Learning C++ by Creating Games with UE4

Unreal Engine 4 is used to create top notch, exciting games by AAA studios, and learning to program in C++ needs some serious motivation.

Learning C++ by Creating Games With UE4 will start with the basics of C++: installing a code editor so you can begin to write C++ code. You will then learn how to write small, self-contained C++ programs that show you how to use the C++ language, without overwhelming you with too much code at the beginning. As we dig into more advanced C++ concepts, you will start to explore the functionality the UE4 engine has to offer. You will use the UE4 editor to create your own world, and then program in some seriously fun gameplay. By the end of this book, you should have a good grasp of how to program in C++.

Who this book is written for

If you are really passionate about games and have always wanted to write your own, this book is perfect for you. It will help you get started with programming in C++ and explore the immense functionalities of UE4.

What you will learn from this book

- Visualize and truly understand C++ programming concepts, such as how data is saved in computer memory and how program flow works
- Write reusable code by grouping lines of code into functions
- Learn how inheritance works-how traits of a base class are passed on to derived classes
- Learn about dynamic allocation of new memory for your program
- Design your own world using the UE4 editor
- Practice programming by coding behaviors into your game world, including player inventory tracking, monsters, and NPCs

Learn C++ programming with a fun, real-world application that allows you to create your own games!
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 8 "Actors and Pawns"
- A synopsis of the book’s content
- More information on Learning C++ by Creating Games with UE4

About the Author

William Sherif is a C++ programmer with more than 8 years' programming experience. He has a wide range of experience in the programming world, from game programming to web programming. He has also worked as a university course instructor (sessional) for 7 years.

He has released several apps on to the iTunes store, including strum and MARSHALL OF THE ELITE SQUADRON.

In the past, he has won acclaim for delivering course material in an easy-to-understand manner.

I'd like to thank my family, for their support when I was writing this book; Mostafa and Fatima, for their hospitality; as well as Ryan and Reda, for letting me write.
Learning C++ by Creating Games with UE4

So, you want to program your own games using Unreal Engine 4 (UE4). You have a great number of reasons to do so:

- UE4 is powerful: UE4 provides some of the most state-of-the-art, beautiful, realistic lighting and physics effects, of the kind used by AAA Studios.
- UE4 is device-agnostic: Code written for UE4 will work on Windows desktop machines, Mac desktop machines, Android devices, and iOS devices (at the time of writing this book—even more devices may be supported in the future).

So, you can use UE4 to write the main parts of your game once, and after that, deploy to iOS and Android Marketplaces without a hitch. (Of course, there will be a few hitches: iOS and Android in app purchases will have to be programmed separately.)

What is a game engine anyway?

A game engine is analogous to a car engine: the game engine is what drives the game. You will tell the engine what you want, and (using C++ code and the UE4 editor) the engine will be responsible for actually making that happen.

You will build your game around the UE4 game engine, similar to how the body and wheels are built around an actual car engine. When you ship a game with UE4, you are basically customizing the UE4 engine and retrofitting it with your own game's graphics, sounds, and code.

What will using UE4 cost me?

The answer, in short, is $19 and 5 percent of sales.

"What?" you say. $19?

That's right. For only $19, you get full access to a world class AAA Engine, complete with a source. This is a great bargain, considering the fact that other engines can cost anywhere from $500 to $1,000 for just a single license.

Why don't I just program my own engine and save the 5 percent?

Take it from me, if you want to create games within a reasonable time frame and you don't have a large team of dedicated engine programmers to help you, you'll want to focus your efforts on what you sell (your game).
Not having to focus on programming a game engine gives you the freedom to think only about how to make the actual game. Not having to maintain and bug-fix your own engine is a load off your mind too.

**A game's overview – the Play-Reward-Growth loop**

I want to show you this diagram now because it contains a core concept that many novice developers might miss when writing their first games. A game can be complete with sound effects, graphics, realistic physics, and yet, still not feel like a game. Why is that?

Starting at the top of the loop, Play actions committed during the game (such as defeating a monster) result in rewards for the player (such as gold or experience). These rewards, in turn, can be used for in-game Growth (such as stats increases or new worlds to explore). This Growth then drives the gameplay in new and interesting ways. For example, a new weapon can change the basic mechanics of fighting, new spells let you take on groups of monsters with a completely different approach, or new modes of transportation can let you reach areas that were previously inaccessible.

This is the basic core loop that creates interesting gameplay. The key is that Play must result in some kind of Reward—think of glittering gold pieces popping out of nasty baddies. For rewards to have a point, it must result in some kind of Growth in the gameplay. Think about how many new locations were unlocked with the hook shot in *The Legend of Zelda*.

A game that is only Play (without Rewards or Growth) won't feel like a game: it will feel only like a really basic prototype of a game. For example, imagine a flight simulator with just an open world and no goals or objectives as well as without the ability to upgrade your plane or weapons. It wouldn't be much of a game.
A game with only Play and Rewards (but no Growth) will feel primitive and simple. The rewards will not satisfy the player if they cannot be used for anything.

A game with only Play and Growth (without Rewards) will just be seen as a mindless increasing challenge, without giving the player a sense of gratification for his achievements.

A game with all three elements will keep the player engaged with an entertaining Play. The Play has a rewarding result (loot drops and story progression), which results in the Growth of the game world. Keeping this loop in mind while you are devising your game will really help you to design a complete game.

**Monetization**

Something you need to think about early in your game's development is your monetization strategy. How will your game make money? If you are trying to start a company, you have to think of what will be your sources of revenue from early on.

