5
Upgrade Activated – Making Bounty Dash with C++

Welcome to Chapter 5! Well done on working through the introduction to C++ chapter! We are now going to utilize and build on those skills to produce your first C++ based Unreal Engine title, `Bounty Dash`! During this chapter, you will leverage your own technical knowledge and that of this book to create a 3D endless runner! This chapter has been designed so that we utilize as much C++ as possible, to create this project so that we may maximize your C++ learning. Once you have completed this chapter and the following, you will be well on your way to becoming a capable Unreal Engine 4 C++ programmer!

During this chapter, you are going to learn some core techniques that will provide you with the knowledge to write effective and efficient gameplay code. You will learn how to create a game character utilizing C++ and to access in level objects from C++. You will learn how to get components and metadata from objects using C++. You will use all of this knowledge to make a game with C++ and Unreal Engine! It should be noted that we are specifically taking a C++ oriented approach when developing this project, meaning that we may sacrifice development efficiency for the sake of learning. The next project started in *Chapter 7*, *Boss Mode Activated – Unreal Robots* will be developed with hybrid, efficient techniques in mind. In a list, we are going to be learning the following topics:

- Creating and extending the C++ basic template
- Creating character objects using C++
- Utilizing blueprint to extend C++ objects for easier asset association
- Triggering audio and animation cues in C++
- Referencing in level objects within C++ objects
- Referencing Blueprint objects in C++ with the UCLASS type

- Creating a game mode with C++
- Obtaining object metadata with C++
- Communicating with the custom C++ game mode

Creating a C++ character

Alright! Let's start the development of Bounty Dash by creating the character that we will be using for this project. Due to the nature of an endless runner, we can create a fairly simple input and movement implementation for the character. This is a great place to start with creating C++ characters, as they can quickly become complicated and involved!

Create the C++ project

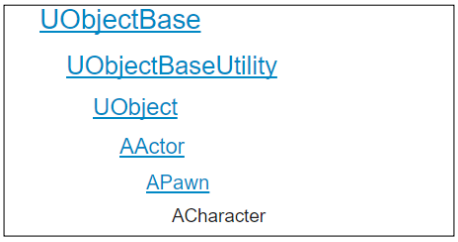
Before we start building the character, we first need to create the project. Open Unreal Engine 4.10 and select the `Basic Code` project template we used for the last chapter. This time call the project `Bounty Dash`. Upon opening the project, select **New Level...** from the **File** dropdown. From the available map options, choose **Default**. This will provide us with a nice simple base to work from. Save this map as `BountyDashMap`.

The gameplay for `BountyDash` will be quite simple, there will be a seemingly infinite progression of objects that the player is running towards. The player must dodge the objects by swapping between three lanes. `Coins` will also appear periodically; the player must collect these coins to boost their score. Once a certain number of coins have been collected, the game will speed up. If the player hits an obstacle, the player will be stalled. If the player is stalled for too long in total, he will be killed by a constantly encroaching wall of death. We will definitely be expanding on these game rules as we work through the project; but for now, this a good base to work from.

The UE4 object hierarchy

As we mentioned in the previous chapters, there is a base object hierarchy that all unreal objects will inherit from. It is important to know the various stages of this hierarchy and the feature set each subsequent base class in a hierarchy brings to the table. The first thing that you must know is that all Unreal objects will inherit from the `UObject` base class. This means that every object that interacts with the engine in some way can be processed and stored generically via the `UObject` type, affording the use of the engine's generic object management systems such as the **Unreal Garbage Collector**.

When it comes to objects that exist in a level, we have already covered a brief description of objects and actors. Before we go any further with our C++ journey, a more descriptive breakdown is required. The object hierarchy for a UE4 `ACharacter` object is as follows:



Each of the classes listed in this hierarchy add to the feature set of the object defining it as an `ACharacter`. Each of the classes featured in this hierarchy can be separated with regards to the role each class assumes. They can be broken down as follows:

- `UObjectBase`: This is the `UObject` base class from which all objects inherit. It includes generic functions that all objects will require. An example of one said function is `GetClass()`, which will return the class type of this object. This means that we are able to type check against all `UObject` classes.
- `UObjectBaseUtility`: This expands the function set that only depends on `UObjectBase`. This includes metadata functions similar to that of `GetClass()` that allows developers to get various information from their objects.
- `UObject`: This is the base class of all objects. As mentioned earlier, any unreal object will inherit from this class. Again, this class includes methods that the epic team wished all unreal objects to have.
- `AActor`: The base class for any object that can be placed or spawned in a level. `AActors` can (and will) include `Actor Components` that control how they move, render, and process. It is from this base class that we are able to make tickable actors that can be seen in levels. It is also important to note that the other main function of the `AAction` base class is the inclusion of the various networking features provided by the engine. This will be covered in *Chapter 9*, *Creating a Networked Shooter*.
- `APawn`: This is the base class for all actors that can be possessed and controlled by either player controllers or AI. `APawns` are the physical representations of players and creatures in levels. This provides the developers with a base class for all controllable objects, which is why a lot of the input functions we have been working with so far (in Blueprint and C++) will return or take an `APawn` reference.

- `ACharacter`: These are similar to pawns, but will include `Meshes`, collision, and in-built movement logic by default. They are responsible for all interactions that take place between a player or AI and the game world. `ACharacters` expand on the networking features mentioned in the `AActor` base class. We have already worked with a Blueprint abstraction of an `ACharacter` in `BarrelHopper`. As we know, `ACharacters` include a `CharacterMovementComponent` that will dictate how the character moves through the game world based on various movement modes.

If you wish to find out more about the ACharacter base class and hierarch address the following API page—`https://docs.unrealengine.com/latest/INT/API/Runtime/Engine/GameFramework/ACharacter/index.html`.

Creating the Character

Alright, it's time to create our character. Start by creating the code files for the character. Open the C++ class wizard (navigate to **File** | **New C++ Class…**). Choose **Character** for the base class and name this new class `BountyDashCharacter`. This will create a new class of type `ABountyDashCharacter` that publically inherits from `UCharacter`. This character will need to be able to navigate between three lanes of incoming objects and coins. It will need to be able to jump and run while playing the corresponding animations. Finally, it should also include a camera that we can use as the main game camera.

What we have been given

As it was with the `AHelloSphere` object we created with the class wizard, some default functionality has been provided with the `ACharacter` base class in mind. Navigate to `BountyDashCharacter.h` now, and observer the following code:

```
UCLASS()
class BOUNTYDASH_API ABountyDashCharacter: public ACharacter
{
GENERATED_BODY()

public:
// Sets default values for this character's properties
ABountyDashCharacter();
```

```
// Called when the game starts or when spawned
virtual void BeginPlay() override;

// Called every frame
virtual void Tick( float DeltaSeconds ) override;

// Called to bind functionality to input
virtual void SetupPlayerInputComponent(class UInputComponent*
                    InputComponent) override;

};
```

There is little here that is new to you, apart from the `SetupPlayerInputComponent()` function. This function is a part of the `APawn` interface (which `ACharacter` inherits from) and allows us to bind Action Axis, Action Mappings, and Input events to functions that we define in the class. As you can see, it is a virtual function that we are overriding, so we may create our own input initialization functionality. We will be doing this by calling `BindAction()`, a function accessed through the `UInputComponent*` handle that is parsed into the function; more on this later.

Now, navigate to `BountyDashCharacter.cpp`. As you can see, each of the virtual functions declared in the header file have been defined with a simple function body that includes the call to the parent function via the `Super` namespace. We will be making edits to all of these functions during the course of this chapter; but for now, let's begin by adding important members and methods (functions) to the class definition.

It is also a very good idea to quickly take a look at the definition of `ACharacter`, so you may familiarize yourself with the components and functions that are provided by the base class. The two componenets we will be interacting with are `UCapsuleComponent` and `USkeletalMeshComponent`.

What we are going to need

We are going to need to add the various member variables and functions to our `ABountyDashCharacter` base class that will make up the functionality for our main feature of the `BountyDash` project!

BountyDashCharacter's members

Navigate to `BountyDashCharacter.h` now; we are going to declare the public members that will be exposed to the engine and subsystems. Underneath the `GENERATED_BODY()` macro include the following members and public keywords:

```
public:
// Array of movement locations
UPROPERTY(EditAnywhere, Category = Logic)
TArray<class ATargetPoint*> TargetArray;

// Character lane swap speed
UPROPERTY(EditAnywhere, Category = Logic)
float CharSpeed;

// Audio component for obstacle hit sound
UPROPERTY(EditAnywhere, Category = Sound)
UAudioComponent* hitObstacleSound;

// Audio component for coin pickup sound
UPROPERTY(EditAnywhere, Category = Sound)
UAudioComponent* dingSound;
```

Firstly, we included an unreal template object `TArray`. It is a template container provided by UE4. This should be the default container to use when looking to store contiguous data that can be edited and referenced by the engine. The `TArray` object provides all of the functionality that we would expect from a template container. It has a similar feature set to that of STL Vector. In this case, `TArray` will contain `ATargetPoint` pointers. These `ATargetpoints` will be used to store the possible lane locations the player can move to during play. Again, we forward declared this type via the class keyword. It is important to note that this is possible as this Array is of `ATargetPoint` pointers. If the array contained whole `ATargetPoint` objects the compiler would throw an error on this incomplete type.

Next, we included the float property `CharSpeed`. This property will act as the interpolation speed with which the character can swap between lanes. This has been exposed via `EditAnywhere` as we will probably want to tweak the default value of this within the editor. Following this float value, we have two `UAudioComponent` handles. The `UAudioComponent` allows us to play sounds from our C++ classes. We included one for a sound to be played when the player hits an obstacle and another for when the player picks up a coin. We have also specified that we wish these to be editable anywhere, so we can assign these sounds from the editor.

Next, let's cover our protected members. This will include the component variables we will use to construct our camera system for the character. Add the following code to the class definition:

```
protected:
// Camera Boom for maintaining camera distance to the player
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera)
class USpringArmComponent* CameraBoom;

// Camera we will use as the target view
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera)
class UCameraComponent* FollowCamera;

// Player Score
UPROPERTY(BlueprintReadOnly)
int32 Score;
```

Here, we have `USpringArmComponent`. This component acts as an arm that positions the camera so that it maintains a given distance away from the character or anchor point while following some physical parameters such as drag, sway, and spring. The following `UCameraComponent` will act as the camera we use for the target view. We will be attaching the later to the former in our constructor. Both of these components have been exposed at a `BlueprintReadOnly` level, meaning blueprints will be able to call functions on these components but not replace them. Lastly, we have an `int32` that represents the score of the player. Again, we exposed this variable with `BluerpintReadOnly`, so Blueprints may read the value but not modify.

We will have two private members for this class that will only be used for internal logic. Add the following code to the class definition:

```
private:
// Data for character lane positioning
short CurrentLocation;
FVector DesiredLocation;
bool bBeingPushed;
```

The use of these members will be detailed when we begin to implement our character lane movement and collision. You will have noticed that these members have not been declared with the `UPROPERTY()` macro, that is because these members will only be used for internal logics and do not need to be engine facing.

