2
Blueprints and Barrels – Your First Game

Welcome to **Blueprints** and **Barrels**! During this chapter, you will be introduced to the **visual scripting** system called Blueprint. This system will be your first entry point toward creating customized functionality and content, using UE4. This chapter will show you the basics of how to create and work with Blueprints to achieve the same level of freedom that any other text based scripting language can provide. You will also expand on the engine navigational skills you learned in the previous chapter as we delve into the more complicated facets of the **Editor** panel and how you can communicate between the different components of the engine. You will learn all this by creating your very first UE game project, **Barrel Hopper**.

This chapter will cover the following points:

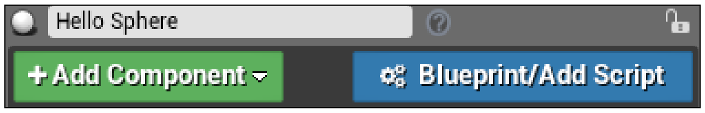
- Creating Blueprints
- Navigating the Blueprint GUI
- How to find Blueprint functions
- Basic coding functionality with Blueprint (logical nodes, objects, and events)
- Game mode Blueprints
- Using UE4's world outliner
- Basic level creation and blocking
- Blueprint collision
- Adding physics dynamics to objects in a world
- Creating a Blueprint character
- Controlling a character through Blueprint
- Debugging Blueprints

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Creating your first Blueprint

Now, it is time to create your first Blueprint. Instead of jumping straight into a new project, I can't think of a better place to create your first Blueprint than our `Hello World` project. It is time to give our `Hello Sphere` an upgrade! The first thing we need to do is create a Blueprint from our actor. To do this, we need to select the `HelloSphere` actor by clicking on the **Actor** in the viewport or selecting the `HelloSphere` name in the **World Outlier**. This will populate our **Details Panel** with the appropriate options. Just next to the **Add Component** button, there is a **Blueprint/Add Script** button, click on this now:



You will be prompted to save the Blueprint we are about to create in some folder within our **Content Browser**. For now, save the blueprint in the Content folder directly. Name this Bluerpint **HelloUnrealBP** and press the green **Create Blueprint** button.

The Blueprint window

You will have been presented with an entirely new window, the Blueprint window. This will be your work area whenever you are editing or creating Blueprints. The Blueprint window boasts a new layout of panels and visual sections:

1. **Components**: Located in the top left-hand side of the window, this panel functions in exactly the same way as the top of the **Details** panel in the editor window. You can use this panel to add and remove components.
2. **My Blueprint**: Located directly under the **Components** panel, this panel is responsible for all of the different technical members of the Blueprint. This is where you can find your **Event Graphs**, **Events**, **Functions**, **Macros**, **Variables**, and **Event Dispatchers**. Each of these elements will be described in detail throughout the chapter.
3. **Details**: This panel functions exactly the same way as is it did in the editor window. This panel exposes all of the publicly exposed properties of your currently selected component within the Blueprint.
4. **Toolbar**: The toolbar is located in center of the screen at the top and features a set of large clickable buttons. These buttons will be explained in the next paragraph.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

5. **Viewport**: This viewport section is responsible for showing the current physical arrangement of the blueprint in 3D space. This section will also be used to navigate the graphs or functions that are created with the Blueprint system. These other panels can be seen as tabs along the top of the viewport window. By default, **Viewport**, Construction Script, and **Event Graph** are shown.

Working with Blueprints

When creating Blueprints, you are frequently going to want to test your recent changes and additional functions as intended. You will also want to make sure that you have minimal errors, warnings, or bugs. The way you will do this is by interfacing with the previously mentioned **Toolbar**. The buttons featured on the **Toolbar** allow you to perform varying actions that assist with having your Blueprint execute in a test environment. Each button performs the follows:

1. **Compile**: This compiles the current Blueprint and will output any errors or warnings. You can specify various parameters for Blueprint compilation that will adjust if the file is saved and how Blueprint compile errors are handled. These parameters can be found by navigating to **Editor Preferences | Content Editors | Blueprint**.
2. **Save**: This saves the Blueprint.
3. **Find in CB**: This opens the content browser and navigate to a selected asset. This lets you locate any assets included in your Blueprint within your content browser. This can be useful if you wish to edit these assets midway through working with a Blueprint.
4. **Class Settings**: This opens a **Details** panel that is responsible for all of the Blueprint Properties such as the Blueprints Parent class, thumbnail appearance, and any *interfaces* the Blueprint is concerned with. These options will be covered when we look at advanced blueprint functionality.
5. **Class Defaults**: This opens a **Details** panel that is responsible for the default values of all of the *publically* exposed variables that can be found in the Blueprint and its components. This lets you specify what state you would like these variable to be in when an instance of this Blueprint is created in a game.
6. **Simulation**: This runs the Blueprint as a simulation. The simulation takes place within the Blueprint viewport. This lets you witness how the Blueprint will function within the game without having to create an instance in the level and running the game.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

7. **Play**: This runs the current project, and functions in the same way as the play button within the editor window. The main difference is the drop-down window located next to the Play button. This lets you select the in-Editor Blueprint instance you wish to scrutinize for debugging. Note that this kind of playing in the editor is known as a **PIE** (**Play In Editor**) session

All of these options provide you the ability to debug and run your Blueprints while working on them in your project.

[Once you are familiar with the engine and you no longer need the tabs describing the title of each panel. You can right-click on the tab and select the **Hide Tab** option, so you can maximize available screen space.]

Blueprint elements

Each Blueprint will include and be composed of certain elements, these elements can be found under the **My Blueprint** panel. Each of these elements will be used by you to create the custom functionality that you require from your Blueprint.

Graphs

Blueprint Graphs are the canvas with which you work to create blueprint functionality. It is within these graphs that you will place and connect your Blueprint nodes. **Event Graphs** house the *Events* that will be fired during a Blueprint's application lifetime. These events can be things like the *Tick* event that will be called every frame or the `ActorBeginOverlap` event that will be called when the Blueprint object is overlapping another. Blueprint functions and Macros will also be constructed using the graph system; however, these graphs will be independent.

Functions

Blueprint functions are a means of wrapping frequently used functionality or functionality that you wish to expose to other objects These functions act in a very similar way to standard C++ functions. You may also specify an encapsulation level when working with Blueprint functions (public, protected, and private). Blueprint functions allow you to specify both input and return parameters.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Macros

Similar to functions but are only internally facing, meaning that they can only be called from within the *Blueprint itself*. Macros are predominantly used for grouping sets of simple instructions that will be called very frequently. The major difference between Blueprint functions and Blueprint macros is that macros are *inlined* when the Blueprint is compiled. As UE4 is a C++ based engine, all blueprints are eventually compiled down into generated C++ that can be then again compiled into executable binary. When I state that the Macros are inlined, I am speaking of traditional C++ inlining to the macro function itself it its generated C++ form.

Variables

Blueprint variables allow you to create data types that can store information. These variables are very similar to **Class Member Variables** in C++. You can specify an encapsulation level and default values for Blueprint variables. It is also important to note that, for every component included in a blueprint, there will be a corresponding variable that will act as a reference to the component. This allows you to access components form within your blueprint graphs.

Event dispatchers

Event dispatchers allow you to bind events to one another. For example, if you wanted some functionality to be performed in Class A when an event in Class B is fired, you can do that using Event dispatchers; or similarly you can set up these delegate relationships within single objects as well. We will cover these in detail at the end of the chapter. This is very similar to **Delegates** in C++ and C#.

Modifying the Hello Sphere Blueprint

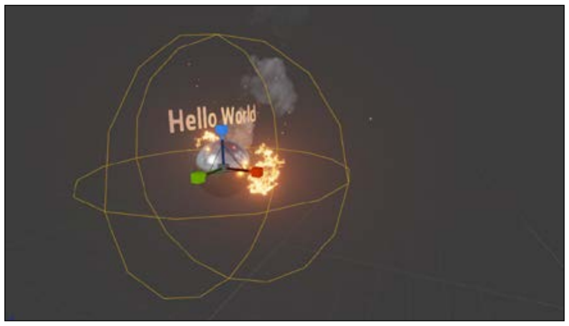
Double-click on the `HelloUnrealBP` to open it. Now, you should be able to see our spherical friend front and center within the **Viewport**. You will also notice that the components we added to our `Hello Sphere` can be found in the **Components** panel and as variables in the **My Blueprint** panel. This is so that we can get references to our components within our **Blueprint Graphs**.

Our goal is to have it, so we can approach our Hello sphere and the 3D text will change from **Hello World** to **Hello Player**. To do this, we are going to need to add another component, work with the Blueprints **Event Graph**, and utilize an **Event**.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

The first thing we need to do is provide a volume we can use to detect other overlapping actors. To do this, we need to add a `SphereCollision` component via the **Add Component** button in the top-left corner of the Blueprint window. Do this now, and name it `SphereCollision`. With the new component selected in the **Components** panel, you should be able to see the collision sphere visualized within the viewport. Scale the sphere volume via the in viewport widget (like we did in Chapter one), so it is approximately 3.5x bigger than our sphere mesh. You should be presented with something that looks similar to this:



The next thing we need to do is ensure that this component will generate **overlap events**. With the sphere collision component selected address, the **Details Panel** on the right-hand side of the Blueprint window. Under the **Collision** section, you will see a series of options, one being a checkbox titled **Generate Overlap Events**, with this checkbox ticked, this Blueprint will fire the `ActorBeginOverlap` event within the event graph when it overlaps another object also signaled to generate either hit or overlap events. It is important to note that both offending bodies must be set to generate these events or nothing will occur.

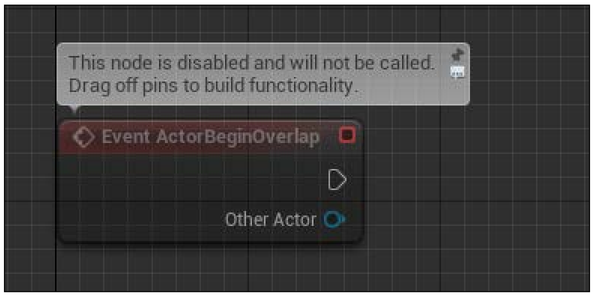
Working with Blueprint graphs

Now, we need to modify the Blueprints **Event Graph**. This will be your first attempt at creating custom functionality with Blueprints. Open the **Event Graph** either by double-clicking on the **Event Graph** element under the **Graphs** section of the **My Blueprint** panel, or selecting the **Event Graph** tab above the **Viewport** panel. You should see a graph with three slightly translucent **Event Nodes** underneath comment bubbles. These are currently disabled events. They have been presented to you by default as they are the most commonly used events; however, until functionality has been appended to these nodes, they will not be triggered.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

You should see a node titled **Event ActorBeginOverlap** as follows:


This node is disabled and will not be called.
Drag off pins to build functionality.
Event ActorBeginOverlap
Other Actor

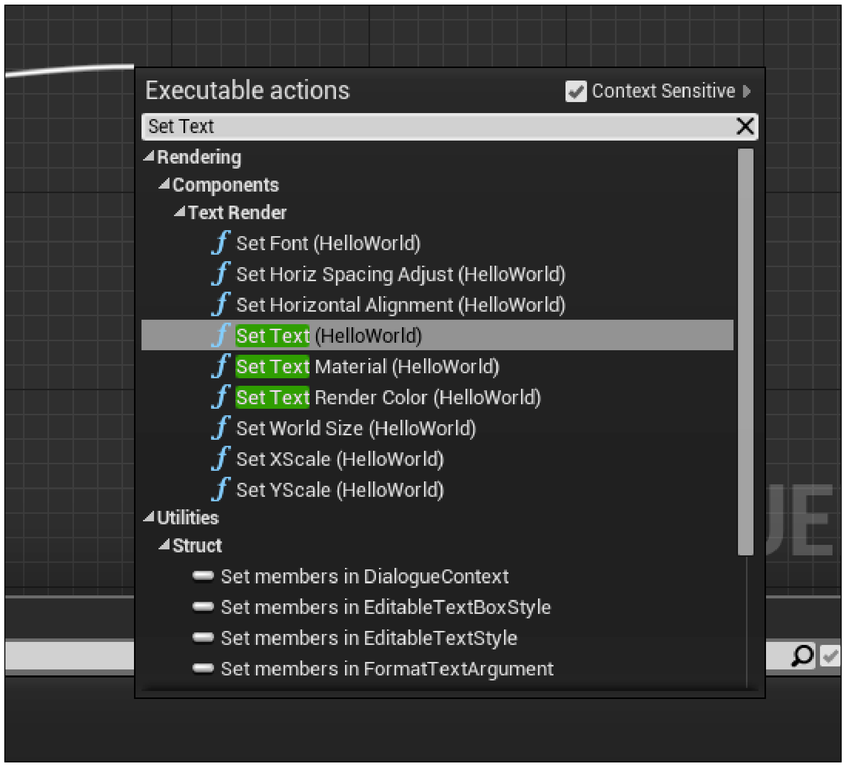
This is the event that will be triggered when other objects overlap our `SphereCollision` component. To enable this node, we need to append functionality to the node. We can do this by left-clicking on the hollow white arrow on the right-hand side of the node and dragging off into the grid, then releasing the left mouse button. Upon releasing the left mouse button you will be presented with a list of available *functions*, *events*, and *logical nodes*. This list is your access point into the Blueprint node library as well as any custom functions, events, or Macros you have created within the Blueprint itself. Functions in blueprint appear in the form of *Nodes*, which are a visual representation of the function within our graph.