Are you going to try to make money from the purchase price, such as *Jamestown, The Banner Saga, Castle Crashers, or Crypt of the Necrodancer*? Or, will you focus on distributing a free game with in-app purchases, such as *Clash of Clans, Candy Crush Saga, or Subway Surfers*?

A class of games for mobile devices (for example, builder games on iOS) make lots of money by allowing the user to pay in order to skip Play and jump straight to the rewards and Growth parts of the loop. The pull to do this can be very powerful; many people spend hundreds of dollars on a single game.

**Why C++**

UE4 is programmed in C++. To write code for UE4, you must know C++.

C++ is a common choice for game programmers because it offers very good performance combined with object-oriented programming features. It's a very powerful and flexible language.

**What This Book Covers**

*Chapter 1, Coding with C++,* talks about getting up and running with your first C++ program.

*Chapter 2, Variables and Memory,* talks about how to create, read, and write variables from computer memory.

*Chapter 3, If, Else, and Switch,* talks about branching the code: that is, allowing different sections of the code to execute, depending on program conditions.
Chapter 4, *Looping*, discusses how we repeat a specific section of code as many times as needed.

Chapter 5, *Functions and Macros*, talks about functions, which are bundles of code that can get called any number of times, as often you wish.

Chapter 6, *Objects, Classes, and Inheritance*, talks about class definitions and instantiating some objects based on a class definition.

Chapter 7, *Dynamic Memory Allocation*, discusses heap-allocated objects as well as low-level C and C++ style arrays.

Chapter 8, *Actors and Pawns*, is the first chapter where we actually delve into UE4 code. We begin by creating a game world to put actors in, and derive an Avatar class from a customized actor.

Chapter 9, *Templates and Commonly Used Containers*, explores UE4 and the C++ STL family of collections of data, called containers. Often, a programming problem can be simplified many times by selecting the right type of container.

Chapter 10, *Inventory System and Pickup Items*, discusses the creation of an inventory system with the ability to pick up new items.

Chapter 11, *Monsters*, teaches how to create monsters that give chase to the player and attack it with weapons.

Chapter 12, *Spell Book*, teaches how to create and cast spells in our game.
Now we will really delve into UE4 code. At first, it is going to look daunting. The UE4 class framework is massive, but don't worry. The framework is massive, so your code doesn't have to be. You will find that you can get a lot done and a lot onto the screen using relatively less code. This is because the UE4 engine code is so extensive and well programmed that they have made it possible to get almost any game-related task done easily. Just call the right functions, and voila, what you want to see will appear on the screen. The entire notion of a framework is that it is designed to let you get the gameplay you want, without having to spend a lot of time in sweating out the details.

**Actors versus pawns**

In this chapter, we will discuss actors and pawns. Although it sounds as if pawns will be a more basic class than actors, it is actually the other way around. A UE4 actor (the actor class) object is the basic type of the things that can be placed in the UE4 game world. In order to place anything in the UE4 world, you must derive from the actor class.

A pawn is an object that represents something that you or the computer's Artificial Intelligence (AI) can control on the screen. The pawn class derives from the actor class, with the additional ability to be controlled by the player directly or by an AI script. When a pawn or actor is controlled by a controller or AI, it is said to be possessed by that controller or AI.

Think of the actor class as a character in a play. Your game world is going to be composed of a bunch of actors, all acting together to make the gameplay work. The game characters, Non-player Characters (NPCs), and even treasure chests will be actors.
Creating a world to put your actors in

Here, we will start from scratch and create a basic level into which we can put our game characters.

The UE4 team has already done a great job of presenting how the world editor can be used to create a world in UE4. I want you to take a moment to create your own world.

First, create a new, blank UE4 project to get started. To do this, in the Unreal Launcher, click on the Launch button beside your most recent engine installation, as shown in the following screenshot:

That will launch the Unreal Editor. The Unreal Editor is used to visually edit your game world. You’re going to spend a lot of time in the Unreal Editor, so please take your time to experiment and play around with it.
I will only cover the basics of how to work with the UE4 editor. You will need to let your creative juices flow, however, and invest some time in order to become familiar with the editor.

To learn more about the UE4 editor, take a look at the Getting Started: Introduction to the UE4 Editor playlist, which is available at https://www.youtube.com/playlist?list=PLZlv_N0_Olgasd4Io0e9Cx9wHoBB7rxFl.

Once you've launched the UE4 editor, you will be presented with the Projects dialog. The following screenshot shows the steps to be performed with numbers corresponding to the order in which they need to be performed:

Perform the following steps to create a project:

1. Select the New Project tab at the top of the screen.
2. Click on the C++ tab (the second subtab).
3. Then select Basic Code from the available projects listing.
4. Set the directory where your project is located (mine is \Unreal Projects\). Choose a hard disk location with a lot of space (the final project will be around 1.5 GB).

5. Name your project. I called mine GoldenEgg.

6. Click on Create Project to finalize project creation.

Once you’ve done this, the UE4 launcher will launch Visual Studio. There will only be a couple of source files in Visual Studio, but we’re not going to touch those now. Make sure that Development Editor is selected from the Configuration Manager dropdown at the top of the screen, as shown in the following screenshot:
Now launch your project by pressing Ctrl + F5 in Visual Studio. You will find yourself in the Unreal Engine 4 editor, as shown in the following screenshot:

The UE4 editor
We will explore the UE4 editor here. We'll start with the controls since it is important to know how to navigate in Unreal.