BountyDashCharacters methods

Now that we have all of the members required for the `ABountyDashCharacter`, let's declare the methods we will be using. First up, we have the default constructor `ABountyDashCharacter()`. This has been generated by the class wizard. Again, we will use the definition of this constructor to perform a large amount of the initialization logic. Following this, we have the provided `BeginPlay()`, `Tick()`, and `SetupPlayerInputComponent()` functions. The `Tick()` function will be used to update the player's position based off the target point, and `BeginPlay()` will be used to possess control for this character and sort our array of `ATargetPoint` objects. Just below this, add the following function:

```
void ScoreUp();
```

This is a very simple function that will be called when a coin is picked up by the player. We will use it to group all of the score increment functionality. It is important to declare this method under public encapsulation, and we will be calling it form another code object. Following that, add the following code. These are the protected functions required for our character:

```
protected:
// Will handle moving the target location of the player left and right
void MoveRight();
void MoveLeft();
// Overlap functions to be used upon capsule component collision
UFUNCTION()
void myOnComponentOverlap(AActor* OtherActor, UPrimitiveComponent*
OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult&
SweepResult);

UFUNCTION()
void myOnComponentEndOverlap(AActor* OtherActor, UPrimitiveComponent*
OtherComp, int32 OtherBodyIndex);
```

The first two methods `MoveRight()` and `MoveLeft()` will be bound to an input event and used to determine which target point the player needs to seek too. We have also included `MyOnComponentOverlap()` and `MyOnComponentEndOverlap()`. These two functions will be provided to the corresponding delegates so we may customize collision reactions for this character. We have to use `OnComponentBeginOverlap` as opposed to `OnActorBeginOverlap` this time as collisions will be taking place on the `UCapsuleComponent` provided by the `ACharacter` base class. We also have the generated `SetupPlayerInputComponent` function generated by the class wizard.

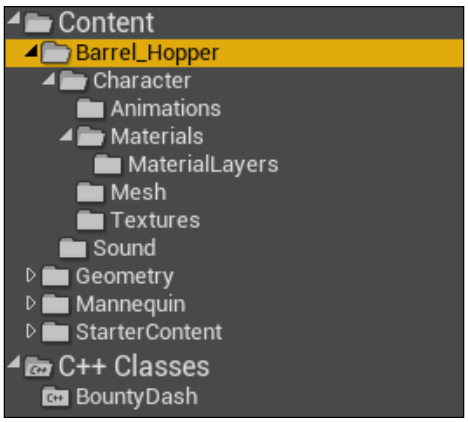
Constructing the Character

Now, we will flesh out some of the function definitions of `ABountyDashCharacter`, starting with the constructor `ABountyDashCharacter::ABountyDashCharacter()`. When adding code to constructor, it is always important to only include code that pertains to the construction of that object. The constructor will be executed when the object is first loaded into the engine. Meaning that an object that you are working within the editor has already had its constructor executed. This is what I like to think about when adding code to the constructor. Will the code I am about to write need to execute before this new actor is placed in the level and the game is played? If yes, then you are writing code in the right place.

We are going to be creating the camera components that are used for the target view, initializing some character movement properties so the character feels good while playing and providing the capsule component collision delegates with our custom functions. The first thing we need to do, however, is initialize the character with the required visual assets. Namely, a skeletal mesh and an animation Blueprint.

Borrowing from the old to make the new

For the purposes of this chapter, we can happily reuse the Skeletal mesh and Animation Blueprint we created for `BarrelHopper`! If you re-open the `BarrleHopper` project and navigate to the location of the `BH_Character_AnimBP` asset. Right-click on the `AnimBP` and select **AssetActions** | **Migrate**. You will see a list of all of the assets that will need to be migrated with `AnimBP` for the migration to be successful. Thankfully, this means that with one migration we can provide the `BountyDash` project with all of the assets, we need for a skeletal character mesh! Choose the **Content** folder of the `BountyDash` project for the destination of the migration. Once the migration is complete, you will see a new `Barrel_Hopper` folder in the content browser of the `BountyDash` project as follows:



This will contain the `BH_Character_AnimBP`, the `SK_Mannequin`, and the `1D_Idle_Run_BS`, we made during the development of `BarrelHopper`.

Assigning Blueprints in code with generated classes

With the appropriate assets migrated, it is time to make changes to the function definitions in `BountyDashCharacter.cpp`. Before we begin to ensure that the included list for the `BountyDashCharacter.cpp` matches this:

```
#include "BountyDash.h"
#include "Animation/AnimInstance.h"
#include "Animation/AnimBlueprint.h"
#include "Engine/TargetPoint.h"
#include "BountyDashGameMode.h"
#include "BountyDashCharacter.h"
```

With those in place, you will have access to everything you need within the scope of this chapter. Now, let's begin to construct `ABountyDashCharacter`. Add the following lines to the `ABountyDashCharacter` constructor:

```
PrimaryActorTick.bCanEverTick = true;

// Set size for collision capsule
GetCapsuleComponent()->InitCapsuleSize(42.f, 96.0f);

ConstructorHelpers::FObjectFinder<UAnimBlueprint> myAnimBP(TEXT("/
Game/Barrel_Hopper/Character/BH_Character_AnimBP.BH_Character_
AnimBP"));

ConstructorHelpers::FObjectFinder<USkeletalMesh> myMesh(TEXT("/Game/
Barrel_Hopper/Character/Mesh/SK_Mannequin.SK_Mannequin"));

if (myMesh.Succeeded() && myAnimBP.Succeeded())
{
GetMesh()->SetSkeletalMesh(myMesh.Object);
GetMesh()->SetAnimInstanceClass(myAnimBP.Object->
GeneratedClass);
}
```

The preceding lines simply leverage the `FObjectFinder` template object to find and assign the skeletal mesh asset that the character will be using along with the accompanying `AnimInstance` asset. An `AnimInstance` is the base class from which all animation blueprints derive from; therefore, we are able to assign the Animation Blueprint we made earlier through the `SetAnimInstanceClass()` function. We are able to make the association between the base class and our abstracted blueprint object using the `GeneratedClass` member of the `UAnimBlueprint` object we attained through `FObjectFinder`. This `GeneratedClass` member holds the resultant C++ class that is generated upon the blueprints compilation. This means that we can associate the Animation Blueprint made in engine with the `ABountyDashCharacter` C++ object! All Blueprints feature this `GeneratedClass` member and is the main way to spawn, create, and assign blueprints using C++ code. Ensure that the paths that you provide to the `FObjectFinder` constructor match that of the paths to your assets within the project **Content** Browser.

Setting up the components

The next step to constructing the character is setting up all of the component properties so the character behaves as expected during gameplay. Start by adjusting some of the mesh and character movement properties:

```
// Rotate and position the mesh so it sits in the capsule component
properly
GetMesh()->SetRelativeLocation(FVector(0.0f, 0.0f,
-GetCapsuleComponent()->GetUnscaledCapsuleHalfHeight()));

GetMesh()->SetRelativeRotation(FRotator(0.0f, -90.0f, 0.0f));

// Configure character movement
GetCharacterMovement()->JumpZVelocity = 1450.0f;
GetCharacterMovement()->GravityScale = 4.5f;
```

We are setting the location of the mesh, so it is half way down the capsule via the `GetUnscaledCapsuleHalfHeight()` function, which will return half of the height of the capsule as it exists in component space. Note that we are using `SetRelativeLocaiton()` as we wish to make these adjustments in component space based off of the parent components transform. The next function simply rotates the character mesh by `-90.0` around the z-axis (yaw). The next thing we are doing is changing the jump velocity and gravity scale of the character movement component. We do this so that we may have a quicker jump on the character, with an increased jump velocity and gravity scale the character will jump quickly and fall quickly.

Following this, we are going to set up our cameras. First, we should create a camera boom so that the camera sits about `500` units away from the character and `160` units up. We can do this with the following code:

```
// Create a camera boom (pulls in towards the player if there is a
collision)
CameraBoom = CreateDefaultSubobject<USpringArmComponent>(TEXT("Camera
Boom"));
check(CameraBoom);

CameraBoom->AttachTo(RootComponent);

// The camera follows at this distance behind the character
CameraBoom->TargetArmLength = 500.0f;

// Offset to player
CameraBoom->AddRelativeLocation(FVector(0.0f, 0.0f, 160.0f));
```

After creating the component, you will notice that we have used the `check` macro. This macro is simply an assert. It will check that the provided condition resolves to true; if not, it will throw an assert at runtime. We then attach the `CameraBoom` to the `RootComponent` (which by default is the capsule component). We then set the length of the arm to be 500 cms long, and then position the arm 160 cm along the z-axis with `AddRelativeLocation()`. Next, we must create the camera itself and attach the camera to `USpringArmComponent`.

Every spring arm features a socket at the end of the arm. A socket is a transform that is updated with the movement of the spring arm component. A socket will maintain its relative location to whatever it is owned by at all times. With this in mind, we can attach the camera we create to this socket, and the camera will be positioned appropriately as the character moves through the world and the spring arm component updates with regards to those previously mentioned physical factors (sway, drag, and spring). We can do this with the following code:

```
// Create a follow camera
FollowCamera = CreateDefaultSubobject<UCameraComponent>(TEXT("FollowC
amera"));
check(FollowCamera);

// Attach the camera to the end of the boom and let the boom adjust to
match the controller orientation
FollowCamera->AttachTo(CameraBoom, USpringArmComponent::SocketName);

// Rotational change to make the camera look down slightly
FollowCamera->AddRelativeRotation(FQuat(FRotator(-10.0f, 0.0f,
0.0f)));
```

We attached `UCameraComponent` to `USpringArmComponet` by the `AttachTo()` function by passing a handle to the spring arm component (`CameraBoom`), and the name of the socket we would like to attach to. We can use the static member `SocketName` of the class `USpringArmComponent` as all `USpringArmComponents` have the same name for the boom socket. We then pitch the camera slightly (-10.0 degrees) so that it will look down at the character during play. Following this, we need to provide a default value for the character speed and provide the overlap functions to the `UCapsupleComponents` `OnComponentBeginOverlap` and `OnComponentEndOverlap` delegates. The following code will do that:

```
// Game Properties
CharSpeed = 10.0f;

GetCapsuleComponent()->OnComponentBeginOverlap.AddDynamic(this, &ABoun
tyDashCharacter::MyOnComponentOverlap);
GetCapsuleComponent()->OnComponentEndOverlap.AddDynamic(this, &ABounty
DashCharacter::MyOnComponentEndOverlap);
```

We are setting the `CharSpeed` value to `10.0f` as that will be the speed at which the character will interpolate between lane positions. The value is exposed with `EditAnywhere`, so if we feel it is to slow once we are testing in an editor we can change it via **Details** panel. After this, you can see that we are adding our custom collision functions to the `OnComponentBegin/EndOverlap` delegate included in the component. This is very similar to what we did in the `HelloSphere` example; yet this time, we are adding the collision methods to a delegate object in one of the owned components not the `AActor` itself.