This list will be context sensitive by default, meaning that you will only see options that are concerned with the **Critical Execution Path** and are related to the Blueprint. This path is the order in which your nodes will execute and it can be denoted by a white line that runs between nodes connected via white arrows. The question is what function do we need to call to change our text? What we need to do is, set the **text parameter** on our `TextRender` component. As our list is context sensitive and we have a `TextRender` component included in our blueprint, we can assume that we will be able to find the function we desire. Given that, simplistically, we wish to **Set Text** let's try searching for that. By typing or clicking in the search box at the top of the list, we will be able to perform a keyword search for our function.

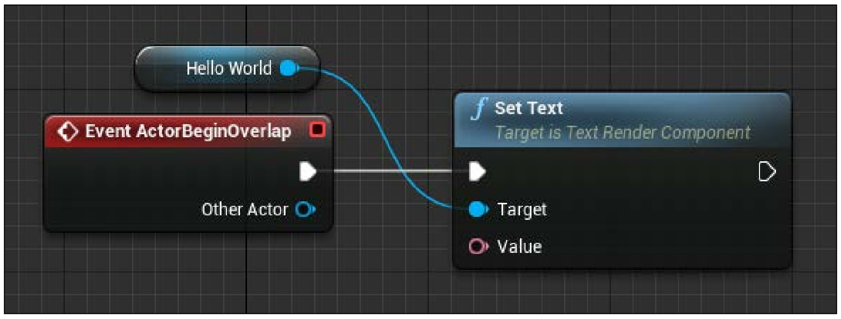
Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

By typing **Set Text**, we are presented with the following options:


Executable actions
Context Sensitive
Set Text
Rendering
Components
Text Render
Set Font (HelloWorld)
Set Horiz Spacing Adjust (HelloWorld)
Set Horizontal Alignment (HelloWorld)
Set Text (HelloWorld)
Set Text Material (HelloWorld)
Set Text Render Color (HelloWorld)
Set World Size (HelloWorld)
Set XScale (HelloWorld)
Set YScale (HelloWorld)
Utilities
Struct
Set members in DialogueContext
Set members in EditableTextBoxStyle
Set members in EditableTextStyle
Set members in FormatTextArgument

As you can see, the context sensitive search has provided us with all of the functions that we can call that either partially include the phrase `Set Text` or are related to the `TextRender` component. You will also notice that all of the function calls also have (`Hello World`) following the function name, this means that if you select that function it will already be set up to affect our `HelloWorld TextRender` component that we included in the Blueprint. Select this option now. You will see something similar to this:



Copyright © 2016. Packt Publishing, Limited. All rights reserved.

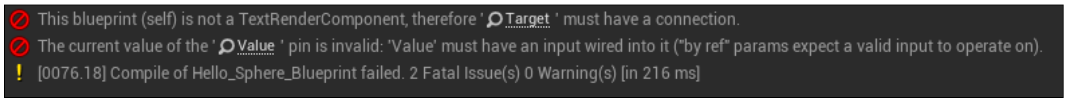
Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Compiling Blueprints

Congratulations! You have created your first Blueprint function! As you can see, two of our nodes are linked via the white line. This means that the `Set Text` function will be hit immediately after the event `ActorBeginOverlap` has been triggered. You will notice that the event has one pin titled **Other Actor** on the right hand side of the node, whereas our `Set Text` function node has two pins titled **target** and **value** on the left-hand side of the node. These are the node's inputs and outputs. Inputs act in exactly the same way as function parameters do in C++ and are found on the left-hand side of the node. This means that those pins are for data that you wish to pass into a function or node, whereas pins on the right are the resultant output of a function or node and are similar to output parameters or a return type in C++.

As you can see, the target pin has already been populated for us. Our **Context Sensitive** node search has provided us with a `Set Text` function that has already set our `TextRender` component to be the target. The small node titled `hello world` is our visual representation of the reference we have to the component. This reference can be found as a variable under the **MyBlueprint** panel. Select this small node and press the *Delete* key. You will notice that the `Target` pin is now empty, and in its place, the field has been populated with the word **self**. This means that when no input is specified for this pin, it will default to use a `self-reference`, that is, a reference to the blueprint you are working from.

On trying and compiling the blueprint by clicking on the **Compile** button in the **Toolbar** panel, you will see that there are two errors preventing successful compilation in the **Compiler Results** panel, which can be found at the bottom of the Blueprint window:



```
This blueprint (self) is not a TextRenderComponent, therefore ' Target ' must have a connection.
The current value of the ' Value ' pin is invalid: 'Value' must have an input wired into it ("by ref" params expect a valid input to operate on).
[0076.18] Compile of Hello_Sphere_Blueprint failed. 2 Fatal Issue(s) 0 Warning(s) [in 216 ms]
```

Each time you compile a blueprint, you will receive a verbose report of any errors or warnings within your blueprint. You may double-click on these report lines to take you to the area of the Blueprint where the error occurs. The first error states that the default parameter of the `Target` input pin `self`, meaning a reference to the `blueprint` we are working from, is not of a compatible type with the function input pin. This makes sense as the Blueprint we are working from is not of type `TextRender` component. The second states that the value of the `Value` pin is invalid. This is because we have not specified any input value for this pin. The final line is a summary of the compilation, including compilation time in milliseconds.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

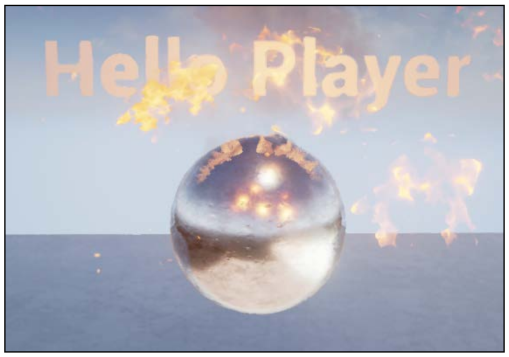
Using Blueprint variables

The first thing we need to do is replace the `HelloWorld` `TextRender` component reference we deleted. To do this, we can click and drag the variable from the **MyBlueprint** panel into the graph, then select **Get** from the drop-down menu. You can then click and drag the pin on the right-hand side of the reference node and connect it with the `Target` input pin on the `Set Text` function node. While you have the `HelloWorld` reference pin dragged, you can see a small box underneath the marker. This will inform you of the action that will take place upon releasing the pin.

> When clicking and dragging variables from the **MyBlueprint** panel, you can hold *Ctrl* to spawn a `get` node or *Alt* to spawn a `set` node.

The next thing we need to do is populate our `Value` pin with the text value that we want to appear when a player moves within our `Sphere Collision Volume`. To do this, we need to summon our function search widget, you can do this by right-clicking on blank space within the graph. We need to search for a `Make Literal` function; these functions are utility functions that allow the creation of temporary variables during Blueprint execution. Specifically, we require a `MakeLiteralText` node. Select this option from the search window and type `Hello Player` in the `Value` box on the left-hand side. Then, connect the **Return Value** pin to the **Value** pin on the `Set Text` Node. Now, Compile! No errors or warnings should present and the compilation will be successful.

Now, we can run our project and see the fruits of our labor! After pressing the Play button in the main editor window we will possess a flying first person avatar, fly close to our hello sphere and you will see the text change!



Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Utilizing the Blueprint palette

We are nearly done with our Hello World Unreal project. There are still a few basic facets of Blueprint usage that have yet to be covered. We are going to extend the functionality of our `HelloSphere` so that it can say hello directly to the **object** that has entered its bounds. To do this, we will need to get the name of the object that has overlapped with the `HelloSphere`, prefix that name with `Hello` and convert the resultant string to a format the `Value` input pin will comply with. We should also change the text back to `hello world` when the object leaves the bounds of our sphere.

We will be finding our desired blueprint functionality in a different way than before: we will use the **Palette Panel**. To bring up the Palette panel, select **Window | Palette** from the drop-down menus located at the top of the screen underneath the tab banner. This will bring up the Palette panel on the right-hand side of the Blueprint window. This panel acts as a search directory for Blueprint library nodes. Nodes found in this panel will not be context sensitive, meaning you will have to populate all input pins yourself. At the top of the **Palette** panel, you can see any nodes that you have specified as a favorites. The bottom half is the search directory titled **Find a Node**. Here, you can specify a class you wish to search for within for the node, and a search bar where you can enter the keywords for your node search.

The first thing we need to do is ensure that our text will reset upon leaving the bounds of our sphere collision volume. To do so, we will need to utilize another event, similar to the `ActorBeginOverlap` there is an `ActorEndOverlap` node. We can find this by searching for it in our **Palette** or by right-clicking on graph. If you used the **Palette** to search for the function to create the node all you need to do is select the desired function from the list and click and drag it into the graph.

With our `ActorEndOverlap` node now in the graph, all we have to do is replicate the previous functionality we created for our `ActorBeginOverlap` event yet instead of using `Hello Player` as our literal value we can use `Hello World!`. You can copy paste blueprint nodes if you wish. All lines and associations will be preserved. I would advise you to keep copy pasting of nodes to a minimum. If you are finding yourself copy pasting functionality quite frequently, it is likely you would be able to wrap the functionality into a blueprint function.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Blueprint meta-data and string manipulation

Now, to create the functionality to print the name of the object that has overlapped our bounds, we are going to need to retrieve two more function nodes and adapt our `ActorBeginOverlap` functionality. Retrieve `GetObjectName` and `Append` now. We will use these two nodes to complete our functionality. As you can see, the `GetObjectName` node takes in an object reference (remember our Unreal Object Hierarchy), which is the base type of all Unreal Objects. This input pin is used to specify which object we wish to get the name from. This kind of information is known as meta-data and is usually present in large polymorphic inheritance chains.

We are going to drag the **Other Actor** pin from our `ActorBeginOverlap` node to this input pin. This will retrieve the name from our overlapping actor as a string. It is important to note that our collision functions `ActorBeginOverlap` and `ActorEndOverlap` all expect the colliding object to be of type **actor**, that is, because `UActor` is the base class from which all objects that can move or be physically present in a scene should inherit from and `UActor` inherits from `UObject` allowing us to make the conversion.

Now that we have our offending actor's name, we need to prefix the resultant string with `Hello` via our `Append` node. By default, our append node will have two input pins, **A** and **B**, in short the resultant string will be whatever value is plugged into **A** plus whatever value is plugged into **B**. For our purposes, we wish to plug our offending actor's name into **B** and leave **A** unconnected, but type `Hello` in the field box provided this will result in `Hello` **B**. Ensure `Hello` is followed by a space.

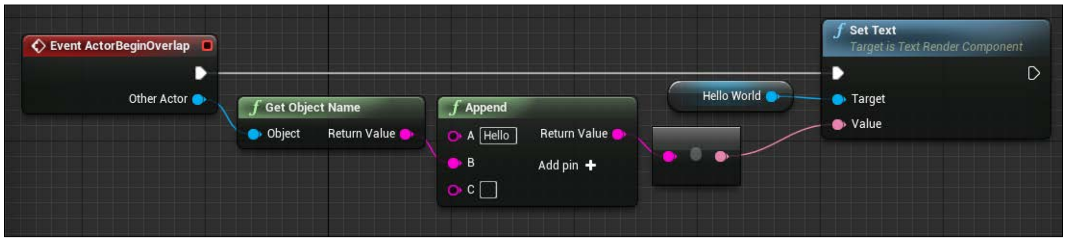
On the right-hand side of our `Append` node, we have an output pin for the resultant string. There is something else on this node; however, there is an **Add Pin +** button. This is because our `Append` node has an `expandable number of input pins`. By clicking this button, we will add a **C** input pin; do this now. We now have an extra input pin on the left-hand side of the node, the resultant string will now be *A+B+C*. Fill the provided text field for **C** with `!` to add some extra enthusiasm.

We are now going to plug the resultant string from this append node into the input pin provided by the `Make Literal Text` node. You will notice that `string` and `text` are two different types. With the `string` output pin, click and drag the mouse over the input pin for the `MakeLiteralText` node. You will see that the little box by the mouse that shows what will happened upon releasing the pin is not a green tick but a box stating **Convert String to Text**.

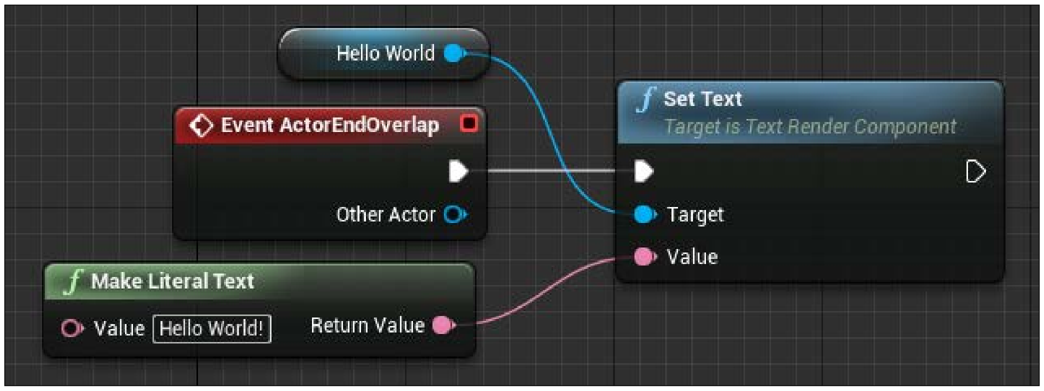
Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

This means that upon releasing the left mouse button the Blueprint will create a **conversion node** that takes in a `string type` and outputs a `text type`. This is very similar to **conversion** in C++. However, conversions between types in Blueprint could be concealing more complicated functionality as some types may require additional steps to convert between. With this step complete, you should see something similar to this:



The node arrangement for the `ActorEndOverlap` event will appear as follows:



Now, run your project and encroach on our spherical friend you will see the text change to something along the lines of **Hello DefaultPawn_1!** Then upon leaving the spheres bounds, it will change back to `Hello World`! Well done, you have completed your first Blueprint! Now that we have created a `hello world` project and a `hello world` blueprint, I think it is time we sank our teeth into your first unreal game project Barrel Hopper! We will be expanding our Blueprint skillset and learning some new editor tricks. We can finally close our `hello world` project!

