asset packages and maps.
- **Intermediate**: It contains temporary files that are generated during the building of the engine or game.
- **Saved**: It contains autosaves, configuration (same *.ini of Config folder) files, and log files. Here, you can find crash logs, hardware information, and swarm options and data.
- **Source**: It contains all the source files for the game divided in to object class definitions (.h files) and object class implementation (.cpp files).

Now, we can open the project solution by double-clicking the `.sln` file or under **File | Open Visual Studio Project** through the editor:

One of the problem of UE3 was that whenever you wrote or modified your unrealScripts to test and see your modifications in the engine, you were obligated to restart the editor, losing a certain amount of time due to closing and opening it a hundred times during the development.
On UE4, this is not needed anymore. You can compile your script directly within the editor, and each modification you make on both side (Code or Engine) is automatically updated.

**Add a new class from the editor**

To add a new class from the editor, we can navigate to **File | New C++ Class...** from the menu bar. A pop-up window similar to the Blueprint one will appear where the editor will ask you to choose the parent class.

Notice that here you can choose to *not* have a parent for your class, which is different from the Blueprint class, where it *needed* to have a parent class.

When you choose a parent, you need to specify a name for it and its path (keep all your code under the `Source` folder). The C++ wizard will add a header and a C++ file for you and, when finished, will ask you whether you want to immediately edit that file:
Getting Started with Unreal Blueprints

For any other parent class that you choose, except none, the wizard will add the most used functions for you on the new class together with the constructor. By default, you will find your class ready to be implemented with the BeginPlay function and the Tick functions:

Now that you know how to create your classes, you are ready to write your own code. We will see what to write and how to debug from Visual Studio in detail in the next chapters.

Summary

Now, you should be able to create a project in UE4 and its Visual Studio solution. You know exactly what Blueprint is and how to read the Blueprint classes written by someone else or found in the Epic examples. You are able to create your own classes and start experimenting with nodes and wires.

In the next chapter, we are going to use what we studied here to create our first game. We will also see how different Blueprint classes communicate among each other and how to use BPS Brushes to create simple environments.
Where to buy this book

You can buy Unreal Engine Game Development Blueprints from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.