Assuming default control

The last thing we need to do is ensure that the `ABountyDashCharacter` we place in the level is possessed by the Player Controller as soon as the game starts. Usually, this would happen automatically when we specify the default pawn in the game mode. However, we may need to adjust some of the exposed members of `ABountyDashCharacter` from the editor. We, therefore, cannot have the game mode spawn a new `ABountyDashCharacter` that is lacking these modified values. This means that we have to assume control of an already existing in level character. Add the following line to the constructor definition:

```
// Poses the input at ID 0 (the default controller)
AutoPossessPlayer = EAutoReceiveInput::Player0;
```

The preceding lines set the `AutoPossessPlayer` member of the `APawn` base class to the `enum` value `Player0`. This will ensure that when the game begins play, the default controller will possess the first `ABountyDashCharacter` we already have in the level. This affords us the ability to modify the members of our character that have been exposed with the `EditAnywhere` specifier from the editor and ensure the character we control has these properties. This is an example of something we would not have to worry about if we were going to be creating a blueprint abstraction of our player as we could simply modify the values in the blueprint object, and then assign that blueprint object type to be the default pawn.

Writing the begin Play function

Now that we have the appropriate construction functionality in place, we can write what our character is going to do at the beginning of play! We need our character to know about all of `ATargetPoints` that exist in the current game world and then add those objects to the `TargetArray` member. To do this, we need to access `Game World`!

Getting In-Editor objects using the Game World

One of the most important features of the UE4 programming tool set is the functionality obtained through the `GetWorld()` function. This function returns a handle to the game world that the calling class currently exists in. The handle is of type `UWorld*` and through this, we can create and destroy objects, get handles to objects already existing in the level, and obtain handles to Player Controllers. We are going to be coupling this `UWorld` handle with another UE4 template object—`TActorIterator`. This object, when specified with a type, will return an iterator that you can increment to provide access to all actors of that type within a game world. You must initialize `TActorIterator` with a game world context. Add the following code to `ABountyDashCharacter::BeginPlay()` underneath `Super::BeginPlay()`:

```
for (TActorIterator<ATargetPoint> TargetIter(GetWorld());
     TargetIter; ++TargetIter)
{
TargetArray.Add(*TargetIter);
}
```

We are creating a for loop with the initialization section used to create a `TActorIterator` object of type `ATargetPoint` called `TargetIter`. We also pass a reference to the world we wish to scrutinize to the object constructor via the `GetWorld()` function, which is part of the `AActor` interface. The condition of the `for` loop checks that the iterator is still valid, and each iteration of the loop we increment the iterator with `++TargetIter`. The resultant loop will create the iterator upon entry, increment the iterator each time the loop iterates until it is invalid. This will iterate over every `ATargetPoint` actor that exists in the game world.

For each `ATargetPoint` we encounter, we add it to the `TargetArray` via the `Add()` function. We need to dereference the iterator when passing the `ATargetPoint` into `Add()` to access the contained `ATargetPoint*` handle.

Sorting TArrays with Lambdas

The next thing we need to do in `BeginPlay()` is ensure that our `TArray` of `ATargetPoints` is sorted properly. We do not know in which order `ATargetPoints` have been found by `TActorIterator` we implemented earlier. We require these points to be sorted in the Array so that we may traverse them in order from furthest left to furthest right when moving the character.

We can do this via a function object or lambda object and pass this as a sorting predicate to `TargetArray`. If that is unclear to you, a function object is simply an object that is only responsible for one function and has no members, a lambda is the new C++11 syntax that allows us to write these on the fly. A predicate is simply a check that is used with sorting algorithms to determine the need for a sort. Add the following code to the `BeginPlay` function now:

```
auto SortPred = [](const AActor& A, const AActor& B)->bool
{
    return(A.GetActorLocation().Y < B.GetActorLocation().Y);
};

TargetArray.Sort(SortPred);
```

We are using a lambda (denoted by `[]`) to create a function that takes in two `const AActor` references and returns a `bool`. Within this lambda, we are simply checking that the Y value of the first `AActor` is less than the second. We then save this Lambda into a `SortPred` handle that is of type `auto`. This is another C++ 11 keyword that allows for the compile time type checking of the right-hand argument (right-hand side of the assignment operator), `auto` then automatically changes to that type during the assignment.

This predicate will be used when sorting the array where actor `A` will be one element of the `TArray` and actor `B` will be the very next. If the statement returns false, it means `B` needs to shift onto the other side of `A`; thus, a sort is needed. We then pass this predicate as a parameter to `Sort()`. We are able to use a predicate expecting `AActor` references here as `ATargetPoint` inherits from `AActor`.

The last thing we need to do is ensure that the character starts the game in the middle most lane. We can do this with some simple math. Add the following code underneath `TargertArray.Sort(SortPred)`:

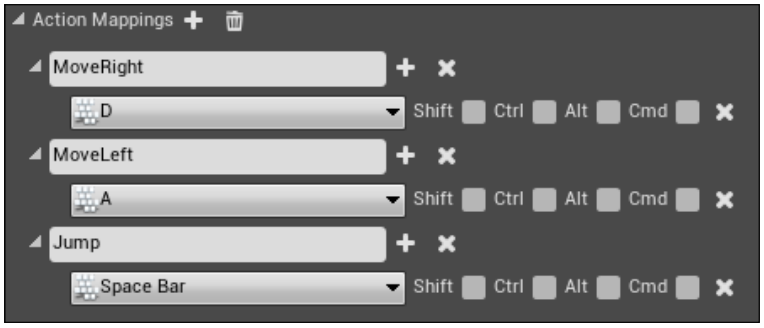
```
CurrentLocation = ((TargetArray.Num() / 2) + (TargetArray.Num() % 2) -
1);
```

This will get the number of elements in the target array, divide that by two, and then add the result of the number of elements modulus `2`. Meaning the remainder of that number is divided by `2`. We then minus the result of this calculation by `1` to ensure that we shift the result to incorporate index `0`. Because we are dealing with integer values truncation of decimal fractions is intended and means the logic is sound.

We are interacting with the member `CurrentLocation` that we declared in the `ABountyDashCharacter` class definition earlier. The purpose of this member is to indicate at which `ATargetPoint` in `TargetArray` the character is currently situated. As it is of type integer, this means that we are able to use it for `TArray` lookups via the subscript operator (`[int]`).

Setting up inputs in C++

Before we write our tick functionality, let's first look at binding inputs to the character. The first thing we need to do is create axis and action mappings in the editor like we always have. Navigate to **Edit** | **Project Settings** in the level editor and then navigate to the **Engine Input** section and add three action mappings **MoveRight**, **MoveLeft**, and **Jump**. Bind them to *D*, *A*, and *space bar*, respectively. Your input map should look as follows:



Binding actions

With the action mappings established, lets now bind these mappings to functions in the `ABountyDashCharacter` object. The `SetupPlayerInputComponent()` function provided by the `APawn` interface (which we have access to via inheritance) will be called by the engine to request bindings. A `UInputComponent` is parsed into the function, and it is with this component that we will bind the functionality. This is done in a very similar manner to binding functions to `OnOverlap` delegates. The following code shows you how to bind the `MoveRight` and `MoveLeft` functions we declared in the `ABountyDashCharacter` class definition. Add this code to `ABountyDashCharacter::SetupPlayerInputComponent()` underneath the `Super` call now:

```
// Set up gameplay key bindings
check(InputComponent);
InputComponent->BindAction("Jump", IE_Pressed, this,
&ACharacter::Jump);

InputComponent->BindAction("Jump", IE_Released, this,
&ACharacter::StopJumping);

InputComponent->BindAction("MoveRight", IE_Pressed, this, &ABountyDash
Character::MoveRight);

InputComponent->BindAction("MoveLeft", IE_Pressed, this, &ABountyDashC
haracter::MoveLeft);
```

Again, we use `check(InputComponent)` to validate the parsed input component. Following that, we bind the `Jump` action when pressed to the `Jump` function found in the `ACharacter` parent class. We do this by passing the name of the action we wish to bind, the state of the action (`IE_PRESSED` or `IE_RELEASED`), the object that owns the function to bind, and a pointer to the function itself to `BindAction()`. As you can see we do this for the `MoveRight` and `MoveLeft` action mappings as well. You should also note that we bind the release event of `Jump` to the function `StopJumping()` found in the `AChracter` base class. This calls into the character movement component and stops any further lift being applied to the character. This allows for things such as short hops and quick jumps.

Next, we need to definite the functions we just bound with meaningful functionality. Add the following code to `BountyDashCharacter.cpp`:

```
void ABountyDashCharacter::MoveRight()
{
if ((Controller != nullptr))
{
    if (CurrentLocation < TargetArray.Num() - 1)
    {
```

```
            ++CurrentLocation;
        }
        else
        {
            // Do Nothing
        }
    }
    }

    void ABountyDashCharacter::MoveLeft()
    {
    if ((Controller != nullptr))
    {
        if (CurrentLocation > 0)
        {
        --CurrentLocation;
        }
        else
        {
              // Do Nothing
        }
    }
    }
```

These two functions are quite simple. We first check that this character is being controlled by something by checking the state of the `Controller` handle. This will be populated with a valid object if something is currently in possession of the character. As we possessed the character with the first player controller at begin play, this should never fail. We then check if the `CurrentLocation` member is within the maxims of the `TargetArray` bounds (either the number of elements in the array less one or greater than zero), and then either increment the current location if we want to move right or decrement the location if we want to move left. These two functions in combination will allow us to traverse the lanes that we set up. We also included checks to see if the player has reached the furthest right or furthest left target point. If so, and the player is still trying to move in that direction, we do nothing.

How our Character is going to Tick

Now, with everything established for our character, we can write the tick function that we are going to use for the character. The main thing `Tick()` will be responsible for is maintaining the characters position with regards to the current `ATargetPoint` we wish the character to occupy. We can do this using the `CurrentLocation` value to look up in `TargetArray`, which point is our target. We can then linearly interpolate or lerp between the character's current position and that of the target location. This will ensure that the character always maintains the correct place when traversing lanes.

Add the following code to `ABountyDashCharacter::Tick()`:

```
void ABountyDashCharacter::Tick(float DeltaSeconds)
{
Super::Tick(DeltaSeconds);

if(TargetArray.Num() > 0)
{
FVector targetLoc = TargetArray[CurrentLocation]-
    >GetActorLocation();
    targetLoc.Z = GetActorLocation().Z;
targetLoc.X = GetActorLocation().X;

if(targetLoc != GetActorLocation())
    {
SetActorLocation(FMath::Lerp(GetActorLocation(), targetLoc, CharSpeed
* DeltaSeconds));
    }
}

}
```

We called the parent tick function via `Super::Tick()`. We then check that our `TargetArray` has been populated with at least one target point. If so we get the location the of the current point. We merge this location with the *X* and *Z* value of the character's location, so we can allow jumping and pushing movement without affecting the `lerp`. We then set the actor location to the output form a `Lerp` function. This function takes in a current location, the location we wish to move too and a delta. This is how much should be interpolated this frame. For us, that is the delta time * the character's interpolation speed (`10.0f` by default).