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

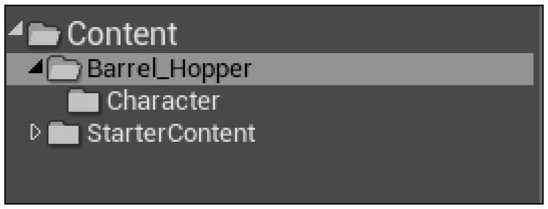
Creating the Barrel Hopper project!

We are going to create our Barrel Hopper project the same way we created our Hello World project. Load the UE4.11 clients and create a new project that is a **Blank Blueprint project** titled **Barrel Hopper**. For this project, we are going to need a few things. We are going to need a **Character** that can move around and jump, a camera that provides us a side on view, barrels that can spawn and will despawn the player when collided with, and a level for all of these elements to exist in.

Creating our Character

We are going to be creating our first UE Character. To construct our character, we will create a new Blueprint that inherits from the UE4 `UCharacter` class. `UCharacters` are designed to be possessed and controlled by various controllers while utilizing UE character movement and physics backend. As `UCharacter` is at the bottom of the `UObject` hierarchy, it is the most developed of the objects, inheriting all of the parent object's public or protected functionality. The relationship between `UCharacters` and the engine will be described in more detail in *Chapter 4*, *Unreal Engine, C++, and You*.

Before we create the character, we need to set up our file hierarchy in the **Content Browser** so that our assets are organized in a logical manner. To create folders simply *right-click* on the folder hierarchy on the left-hand side of the **Content Browser** panel and select **New Folder**. Create a folder now titled **Barrel_Hopper**, and within that folder, create one titled **Character**. Every time we make a new category of object, it is a good idea to add a folder in the content browser that will let you group assets together that are associated with that category of object. You should have a hierarchy that looks similar to this:



Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

To create the character blueprint, select the newly created **character** folder and right-click within the large area in the center of the panel. This opens up a menu of objects that we could create within this folder as new content. Click on the **Blueprint** option now. This will open the blueprint creation wizard; from here, you can select which class you would like to inherit from to create the Blueprint. For our purposes, we wish to create a character so select the **Character** option. This will create a new blueprint within our `Character` folder, name this character `BH_Character`. If is not already, open this blueprint now.

As you can see our `BH_Character` blueprint is void of any functionality and only has a few components. The `Capsule` component will act as the root component and will be used as the **colliding volume** for the character. The `Arrow` component is an in editor tool that shows us which way our character's forward vector points while being placed in a scene. The `Mesh` component is used to visually represent our character; at the moment, we have no mesh specified so nothing is shown.

Finally, the last component present is the `CharacterMovement` component. This component is responsible for all character movement and the variables that dictate how the character will move through the 3D world. You will also note that all of these components are specified as inherited that is because these components exist in the `UCharacter` code created base class that our new `BH_Chracter` blueprint inherits from.

I would strongly suggest looking at the Details panel each time you encounter a new component so that you may better understand the workings of the component and how you can manipulate it for your needs. Do this now for the `CharacterMovement` component; we will be changing a few of these variables later. As you can see, we have the ability to set the walking speed of the character, how much gravity affects the character, and other movement based variables.

Bringing our character to life

Currently, we only have a shell for a character that we need to fill to fully realize our **Barrel Hopper Character**! The first thing we need to do is attain the assets that we require to visualize and later animate our character. To do this, we are going to transfer some of Unreal Engine's provided content to our blank project. While keeping our current project open, create another new blueprint project. This time we want to create a project from the **Side Scroller template**. We are going to be migrating their mannequin assets to our current content folder. To do this, we need to navigate to the folder we wish to migrate from the new `Side Scroller` project via the **Content Browser**.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

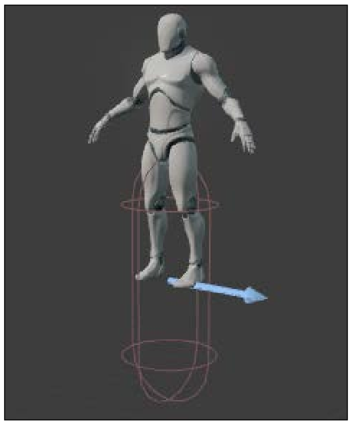
In our case, we need to navigate to **Content | Mannequin**. Right-click on the **Mannequin** folder and choose the **migrate** option. This will prompt you to select the content folder you wish to migrate these assets too. Direct this path to the content folder of our `Barrel Hopper` project. Your content folder should be located at (Project Directory) `\BarrelHopper\Content\`.

If you do not know where your content directory of `Barrel Hopper` is, you can right-click on any of the assets in the **Content** browser and select Show in Explorer. Then, copy and paste the path from the search bar at the top.

Once you have migrated the files, return to your original project to find that the migrated mannequin folder can be found within our content browser. Now, all we need to do is tidy up our content browser by moving around some of the assets. Under **Mannequin | Character**, there should be three folders **Materials**, **Mesh**, and **Textures**. Highlight these folders by shift clicking them, then click and drag the folders into our **Character** folder. You will be prompted to choose, **copy**, or **move** the folders; select the move option. We will be returning to our Mannequin folder to later retrieve the animation assets we require in *Chapter 3*, *Advanced Blueprint, Animation, and Sound*.

Giving our character a mesh

Now that we have the visual assets we require for our character, we can begin to construct our character blueprint. From within the Blueprint window for our `BH_Character`, select the `Mesh` component. In the **Details** panel, you will see a drop-down field for **Skeletal Mesh**. Using this drop-down field, select `SK_Mannequin`. You will see that, when the mesh is brought into our Character blueprint, the mesh sits too high within our capsule and is facing the wrong way.

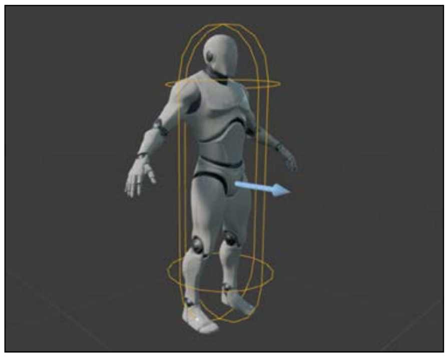


Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

You will also notice that the capsule is just too small for our new mesh. We could scale the mesh so that it fits within the capsule, but that will not always work. Instead, you can change the width and height of the `Capsule` component so that it encompasses the character mesh vertically and is roughly 1.5x wider than the **chest** of the character. In this instance, we only want our colliding volume to encompass the core body of the character, we do not require rough collision on body extremities such as **arms**. To do this, select the `capsule` component and adjust the **Capsule Half Height** and **Capsule radius** variables found under the **Shape** section.

To fix the mesh displacement issue, select the `Mesh` component and use the transform widgets to bring the mesh down so that the heels of the feet are in line with the bottom of the capsule (~ -85cm on the z-axis) and rotate the mesh so that it is facing down the direction of the `Arrow` component. Your finished alignment should look something like this:



[You can adjust the 'step' amount for rotations and translation using the transform widget via the numbered button next to the orange grid in the top right hand of the viewport. Clicking on the orange grid will disable snapping all together, and you will have free range of movement.]

Creating the character's camera

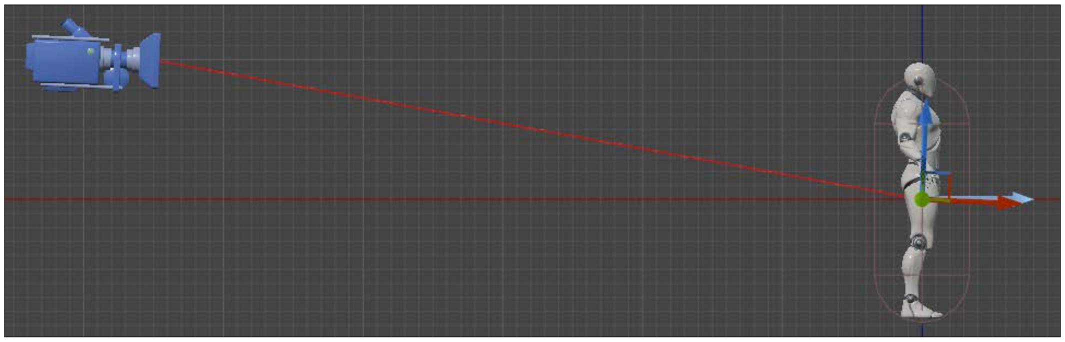
We need to add a camera to our character blueprint, so we can have our viewport update with our character's movements. This requires two new components—a `Camera` component that will act as the main camera for our game scene, and a `SpringArm` component that will be used to position the camera appropriately in 3D space. Add these two components to the Blueprint now. Ensure that they are child components of the `Capsule` component as we wish their positions to update relative to the capsule component. Also ensure that the `Camera` component is a child of the `SpringArm` component, as we will be using the spring arm to position and rotate the camera.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

With the `SpringArm` component selected, check the **Details Panel** for the property titled **Arm Length**, and set this to **550cm** now. As you can see, our camera is pushed back away from our character. We need to ensure our spring arm is positioned so that the camera will maintain a side on view as our character traverses the world. Pitching the camera up slightly as well will provide a better view of our level to the end user.

We can do this by setting variables in the **Transform** section of **Details** panel. Set the Y value (Pitch) of the transform **Rotation** property now to **-2.5 now**. Also, underneath arm length, there is a vector property titled **Socket offset**, set the Z component of this offset to be **75cm**. This will displace the camera upwards along the z-axis. You will see something similar to this:



As you can see, from within the Blueprint Editor Viewport, our camera does not currently take a side on view of the Character. That is ok. In the world we are about to create, we need to ensure that the camera continues to look down the **x-axis** regardless of the character's current rotation. We will then limit the character's movement, so it can only move along the axis that is at right angles with the look direction of the camera. With both of these constraints in place, we will have a suitable side scrolling environment. To ensure that our camera does not rotate with our character, select the `SpringArm` component, within the **Details** panel for this component. Under the **Camera Settings** section, there are three checkboxes: **inherit yaw**, **inherit pitch**, and **inherit roll**. Uncheck all of these checkboxes and save the blueprint. We also need to ensure our character does not inherit its rotation from its owning controller. What controllers are and how to use them will be covered later in this book. For now, select **BH_Character(Self)** from the **Components** panel. Then within the **Details** panel, uncheck the property **Use Controller Rotation Yaw**; it can be found in the **Pawn** section.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Game modes and how to make them

Game Modes in Unreal Engine are a set of instructions that inform the engine which objects to use for things such as the **players default pawn** (default character to use), the **game rules**, and what state objects to use for processes such as **HUD**, **Player states**, and **Game states**. We will not only be creating custom game modes but also custom objects for each of the game mode classes. To create a game mode, simply open the blueprint wizard by right-clicking on the appropriate content browser folder and choose to create **Game Mode Blueprint**.

As we are creating a new category of objects, I would also advise creating a `Game_Mode` folder in our content browser. Name this new game mode `BH_GameMode` and save it within the new `Game_Mode` folder. When you open the game mode blueprint, you will be presented with the standard Blueprint window. You will notice that this object does not have many components, only itself and a default scene root. This is because game modes will not be placed in the scene, they are simple objects that we will set as an engine parameter and do not require a physical form within a 3D scene.

By selecting the `BH_GameMode(Self)` component, you will see the **Details** panel houses and the aforementioned game mode classes. Under the **Classes** section, in the **Details** panel, you can see a field titled **Default Pawn Class**. Click on the drop-down field and select our `BH_Character`. This drop-down list will automatically populate with any objects that can be used as **Default Pawns**. We will eventually be setting most of these class references to our custom objects, but for now, everything else can remain as is. What we have just done is create a game mode that informs the engine that when a controllable player is spawned into a world to use our `BH_Character` object by default.

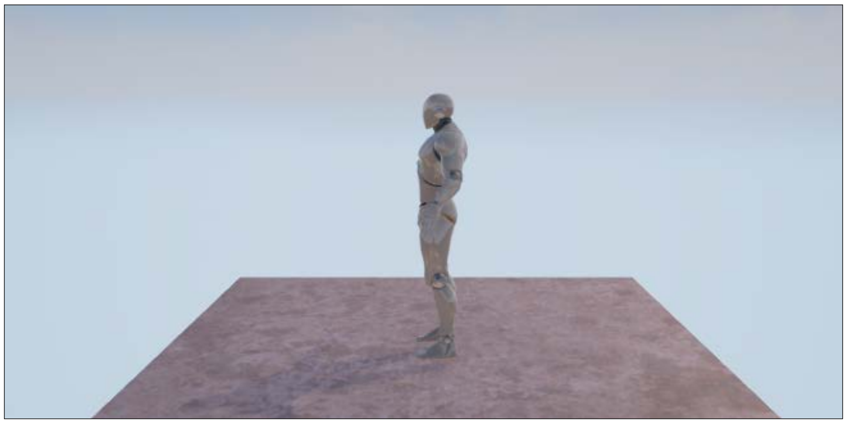
All we need to do now is set the game mode in the engine so that when we press the **Play** button our `BH_Character` is used instead of the free flying first person avatar. To do this, go to the **Edit** drop-down menu at the top left hand side of the **Editor** window; from this window, select **project settings**. This will bring up a **Project Settings** window. This is where you can set a multitude of options on both your current project and the engine itself.