Editor controls
If you've never used a 3D editor before, the controls can be quite hard to learn. These are the basic navigation controls while in edit mode:

- Use the arrow keys to move around in the scene
- Press Page Up or Page Down to go up and down vertically
- Left mouse click + drag it left or right to change the direction you are facing
- Left mouse click + drag it up or down to dolly (move the camera forward and backward, same as pressing up/down arrow keys)
Actors and Pawns

- Right mouse click + drag to change the direction you are facing
- Middle mouse click + drag to pan the view
- Right mouse click and the W, A, S, and D keys to move around the scene

Play mode controls
Click on the Play button in the bar at the top, as shown in the following screenshot. This will launch the play mode.

Once you click on the Play button, the controls change. In play mode, the controls are as follows:

- The W, A, S, and D keys for movement
- The left or right arrow keys to look toward the left and right, respectively
- The mouse's motion to change the direction in which you look
- The Esc key to exit play mode and return to edit mode

What I suggest you do at this point is try to add a bunch of shapes and objects into the scene and try to color them with different materials.

Adding objects to the scene
Adding objects to the scene is as easy as dragging and dropping them in from the Content Browser tab. The Content Browser tab appears, by default, docked at the left-hand side of the window. If it isn't seen, simply select Window and navigate to Content Browser in order to make it appear.
Make sure that the Content Browser is visible in order to add objects to your level.

Next, select the **Props** folder on the left-hand side of the **Content Browser**.

Drag and drop things from the Content Browser into your game world.
Actors and Pawns

To resize an object, press R on your keyboard. The manipulators around the object will appear as boxes, which denotes resize mode.

Press R on your keyboard to resize an object

In order to change the material that is used to paint the object, simply drag and drop a new material from the **Content Browser** window inside the **Materials** folder.

Drag and drop a material from the Content Browser's Materials folder to color things with a new color
Materials are like paints. You can coat an object with any material you want by simply dragging and dropping the material you desire onto the object you desire it to be coated on. Materials are only skin-deep: they don't change the other properties of an object (such as weight).

Starting from scratch
If you want to start creating a level from scratch, simply click on File and navigate to New Level..., as shown here:

You can then select between Default and Empty Level. I think selecting Empty Level is a good idea, for the reasons that are mentioned later.
The new level will be completely black in color to start with. Try dragging and dropping some objects from the **Content Browser** tab again.

This time, I added a resized shapes / box for the ground plane and textured it with moss, a couple of **Props / SM_Rocks**, **Particles / P_Fire**, and most importantly, a light source.

Be sure to save your map. Here's a snapshot of my map (how does yours look?):

![Snapshot of a map](image)

If you want to change the default level that opens when you launch the editor, go to **Project Settings | Maps & Modes**; then you will see a **Game Default Map** and **Editor Startup Map** setting, as shown in the following screenshot:

![Screenshot of Project Settings](image)

**Adding light sources**

Note that if your scene appears completely black, it is possible that you forgot to put a light source into it.
In the previous scene, the P_Fire particle emitter acts as a light source, but it only emits a small amount of light. To make sure that everything appears well-lit in your scene, you should add a light source, as follows:

1. Go to **Window** and then click on **Modes** to ensure that the light sources panel is shown:

2. Then, from the **Modes** panel, drag one of the **Lights** object into the scene:

3. Select the lightbulb and box icon (it looks like a mushroom, but it isn't).
4. Click on **Lights** in the left-hand side panel.
5. Select the type of light you want and just pull it into your scene.

If you don't have a light source, your scene will appear completely black.
Collision volumes

You might have noticed that, so far, the camera just passes through all the scene geometry, even in play mode. That's not good. Let's make it such that the player can't just walk through the rocks in our scene.

There are a few different types of collision volumes. Generally, perfect mesh-mesh collisions are way too expensive to do at runtime. Instead, we use an approximation (a bounding volume) to guess the collision volume.

Adding collision detection for the objects editor

The first thing we have to do is associate a collision volume with each of the rocks in the scene.

We can do this from the UE4 editor as follows:

1. Click on an object in the scene for which you want to add a collision volume.
2. Right-click on this object in the Scene Outliner tab (the default appears on the right-hand side of the screen) and select edit, as shown in the following screenshot:

You will find yourself in the mesh editor.
3. Ensure that the collision volume is highlighted, at the top of the screen:

4. Go to the Collision menu and then click on Add Capsule Simplified Collision:
5. The collision volume, when added successfully, will appear as a bunch of lines surrounding the object, as shown in the following images:

![The default collision capsule (left) and manually resized versions (right)](image)

6. You can resize (R), rotate (E), move (W), and change the collision volume as you wish, the same way you would manipulate an object in the UE4 editor.

7. When you're done with adding collision meshes, try to click on Play; you will notice that you can no longer pass through your collidable objects.

Adding an actor to the scene
Now that we have a scene up and running, we need to add an actor to the scene. Let's first add an avatar for the player, complete with a collision volume. To do this, we'll have to inherit from a UE4 GameFramework class.

Creating a player entity
In order to create an onscreen representation of the player, we'll need to derive from the Character class in Unreal.

Inheriting from UE4 GameFramework classes
UE4 makes it easy to inherit from the base framework classes. All you have to do is perform the following steps:

1. Open your project in the UE4 editor.
2. Go to File and then select Add Code to Project....
Navigating to File | Add Code To Project... will allow you to derive from any of the UE4 GameFramework classes.

3. From here, choose the base class you want to derive from. You have **Character**, **Pawn**, **Actor**, and so on, but for now, we will derive from **Character**:

Select the UE4 class you want to derive from
4. Click on **Next >** to get this dialog box, where you name the class. I named my player’s class **Avatar**.