Compile time

With all of that in place, it is nearly time to compile! First, we need to provide empty function definitions for the component overlap functions and `ScoreUp()`. We will be populating these function definitions with code soon! The following will suffice for now:

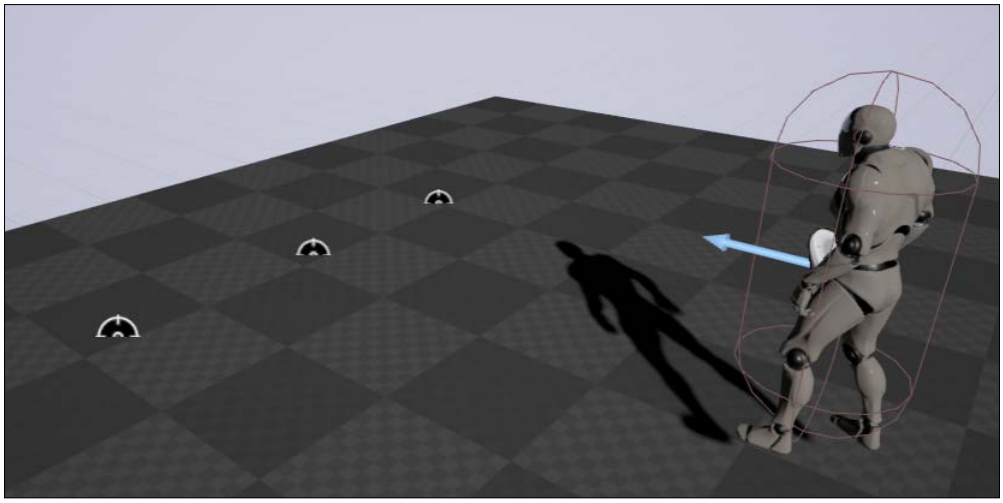
```
void ABountyDashCharacter::myOnComponentOverlap(AActor* OtherActor,
UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep,
const FHitResult& SweepResult)
{
}
```

```
void ABountyDashCharacter::myOnComponentEndOverlap(AActor* OtherActor,
UPrimitiveComponent* OtherComp, int32 OtherBodyIndex)
{
}

void ABountyDashCharacter::ScoreUp()
{
}
```

Ok, now we can compile the code!

Creating the C++ world objects

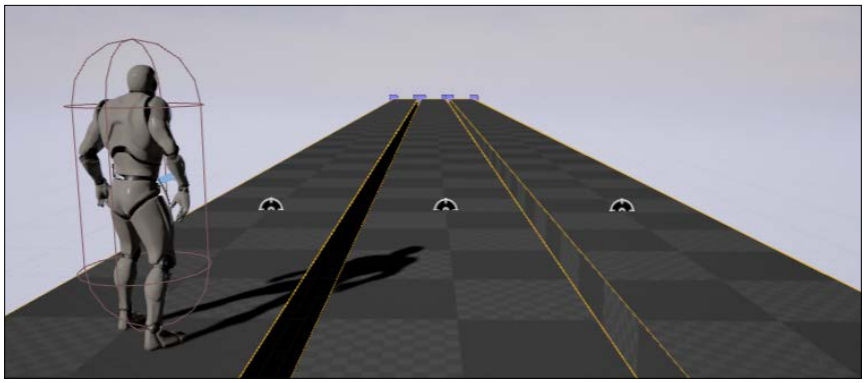
With the character constructed, we can now start to build the level. We are going to be creating a block out for the lanes we will be using for the level. We can then use this block out to construct a mesh we can reference in code. Before we get into the level creation, we should ensure the functionality we implemented for the character works as intended. With the `BountyDashMap` open, navigate to the `C++ classes` folder of the content browser. Here, you will be able to see the `BountyDashCharacter`! Drag and drop the character into the game level onto the platform. Then, search for `TargetPoint` in the **Modes** panel. Drag and drop three of these target points into the game level, and you should be presented with the following:



Now, press the **Play** button to enter the PIE mode (Play In Editor). The character should be automatically possessed and used for input! Also, ensure that when you press *A* or *D* the character moves to the next available target point!

Now that we have the base of the character implemented, we should start to build the level. We require three lanes for the player to run down and obstacles for the player to dodge. For now, we should focus on the lanes the player will be running on. Let's start by blocking out how the lanes will appear in the level. Drag a BSP box brush into the game world. You can find the Box brush in **Modes** panel under the **BSP** section under the name **Box**. Place the box at world location (0.0f, 0.0f, and -100.0f). This will place the box in the center of the world. Now, change the **X** property of the box under the **Brush settings** section of the **Details** panel to 10,000.

We require this lane to be long, so that later on we can hide the end using fog without obscuring the objects the player will need to dodge. Next, we need to click on and drag two more copies of this box. You can do this by holding *ALT* while moving an object via the transform widget. Position one box copy at world location (0.0f, -230.0f, and -100) and the next at (0.0f, 230, and -100). The last thing we need to do to finish blocking the level is place the **Target** points in the center of each lane. You should be presented with this when you are done!



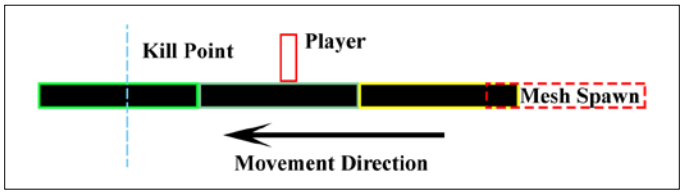
Converting BSP brushes to a static mesh

The next thing we need to do is convert the lane brushes we made into one mesh so that we can reference it within our code base. Select all of the boxes in the scene. You can do this by holding *CTRL* while selecting the box brushes in the editor. With all of the brushes selected, address the **Details** panel. Ensure that the transform of your selection is positioned in the middle of the three brushes. If it is not, you can either reselect the brushes in a different order or group the brushes by pressing *CTRL-G* while the boxes are selected. This is important as the position of the transform widget shows what the origin of the generated mesh will be. With the grouping or boxes selected address the **Brush Settings** section in the **Details** panel, there is a small white expansion arrow at the bottom of the section, click on this now. You will then be presented with a **create static mesh** button, press this now. Name this mesh `Floor_Mesh_BountyDash` and save it under `Geometry/Meshes/` of the content folder.

Smoke and mirrors with C++ objects

We are going to be creating the illusion of movement within our level. You may have noticed that we have not included any facilities in our character to move forward in the game world. That is because our character will never advance past his X positon at 0. Instead, we are going to be moving the world toward and past him. This way we can create a very easy spawning and processing logic for the obstacles and game world without having to worry about continuously spawning objects the player can move past further and further down the x-axis.

We require some of the level assets to move through the world so we can establish the illusion of movement for the character. One of these moving objects will be the floor. This requires some logic that will reposition floor meshes as they reach a certain depth behind the character. We will be creating a swap chain of sorts that will work with three meshes. The meshes will be positioned in a contiguous line. As the meshes move underneath and behind the player, we need to move any mesh that is far enough behind the player's back, to the front of the swap chain. The effect is a never ending chain of floor meshes constantly flowing underneath the player. The following diagram may help to understand the concept:



Obstacles and coin pickups will follow a similar logic. However, they will simply be destroyed upon reaching the Kill point in the preceding diagram.

Modifying the BountyDashGameMode

Before we start to create code classes that will feature in our world, we are going to modify `BountyDashGameMode` that was generated when the project was created. The game mode is going to responsible for all of the game state variables and rules. Later on, we are going to be using the game mode to determine how the player respawns and when the game is lost.

BountyDashGameMode class definition

The game mode is going to be fairly simple. We are going to a add a few member variables that will hold the current state of the game such as game speed, game level, and the number of coins needed to increase the game speed. Navigate to `BountyDashGameMode.h` and add the following code:

```
UCLASS(minimalapi)
class ABountyDashGameMode : public AGameMode
{
GENERATED_BODY()

UPROPERTY()
float gameSpeed;

UPROPERTY()
int32 gameLevel;
```

As you can see, we have two private member variables `gameSpeed` and `gameLevel`. These are private as we wish no other object to be able to modify the contents of these values. You will also note that the class has been specified with `minimalapi`. This specifier effectively informs the engine that other code modules will not need information from this object outside of the class type. This means that you will be able to cast to this class type but functions cannot be called within other modules. This is specified as a way to optimize compile times as no module outside of this project API will require interactions with our game mode.

Next, we declare the public functions and members we will be using within our game mode. Add the following code to the `ABountyDashGameMode` class definition:

```
public:
ABountyDashGameMode();

    void CharScoreUp(unsigned int charScore);

    UFUNCTION()
    float GetInvGameSpeed();

    UFUNCTION()
    float GetGameSpeed();

    UFUNCTION()
    int32 GetGameLevel();
```

The `CharScroreUp()` function takes in the player's current score (held by the player) and changes game values based on that score. This means we are able to make the game more difficult, as the player scores more points. The next three functions are simply the accessor methods we can use to get the private data of this class in other objects.

Next, we need to declare our protected members we exposed to be `EditAnywhere`, so we may adjust these from the editor for testing purposes.

```
protected:

UPROPERTY(EditAnywhere, BlueprintReadOnly)
int32 numCoinsForSpeedIncrease;

UPROPERTY(EditAnywhere, BlueprintReadWrite)
float gameSpeedIncrease;

};
```

The `numCoinsForSpeedIncrease` variable will determine how many coins it takes to increase the speed of the game and the `gameSpeedIncrease` value will determine how much faster the objects move when the `numCoinsForSpeedIncrease` value has been met.

BountyDashGameMode function definitions

Let's begin add some definitions to the `BountyDashGameMode` functions. They will be very simple at this point. Let's start by providing some default values for our member variables within the constructor and by assigning the class to be used for our default pawn. Add the definition for the `ABountyDashGameMode` constructor:

```
ABountyDashGameMode::ABountyDashGameMode()
{
    // set default pawn class to our ABountyDashCharacter
    DefaultPawnClass = ABountyDashCharacter::StaticClass();

    numCoinsForSpeedIncrease = 5;
    gameSpeed = 10.0f;
    gameSpeedIncrease = 5.0f;
    gameLevel = 1;
}
```

Here, we are setting the default pawn class by calling `StaticClass()` on `ABountyDashCharacter`. As we have just referenced the `ABountyDashCharacter` type, ensure that `#include "BountyDashCharacter.h"` is add to the `BountyDashGameMode.cpp` include list. The `StaticClass()` function is provided by default for all objects and returns the class type information of the object as a `UClass*`. We then establish some default values for member variables. The player will have to pick up five coins to increase level, the game speed is set to 10.0f (10m/s) and the game will speed up by 5.0f (5m/s) every time the coin quota is reached. Next, let's add a definition for the `CharScoreUp()` function:

```
void
ABountyDashGameMode::CharScoreUp(unsigned int charScore)
{
    if (charScore != 0 &&
       charScore % numCoinsForSpeedIncrease == 0)
    {
        gameSpeed += gameSpeedIncrease;
        gameLevel++;
    }
}
```

This function is quite self-explanatory. The character's current score is passed into the function. We then check that the character's score is not currently 0 and that if the remainder of our character score is 0 after being divided by the number of coins needed for a speed increase, that is, if it divided equally thus the quota has been reached. We then increase the game speed by the `gameSpeedIncrease` value and then increment the level.