On the left-hand side of the panel is a list of categories, one of which is **Maps & Modes**; choose this now. Here, you can set which Map to use, how Local multiplayer will be laid out, and the **Default Game Mode**. We are only concerned with the latter at this stage. Under the **Default Modes** section, you can see a **Default Game Mode** field, click the drop-down menu, and select our `BH_GameMode`.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Now, navigate back to the main level editor window and select the `Player Start` object within the scene. This will be where our player spawns. Rotate this Player Start object to 90 degrees on the z-axis. This will rotate the Player Start so that the arrow is pointing up the y-axis. Now, press play and you should see our new character standing side on to the camera!



Creating and receiving input events

Now that we can see our character in the game world, it is time to get him moving! We are going to create some input events via our project settings, then receive those events in our `BH_Character` Blueprint. Let's start by setting up some input axis mappings and input action mappings. These mappings simply bind an input, that is, a key press or mouse change, to an axis event or action event that we can receive in our objects. Axis mappings should be used when you wish to receive varying levels from an input, these are most commonly used for analogue inputs such as joysticks and mouse movements. Whereas action mappings should be used when you wish to input from something that has no variable movement, for example, a button or key press.

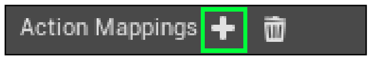
Creating the input Events

Sometimes, you may need to create action or axis mappings that receive input from multiple sources. You may need to create an axis mapping for aiming that can take in input from **both** the mouse and analog sticks. In our case, we will be creating an action mapping for jumping and an axis mapping for moving. Open the **Project Settings Window** again via **Edit** | **Project** settings. Address the list of categories on the left-hand side. This time select **Input**, which is under the **Engine** section. This will show the input settings for our current project.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

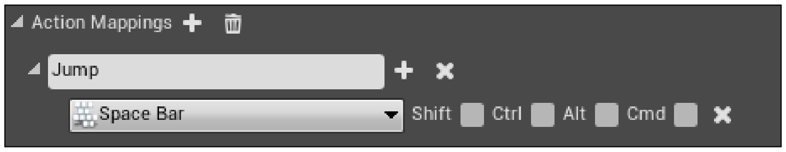
Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

You create mappings by clicking on the white plus (+) next to the desired mapping category. Press the white plus next to the **Action Mappings** field now:



You may need to click on the small arrow next to **Action Mappings** to expand the category. You should see a filed titled `NewActionMapping_0` rename this mapping to **Jump** now. Now, press the small arrow next to this field. We can now map an input to this mapping. This means, when we press the specified input, in our case it will be the *space bar*, an action event tilted `InputAction Jump` will trigger in Blueprints that has included the new mapping in its event graph.

Just underneath the mapping name is a small drop-down followed by four modifiers. Clicking this dropdown will open a search for inputs, in this search type **Space Bar** and select the corresponding input. The modifier checkboxes specify whether or not the modifier key must be pressed for the input to register. You should see something like this:

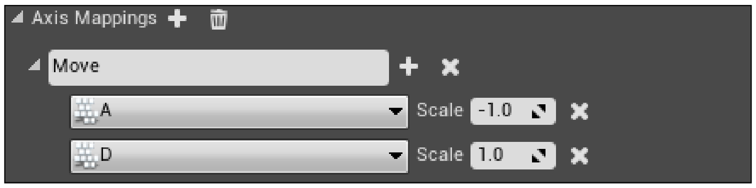


Now, we need to make an axis mapping for movement, just below the **Action Mappings** category is **Axis Mappings**. Again, click on the small plus on the right-hand side of this category; then, expand the field. Name this new mapping to **Move**. By default, only one input field is present. For movement, however, we need two; one for each direction of movement along a given axis, in our case left and right along the y-axis. Fortunately, as our game Barrel Hopper only has a movement on one plane due to its side scrolling nature, we only require input for horizontal movement. Thus, one axis mapping with two inputs. If we were dealing with a game with complete 3D movement, we would require two more mappings for each axis we would like to move along.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Press the small white plus sign, next to the **Move** mapping. This will provide another input field. This time our input fields are followed by a value called **scale**. This scale parameter is the value that will be parsed through the function when the input device is at full input (analog stick fully pressed in one direction). Despite our key presses having no range of variable output, we will still be using key presses for our input, we may at some point wish to provide controller support without having to create a new mapping, and then in turn not needing to edit all of our Blueprints! Set the first input to *A* and set the **scale** to **-1.0**, and set the second input to *D* and the **scale** to **1.0.** You should see something similar to this:



Receiving input events

All we need to do now is receive our newly created events in our character. Luckily, the `CharacterMovement` component largely handles the output from these events. Open the `BH_Character` blueprint and navigate to the event graph, use either the **Pallet** or the right-click on menu to find our newly created axis mapping `InputAxis Move` and our newly created action mapping `InputAction Jump`. You will find the axis mapping under **Input** | **AxisEvents** and the action mapping jump under **Input** | **ActionEvents**. Create both of these nodes now. As you can see our `InputAxis Move` node outputs a **float** axis value (this will exist between 1.0 and -1.0, thanks to our scale values in our `Move` mapping). We need to plug this floating point value into some meaningful function to get our character to move.

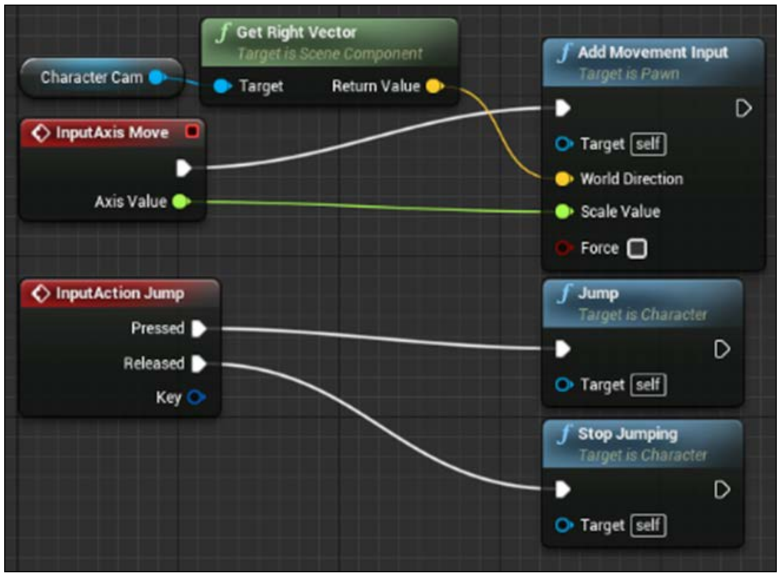
The function we can use is `AddMovementInput`, find this node now. This node takes in a **Target**, which is the `Pawn` to be moved, a **World Direction** to move in, a **Scale Value** to move by, and a **Force** Boolean we can tick to always force movement. Leave the **Target** pin as the default self-reference, connect the **Axis value** pin to the **Scale Value** input pin, and leave **Force** unchecked. For the **World Direction**, we are going to use the right vector of our `Camera` component! To do this, drag the reference to our `Camera` component into the graph, create a `get` node, click and drag from this node's output pin to bring up the context sensitive function search. Search for the `Get Right Vector` node. This will create a function node that outputs the right vector of the `Camera` component. Plug the output vector pin from this `Get Right Vector` node into the **World Direction** input pin of the `AddMovementInput` node.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Now, run the project! When you press *A* and *D* you should move left and right relative to the direction the camera is looking in! You will notice that our character does not currently turn to face the direction it is moving in. To fix this, check the checkbox titled **Orient Rotation to Movement** in the `Character Movement` component's details panel.

Jumping is even easier, return to the `BH_Character` Blueprint. If you have not already created the `InputAction Jump` node, do it now. Action mappings will fire a pressed event when the input is pressed and then will fire a release event when the same input is **released**. Drag from the pressed pin of our `InputAction Jump` node and from the context sensitive search find the function node `Jump`. Do the same from the released pin, but instead find the function node `Stop Jumping`. These functions will inform the Character Movement component when it should initiate a jump action and if it should stop the jump action early; in games, this can be used to create jump actions that jump higher if the input is held down. Your current Blueprint layout should look something like this:



Tweaking the character movement component

You will notice that, if you jump with your character now, there is no air control at all. You may have also noticed that the character turns fairly slowly, these are small but very significant factors that can affect the general feel of your game. Most of the time these slight changes are influenced by a single movement property.

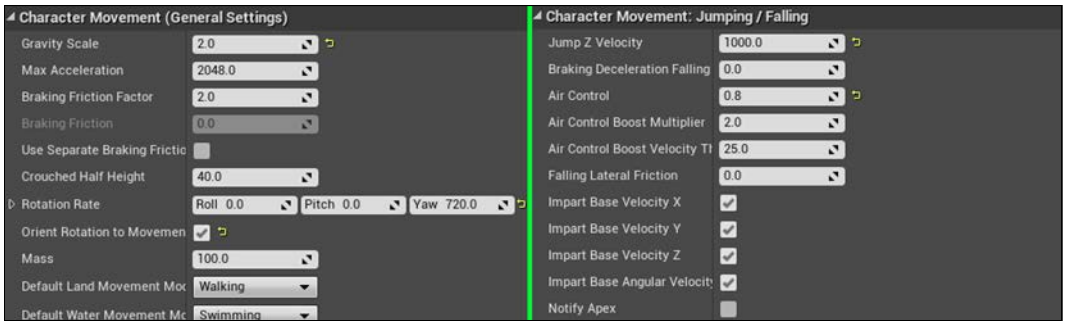
Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Thankfully, our `Character Movement` component has all of our movement properties grouped in a similar place. This is where the **Details** panel for the `Character Movement Component` comes into play. Each section of this **Details** panel is responsible for a different area of 3D movement. The first two we are concerned with are **Character Movement (General Settings)** and **Character Movement: Jumping / Falling**. It is within these sections that you will find the parameters that drive specific parts of character movement such as turn rate, fall speed, air movement, and gravity influence.

To give our character a better movement feel, we are going to change some of these variables. The first one we are going to modify is the **Gravity Scale**. One of the best ways to prevent a character from feeling floaty in a game world is to increase the jump and fall speed of the character. Change the **gravity scale** to **2.0**; it can be found under the **General Settings** section of the `Character Movement component` **Details** panel. This will make our character accelerate downwards faster when falling.

Under the same section you will find **Rotation Rate**; change the yaw of Rotation rate to **720**. This will make our player rotate much faster when moving from left to right. **Orient Rotation to Movement** is already checked, so we can leave that property alone. Under the **Jumping / Falling** section of the **Details Panel** change the **Jump Z velocity** to **1000.0**, the air control parameter to **0.8** and we will be done! You can tweak these parameters later to change how your character moves through 3D space later if you wish. For now each section should look similar to this:



Character Movement (General Settings)

Gravity Scale	2.0
Max Acceleration	2048.0
Braking Friction Factor	2.0
Braking Friction	0.0
Use Separate Braking Frictic	☐
Crouched Half Height	40.0
Rotation Rate	Roll 0.0 Pitch 0.0 Yaw 720.0
Orient Rotation to Movemen	✓
Mass	100.0
Default Land Movement Moc	Walking
Default Water Movement Mc	Swimming

Character Movement: Jumping / Falling

Jump Z Velocity	1000.0
Braking Deceleration Falling	0.0
Air Control	0.8
Air Control Boost Multiplier	2.0
Air Control Boost Velocity Tl	25.0
Falling Lateral Friction	0.0
Impart Base Velocity X	✓
Impart Base Velocity Y	✓
Impart Base Velocity Z	✓
Impart Base Angular Velocit	✓
Notify Apex	☐

Now, run the project again and test how those small changes have modified the feel of moving our character. In the next section of this chapter, we will be creating the game world within which we will be moving around. Feel free to tweak these values once the level has been created to adjust the movement to your liking.

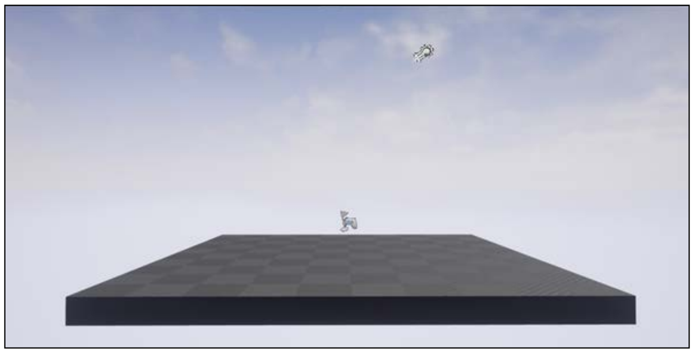
Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

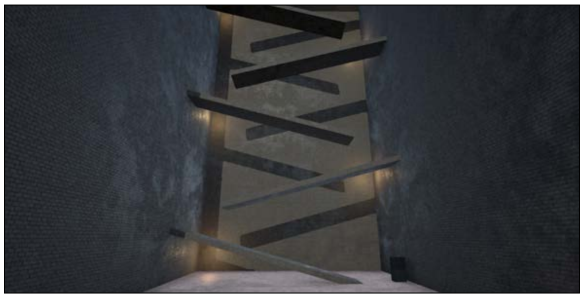
[You can tell when a variable has been changed from its default value when there is a small yellow `undo` arrow on the right-hand side of the variable field]

Building the level!