5. Finally, click on **Create Class** to create the class in code, as shown in the preceding screenshot.

Let UE4 refresh your Visual Studio project when it asks you. Open the new Avatar.h file from the **Solution Explorer**.

The code that UE4 generates will look a little weird. Remember the macros that I suggested you avoid in Chapter 5, Functions and Macros. The UE4 code uses macros extensively. These macros are used to copy and paste boilerplate starter code that lets your code integrate with the UE4 editor.

The contents of the Avatar.h file are shown in the following code:

```cpp
#pragma once
// Avatar.h code file
#include "GameFramework/Character.h"
#include "Avatar.generated.h"
UCLASS()
class MYPROJECT_API AAvatar : public ACharacter
{
  GENERATED_UCLASS_BODY()
};
```
Let's talk about macros for a moment.

The \texttt{UCLASS()} macro basically makes your C++ code class available in the UE4 editor. The \texttt{GENERATED_UCLASS_BODY()} macro copies and pastes code that UE4 needs to make your class function properly as a UE4 class.

For \texttt{UCLASS()} and \texttt{GENERATED_UCLASS_BODY()}, you don't truly need to understand how UE4 works its magic. You just need to make sure that they are present at the right spot (where they were when you generated the class).

**Associating a model with the Avatar class**

Now we need to associate a model with our character object. In order to do this, we need a model to play with. Fortunately, there is a whole pack of sample models available from the UE4 marketplace for free.

**Downloading free models**

To create the player object, we'll download the \textit{Animation Starter Pack} file (which is free) from the \textit{Marketplace} tab.

![Animation Starter Pack in Marketplace](image)

From the Unreal Launcher, click on Marketplace and search for Animation Starter Pack, which is free at the time of writing this book.
After you've downloaded the **Animation Starter Pack** file, you will be able to add it to any of the projects you've previously created, as shown in the following screenshot:

When you click on **Add to project** under **Animation Starter Pack**, you will get this pop up, asking which project to add the pack to:

Simply select your project and the new artwork will be available in your **Content Browser**.
Loading the mesh

In general, it is considered a bad practice to hardcode your assets into the game. Hardcoding means that you write C++ code that specifies the asset to load. However, hardcoding means the loaded asset is part of the final executable, which will mean that changing the asset that is loaded wouldn't be modifiable at runtime. This is a bad practice. It is much better to be able to change the asset loaded during runtime.

For this reason, we're going to use the UE4 blueprints feature to set up the model mesh and collision capsule of our Avatar class.

Creating a blueprint from our C++ class

1. This is really easy. Open the Class Viewer tab by navigating to Window and then clicking on Class Viewer, as shown here:
2. In the **Class Viewer** dialog, start typing in the name of your C++ class. If you have properly created and exported the class from your C++ code, it will appear, as shown in the following screenshot:

![Class Viewer Screenshot](image)

If your Avatar class does not show up, close the editor and compile/run the C++ project again.

3. Right-click on the class that you want to create a blueprint of (in my case, it's my Avatar class).

4. Name your blueprint something unique. I called my blueprint **BP_Avatar**.

5. Now, open this blueprint for editing, by double-clicking on **BP_Avatar** (it will appear in the **Class Viewer** tab after you add it, just under Avatar), as shown in the following screenshot:

![Blueprint Screenshot](image)
6. You will be presented with the blueprints window for your new **BP_Avatar** object, as shown here:

![Blueprint Window](image)

From this window, you can attach a model to the **Avatar** class visually. Again, this is the recommended pattern since artists will typically be the ones setting up their assets for game designers to play with.
7. To set up the default mesh, click on the Defaults button at the top. Scroll down through the properties until you come across Mesh.

8. Click on the dropdown and select HeroTPP for your mesh, as shown in the preceding screenshot.

9. If HeroTPP doesn't appear in the dropdown, make sure that you download and add the Animation Starter Pack to your project. Alternatively, you can add the yellow TutorialTPP model to your project if you select Show Engine Content under View Options:
10. What about the collision volume? Click on the Components tab in the blueprint editor for your avatar:

If your capsule doesn't encapsulate your model, adjust the model so that it fits.

If your model ended up like mine, the capsule is off the mark! We need to adjust it.
11. Click on the blue Avatar model and press the W key. Move him down until he fits inside the capsule. If the capsule isn't big enough, you can adjust its size in the Details tab under Capsule Height and Capsule Radius, as shown in the following screenshot:

![Screenshot showing capsule height and radius adjustments](image-url)

You can stretch your capsule by adjusting the Capsule Height property.

12. Now, we’re ready to add this avatar to the game world. Click and drag your BP_Avatar model from the Class Viewer tab to your scene in the UE4 editor.
The pose of Avatar is called the T-pose. Animators often leave their characters in this default pose. Animations can be applied to the character in order to make them change this default pose to something more interesting. You want him animated, you say! Well, that's easy.

Under the Defaults tab in the blueprint editor, just above Mesh, there is an Animation section where you can select the active animation on your Mesh. If you wish to use a certain animation asset, simply click on the drop-down menu and choose the animation you desire to show.

A better thing to do, however, is to use a blueprint for the animation. This way, an artist can properly set the animation based on what the character is doing. If you select Use Animation Blueprint from Animation Mode and then select ASP_HeroTPP_AnimBlueprint from the drop-down menu, the character will appear to behave much better in the game, because the animation will be adjusted by the blueprint (which would have been done by an artist) as the character moves.
We can't cover everything here. Animation blueprints are covered in Chapter 11, Monsters. If you're really interested in animation, it also wouldn't be a bad idea to sit through a couple of Gnomon Workshop tutorials on IK, animation, and rigging, such as Alex Alvarez's Rigging 101 class at http://www.thegnomonworkshop.com/store/product/768/Rigging-101.