The last thing we need to add is the accessor methods described earlier. They do not require too much explanation short of the `GetInvGameSpeed()` function. This function will be used by objects that wish to be pushed down the x-axis at the game speed:

```
float
ABountyDashGameMode::GetInvGameSpeed()
{
    return -gameSpeed;
}

float
ABountyDashGameMode::GetGameSpeed()
{
    return gameSpeed;
}
```

```
int32 ABountyDashGameMode::GetGameLevel()
{
    return gameLevel;
}
```

Getting our game mode via Template functions

The `ABountyDashGame` mode now contains information and functionality that will be required by most of the `BountyDash` objects we create going forward. We need to create a light-weight method to retrieve our custom game mode ensuring that the type of information is preserved. We can do this by creating a template function that will take in a world context and return the correct game mode handle. Traditionally, we could just use a direct cast to `ABountyDashGameMode`; however, this would require including `BountyDashGameMode.h` in `BountyDash.h`. As not all of our objects will require knowledge of the game mode, this is wasteful. Navigate to the `BoutyDash.h` file now. You will be presented with the following:

```
#pragma once

#include "Engine.h"
```

What currently exists in the file is very simple, `#pragma once` has again been used to ensure the compiler only builds and includes the file once. Then `Engine.h` has been included, so every other object in `BOUNTYDASH_API` (they include `BountyDash.h` by default) has access to the functions within `Engine.h`. This is a good place to include utility functions you wish all objects to have access to. In this file, include the following lines of code:

```
template<typename T>
T* GetCustomGameMode(UWorld* worldContext)
{
    return Cast<T>(worldContext->GetAuthGameMode());
}
```

This code, simply put, is a template function that takes in a game world handle. Get the game mode from this context via the `GetAuthGameMode()` function, then cast this game mode to the template type provided to the function. We must cast to the template type as the `GetAuthGameMode()` simply returns a `AGameMode` handle. Now, with that in place, let's begin coding our never ending floor!

Coding the floor

The construction of the floor will be quite simple in essence, as we only need a few variables and a tick function to achieve the functionality we need. Use the class wizard to create a class named **Floor** that inherits from `AActor`. We will start by modifying the class definition found in `Floor.h` navigate to this file now.

Floor class definition

The class definition for the floor is very basic. All we need is a `Tick()` function and some accessor methods, so we may provide some information about the floor to other objects. I have also removed the `BeginPlay` function provided by default by the class wizard as it is not needed. The following is what you will need to write for the `AFloor` class definition in its entirety. Replace what is present in `Floor.h` with this now (keeping the #include list intact):

```
UCLASS()
class BOUNTYDASH_API AFloor : public AActor
{
GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AFloor();

    // Called every frame
    virtual void Tick( float DeltaSeconds ) override;

    float GetKillPoint();
    float GetSpawnPoint();

protected:
    UPROPERTY(EditAnywhere)
    TArray<USceneComponent*> FloorMeshScenes;

    UPROPERTY(EditAnywhere)
    TArray<UStaticMeshComponent*> FloorMeshes;

    UPROPERTY(EditAnywhere)
    UBoxComponent* CollisionBox;

    int32 NumRepeatingMesh;
    float KillPoint;
    float SpawnPoint;
};
```

We have three `UPROPERTY` declared members. The first two being `TArrays` that will hold handles to the `USceneComponent` and `UMeshComponent` objects that will make up the floor. We require the `TArray` of scene components, as the `USceneComponent` objects provide us with a world transform that we can apply translations to so that we may update the position of the generated floor mesh pieces. The last `UPROPERTY` is a collision box that will be used for the actual player collisions to prevent the player from falling through the moving floor. The reason we are using a `BoxComponent` instead of the meshes for collision is that we do not want the player to translate with the moving meshes. Due to surface friction simulation, having the character collide with any of the moving meshes will cause the player to move with the mesh.

The last three members are protected and do not require any `UPROPRTY` specification. We are simply going to use the two float values, `KillPoint` and `SpawnPoint`, to save output calculations from the constructor so we may use them in the `Tick()` function. The integer value `NumRepeatingMesh` will be used to determine how many meshes we will have in the chain.

Floor function definitions

As always, we will start with the constructor of the floor. It is here that we will be performing the bulk of our calculations for this object. We will be creating `USceneComponents` and `UMeshComponents` that we are going to use to make up our moving floor. With dynamic programming in mind, we should establish the construction algorithm so that we can create any number of meshes in the moving line. Also as we will be getting the speed of the floors movement form the game mode, ensure that `#include "BountyDashGameMode.h"` is included in `Floor.cpp`.

AFloor::AFloor() constructor

Start by adding the following lines to the `AFloor` constructor `AFloor::AFloor()` found in `Floor.cpp`:

```
RootComponent =CreateDefaultSubobject<USceneComponent>(TEXT("Root"));

ConstructorHelpers::FObjectFinder<UStaticMesh>myMesh(TEXT(
"/Game/Barrel_Hopper/Geometry/Floor_Mesh_BountyDash.Floor_Mesh_
BountyDash"));

ConstructorHelpers::FObjectFinder<UMaterial>myMaterial(TEXT(
"/Game/StarterContent/Materials/M_Concrete_Tiles.M_Concrete_Tiles"));
```

To start with, we are simply using `FObjectFinders` to find the assets we require for the mesh. For the `myMesh` finder, ensure you parse the reference location of the static floor mesh we created earlier. We also create a scene component to be used as the root component for the floor object. Next, we are going to be checking the success of the mesh acquisition, and then establishing some variables for the mesh placement logic:

```
if (myMesh.Succeeded())
{
    NumRepeatingMesh = 3;

    FBoxSphereBounds myBounds = myMesh.Object->GetBounds();
    float XBounds = myBounds.BoxExtent.X * 2;
    float ScenePos = ((XBounds * (NumRepeatingMesh - 1)) / 2.0f) * -1;

    KillPoint = ScenePos - (XBounds * 0.5f);
    SpawnPoint = (ScenePos * -1) + (XBounds * 0.5f);
```

Note that we have just opened an `if` statement without closing the scope; from time to time, I will split segments of code within a scope across multiple pages. If you are ever lost as to the current scope, we are working from look for this comment `; <-- Closing If(MyMesh.Succed())` or in the future, a similarly named comment.

Firstly, we are initializing the `NumRepeatingMesh` value with three. We are using a variable here instead of a hard coded value so that we may update the number of meshes in the chain without having to refactor the remaining code base.

We then get the bounds of the mesh object using the `GetBounds()` function on the mesh asset we just retrieved. This returns a `FBoxSphereBounds` struct, which will provide you with all of the bounding information of a static mesh asset. We then use the `X` component of the member `BoxExtent` to initialize Xbounds. BoxExtent is a vector that holds the extent of the bounding box of this mesh. We save the X component of this vector, so we can use it for mesh chain placement logic. We have doubled this value as the `BoxExtent` vector that represents the extent of the box from origin to one corner of the mesh. Meaning if we wish for the total bounds of the mesh, we must double any of the `BoxExtent` components.

Next, we calculate the initial scene position of the first `USceneCompoennt` we will be attaching a mesh to and storing in the `ScenePos` array. We can determine this position by getting the total length of all of the meshes in the chain `(XBounds * (numRepeatingMesh – 1)`, then halve the resulting value so we can get the distance the first `SceneComponent` will be from the origin along the x-axis. We also multiply this value by `-1` to make it negative as we wish to start our mesh chain behind the character (at X position 0).

We then use `ScenePos` to specify our `killPoint`, which represents the point in space at which floor mesh pieces should get to before swapping back to the start of the chain. For the purposes the swap chain, whenever a scene component is half a mesh piece length behind the position of the first scene component in the chain, it should be moved to the other side of the chain. With all of our variables in place, we can now iterate through the number of meshes we desire (3) and create the appropriate components. Add the following code to the scope of the if statement we just opened:

```
for (int i = 0; i < NumRepeatingMesh; ++i)
{
// Initialize Scene
FString SceneName = "Scene" + FString::FromInt(i);
FName SceneID = FName(*SceneName);
USceneComponent* thisScene = CreateDefaultSubobject<USceneComponent>(
SceneID);
check(thisScene);

thisScene->AttachTo(RootComponent);
thisScene->SetRelativeLocation(FVector(ScenePos, 0.0f, 0.0f));
ScenePos += XBounds;

floorMeshScenes.Add(thisScene);
```

Firstly, we are creating a name for the scene component by appending `Scene` with the iteration value we are up too. We then convert this appended `FString` to an `FName`; then provide this to the `CreateDefaultSubobject` template function. With the resultant `USceneComponent` handle, we call `AttachTo()` to bind it to the root component. Then, we set the `RelativeLocation` of `USceneComponent`. Here, we are parsing in the `ScenePos` value we calculated earlier as the `X` component of the new relative location. The relative location of this component will always be based off the position of the root `SceneComponent` we created earlier.

With `USceneCompoennt` appropriately placed, we then increment the `ScenePos` value by that of the `XBounds` value. This will ensure that subsequent `USceneComponents` created in this loop will be placed in an entire mesh length away from the previous, forming a contiguous chain of meshes attached to scene components. Lastly, we add this new `USceneComponent` to `floorMeshScenes`, so we may later perform translations on the components. Next, we will construct the mesh components by adding the following code to the loop:

```
// Initialize Mesh
FString MeshName = "Mesh" + FString::FromInt(i);
UStaticMeshComponent* thisMesh = CreateDefaultSubobject<UStaticMeshCom
ponent>(FName(*MeshName));
check(thisMesh);
```

```
thisMesh->AttachTo(FloorMeshScenes[i]);
thisMesh->SetRelativeLocation(FVector(0.0f, 0.0f, 0.0f));
thisMesh->SetCollisionProfileName(TEXT("OverlapAllDynamic"));

if (myMaterial.Succeeded())
{
   thisMesh->SetStaticMesh(myMesh.Object);
   thisMesh->SetMaterial(0, myMaterial.Object);
}

FloorMeshes.Add(thisMesh);
} // <--Closing For(int i = 0; i < numReapeatingMesh; ++i)
```

As you can see, we performed a similar name creation process for `UMeshComponents` as we did for `USceneComponents`. The construction process following is quite simple. We attach the mesh to the scene component so the mesh will follow any translations we apply to the parent `USceneComponent`. We then ensure that the mesh's origin will be centered around `USceneComponent` by setting the Mesh's **relative** location to be (0.0f, 0.0f, and 0.0f). We then ensure that the meshes do not collide with anything in the game world, we do that with the `SetCollisionProfileName()` function.

If you remember when we used this function earlier, you provide a profile name you wish the object to use the collision properties from. In our case, we wish this mesh to overlap all dynamic objects; thus, we parse `OverlapAllDynamic`. Without this line of code, the character may collide with the moving floor meshes, and that will drag the player along at the same speed thus breaking the illusion of motion we are trying to create.

Lastly, we assign the static mesh object and material we obtained earlier with the `FObjectFinders`. We ensure that we add this new mesh object to the `FloorMeshes` array in case we need them later. We also close the loop scope we created earlier.