UE provides a very in-depth and multi-use editor. From this editor, developers are able to create amazing and high-detail levels, the combination of editor tools present in UE4 creates a developer environment where the most limiting factor is the imagination! For the scope of this chapter, we will be using a limited tool set of the editor to act as an introduction to building levels with UE4. Each of the game projects you will create during the course of this book will require a new approach to building levels. This chapter will teach you the fundamentals of level creation and manipulation while teaching you some tricks to make position world geometry a breeze. We are going to take the default level arrangement from this:



To a more realized gameplay level, like this:



For this process, we are going to be creating a new level. To do this navigate to **File | New Level** and select the option default. Save this level as `Barrel_Hopper_Map`.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

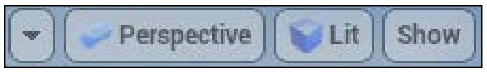
Camera tips and tricks

It's time for us to learn about the various camera and viewport tricks that we can use when working with levels to make creating interesting world environments that is much easier.

Camera settings

There is nothing worse than spending hours placing objects in a 3D scene, only to find that once the camera has moved, everything is out of alignment! It is important to constantly check how the world geometry appears from multiple angles, as you are positioning your assets in a scene. This is particularly important when placing blocking volumes that prevent players from accessing areas of your map you do not wish them too.

In the top left-hand corner of any viewport are a series of view buttons. They look like this:



Clicking on the downward arrow opens up a drop-down menu that has a large selection of settings that you can apply to your viewport camera. Here, you can set things such as **Field of View** and **Far View Plane**, if you would like to have runtime statistics like Frames per Second displayed and many more camera-specific options. Most of these options will be explored over the course of the book.

One of the most important buttons on this bar is the one that currently says **Perspective**. It is this button that allows you to swap between camera perspective types, with the default being 3D perspective view. The other options for this button are the available orthographic views. What this means is that these view modes allow you to change how your world is viewed through the Editor window. If you select the **Front** option, it will change the camera from a perspective camera affected by **FOV** to an **orthographic** camera that looks in the opposite direction of the world's x-axis. This is very important to a level designer, as you are able to view the layout of your game world without a FOV bias from multiple angles.

The button titled **Lit** allows you to specify the rendering mode through which you would like to view your world. Do you want to see the level unaffected by `postprocessing`? Do you want to see your level unaffected by lights? All of these options are here. It is important to use these options to remove **visual clutter** when dealing with levels that are heavily affected by post-processing and visual effects.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

The final option allows you to specify which objects you would like to see in your viewport. If you want to view your entire world without any of your currently placed static meshes or BSP volumes, you can specify this here. It is also through this option that you can specify specific post-processing effects that you wish to view your world with.

Controlling the camera

On top of the basic camera controls there are a couple more advanced methods of camera manipulation. One of the control techniques covered in the tutorial that you saw when you opened Unreal for the first time was focusing. Pressing *F* will move the camera, so the currently selected object is front and center in the screen. There are some object-specific camera controls that expand this concept. You can press and hold *ALT* while pressing the left-mouse button to orbit the focused object; if you do the same thing while pressing the right-mouse button, you are able to dolly in and out from the object. You may also pan the camera up, right, left, and down by holding the middle mouse button.

> For a complete list of in editor controls address `https://docs.unrealengine.com/latest/INT/Engine/UI/LevelEditor/Viewports/ViewportControls/index.html`.

Creating the level

At the moment, our newly created map has a **Player Start**, a **Sky sphere**, a **light source**, some **atmospheric fog** and a **floor**. We are going to change this level to match something similar to the preceding image. Through this process, we are going to learn some level construction tips, what **BSP geometry is**, and how we can use them, as well as how we can apply and modify **materials**. In the game **Barrel Hopper**, we will be getting the player to scale sloping ramps to reach the top of a level while trying to jump over barrels that are rolling down the ramps. We need to construct a level that accommodates these gameplay requirements.

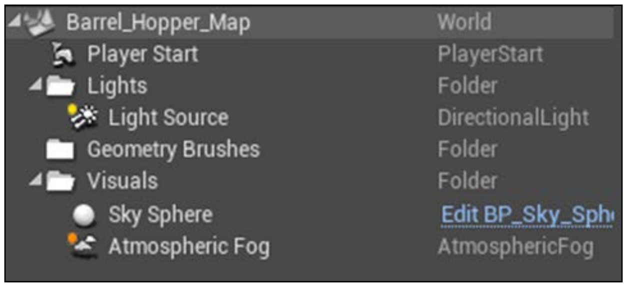
Blocking geometry

The first thing we are going to do is roughly place our level geometry that we will later be replacing with static mesh actors; this is called **blocking**, and we are going to do this using **Geometry Brushes**. This process allows us to estimate the level geometry without having to create custom mesh assets in a third party tool such as Maya or 3DsMax. Fortunately, the Geometry Brush that we are going to use to **block** our level can be converted to static mesh actors, so we will be able to create our entire level using UE 4.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

To ensure our World Outlier remains tidy and organized before we begin create a folder structure, we can group similar objects in our 3D world. Do this by clicking the small + folder button in the top right-hand corner of the **World Outliner** panel. Create three folders now, **Lights**, **Geometry Brushes**, and **Visuals**. Match the folder structure show here:



Barrel_Hopper_Map	World
Player Start	PlayerStart
Lights	Folder
Light Source	DirectionalLight
Geometry Brushes	Folder
Visuals	Folder
Sky Sphere	Edit BP_Sky_Sph
Atmospheric Fog	AtmosphericFog

Whenever we create a new **Geometry Brush** ensure that it is categorized under the similarly named folder in the **World Outlier**. Now what we need to do is select the static mesh actor titled **Floor** and delete it. We are going to be creating all of our level geometry from scratch.

Geometry Brushes and how to use them

We are going to use **Geometry Brushes** to block our scene. These brushes are used by UE 4 to add geometry to a scene that has prebuilt behavior like collision, tessellation, and material tiling. They can be used by developers to effectively block a 3D scene then be converted to static mesh assets for later optimization of the 3D scene. The reasons this is beneficial to developers is the huge amount of time saving this process can afford the level designers. It also eliminates a major development hold up the level designer can continue to create without having to wait for assets from the 3D modelers.

From the **Modes** panel on the left-hand side of the Editor window, ensure that the place mode is active and navigate to the **BSP** section. Here, you can see a list of basic primitive types such as **Box**, **Cone**, **Cylinder**, and so on. From this list, click and drag the **Box** option into the 3D scene. Congratulations, you just created your first **Geometry Brush**! As you can see, it is currently called **Box Brush** in the world outlier. Change this now to **Floor Brush**.

Currently with the brush selected, you should see a Details Panel that contains a **Transform** section, a **Brush Settings** section, and an **Actor** section. If you see something different, but you can see that the brush object is obviously highlighted, it means that you have selected a single face on the brush. I will come back to this later. If this is the case, deselect the brush and then select the brush again by clicking on it once.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

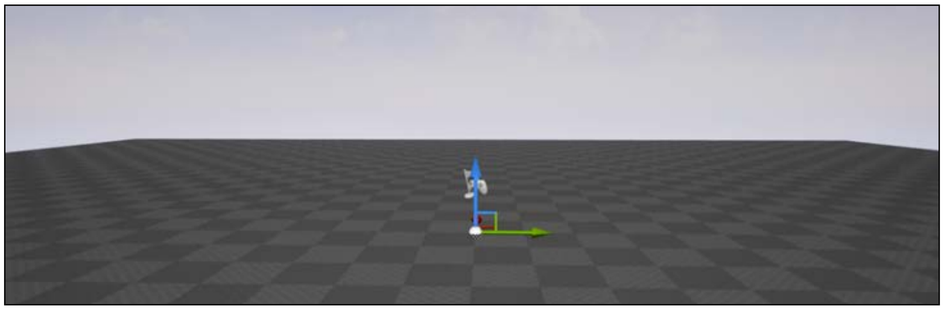
The section that we are most concerned with is the **Brush Settings** section. It is here that we can change the **proportions** of the brush.

It is important to use the **Brush Settings** section to modify the proportions of the brush as opposed to using the scale of the object through the transform properties. The reason being that when we change proportions using **Brush Settings**, the way a material will appear on the surface of the brush will be preserved. If we were to use the scaling tools, we would have to adjust the UV's of the brush. A UV is simply how we choose a specific pixel from a texture when it is mapped to a surface. When a surface is scaled, the UV's are stretched resulting in the texture stretching as well.

Placing the geometry

For now, we can ignore the other settings. The first thing we need to do is ensure our floor brush is centered in our world at the 3D co-ordinates of 0, 0, 0; we will call this **origin**. We then need to proportion our brush, so we can use it as the floor in our 3D world.

We can do this by changing the **Brush Settings**. Change the **X** value of the brush to **5000cm**, the **Y** value to **5000cm**, and the **Z** value to **25cm**. Next, we need to position the brush. Change **Transform Settings** so that all numbers along the **Location** row are **0**. This will shift our floor to **origin**. Now, because our brush is now 25 units in height, we should adjust the location of the brush so the top side of the brush is flush with the zero plane. To do this, simply change the **Z** value of the location row in the **Transform Settings** to **-12.5cm**. You should now be presented with something similar to this:

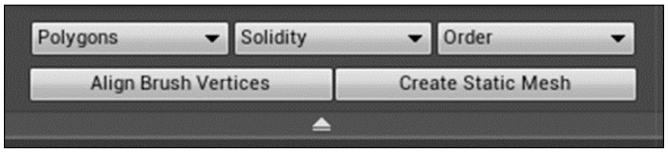


Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Converting a geometry Brush to a static mesh

The next thing we should do is place a **Brush** in our scene that we can use to represent the size and scale of our character, so we have a persistent visual reference for when we construct our level. To do this drag another **Box Brush** into the scene. This time proportion the brush using the Brush setting to the following—**X 100cm**, **Y 100cm**, and **Z 200cm**. We are using these dimensions as that is roughly the bounds within which our character occupies. Now, we are going to make this box a static mesh actor and place it in the scene. This will create a mesh object that the character will be able to pass through and our camera will not collide with. To do this, select the brush and under the **Brush Settings** section click on the white drop-down arrow at the bottom of the section to expand the option list. You will be presented with these options:



Press the **Create Static Mesh** option now and save the resultant mesh under the `Barrel_Hopper` folder. You will notice that our Brush has been replaced with a `static mesh`! We will use this mesh as an *approximation volume* when building the rest of the level. Now is a good time to update our folder hierarchy in our Content browser, as it currently contains a `Mesh` and a `Level` with no containing folder. Add two new folders to our content browser: **Levels** and `Static_Meshes`. Place the new mesh under the `Static_Meshes` folder and place our new `Barrel_Hopper_Level` in the **Levels** folder. Create a similar `Static_Meshes` folder in the **world outlier** and place the new mesh under this folder. From this point forward, I will leave organizing your **content browser** and **world outlier** up to you for the sake of brevity.

Now, we need to create the walls and the ramps the barrels are going to roll down. Let's start by creating the walls. Drag a new **Box Brush** into the world. This time proportion this Brush using the brush settings with **5000cm** on the **X**, **25cm** on the **Y**, and **5000cm** on the **Z**. Then, you can place the brush using the location settings at: **X 0cm, Y -1100cm**, and **Z 2500cm**.

Applying materials to geometry brushes

We can then duplicate this brush and use it for our other wall; but before we do that, we want to apply a material to our brush. Now, we need to select a face of the brush, do this by holding *Ctrl* and *Shift* then clicking on a face of the brush. This will present you with a new section list in the Details panel that has properties specific to single face of the brush.

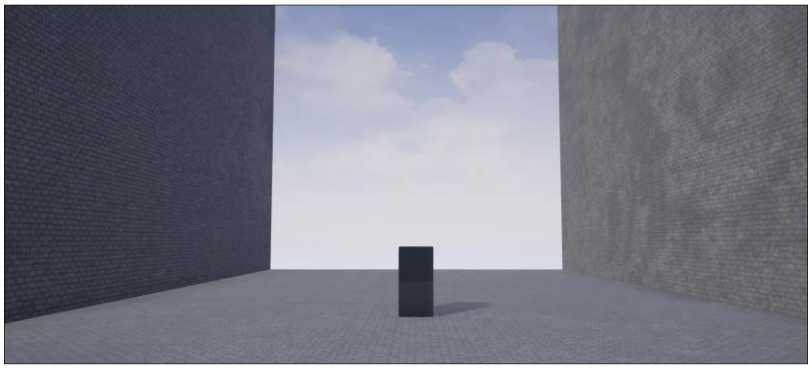
Copyright © 2016. Packt Publishing, Limited. All rights reserved.

We need to ensure that we select all faces of the brush before applying the material, so it is applied to all faces. Do this by clicking on the select dropdown under the **Geometry** section and then by clicking **Select all adjacent faces**. This will select all faces of the currently selected brush. With all of the faces selected, click the dropdown under the **Surface Materials** section and find and select **M_Brick_Clay_Beveled**. The entire brush should now appear to be made of beveled clay bricks!

To **duplicate** our wall brush we can do one of two things, we can copy paste the element via our **world outlier** or we can hold *Alt* while transforming the brush using the in-viewport transform widget. Do this now by holding *Alt* and translating the wall brush along the y-axis using the green arrow of the translation transform widget. You will notice that the original brush remains in place and you have instead dragged a duplicate of the brush off the original. Position this brush at **X 0cm, Y 1100cm**, and **Z 2500cm** using the **Transform** settings of the new brush now. While you are at it, you should also apply the `M_Brick_Clay_Beveled` material to the floor brush now as well!

> If you ever want to have an object you are trying to move snap to an underlying surface, press the *End* key.