One more thing: let's make the camera for the Avatar appear behind it. This will give you a third person's point-of-view, which will allow you to see the whole character, as shown in the following screenshot with the corresponding steps:

1. In the BP_Avatar blueprint editor, click on the Components tab.
2. Click on Add Component.
3. Choose to add a Camera.

A camera will appear in the viewport. You can click on the camera and move it around. Position the camera so that it is somewhere behind the player. Make sure that the blue arrow on the player is facing the same direction as the camera. If it isn't, rotate the Avatar model mesh so that it faces the same direction as its blue-colored arrow.
The blue-colored arrow on your model mesh indicates the forward direction for the model mesh. Make sure that the camera's opening faces the same direction as the character's forward vector.

**Writing C++ code that controls the game's character**

When you launch your UE4 game, you might notice that the camera is a default, free-flying camera. What we will do now is make the starting character an instance of our `Avatar` class and control our character using the keyboard.

**Making the player an instance of the Avatar class**

In the Unreal Editor, create a subclass of `Game Mode` by navigating to File | Add Code To Project... and selecting Game Mode. I named mine `GameModeGoldenEgg`. 
The UE4 GameMode contains the rules of the game and describes how the game is played to the engine. We will work more with our GameMode class later. For now, we need to subclass it.

Recompile your project from Visual Studio, so you can create a GameModeGoldenEgg blueprint. Create the GameMode blueprint by going to the Blueprints icon in the menu bar at the top, clicking on GameMode, and then selecting + Create | GameModeGoldenEgg (or whatever you named your GameMode subclass in step 1).

1. Name your blueprint; I called mine BP_GameModeGoldenEgg, as shown in the following screenshot:

2. Your newly created blueprint will open in the blueprint editor. If it doesn't, you can open the BP_GameModeGoldenEgg class from the Class Viewer tab.
3. Select your **BP_Avatar** class from the **Default Pawn Class** panel, as shown in the following screenshot. The **Default Pawn Class** panel is the type of object that will be used for the player.

![Default Pawn Class Panel](image)

4. Now, launch your game. You can see a back view as the camera is placed behind the place, as shown here:

![Game View](image)

You'll notice that you can't move. Why is that? The answer is because we haven't set up the controller inputs yet.
Setting up controller inputs

1. To set up controller inputs, go to Settings | Project Settings:

2. Next, in the left-hand side panel, scroll down until you see Input under Engine.

3. On the right-hand side, you can set up some Bindings. Click on the small arrow next to Axis Mappings in order to expand it. Add just two axis mappings to start, one called Forward (connected to the keyboard letter W) and one called Strafe (connected to the keyboard letter D). Remember the names that you set; we will look them up in C++ code in just a moment.

In the Avatar.h constructor, you need to add three member function declarations, as shown here:

```cpp
UCLASS()
class GOLDENEGG_API AAvatar : public ACharacter
{
    GENERATED_UCLASS_BODY()

    // New! These 3 new member function declarations
    // they will be used to move our player around!
    void SetupPlayerInputComponent(class UInputComponent* InputComponent) override;
    void MoveForward( float amount );
    void MoveRight( float amount );
};
```

Notice how the first member function we're adding (SetupPlayerInputComponent) is an override of a virtual function. SetupPlayerInputComponent is a virtual function in the APawn base class.

In the Avatar.cpp file, you need to put the function bodies. Add the following member function definitions:

```cpp
void AAvatar::SetupPlayerInputComponent(class UInputComponent* InputComponent)
{
    check(InputComponent);
    InputComponent->BindAxis("Forward", this, &AAvatar::MoveForward);
    InputComponent->BindAxis("Strafe", this, &AAvatar::MoveRight);
}
```

This member function looks up the Forward and Strafe axis bindings that we just created in Unreal Editor and connects them to the member functions inside the this class. Which member functions should we connect to? Why, we should connect to AAvatar::MoveForward and AAvatar::MoveRight. Here are the member function definitions for these two functions:

```cpp
void AAvatar::MoveForward( float amount )
{
    // Don't enter the body of this function if Controller is
    // not set up yet, or if the amount to move is equal to 0
    if( Controller && amount )
    {
        FVector fwd = GetActorForwardVector();
        // we call AddMovementInput to actually move the
```
Actors and Pawns

    // player by `amount` in the `fwd` direction
    AddMovementInput(fwd, amount);

void AAvatar::MoveRight(float amount)
{
    if (Controller && amount)
    {
        FVector right = GetActorRightVector();
        AddMovementInput(right, amount);
    }
}

The Controller object and the AddMovementInput functions are defined in the APawn base class. Since the Avatar class derives from ACharacter, which in turn derives from APawn, we get free use of all the member functions in the base class APawn. Now do you see the beauty of inheritance and code reuse?

Exercise
Add axis bindings and C++ functions to move the player to the left and back.

Here' a hint: you only need to add axis bindings if you realize going backwards is only the negative of going forward.

Solution
Enter two extra axis bindings by navigating to Settings | Project Settings... | Input, as shown here:
Scale the S and A inputs by -1.0. This will invert the axis. So pressing the S key in the game will move the player forward. Try it!

Alternatively, you can define two completely separate member functions in your AAvatar class, as follows, and bind the A and S keys to AAvatar::MoveLeft and AAvatar::MoveBack, respectively:

```cpp
void AAvatar::MoveLeft( float amount )
{
    if( Controller && amount )
    {
        FVector left = -GetActorRightVector();
        AddMovementInput(left, amount);
    }
}

void AAvatar::MoveBack( float amount )
{
    if( Controller && amount )
    {
        FVector back = -GetActorForwardVector();
        AddMovementInput(back, amount);
    }
}
```
**Actors and Pawns**

**Yaw and pitch**

We can change the direction in which the player looks by setting the yaw and pitch of the controller.