The next thing we are going to do is create the collision box that will be used for character collisions. With the box set to collide with everything and the meshes set to overlap everything, we will be able to collide on the stationary box while the meshes whip past under our feet. The following code will create the box collider:

```
collisionBox =CreateDefaultSubobject<UBoxComponent>(TEXT("CollsionB
ox"));
check(collisionBox);

collisionBox->AttachTo(RootComponent);
collisionBox->SetBoxExtent(FVector(spawnPoint, myBounds.BoxExtent.Y,
myBounds.BoxExtent.Z));
collisionBox->SetCollisionProfileName(TEXT("BlockAllDynamic"));

} // <-- Closing if(myMesh.Succeeded())
```

As you can see, we initialize `UBoxComponent` as we always initialize components. We then attach the box to the root component as we do not wish to move it. We also set the box extent to be that of the length of the entire swap chain by setting the `spawnPoint` value as the X bounds of the collider. We set the collision profile to `BlockAllDynamic`. This means that it will block any dynamic actors such as our Character! Note that we have also closed the scope of the if statement opened earlier. With the constructor definition finished, we might as well define the accessor methods for `spawnPoint` and `killPoint` before we move onto the `Tick()` function:

```
float AFloor::GetKillPoint()
{
    return KillPoint;
}

float AFloor::GetSpawnPoint()
{
    return SpawnPoint;
}
```

AFloor::Tick()

Now, it is time to write the function that will move the meshes and ensure they move back to the start of the chain when they reach `KillPoint`. Add the following code to the `Tick()` function found in `Floor.cpp`:

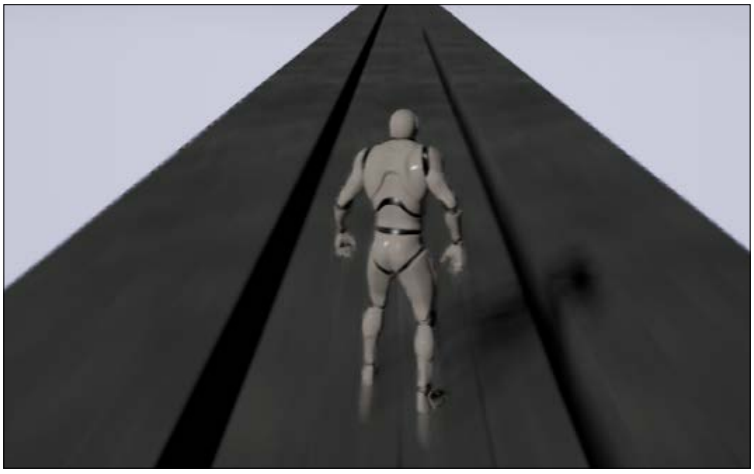
```
for (auto Scene : FloorMeshScenes)
{
Scene->AddLocalOffset(FVector(GetCustomGameMode
<ABountyDashGameMode>(GetWorld())->GetInvGameSpeed(), 0.0f, 0.0f));

if (Scene->GetComponentTransform().GetLocation().X <= KillPoint)
{
    Scene->SetRelativeLocation(FVector(SpawnPoint, 0.0f, 0.0f));
}
}
```

Here, we are using a C++ 11 range for loop. Meaning that for each element inside of `FloorMeshScenes`, it will populate the scene handle of type auto with a pointer to whatever type is contained by `FloorMeshScenes`, in this case `USceneComponent*`. For every scene component contained within `FloorMeshScenes`, we are adding a local offset to each frame. The amount we offset each frame is dependent on the current game speed.

We are getting the game speed from the game mode via the template function we wrote earlier. As you can see, we specified the template function to be of type `ABountyDashGameMode`, thus we will have access to the bounty dash game mode functionality. We have done this so that the floor will move faster under the player's feet as the speed of the game increases. The next thing we do is check the X value of the Scene components location. If this value is equal to or less than the value stored in `KillPoint`, we reposition the scene component back to the spawn point. As we attached the meshes to these `USceenComponents` earlier, the meshes will also translate with the scene components. Lastly, ensure that you have added #include n`BountyDashGameMode.h`" to the `.cppts` include list.

Placing the Floor in the level!

We are done making the floor! Compile the code and return to the level editor. We can now place this new floor object in the level! Delete the static mesh that would have replaced our earlier box brushes and drag and drop the **Floor** object into the scene. The floor object can be found under the `C++ classes` folder of the content browser. Select the `Floor` in the level and ensure that its location is set too (0.0f, 0.0f, and -100.f). This will place the floor just below the player's feet around origin. Also ensure the `ATargetPoints` we placed earlier are in the right positions above the lanes. With this all in place, you should be able to press play and observe the floor moving underneath the player indefinitely. You should see something similar to this:



You will notice that as you move between the lanes by pressing *A* and *D* the player maintains the X Position of the target points but nicely travels to the center of each lane.

Creating the obstacles

The next step for this project is to create the obstacles that will come flying at the player. These obstacles are going to be very simple and contain only a few members and functions. These obstacles are to only serve as a blockade for the player and all of the collision with the obstacles will be handled by the player itself. Use the class wizard to create a new class named **Obstacle** and inherit this object from `AActor`. Once the class has been generated, modify the class definition found in `Obstacle.h`, so it appears as follows:

```
UCLASS(BlueprintType)
class BOUNTYDASH_API AObstacle: public AActor
{
GENERATED_BODY()

    float KillPoint;

public:
    // Sets default values for this actor's properties
    AObstacle ();

    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

    // Called every frame
    virtual void Tick( float DeltaSeconds ) override;

void SetKillPoint(float point);
float GetKillPoint();

protected:
    UFUNCITON()
    virtual void MyOnActorOverlap(AActor* otherActor);

    UFUNCTION()
    virtual void MyOnActorEndOverlap(AActor* otherActor);

public:
UPROPERTY(EditAnywhere, BlueprintReadWrite)
    USphereComponent* Collider;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    UStaticMeshComponent* Mesh;
};
```

You will note that the class has been declared with the `BlueprintType` specifier! This object is simple enough to justify extension into blueprint as there is no new learning to be found within this simple object, and we can use blueprint for convenience. For this class, we added a private member `KillPoint` that will be used to determine when the `AObstacle` should destroy itself. We also added the accessor methods for this private member. You will notice that we added the `MyActorBeginOverlap` and `MyActorEndOverlap` functions that we will be providing to the appropriate delegates, so we can provide custom collision response for this object. We also declared these functions as virtual; this is so we can override these collision functions in child classes of `AObstacle`.

The definitions of these functions are not too complicated either. Ensure that you have included `#include "BountyDashGameMode.h"` in the `Obstacle.cpp`. Then, we can begin filling out our function definitions. The following is code what we will use for the constructor:

```
AObstacle::AObstacle()
{
PrimaryActorTick.bCanEverTick = true;

Collider = CreateDefaultSubobject<USphereComponent>(TEXT("Collider"));
check(Collider);

RootComponent = Collider;
Collider ->SetCollisionProfileName("OverlapAllDynamic");

Mesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("Mesh"));
check(Mesh);
Mesh ->AttachTo(Collider);
Mesh ->SetCollisionResponseToAllChannels(ECR_Ignore);
KillPoint = -20000.0f;

OnActorBeginOverlap.AddDynamic(this, &AObstacle::MyOnActorOverlap);
OnActorBeginOverlap.AddDynamic(this, &AObstacle::MyOnActorEndOverlap);
}
```

The only thing of note within this constructor is again we set the mesh of this object to ignore collision response to all channels meaning. The mesh will not affect collision in any way. We have also initialized kill point with a default value of `-20000.0f`. Following that we are binding the custom `MyOnActorOverlap` and `MyOnActorEndOverlap` function to the appropriate delegates.

The `Tick()` function of this object is responsible for translating the obstacle during play. Add the following code to the Tick function of `AObstacle`:

```
void AObstacle::Tick( float DeltaTime )
{
    Super::Tick( DeltaTime );
float gameSpeed = GetCustomGameMode<ABountyDashGameMode>(GetWorld())->
GetInvGameSpeed();

    AddActorLocalOffset(FVector(gameSpeed, 0.0f, 0.0f));

    if (GetActorLocation().X < KillPoint)
    {
        Destroy();
    }
}
```

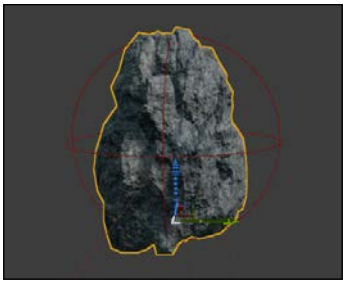
As you can see, the tick function will add an offset to the `AObstacle` each frame along the x-axis via the `AddActorLocalOffset` function. The value of the offset is determined by the game speed set in the game mode. Again, we are using the template function we created earlier to get the game mode to call `GetInvGameSpeed()`. The `AObstacle` is also responsible for its own destruction, upon reaching a maximum bounds defined by `killPoint` the `AObstacle` will destroy itself.

The last thing we need to add is the function definitions for the `OnOverlap` functions and `KillPoint` accessors:

```
void AObstacle::MyOnActorOverlap(AActor* otherActor)
{

}
void AObstacle::MyOnActorEndOverlap(AActor* otherActor)
{

}

void AObstacle::SetKillPoint(float point)
{
   killPoint = point;
}

float AObstacle::GetKillPoint()
{
    return killPoint;
}
```

Now, let's abstract this class into blueprint. Compile the code and go back to the game editor. Within the content folder, create a new blueprint object that inherits form the `Obstacle` class we just made, name it `RockObstacleBP`. Within this blueprint, we need to make some adjustments. Select the **collider** component we created and expand the shape sections in **Details** panel. Change the **Sphere radius** property to `100.0f`. Next, select the mesh component and expand the **Static Mesh** section; from the provided drop down, choose the `SM_Rock` mesh. Next, expand the transform section of the `Mesh` component details panel and match these values:

You should end up with an object that looks similar to this:



Spawning actors from C++!

Despite the Obstacles being fairly easy to implement from a C++ standpoint, the complication comes from the spawning system we will be using to create these objects in game. We will leverage a similar system to the player's movement by basing the spawn locations off of `ATargetPoints` that are already in the scene. We can then randomly select a spawn target when we require a new object to spawn. Open the class wizard now, and create a class that inherits from **Actor** and call it `ObstacleSpawner`. We inherit from `AActor` as even though this object does not have a physical presence in the scene, we still require the `ObstacleSpawner` to tick.

The first issue we are going to encounter is our current target points give us a good indication of the `Y` positon for our spawns but the X position is centered around origin. This is undesirable for the obstacle spawn point as we would like to spawn these objects a fair distance away from the player so we can do two things. One, obscure the **popping** of spawning the objects via fog and two, present the player with enough obstacle information so they may dodge them at high speeds. This means we are going to require some information from our floor object, we can use the `KillPoint` and `SpawnPoint` members of the floor to determine the spawn location and kill location of the Obstacles.