With both of our walls in place, the **light** from our **light source** is being obstructed and our world is beginning to look quite dark. To remedy this, we need to place a **Skylight** in the scene. A Skylight is an object that captures the color of your world outside a distance threshold from the Skylight's position and applies that as a uniform light within the scene. As we have a skybox with a light blue ambient color, the Skylight will provide a light to the scene that matches this color. Click on and drag a **Skylight** into the scene from the **Modes Panel** in the **Place Tab** under the **Lights** section. You should now see something that looks like this:



Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

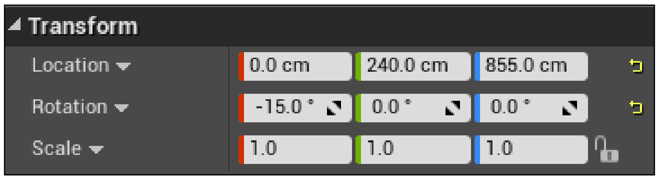
Level building and trigonometry!

We are starting to form the basic bounds of our level! You may have notice that we are starting to cut off large chunks of our **brush geometry** from the player; we will deal with this later. For now, we need to construct our **ramps**! This will require yet another **Box Brush**. Drag a new **Box Brush** into the scene. We want this brush to be roughly three-fourth of the length between our two walls, so there is a 550 cm space between the edge of the ramp and the wall. This will ensure that our barrels can fall through the gap, and our player can jump up to the next ramp. We also want our ramp to be sloped so that our barrels will roll down towards the climbing player. Let's use 15 degrees.

We can use some basic **trigonometry** to figure the length of our ramp. If the ramp is sloped at an angle of 15 degrees and our adjacent side length is *1650 (3/4 of 2,200)*. Then, the length of our ramp is roughly 1708 as *1650/cos(15)* = *1708*. We can increase this to 1800 so that we have some leeway for our player, and so the ramps are easier to place in our level.

With our new length of ramp, we can angle and place this ramp. Using the **transform settings**, place our first ramp at: **X 0, Y -240,** and **Z 200**. Make sure that the rotation of the ramp is set to **15** in the **Roll** value (the first column). From here, all we need to do is **duplicate** the ramp, **raise** it by an amount, and **inverse** the Y position and the Roll angel. The separation for each ramp that I used is 655. I based this number off of the height if the character block out mesh we made earlier, which is 200 units and accounted for the total height of the ramp being 442 (also calculated using basic trig, this time 1650 * Tan(15)). This number was then rounded to 655.

Before duplicating the ramp brushes, ensure that you apply `M_Brick_Clay_Beveled` to all of the brush faces. Now, select the first ramp brush, duplicate it, and then transform the brush by 655 units on the z-axis. Now, you need to flip the rotation of the ramp and position it correctly along the y-axis. As our ramps will alternate symmetrically, you can simply inverse the Y value of the location transform and the Roll value of the Rotation Transform. Your second Ramps Transform matrix should look like this:



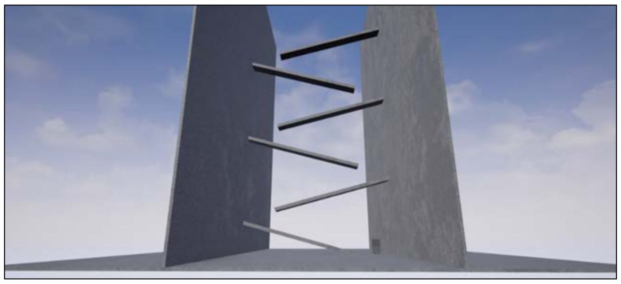
Transform

Location	0.0 cm	240.0 cm	855.0 cm
Rotation	-15.0 °	0.0 °	0.0 °
Scale	1.0	1.0	1.0

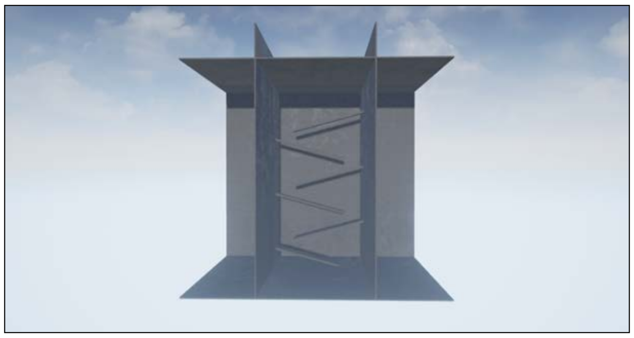
Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

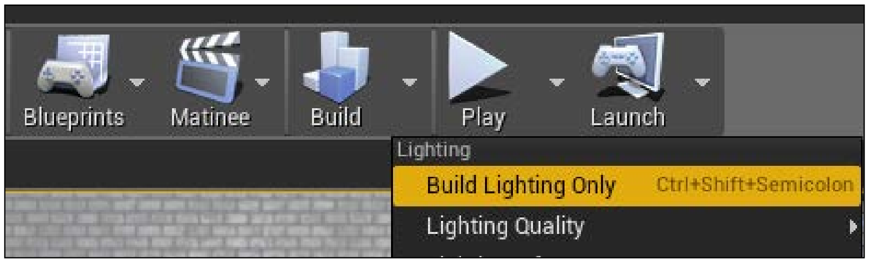
Repeat this process until you have six ramps and can see something similar to this:



We are nearly done with blocking our level! All we need to do now is close off our play area with a roof and a back wall! This is very easily done as we can use a duplicate of our floor brush as the roof, and we can use a duplicate of one of the wall brushes as the back wall. Set the floor brush duplicates position to—**X 0, Y 0,** and **Z 4360**. Yaw the wall brush duplicate by 90 and set its position at—**X 950, Y 0,** and **Z 2500**. You should now have something that looks very similar to this:



We are done on blocking your level! If you aren't happy with the way your level is lit, try adjusting the **Light source** directional light to your liking. If you find that all of the shadows in your level have the word preview repeated through them; it means you need to rebuild your lighting. To do so, click on the drop-down arrow in the **Build** icon under the **Toolbar** panel and select **Build Lighting Only**:



Copyright © 2016. Packt Publishing, Limited. All rights reserved.

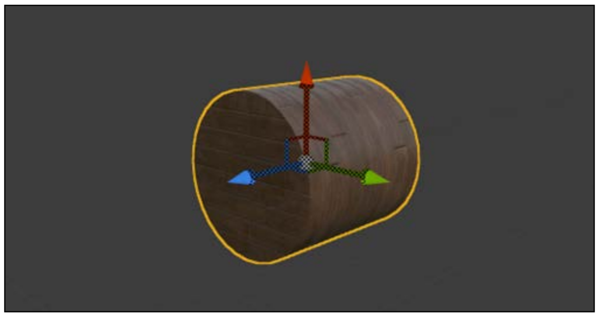
Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Now that we have blocked our level, we can add the other assets to our scene so we can start to form the gameplay of Barrel Hopper!

Getting our barrels rolling

With our level **blocked** out we can begin to place our **Gameplay-centric** actors into the level and add the appropriate functionality to them. In this case, we will be creating the **barrels**. We are going to need to create a barrel object that rolls with physics, destroys the player when the two collide and destroys itself when it reaches the bottom of the level. We are also going to have to create an object that handles spawning these barrels at certain intervals.

The first thing we need to do is create a new blueprint that inherits from Actor called **BH_Barrel**. It is going to be a physics-driven actor that will be one of the key elements in the game world, so create this new blueprint now . `BH_Barrel` will start with the same default components we are used to seeing when working with a new actor. We need to add a visual component to our `BH_Barrel`. Add a **Cylinder shape** via the **Add Component** button now. The **Cylinder** will be used as our `Barrel Mesh`. Let's make an attempt to have our grey scale cylinder look a bit more like a barrel by applying the `M_Wood_Floor_Walnut_Worn` material to the mesh. You can do this by selecting the `Cylinder` component and modifying **Element 0** under the **materials** section in the **Details Panel**. Also, rotate the `Cylinder` component around the y-axis by 90 degrees so lays on its side. You should be presented with something similar to this:



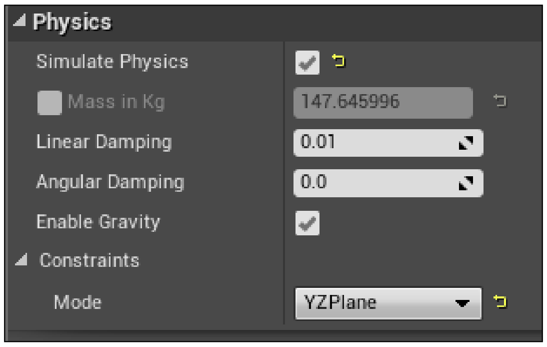
Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Applying physics to objects

Adding **physics dynamics** to an object in UE 4 is actually quite simple given a basic 3D shape. In the case of our Barrel, we already have the perfect shape, a cylinder! What we are going to do is enable physics simulations on our `Cylinder` component. We can do this by selecting the component and navigating to the **Physics** section of the **Details** panel. At the top of the section is a checkbox titled **Simulate Physics**, tick this now. Bam! You have applied physics dynamics to this object! The UE physics layer will now handle all interactions between this object and other physical objects in a 3D scene given the right collision settings.

There are a few more steps we need to take before we are ready to spawn our `BH_Barrels` in the game world. We also need to **constrain** our barrel so that is does not move in an undesirable way. Because we are creating a **side scroller**, we only want freedom of movement on a certain plane. In our case, it is the **YZ plane.** The axis along which the character can move (x-axis and z-axis) are also the only axis we wish our physics bodies to have freedom of movement along. To constrain our body in this way, expand the **Constraints** field under the **Physics** section of the **Details** panel and select **YZPlane** from the drop-down list titled mode. The Physics section of the Cylinder's Details panel should appear as follows:



Physics

Simulate Physics	✓
Mass in Kg	147.645996
Linear Damping	0.01
Angular Damping	0.0
Enable Gravity	✓
Constraints	
Mode	YZPlane

Barrel spawners and Blueprint timers

Now, we need to create the object that will **spawn** the `BH_Barrels`, so they can roll down the ramps and act as hazards to the player trying to reach the top of the level. This will be a simple object that only consists of the default components. However, we will be utilizing a new **event** in the graph of this blueprint. We will be using **Event Tick** to set up a timer so that this object spawns a new barrel whenever a certain threshold of time is reached.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Creating Blueprint variables

Create a new Blueprint that inherits from `Actor` now, and call it **BH_Barrel_ Spawner**. We are not going to add any new components to this object, so you can immediately navigate to the **Event Graph**. To set up our timer, we are going to create two new **variables** of type `float`. To create these variables, click on the small white plus next to **Variables** in the **My Blueprint** panel. By default, the Blueprint will create new variables of type `boolean`. Click on the white plus twice now and call the first variable `barrelTimer` and the second `timeSinceLastBarrel`. We need to change the type of these variables to be `float`. We can do this by selecting the variables within the **My Blueprint** panel under the **Variables** section and addressing the **Details** panel. The **Variable Type** property can be changed via the provided dropdown. Change the type of both of these variables to `Float` now, which is denoted by a green bar symbol in the dropdown.

From this details panel, we are also able to modify the various properties of the variable. You will note that we are able to set the **Default Value** of this variable. The value we provide here will be the value that is used upon object creation. This means we can specify a time in seconds that we wish to use as the rate at which barrels will spawn. For now, set the default value to **3.0**. Create another float variable now and name it `timeSinceLastBarrel` and set the **Default Value** to `0`.

Event tick

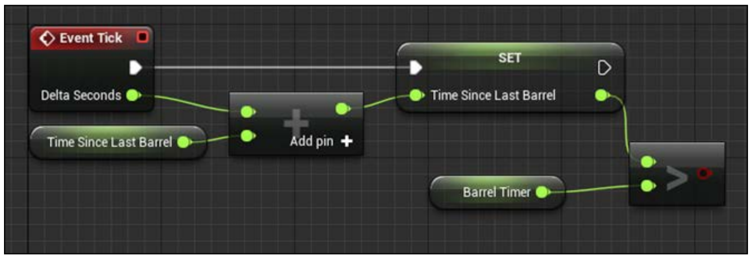
Now, we are going to use those two variables in combination with **Event Tick** to create our timer. The `Event Tick` node is provided by default in the event graph; however, it should currently be translucent as there is no functionality being extended from the node. The `Event tick` node outputs a `float` variable called **Delta Seconds**. This variable represents the time in seconds since the last frame, meaning that every time this event is hit it provides you with the change in time in seconds since the **last** time the event was hit. We can accumulate the value of this variable over time to gauge how much time in seconds has passed in total. What we are going to do is accumulate delta seconds into the variable `timeSinceLastBarrel`; then, when the value saved within is larger than our `barrelTimer` value, we will spawn a `BH_Barrel`.