All we have to do here is add in new axis bindings for the mouse, as shown in the following screenshot:

![Screenshot showing mouse axis bindings](image)

From C++, you need to add in two new member function declarations to **AAvatar.h**:

```cpp
void Yaw( float amount );
void Pitch( float amount );
```

The bodies of these member functions will go in the **AAvatar.cpp** file:

```cpp
void AAvatar::Yaw( float amount )
{
    AddControllerYawInput(200.f * amount * GetWorld()->GetDeltaSeconds());
}

void AAvatar::Pitch( float amount )
{
    AddControllerPitchInput(200.f * amount * GetWorld()->GetDeltaSeconds());
}
```

Then, add two lines to **SetupPlayerInputComponent**:

```cpp
void AAvatar::SetupPlayerInputComponent(class UInputComponent* InputComponent)
```
Here, notice how I’ve multiplied the amount values in the Yaw and Pitch functions by 200. This number represents the mouse’s sensitivity. You can (should) add a float member to the Avatar class in order to avoid hardcoding this sensitivity number.

GetWorld()->GetDeltaSeconds() gives you the amount of time that passed between the last frame and this frame. It isn’t a lot: GetDeltaSeconds() should be around 16 milliseconds (0.016 s) most of the time (if your game is running at 60 fps).

So, now we have player input and control. To add new functionality to your Avatar, this is all that you have to do:

1. Bind your key or mouse actions by going to Settings | Project Settings | Input.
2. Add a member function to run when that key is pressed.
3. Add a line to SetupPlayerInputComponent, connecting the name of the bound input to the member function we want to run when that key is pushed.

Creating non-player character entities

So, we need to create a few NPCs (non-playable characters). NPCs are characters within the game that help the player. Some offer special items, some are shop vendors, and some have information to give to the player. In this game, they will react to the player as he gets near. Let’s program in some of this behavior.

First, create another subclass of Character. In the UE4 Editor, go to File | Add Code To Project... and choose the Character class from which you can make a subclass. Name your subclass NPC.

Now, edit your code in Visual Studio. Each NPC will have a message to tell the player, so we add in a UPROPERTY() FString property to the NPC class.

FStrings are UE4’s version of C++’s <string> type. When programming in UE4, you should use the FString objects over C++ STL’s string objects. In general, you should preferably use UE4’s built-in types, as they guarantee cross-platform compatibility.
How to add the `UPROPERTY()` `FString` property to the NPC class is shown in the following code:

```cpp
UCLASS()
class GOLDENEGG_API ANPC : public ACharacter
{
    GENERATED_UCLASS_BODY()
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Collision)
    TSubobjectPtr<class USphereComponent> ProxSphere;
    // This is the NPC's message that he has to tell us.
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = NPCMessage)
    FString NpcMessage;
    // When you create a blueprint from this class, you want to be
    // able to edit that message in blueprints,
    // that's why we have the EditAnywhere and BlueprintReadWrite
    // properties.
}
```

Notice that we put the `EditAnywhere` and `BlueprintReadWrite` properties into the `UPROPERTY` macro. This will make the `NpcMessage` editable in blueprints.

Full descriptions of all the UE4 property specifiers are available at https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Reference/Properties/index.html.

Recompile your project (as we did for the `Avatar` class). Then, go to the Class Viewer, right click on your `NPC` class, and create a blueprint from it.

Each NPC character you want to create can be a blueprint based off of the `NPC` class. Name each blueprint something unique, as we'll be selecting a different model mesh and message for each NPC that appears, as shown in the following screenshot:
Now, open the blueprint, select skeletal **mesh** from the **Add Components**, and adjust the capsule (as we did for **BP_Avatar**). You can also change the material of your new character so that he looks different from the player.

Change the material of your character in your mesh’s properties. Under the Rendering tab, click on the + icon to add a new material. Then, click on the small capsule-shaped item to select a material to render with.
In the **Defaults** tab, search for the `NpcMessage` property. This is our connection between C++ code and blueprints: because we entered a `UPROPERTY()` function on the `FString NpcMessage` variable, that property appears editable within UE4, as shown in the following screenshot:

![Image of UE4 editor showing NpcMessage property](image)

Now, drag **BP_NPC_Owen** into the scene. You can create a second or third character as well, and be sure to give them unique names, appearances, and messages.

![Image of Content Browser with NPC blueprints](image)

I've created two blueprints for NPCs based on the NPC base classes, **BP_NPC_Justin** and **BP_NPC_Owen**. They have different appearances and different messages for the player.
Displaying a quote from each NPC dialog box

To display a dialog box, we need a custom (heads-up display) HUD. In the UE4 editor, go to File | Add Code To Project... and choose the HUD class from which the subclass is created. Name your subclass as you wish; I've named mine MyHUD.

After you have created the MyHUD class, let Visual Studio reload. We will make some code edits.