Obstacle Spawner class definition

This will be another fairly simple object. It will require a `BeginPlay` function, so we may find the floor and all the target points we require for spawning. We also require a `Tick` function so that we may process spawning logic on a per frame basis. Thankfully, both of these are provided by default by the class wizard. We created a protected `SpawnObstacle()` function, so we may group that functionality together. We are also going to require a few `UPRORERTY` declared members that can be edited from the level editor. We need a list of obstacle types to spawn; we can then randomly select one of the types each time we spawn an obstacle. We also require the spawn targets (though we will be populating these upon begin play). Finally, we will need a spawn time that we can set for the interval between obstacles spawning. To accommodate for all of this, navigate to `ObstacleSpawner.h` now and modify the class definition to match the following:

```
UCLASS()
class BOUNTYDASH_API AObstacleSpawner : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AObstacleSpawner();

    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

    // Called every frame
    virtual void Tick( float DeltaSeconds ) override;


protected:
```

```
    void SpawnObstacle();

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    TArray<TSubclassof<class AObstacle*>> ObstaclesToSpawn;

    UPROPERTY()
    TArray<class ATargetPoint*>SpawnTargets;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    float SpawnTimer;

    UPROPERTY()
    USceneComponent* scene;
private:
    float KillPoint;
    float SpawnPoint;
   float TimeSinceLastSpawn;
};
```

I have again used `TArrays` for our containers of obstacle objects and spawn targets. As you can see the obstacle list is of type `TSubclassof<class AObstacle>>.` This means that the objects in this `TArray` will be class types that inherit from `AObscatle`. This is very useful as not only will we be able to use these array elements for spawn information; the engine will also filter our search when adding object types to this array from the editor! With these class types, we will be able to spawn objects that inherit from `AObject` (including blueprints!) when required. We also included a scene object, so we can arbitrarily place `AObstacleSpawner` in the level somewhere and two private members that will hold the kill and spawn point of the objects. The last element is a `float` timer that will be used to gauge how much time has passed since the last obstacle spawn.

Obstacle Spawner function definitions

Okay, now we can create the body of the `AObstacleSpawner` object. Before we do, ensure the include list in `ObstacleSpawner.cpp` is as follows:

```
#include "BountyDash.h"
#include "BountyDashGameMode.h"
#include "Engine/TargetPoint.h"
#include "Floor.h"
#include "Obstacle.h"
#include "ObstacleSpawner.h"
```

Following this, we have a very simple constructor that establishes the root scene component:

```
// Sets default values
AObstacleSpawner::AObstacleSpawner()
{
// Set this actor to call Tick() every frame.  You can turn this off
to improve performance if you don't need it.
PrimaryActorTick.bCanEverTick = true;

Scene = CreateDefaultSubobject<USceneComponent>(TEXT("Root"));
check(Scene);
RootComponent = scene;

SpawnTimer = 1.5f;
}
```

Following the constructor, we have `BeginPlay()`. Inside this function, we are going to do a few things. First, we are simply performing the same in level object retrieval we executed in `ABountyDashCarhacter` to get the location of `ATargetPoints`. However, this object also requires information from the floor object in the level. We are also going to get the `Floor` object the same way we did with the `ATargetPoints` by utilizing `TActorIterators`. We will then get the required kill and spawn point information. We will also set `TimeSinceLastSpawn` to `SpawnTimer`, so we begin spawning objects instantaneously:

```
// Called when the game starts or when spawned
void AObstacleSpawner::BeginPlay()
{
    Super::BeginPlay();

for(TActorIterator<ATargetPoint> TargetIter(GetWorld()); TargetIter;
    ++TargetIter)
    {
        SpawnTargets.Add(*TargetIter);
    }

for (TActorIterator<AFloor> FloorIter(GetWorld()); FloorIter;
       ++FloorIter)
    {
        if (FloorIter->GetWorld() == GetWorld())
        {
            KillPoint = FloorIter->GetKillPoint();
            SpawnPoint = FloorIter->GetSpawnPoint();
        }
```

```
        }
        TimeSinceLastSpawn = SpawnTimer;
    }
```

The next function we will detail is `Tick()`, which is responsible for the bulk of the `AObstacleSpawner` functionality. Within this function, we need to check if we require a new object to be spawned based on the amount of time that has passed since we last spawned an object. Add the following code to `AObstacleSpawner::Tick()` underneath `Super::Tick()` now:

```
    TimeSinceLastSpawn += DeltaTime;

    float trueSpawnTime = spawnTime / (float)GetCustomGameMode
    <ABountyDashGameMode>(GetWorld())->GetGameLevel();

    if (TimeSinceLastSpawn > trueSpawnTime)
    {
        timeSinceLastSpawn = 0.0f;
        SpawnObstacle ();
    }
```

Here, we are accumulating the delta time in `TimeSinceLastSpawn`, so we may gauge how much real time has passed since the last obstacle was spawned. We then calculate the `trueSpawnTime` of the `AObstacleSpawner`. This is based on a base `SpawnTime`, which is divided by the current game level retrieved from the game mode via the `GetCustomGamMode()` template function. This means that as the game level increases and the obstacles begin to move faster, the obstacle spawner will also spawn objects at a faster rate. If the accumulated `timeSinceLastSpawn` is greater than the calculated `trueSpawnTime` we need to call `SpawnObject()` and reset the `timeSinceLastSpawn` timer to `0.0f`.

Getting information from components in C++

Now, we need to write the spawn function. This spawn function is going to have to retrieve some information from the components of the object that is being spawned. As we allowed our `AObstacle` class to be extended into blueprint, we also exposed the object to a level of versatility we must compensate for in the codebase. With the ability to customize the mesh and bounds of the Sphere Collider that makes up any given obstacle, we must be sure we spawn the obstacle in the right place regardless of size!

To do this, we are going to need to obtain information form the components contained within the spawned `AObstacle` class. This can be done via `GetComponentByClass()`. It will take the `UClass*` of the component you wish to retrieve and will return a handle to the component if it has been found. We can then cast this handle to the appropriate type and retrieve the information we require! Let's begin detailing the spawn function by adding the following code to `ObstacleSpawner.cpp`:

```
void AObstacleSpawner::SpawnObstacle()
{
if (SpawnTargets.Num() > 0 && ObstaclesToSpawn.Num() > 0)
{
    short Spawner = FMath::Rand() % SpawnTargets.Num();
    short Obstical = FMath::Rand() % ObstaclesToSpawn.Num();
    float CapsuleOffset = 0.0f;
```

Here, we ensure that both of the arrays have been populated with at least one valid member. We then generate the random look up integers that we will use to access the `SpawnTargets` and `obstacleToSpawn` arrays. This means that every time we spawn an object, both the lane spawned in and the type of the object will be randomized. We do this by generating a random value with `FMath::Rand()` and then we find the remainder of that number divided by the number of elements in the corresponding array. The result will be a random number that exists between 0 and the number of objects in either array minus one, which is perfect for our needs. Continue by adding the following code:

```
FActorSpawnParameters SpawnInfo;

FTransform myTrans = SpawnTargets[Spawner]->GetTransform();
myTrans.SetLocation(FVector(SpawnPoint, myTrans.GetLocation().Y,
myTrans.GetLocation().Z));
```

Here, we are using a `struct` called `FActorSpawnParameters`. The default values of this `struct` are fine for our purposes. We will soon be parsing this `struct` to a function in our world context. After that, we create a transform that we will be providing to world context as well. The transform of the spawner will suffice apart from the X Component of the location. We need to adjust this so the `X` value of the spawn transform matches the spawn point we retrieved from the floor. We do this by setting the `X` component of the spawn transforms location to be the `spawnPoint` value we received earlier, and the other components of the location vector to be the `Y` and `Z` components of the current location.

The next thing we must do is actually spawn the object! We are going to utilize a template function called `SpawnActor()` that can be called form the `UWorld*` handle returned by `GetWorld()`. This function will spawn an object of a specified type in the game world at a specified location. The type of the object is determined by passing a `UClass*` handle that holds the object type we wish to spawn. The transform and spawn parameters of the object are also determined by the corresponding input parameters of `SpawnActor()`. The template type of the function will dictate the type of object that is spawned and the handle that is returned from the function. In our case, we require `AObstacle` to be spawned. Add the following code to the `SpawnObstacle` function:

```
AObstacle* newObs = GetWorld()-> SpawnActor<AObstacle>(obstacleToSpawn
[Obstical, myTrans, SpawnInfo);

if(newObs)
{
newObs->SetKillPoint(KillPoint);
```

As you can see, we are using `SpawnActor()` with a template type of `AObstacle`. We use the random look up integer we generated before to retrieve the class type from the `obstacleToSpawn` array. We also provide the transform and spawn parameters we created earlier to `SpawnActor()`. If the new `AObstacle` was created successfully, we then save the return of this function into an `AObstacle` handle that we will use to set the kill point of the obstacle via `SetKillPoint()`.

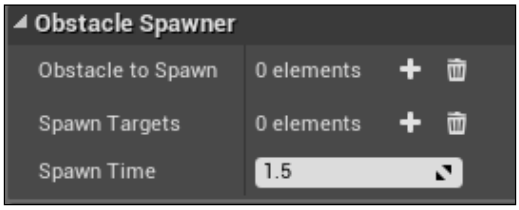
We must now adjust the height of this object. The object will more than likely spawn in the ground in its current state. We need to obtain access to the sphere component of the obstacle so that we may get the radius of this capsule and adjust the position of the obstacle, so it sits above the ground. We can use the capsule as a reliable resource as it is the root component of the obstacle, thus we can move the obstacle entirely out of the ground if we assume the base of the sphere will line up with the base of the mesh. Add the following code to the `SpawnObstacle()` function:

```
USphereComponent* obsSphere = Cast<USphereComponent>
(newObs->GetComponentByClass(USphereComponent::StaticClass()));

if (obsSphere)
{
newObs->AddActorLocalOffset(FVector(0.0f, 0.0f, obsSphere->
GetUnscaledSphereRadius()));
}
}//<-- Closing if(newObs)
}//<-- Closing if(SpawnTargets.Num() > 0 && obstacleToSpawn.Num() > 0)
```

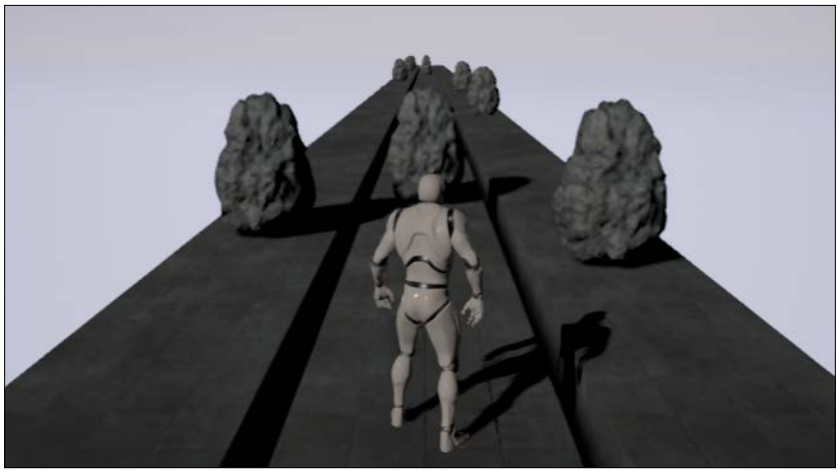
Here, we are getting the sphere component out of the `newObs` handle we obtained from `SpawnActor()` via `GetComponentByClass()`, which was earlier mentioned. We pass the class type of a `USphereComponent` via the static function `StaticClass()` to the function. This will return a valid handle if the `newObs` contains `USphereComponent` (which we know it does). We then cast the result of this function to `USphereComponent*` and save it in the `obsSphere` handle. We ensure this handle is valid; if it is, we can then offset the actor we just spawned on the z-axis by the unscaled radius of the sphere component. This will result in all obstacles spawned be in line with the top of the floor!