The first thing you need to do is add **Delta Seconds** to `timeSinceLastBarrel` and then save the result back into `timeSinceLastBarrel`. You can do this by creating a `get` reference to `timeSinceLastBarrel`, then dragging a line off of the output pin from the reference, and then search for **float + float**. This will give you a **+** node with two green inputs and one green output. By default, one of the input pins is populated by `timeSinceLastBarrel`. Plug **Delta Seconds** into the other input pin. From the output pin, click on and drag off another line, and this time, search for **Set Time Since Last Barrel** and parse in the result of the + node.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Now, we need to check if our accumulated value is larger than our timer value. To do this, drag from the output pin of the Set node we just created and summon the **float > float** node. This node features two float input values, the top input will be compared against the bottom, that is, **is top > bottom**. This node will output a Boolean variable that will be the result of this condition. What we need to do is plug this value into a branch node. Before we go into branch nodes, lets quickly check your Blueprint is coming along in the right direction. At the moment, you should see something similar to this:



Branch nodes

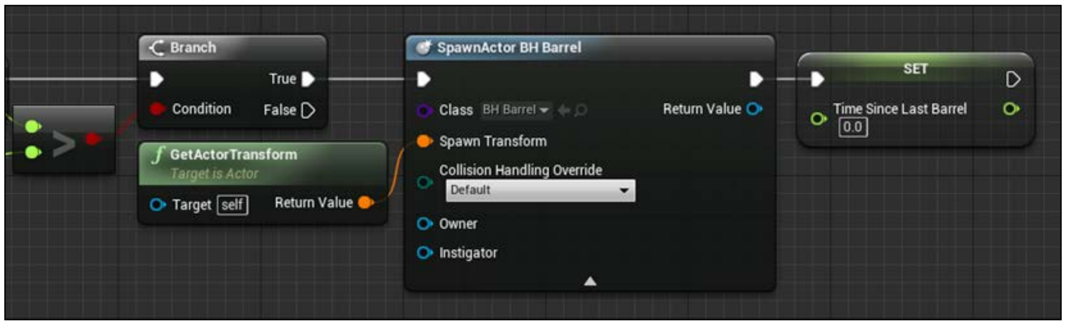
Branch nodes act in a very similar way to `if` statements in C++. Drag off from the output pin of the > node and search for **Branch**. This will present you with a new type of node. This is a **logical node** and can affect the **critical execution path**. As you can see, it takes in a Boolean input as well as the execution path but outputs two execution path options. One for when the input pin has a value of `true`, and another when the input value is `false`. From the true execution path option, drag out a line and search for Spawn Actor From Class. This node will spawn our `BH_Barrel`. The node itself takes in a few inputs they represent the following:

- **Class**: The type of object you wish to spawn. Set this to `BH_Barrel` now.
- **Spawn Transform**: The transform we wish the object to have upon spawning. Summon a node called `GetActorTransform` and plug the orange output pin into this input.
- **Collision Handling Override**: This is an `enum` that represents how the object will handle collision if it is spawned inside another object. Leave this as default.
- **Instigator**: A reference to a pawn that is responsible for damage caused by the spawned actor. Leave this one blank.

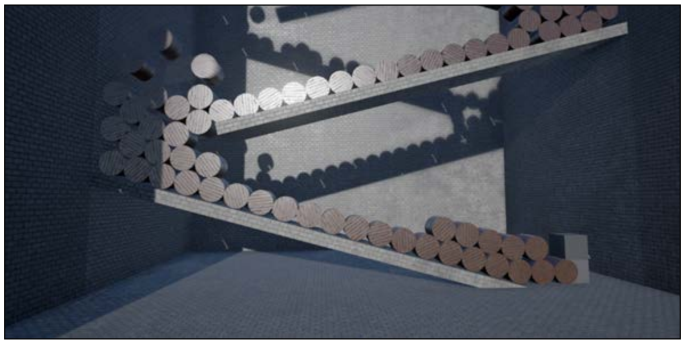
Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Finally, drag another execution line from this node and then set our `timeSinceLastBarrel` variable back to zero via a set node. Your second half of the graph layout should look similar to this:



Now, let's drag `BH_Barrel_Spawner` into our level and see if they work! Return to the Editor window and place one spawner about 450 units above where each ramp meets a wall. With all of the spawners in place, press the **Play** button in the **Toolbar** panel then immediately press the Eject option that appears. This will let you fly around your level mid-simulation without being constrained to the **Default Player Pawn**. You should be able to see a legion of barrels being spawned every 3.0 seconds! You will also notice a very immediate problem. There is nothing that can get rid of our barrels, so they just keep building up!



Trigger volumes and destroying Actors

What we need to do is place some trigger volumes in the scene that will destroy any barrel that enters its bounds. A trigger volume is an area in 3D space that will fire an event if any actor overlaps the 3D volume. We will be creating volumes that are cuboid in shape. From the **Place** tab of the **Modes** panel, navigate to the **Basic** section. From here, you will see an object called **Box Trigger**, click on and drag a box trigger into the scene now. Call this new trigger **BH_BarrelKiller**.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

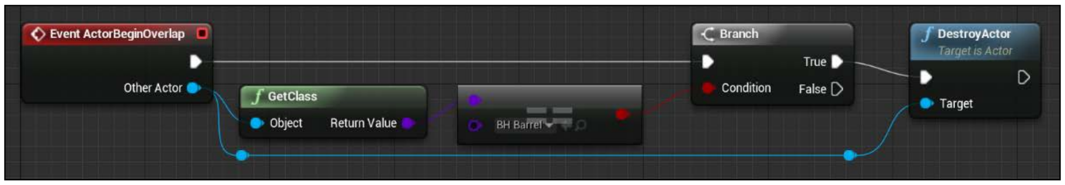
Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Then, click on the **Blueprint/Add script** button. This will create a new Blueprint titled `BH_BarrelKiller_Blueprint`. Again, we are not going to be adding any components to this new blueprint. We are only going to be extending functionality from the Event `ActorBeginOverlap` node.

What we need to do is check if the actor that has just overlapped our new trigger volume is of type `BH_Barrel`. If it has, we need to destroy the actor! Doing this is easy. Drag a line from the **Other Actor** output pin featured on the `ActorBeginOverlap` node and search for a function titled `GetClass`. This node has an output pin that is the class type of the object that is parsed into the node. We can use this class type output pin to determine if the object is of type `BH_Barrel`.

Drag a line from this class output pin and search for *Equal (Class) or type = for short*. This will spawn a class comparison node that will check if the top class input is equal to the bottom class input. By default, the top input pin will be populated with the class pin we just dragged from; the bottom pin we will not connect. Simply select `BH_Barrel` from the provided drop-down list. The comparison node will have a Boolean output pin that will represent the result of the comparison; if the comparison was resolved to be true, we want to destroy the overlapping actor.

We will do what we did in our timer and create a branch node from this Boolean output. From the true execution path, search for a node called `DestroyActor`. This node takes in an actor to destroy. For this input pin, we will drag the Other Actor reference from the `Event ActorBeginOverlap` node. Instead of dragging this pin directly, we are going to keep our graph tidy by utilizing something called a **reroute node**. It is simply something we can use to steer the paths of our graph lines. After dragging from the output pin of the `ActorBeginOverlap` node type reroute and select the **Add Reroute node...** option. Use these reroute nodes to create a clean path from the Event node to the final Destroy Actor function node. I create a path that looked like this:



You will notice that you can have multiple lines from an output pin. This is because this reference can be accessed by multiple things along the execution path.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

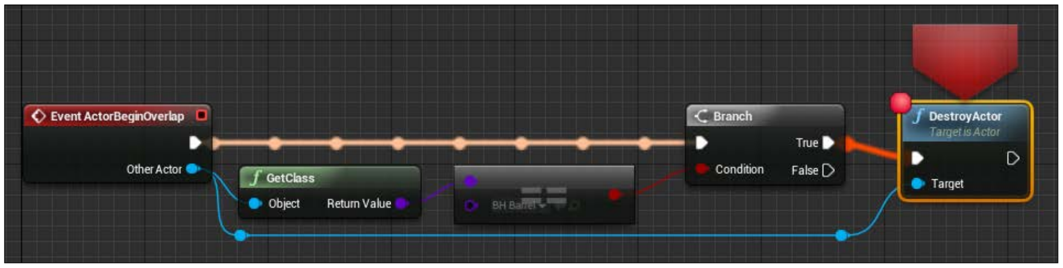
Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

[If you want to remove all lines from a pin in one go simply hold *Alt* then click on the pin you would like to remove lines from. Similarly, if you want to shift multiple lines to a different reference, hold *Ctrl* and then click and drag from the pin.]

Debugging our Blueprints

Much like C++ code, UE allows us to debug our Blueprints. This is a very useful tool that Unreal has provided for us. Let's take our `BH_BarrelKiller_Blueprint` as an example. We have just created a node cluster that performs functionality based on a logical check. If we were developing in C++ code and our desired functionality was not taking place, we would place breakpoints throughout the areas of code we wish to investigate, so we can closer inspect our game state at runtime. We can perform the same debugging functionality with Blueprints. We have scripted some blueprint functionality that should only destroy actors when they are of type `BH_Barrel`. So, let's place a breakpoint on our `Destroy Actor` function node by right-clicking on the node and selecting **Add breakpoint**. Nodes with breakpoints can be denoted by a small red circle in the top left-hand corner of the node.

Now place an instance of our `BH_BarrelKiller` at the base of our first ramp and run the project! Eventually our breakpoint will be hit and you will be presented with something that looks like this:

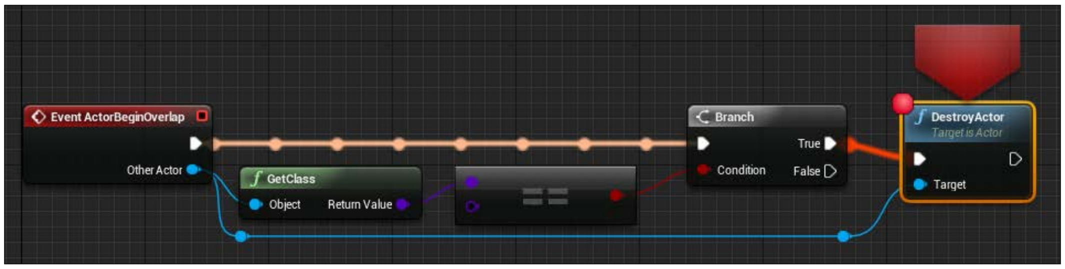


The giant red arrow denotes the node on which the breakpoint was hit. You will also notice that the execution path has been highlighted and features a series of nodules translating toward the node. This is used to show you which execution path is currently being taken! This is very important when working with blueprints as the number of execution paths that stem from one event can become quite numerous; this highlight shows you exactly what order your node set was executed in.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Another useful tool that is provided to us is the preservation of runtime debug values. In other words, the variable states and temporary variable states are preserved when a breakpoint is hit, and you can inspect each value to see if the correct values have been stored. Given our current breakpoint, we are interested in two variables. The first being the type that was returned from the `Get Class` function, and second being the Boolean result of our comparison node. If you hover your mouse over the purple class input pin on our comparison node, you will see runtime debug information about that value. As shown here:



As you can see, the value is currently set to `BH_Barrel.BH_Barrel_C`, which means the class type is of our `BH_Barrel` Blueprint, which is exactly what we are looking for. You can also see the result of our comparison node by hovering over the input pin on our branch node. You will see that it has been set to `true`, which is why our true execution path from the branch node has been hit.

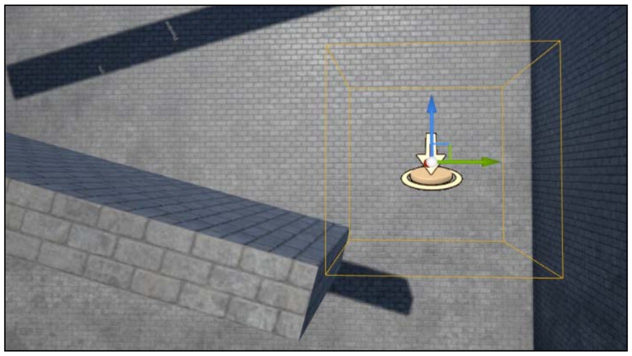
The final debugging tool that will be covered this chapter is the **Debug Filter**, which can be found in the **Toolbar** panel at the top of the Blueprint window next to the **Play** button. This filter allows you to isolate one Blueprint object you wish to test for debugging. Perhaps, only one of your Blueprint instances is not working. If you know the name of the object in the world outlier, you can specify the name here and you will only receive debugging output relevant to the specified object, this includes breakpoints!

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Masking our destruction with particles

With our new trigger volume created, place one `BH_BarrelKiller_Blueprint` at the end of each ramp so that the volume is floating in mid-air but is touching the edge of the ramp as seen here:



Now, run the project again and watch as our barrels destroy themselves when they reach the end of each ramp! Right now, it is quite jarring to have the barrels simply disappear when they are removed. We can fix this with a simple particle effect played upon the destruction of the barrel. Open the **BH_Barrel** blueprint again. We are going utilize a new event node called `Event Destroyed`. Navigate to the event graph and search for this new event node. The event will be triggered every time an actor of this type is destroyed. Thankfully, the `DestroyActor` function we are using in our **BarrelKiller** object will trigger this event!

Drag a line from the execution arrow and search for a function node called **Spawn Emitter at location**. This node takes in an emitter template to spawn, a location to spawn the emitter at, a rotation to spawn the emitter with, and a Boolean titled **auto destroy**. The last parameter means the emitter will destroy itself upon completing the particle simulation. Set **Emitter Template** to **P_explosion** via the provided drop-down menu.