Displaying messages on the HUD

Inside the MyHUD class, we need to implement the DrawHUD() function in order to draw our messages to the HUD and to initialize a font to draw to the HUD with, as shown in the following code:

```cpp
UCLASS()
class GOLDENEGG_API AMyHUD : public AHUD
{
    GENERATED_UCLASS_BODY()
    // The font used to render the text in the HUD.
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = HUDFont)
    UFont* hudFont;
    // Add this function to be able to draw to the HUD!
    virtual void DrawHUD() override;
};
```
The HUD font will be set in a blueprinted version of the AMyHUD class. The DrawHUD() function runs once per frame. In order to draw within the frame, add a function to the AMyHUD.cpp file:

```cpp
void AMyHUD::DrawHUD()
{
    // call superclass DrawHUD() function first
    Super::DrawHUD();
    // then proceed to draw your stuff.
    // we can draw lines..
    DrawLine( 200, 300, 400, 500, FLinearColor::Blue );
    // and we can draw text!
    DrawText( "Greetings from Unreal!", FVector2D( 0, 0 ), hudFont,
              FVector2D( 1, 1 ), FColor::White );
}
```

Wait! We haven't initialized our font yet. To do this, we need to set it up in blueprints. Compile and run your Visual Studio project. Once you are in the editor, go to the Blueprints menu at the top and navigate to GameMode | HUD | + Create | MyHUD.

Creating a blueprint of the MyHUD class
I called mine BP_MyHUD. Edit BP_MyHUD and select a font from the drop-down menu under **HUDFont**:

I selected RobotoDistanceField for my HUD font

Next, edit your **Game Mode** blueprint (BP_GameModeGoldenEgg) and select your new BP_MyHUD (not MyHUD class) for the **HUD Class** panel:
Test your program by running it! You should see text printed on the screen.

Using TArray<Message>

Each message we want to display for the player will have a few properties:

- An FString variable for the message
- A float variable for the time to display it
- An FColor variable for the color of the message

So it makes sense for us to write a little struct function to contain all this information.

At the top of MyHUD.h, insert the following struct declaration:

```cpp
struct Message
{
    FString message;
    float time;
    FColor color;
    Message()
    {
        // Set the default time.
        time = 5.f;
        color = FColor::White;
    }
    Message( FString iMessage, float iT ime, FColor iColor )
    {
```
message = iMessage;
time = iTime;
color = iColor;
};

An enhanced version of the Message structure (with a background color) is in the code package for this chapter. We used simpler code here so that it'd be easier to understand the chapter.

Now, inside the AMyHUD class, we want to add a TArray of these messages. A TArray is a UE4-defined special type of dynamically growable C++ array. We will cover the detailed use of TArray in the next chapter, but this simple usage of TArray should be a nice introduction to garner your interest in the usefulness of arrays in games. This will be declared as TArray<Message>:

```
UCLASS()
class GOLDENEGG_API AMyHUD : public AHUD
{
    GENERATED_UCLASS_BODY()
    // The font used to render the text in the HUD.
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = HUDFont)
    UFont* hudFont;
    // New! An array of messages for display
    TArray<Message> messages;
    virtual void DrawHUD() override;
    // New! A function to be able to add a message to display
    void addMessage( Message msg );
};
```

Now, whenever the NPC has a message to display, we're just need to call AMyHud::addMessage() with our message. The message will be added to TArray of the messages to be displayed. When a message expires (after a certain amount of time), it will be removed from the HUD.

Inside the AMyHUD.cpp file, add the following code:

```
void AMyHUD::DrawHUD()
{
    Super::DrawHUD();
    // iterate from back to front thru the list, so if we remove
    // an item while iterating, there won't be any problems
    for( int c = messages.Num() - 1; c >= 0; c-- )
    {
        // draw the background box the right size
```
// for the message
float outputWidth, outputHeight, pad=10.f;
GetTextSize( messages[c].message, outputWidth, outputHeight, hudFont, 1.f );

float messageH = outputHeight + 2.f*pad;
float x = 0.f, y = c*messageH;

// black backing
DrawRect( FLinearColor::Black, x, y, Canvas->SizeX, messageH );
// draw our message using the hudFont
DrawText( messages[c].message, messages[c].color, x + pad, y + pad, hudFont );

// reduce lifetime by the time that passed since last frame.
messages[c].time -= GetWorld()->GetDeltaSeconds();

// if the message's time is up, remove it
if( messages[c].time < 0 )
{
    messages.RemoveAt( c );
}

void AMyHUD::addMessage( Message msg )
{
    messages.Add( msg );
}

The AMyHUD::DrawHUD() function now draws all the messages in the messages array, and arranges each message in the messages array by the amount of time that passed since the last frame. Expired messages are removed from the messages collection once their time value drops below 0.

Exercise

Refactor the DrawHUD() function so that the code that draws the messages to the screen is in a separate function called DrawMessages().

The Canvas variable is only available in DrawHUD(), so you will have to save Canvas->SizeX and Canvas->SizeY in class-level variables.
Refactoring means to change the way code works internally so that it is more organized or easier to read but still has the same apparent result to the user running the program. Refactoring often is a good practice. The reason why refactoring occurs is because nobody knows exactly what the final code should look like once they start writing it.

**Solution**

See the AMyHUD::DrawMessages() function in the code package for this chapter.

**Triggering an event when it is near an NPC**

To trigger an event near the NPC, we need to set an additional collision detection volume that is a bit wider than the default capsule shape. The additional collision detection volume will be a sphere around each NPC. When the player steps into the NPC sphere, the NPC reacts and displays a message.

![Image of NPC with additional sphere]

We're going to add the dark red sphere to the NPC so that he can tell when the player is nearby.
Inside your NPC.h class file, add the following code in order to declare ProxSphere and UFUNCTION called Prox:

```cpp
UCLASS()
class GOLDENEGG_API ANPC : public ACharacter
{
    GENERATED_UCLASS_BODY()
    // This is the NPC's message that he has to tell us.
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = NPCMessage)
    FString NpcMessage;
    // The sphere that the player can collide with to get item
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Collision)
    TSubobjectPtr<class USphereComponent> ProxSphere;
    // This is a bit weird and not what you'd expect,
    // but it happens because this is a BlueprintNativeEvent
    UFUNCTION(BlueprintNativeEvent, Category = "Collision")
    void Prox(AActor* OtherActor, UPrimitiveComponent* OtherComp,
              int32 OtherBodyIndex, bool bFromSweep, const FHitResult & SweepResult);
};
```

This looks a bit messy, but it is actually not that complicated. Here, we declare an extra bounding sphere volume called ProxSphere, which detects when the player is near the NPC.

In the NPC.cpp file, we need to add the following code in order to complete the proximity detection:

```cpp
ANPC::ANPC(const class FPostConstructInitializeProperties& PCIP) : Super(PCIP)
{
    ProxSphere = PCIP.CreateDefaultSubobject<USphereComponent>(this, TEXT("Proximity Sphere"));
    ProxSphere->AttachTo(RootComponent);
    ProxSphere->SetSphereRadius(32.f);
    // Code to make ANPC::Prox() run when this proximity sphere overlaps another actor.
    ProxSphere->OnComponentBeginOverlap.AddDynamic(this, &ANPC::Prox);
    NpcMessage = "Hi, I'm Owen";//default message, can be edited
                      // in blueprints
}
```
// Note! Although this was declared ANPC::Prox() in the header,  
// it is now ANPC::Prox_Implementation here.

void ANPC::Prox_Implementation( AActor* OtherActor,  
    UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool  
    bFromSweep, const FHitResult & SweepResult )
{
    // This is where our code will go for what happens  
    // when there is an intersection
}

Make the NPC display something to the HUD when something is nearby

When the player is near the NPC sphere collision volume, display a message to the 
HUD that alerts the player about what the NPC is saying.

This is the complete implementation of ANPC::Prox_Implementation:

    void ANPC::Prox_Implementation( AActor* OtherActor,  
        UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep,  
        const FHitResult & SweepResult )
{
    // if the overlapped actor is not the player,  
    // you should just simply return from the function
    if( Cast<AAvatar>( OtherActor ) == nullptr )
    {
        return;
    }

    APlayerController* PController = GetWorld()->
        GetFirstPlayerController();
    if( PController )
    {
        AMyHUD * hud = Cast<AMyHUD>( PController->GetHUD() );
        hud->addMessage( Message( NpcMessage, 5.f, FColor::White ) );
    }
}
The first thing we do in this function is to cast OtherActor (the thing that came near the NPC) to AAvatar. The cast succeeds (and is not nullptr) when OtherActor is an AAvatar object. We get the HUD object (which happens to be attached to the player controller) and pass a message from the NPC to the HUD. The message is displayed whenever the player is within the red bounding sphere surrounding the NPC.

Exercises

1. Add a UPROPERTY function name for the NPC’s name so that the name of the NPC is editable in blueprints, similar to the message that the NPC has for the player. Show the NPC’s name in the output.

2. Add a UPROPERTY function (type UTexture2D*) for the NPC’s face texture. Draw the NPC’s face beside its message in the output.

3. Render the player’s HP as a bar (filled rectangle).
Solutions

Add this property to the **ANPC** class:

```cpp
    // This is the NPC's name
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = NPCMessage)
    FString name;
```

Then, in **ANPC::Prox_Implementation**, change the string passed to the HUD to:

```
    name + FString(": ") + message
```

This way, the NPC's name will be attached to the message.

Add this property to the **ANPC** class:

```cpp
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = NPCMessage)
    UTexture2D* Face;
```

Then you can select face icons to be attached to the NPC's face in blueprints.

Attach a texture to your `struct Message`:

```cpp
    UTexture2D* tex;
```

To render these icons, you need to add a call to `DrawTexture()` with the right texture passed in to it:

```
    DrawTexture( messages[c].tex, x, y, messageH, messageH, 0, 0, 1, 1 );
```

Be sure to check whether the texture is valid before you render it. The icons should look similar to what is shown here, at the top of the screen:
This is how a function to draw the player’s remaining health in a bar will look:

```cpp
void AMyHUD::DrawHealthbar()
{
    // Draw the healthbar.
    AAvatar *avatar = Cast<AAvatar>(
        UGameplayStatics::GetPlayerPawn(GetWorld(), 0) );
    float barWidth=200, barHeight=50, barPad=12, barMargin=50;
    float percHp = avatar->Hp / avatar->MaxHp;
    DrawRect( FLinearColor( 0, 0, 0, 1 ), Canvas->SizeX - barWidth -
        barPad - barMargin, Canvas->SizeY - barHeight - barPad -
        barMargin, barWidth + 2*barPad, barHeight + 2*barPad );
    DrawRect( FLinearColor( 1-percHp, percHp, 0, 1 ), Canvas->SizeX
        - barWidth - barMargin, Canvas->SizeY - barHeight - barMargin,
        barWidth*percHp, barHeight );
}
```

**Summary**

In this chapter, we went through a lot of material. We showed you how to create a character and display it on the screen, control your character with axis bindings, and create and display NPCs that can post messages to the HUD.

In the upcoming chapters, we will develop our game further by adding an *Inventory System and Pickup Items* in Chapter 10, as well as the code and the concept to account for what the player is carrying. Before we do that, though, we will do an in-depth exploration of some of the UE4 container types in *Chapter 9, Templates and Commonly Used Containers*. 
Where to buy this book
You can buy Learning C++ by Creating Games with UE4 from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.