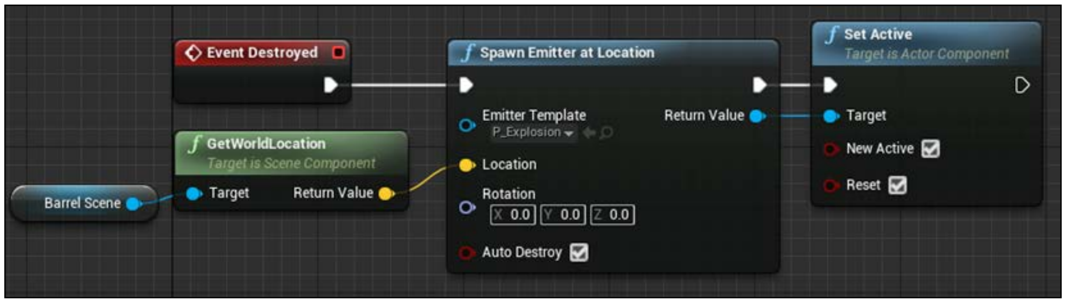
Setting the location is going to be tricky as our Mesh does not have a world location. If we were to use the location of our `DefaultSceenRoot` component, it would play the particle effect at the Barrels original spawn location. What we can do is attach a `scene` component as a child of our cylinder mesh! Remember the object hierarchy we covered in *Chapter 1*, *Introduction to Unreal Engine 4*? If the new scene component is a child of the Cylinder mesh, it will translate with the Mesh as it rolls down the ramp! Create a new scene component via the **Add component** button and call it **Barrel_Scene** now. Then drag this new component onto the Cylinder mesh component in the **components** panel to child it to the `Cylinder` component.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Now returning to our event graph, the value we are going to plug into the **Location** input pin of the `Spawn Emitter` node is the **world location** of our new `Barrel Scene` component. Drag a get reference to this new component from the **variables** section of the **My Blueprint** panel. Drag a line from the reference output pin and search for the `GetWorldLocation` function node. This will output a world vector you can use as valid input for the **Location** input pin. We can leave the rotation values as the default provided, but ensure that auto destroy is checked; otherwise, our spawned emitters will never destroy themselves, and we will have a similar issue to the one we were having with our never ending barrels!

After the Spawn Emitter at Location node, drag the output pin of this node **Return Value**, which is a reference to the newly spawned emitter and search for a function titled **Set Active**. This will ensue that the emitter is set to activate upon spawn if it is not already specified by the emitter by default. Here, there is two Boolean input pins titled **New Active** and **Reset**. Check both of these. You should be presented with something similar to this:



Run the project again, and you will see explosions where there was once nothing! It is quite enjoyable to watch the barrel detonate at the end of the ramps. The next thing we need to do to complete basic gameplay required for Barrel Hopper is kill the player!

Respawning the player

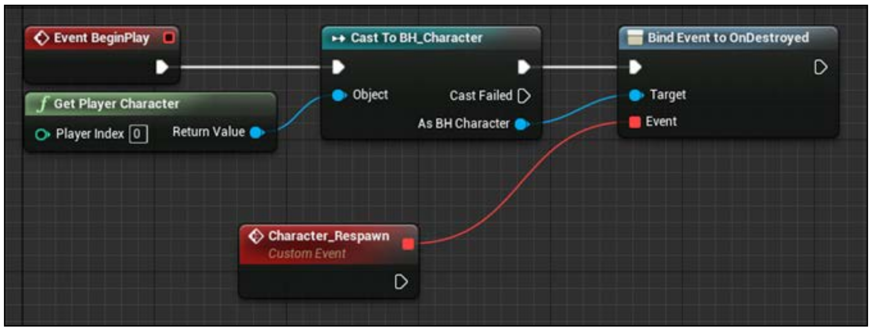
One of the key elements of `BarrelHopper` is punishing the player for unsuccessfully jumping over a barrel. What we need to do is reset the player to the default starting position every time it is hit a by a barrel. The goal is for the player to reach the top of the level in the shortest amount of time possible. In this chapter, we are going to create a rudimentary respawn system that utilizes Event **Delegate**. What we want to do is have our game mode respawn our player after a certain amount of time when the `OnDestroyed` event is fired from within our `BH_Character`. We are going to have to detect collisions with our barrels, destroy our actor, and then ensure that the desired respawning functionality takes place. We will do this via a delegate.

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

To begin, open the `BH_Gamemode` blueprint we created earlier in the chapter and navigate to the event graph. We need to graph some functionality that will get the Player Character reference, cast the Player Character to the `BH_Character` type, and then **bind** the `OnDestroyed` event of the `BH_Character` to a custom event in the game mode. Next, right-click on the event graph and type **Add Custom Event**. This will create a new event node in the graph, rename this node to **Character_Respawn** now. You will notice that on the top right-hand of the event node is a little red square. We will come back to this later.

Next, drag an execution line from the Event `BeginPlay` node that is present by default in the event graph; then, search for the function **Cast To BH_Character**. This function will attempt to cast from any provided object to our `BH_Character`. The node has one input pin for the object you wish to cast from, and one output pin for the resultant `BH_Character` reference. You will notice that the node has two execution path pins: One for a successful cast and another for a failed cast. Before we go any further, we need to provide this cast with a target. Right-click on the graph and search for the node **GetPlayerCharacter**. This node takes in a player ID (0 by default) and outputs a reference to a character object. Plug this output reference into the cast node. This cast will be successful as our `BH_Character` inherits from character.

Drag a line from the output reference of our cast node and search for the **Bind Event to OnDestroyed** node. This will provide you with a node that has an execution path input pin, a reference to the target input pin, and a small red square input pin called **Event**. Remember the similar looking small red square on our custom event `Character_Respawn` we made earlier? We are going to join these pins together, thus, binding them. Now, whenever `OnDestroyed` is hit in our `BH_Character`, `Character_Respawn` will also be hit in our game mode. You graph should look similar to something like this:



Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

This kind of binding is known as a delegate. `Character_Respawn` is now a delegate of the `OnDestroyed` event. Simply put, delegates are triggered for execution when the owning function or event is triggered. We are able to bind multiple delegates to a single event if we wish. This means we can have different objects execute functionality based off of one event.

Delay nodes

Now, we need to create some functionality for this newly bound event to execute. Let's start by creating a **delay** node. It is used to stall execution along an execution path for a given period of time. To create one, simply right-click on the event graph and search for **Delay**. Connect the execution pin from our `Character_Respawn` event to that of the `Delay` node. Then under the **duration** value input **1.5** as we wish for the player have a respawn timer of **1.5** seconds. The next thing we need to do is spawn a new character. To do this create a **SpawnActor From Class** node; this time specify the class type to be `BH_Character`.

Now, we have encountered our first problem, how are we going to specify which transform to use for our player start? Easy, we can set it to the initial position of our player when it is created for the first time. For now, create a variable in the `BH_GameMode` blueprint titled `spawnTransform` that is of type `transform`. Then, get a reference to this variable in your graph, and plug it into the **Spawn Transform** input pin of our spawn actor node.

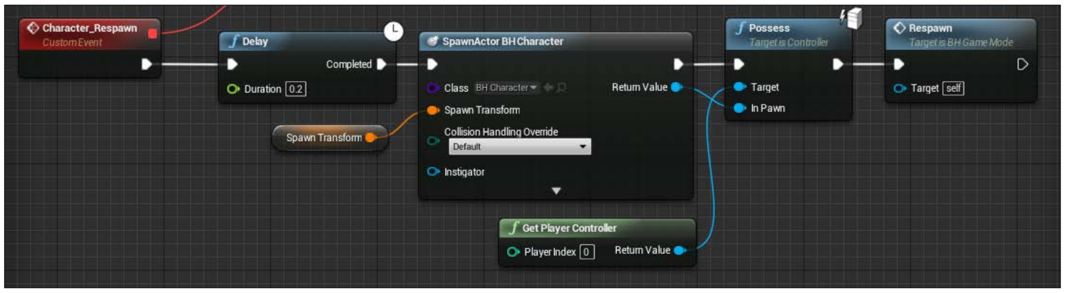
Player controllers

The next thing we need to do is ensure that our player can control the new character we just created. This introduces you to a new concept **Player Controllers**. The only reason we can control our `BH_Character` is that by default at game start up, our character is assigned our **Player Controller**, meaning that any input we provide to the game through the keyboard or mouse is sent to our `BH_Character`. In a single player game such as Barrel Hopper, this is easy as there is only one Player Controller. In local-multiplayer games, this can become a much bigger challenge as there is a Player Controller for each player connected to the game.

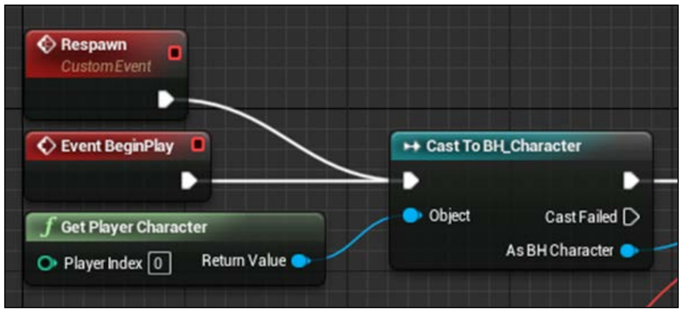
Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

For now, all we need to do is find a node called **Get Player Controller** that takes in a player index and returns the associated controller. We can leave the input index at 0 as there is only one player. Drag a line from the output pin and search for a function node called Posses. This node will take in a **target**, which is the player controller and **In Pawn**. This will then direct any inputs from the target controller to the provided pawn. Plug the output reference pin from our **Spawn Actor BH_Character** node and plug it into this in pawn pin. Your complete functionality should look like this:



The last thing we need to do in our BH_Gamemode is ensure that this newly created character has its `OnDestroyed` event bound to the `Character_Respawn` event in the game mode as the Begin Play event we used previously will not be called again. This is easy to solve. Simply create another custom event within our game mode, and call it respawn. Have the event node for our respawn have its execution path feed into the functionality that trails our `Begin Play` event. Next, call the new respawn event at the end of the `Character_Respawn` functionality chain. This ensures that our new character's `OnDestroyed` event is bound to the `Character_Respawn` event. You can see the call to the respawn event in the preceding image. The graph around our being play event now looks like this as well:

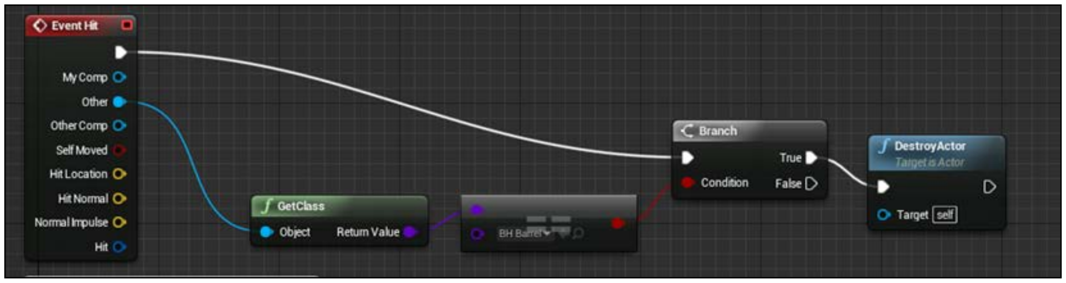


Copyright © 2016. Packt Publishing, Limited. All rights reserved.

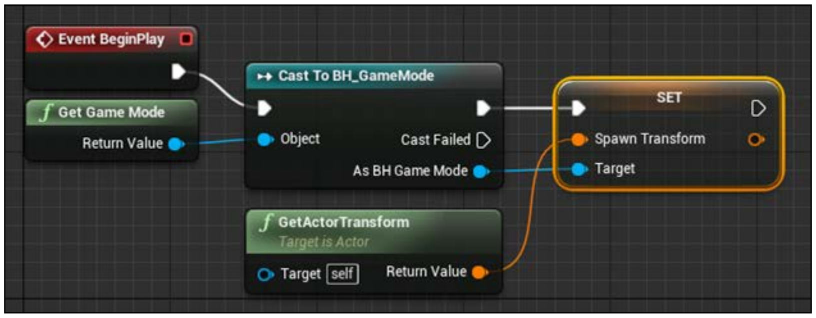
Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.

Hit Events

Now, we have to detect for barrel collisions from within our player. To do this, all we need to do is utilize an event called `Event Hit`. This event functions in a very similar way to **OnActorBeginOverlap**. The only difference is this event is called when a physical collision takes place. It is important that we use this event when dealing with collisions between `BH_Character` and `BH_Barrel`, as both objects contain rigid physics bodies that will not allow for penetration; thus, no overlapping. The process that will follow Event hit is nearly identical. We will be taking the other output pin from Event hit which is a reference to the offending actor and checking to see if it is of type barrel. If this is true, we can destroy our character! The required functionality can be seen here:



All we have left to do is set the value of that `spawnTransform` variable when our character is created for the first time. To do this, all we have to do is get the game mode in our `BH_Character`, cast it to `BH_Gamemode`, and then set the `spawnTransform` variable to the `transform` of the `BH_Character`. Do all of this when event `beginPlay` is hit, and you can ensure that the transform specified is that of the first spawn location. You can see the required functionality here:



Now, run the project and enjoy the base game of what is `Barrel Hopper`!!

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Summary

Great job! You made it this far! You are now familiar with the basics of Blueprint and level creation. You created a blueprint character from scratch that you then brought to life through input events and the character movement components. You learned about the Blueprint window and a large portion of the intricacies and details of that new work environment. You summoned prebuilt nodes via the context sensitive search and blueprint pallet. You created your very own custom functionality using blueprint and how to debug that functionality! You have learned how to utilize brushes to block out level approximations so that you can get onto developing the game. You created a legion of rolling barrels that can bring doom to the players of `Barrel Hopper`. You have actually done quite a lot so far and there is much more to come!

In the next chapter, you will learn all about the animation system provided by Unreal and how you can utilize these systems through Blueprint and provided Editor tools. You will learn how to load and play sounds in the 3D. You will also learn how to bind animation events and triggers to sounds and other actions within the blueprint! We will also be providing that final layer of polish to `Barrel Hopper` to turn it from a basic game loop to a polished mini-game! I look forward to showing how to do all of this and more in the next chapter!

Copyright © 2016. Packt Publishing, Limited. All rights reserved.

Carnall, Benjamin. Unreal Engine 4.X By Example, Packt Publishing, Limited, 2016. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=4620807.
Created from hud on 2019-12-18 16:44:55.