Ensuring the Obstacle Spawner works

Okay, now it is time to bring the obstacle spawner into the scene. Be sure to compile the code then navigate to the **C++ classes** folder of the content browser. From here, drag and drop `ObstacleSpawner` into the scene. Select the new `ObstacleSpawner` via the **World Outlier** and address the **Details** panel. You will see the exposed members under the `ObstacleSpawner` section like so:



Now, add the `RockObstacleBP` we made earlier to the `ObstacleToSpawn` array. Press the small white plus next to the property in **Details** panel; this will add an element to the `TArray` that you will then be able to customize. Select the dropdown that currently says **None**. Within this drop down search for the `RockObstacleBP` and select it. If you wish to create and add more obstacle types to this array feel free. We do not need to add any members to the **Spawn Targets** property as that will happen automatically. Now, press **Play** and behold a legion of moving rocks!



Minting the coin object

We are nearly ready to start playing `BountyDash`! First, we need to create a coin system. Coins will function in a very similar way to obstacles, but they require other specific functionality that warrants their own class. What we can do, however, is create the coin as a child class of `AObstacle`! This means that we do not have to write duplicate functionality! Use the class wizard to create another class called `Coin`; however, this time be sure to make the parent class `AObstacle`.

Coin class definition

Once the class generation is complete, navigate to the `Coin.h` file. You will notice that we are given no default functionality and we must specify it all. Still much like the `AObstacle`, our class definition is going to be very minimal. Most of the functionality we require for the coin has already been included in the `AObstacle` base class. All we need to do here is override the `Tick()` and `MyOnActorOverlap()` functions, so we can add additional coin functionality.

The following is the class definition for the `ACoin` object:

```
UCLASS()
class BOUNTYDASH_API ACoin : public AObstacle
{
    GENERATED_BODY()

    ACoin();
```

```
    // Called every frame
    virtual void Tick(float DeltaSeconds) override;

    UFUNCTION()
    virtual void MyOnActorOverlap(AActor* otherActor) override;
};
```

Simple right? As you can see, we declared a `ACoin()` constructor and we are overriding the `virtual Tick` function. You will notice that we have also included the `UFUNCITON` macro above the `MyOnActorOverlap` function that we will be passing to the overlap delegate for this actor so that the delegate association succeeds.

Coin function definitons

Let's now define how the coin is going to work navigate to `coin.cpp` now. Start by adding the following to the include list if they are not already present:

```
#include "BountyDash.h"
#include "BountyDashCharacter.h"
#include "BountyDashGameMode.h"
#include "Coin.h"
```

Now, we are going to define our coin's constructor. `AObstacle` base class already constructs the mesh and collider objects, so we do not have to do that here. It also provides the `MyOnActorBeginOverlap` function to the `OnActorBeginOverlap` delegate. Because the `MyOnActorBeginOverlap` function was specified as virtual and we have overridden this function in `ACoin`, we do not need to re-bind the functions. We can define our constructor with no functionality as the following:

```
ACoin::ACoin()
{

}
```

Following that, we can define the `Tick()` function. This function is going to be used to rotate the mesh component of the object so it spins while moving toward the player. Add the following to `Tick()` within `Coin.cpp`:

```
void ACoin::Tick(float DeltaTime)
{
Super::Tick(DeltaTime);
    Mesh->AddLocalRotation(FRotator(5.0f, 0.0f, 0.0f));
}
```

Here, you can see we are calling `Super::Tick()`, so the functionality we implemented in `AObstacle` is executed. We then add a local rotation to the mesh component every frame via the `AddLocalRotation` function. We yaw the coin roughly at 5 degrees per frame.

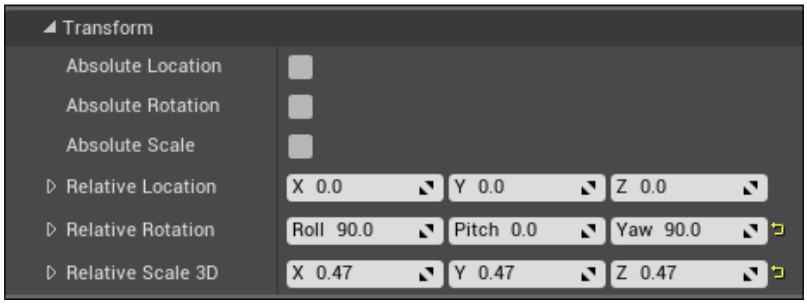
Ok, now we can define our overridden `MyOnActorOverlap` functionality. We will be detailing the collisions between objects in more detail later in the chapter. For now, we will write code that will handle when a coin is spawned on top of an obstacle object. This may happen from time to time as they will be sharing spawning locations but not necessarily have exclusive spawn times. We need to detect if the coin is currently colliding with an obstacle. If so, we need to adjust the height of the coin object, we can do that with the following code:

```
void ACoin::MyOnActorOverlap(AActor* otherActor)
{
    if (otherActor->GetClass()->IsChildOf(AObstacle::StaticClass()))
    {
USphereComponent* thisSphere = Cast<USphereComponent>(GetComponentByCl
ass(USphereComponent::StaticClass()));

USphereComponent* otherSpehre = Cast<USphereComponent>(otherActor-> Ge
tComponentByClass(USphereComponent::StaticClass()));

        if (otherSpehre)
        {
AddActorLocalOffset(FVector(0.0f, 0.0f, (otherSpehre->
GetUnscaledSphereRadius() * 2.0f) + Collider->
GetUnscaledSphereRadius()));
        }
    }
}
```

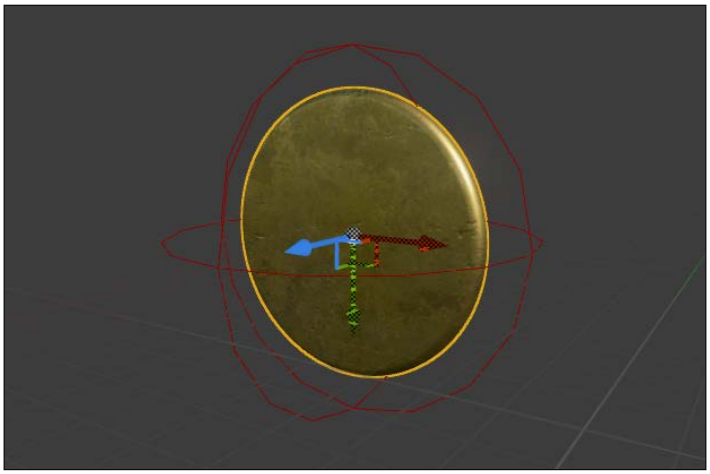
The first thing we do is use the class type information of the `AActor` object to see if the offending actor is either a `AObstacle` or a child of `AObstacle`. We do this with the `IsChildOf()` function. This function can be called on any actor to determine if it is a child of a specific class type or of that type itself. For the input parameter, we parse the class type of `AObstacle` via `StaticClass()`. If this check succeeds, we need to get information from the sphere component of the other actor now proven to be of type `AObstacle`. We do this as we did previously using `GetComponentByClass()` and parsing the class information of the type of component we require. We then check that this function returned a valid handle. If so, we will add a local offset to the coin actor. We will only be adding an offset on the z-axis. We will offset by the `otherSphere` radius * 2 plus the radius of the coin sphere so that the coin will be placed on top of the obstacle mesh.

Making the coin Blueprint!

Let's create the blueprint that is going to extend from the coin object. Create a new `Blueprint` class somewhere in the content browser and inherit the blueprint from the `Coin` class we just made. Call this class `CoinBP`. We are going to assemble this blueprint in the same way we did `RockObstacleBP`. For the collider component, set the **Sphere Radius** property of the **Shape** section in the details panel to `60.0f`. Then, set the **Static Mesh** property of the **Static Mesh** section of the **mesh** component to `SM_AssetPlatform`. After that, add a member to the **Override Materials** list in the **Rendering** section of **Details** panel. Set this member to be the `M_Metal_Gold` material. Then, change the transform settings of this component so they match the following:



You will then be presented with a blueprint object that looks like this!



Making it rain coins, creating the coin spawner

We now need to create a system that will spawn the coins. This time, we cannot use the `AObstacleSpawner` we created previously as a base class. This is because the spawning logic for the coin spawner is much more involved. For the coins in `BountyDash`, we need to construct the system so the coins will spawn in sets of a random number. Also, within each set we need to ensure that the coins spawn far enough apart. The result will be periodically spawning streams of coins the player can pick up. Open the class wizard again and create a class that inherits from Actor; call it `CoinSpawner`. This class is going to be very minimal with regards to methods, but it will have quite a few properties.

Coin Spawner class Definition

We will be keeping most of the default functionality provided by the class wizard. The only function we will be removing is the `Tick()` method, as we are going to utilize the `FTimerManager` object to drive our spawning functionality. More on this later. Ensure that you remove the Tick function declaration and add the following publically declared member variables:

```
public:
    UPROPERTY(EditAnywhere)
    TSubclassOf<ACoin> CoinObject;

    UPROPERTY()
    TArray<class ATargetPoint*> SpawnTransforms;

    UPROPERTY()
    USceneComponent* Root;

    UPROPERTY(EditAnywhere)
    int32 MaxSetCoins;

    UPROPERTY(EditAnywhere)
    int32 MinSetCoins;

    UPROPERTY(EditAnywhere)
    float CoinSetTimeInterval;

    UPROPERTY(EditAnywhere)
    float CoinTimeInterval;

    UPROPERTY(EditAnywhere)
    float MovementTimeInterval;
```

Here, we have a `TSubclassOf<class ACoin>` handle called `CoinObject`. This has been declared as `EditAnywhere`, as we wish to edit this value from the editor much like the obstacle `TArray` of `AObstacleSpawner`. It will act as the class object that we use when spawning the coins. We included another `TArray` of `ATargetpoints` to be used for our coin spawn locations. Following that is another `USceneComponent` that will be used to arbitrarily position the spawner in the world. Underneath that, we have all of the properties that we are going to use to dictate the spawning logic for the coins. You will notice that they have all been declared with the `UPROPERTY` macro and specified, so we may edit the values from the editor. `MaxSetCoins` and `MinSetCoins` will be used to determine a random number of coins to spawn within a set between those two maxims. Next, we have all of the `float` timers that will dictate how long it takes between each major spawner action. We have one for each set of coins, each individual coin within a set, and a movement interval.

Following that, we have our protected members that will be used for internal logic and thus do not need to be exposed:

```
protected:
    void SpawnCoin();
    void SpawnCoinSet();
    void MoveSpawner();

    int32 NumCoinsToSpawn;

    float KillPoint;
    float SpawnPoint;
    int32 TargetLoc;

    FTimerHandle CoinSetTimerHandle;
    FTimerHandle CoinTimerHandle;
    FTimerHandle SpawnMoveTimerHandle;
```

We declared several spawning functions. These functions are going to be used to invoke the various spawning methods we require for our coins. We then declare a series of variables. The first `NumCoinsToSpawn` is an integer value that will hold the number of coins we need to spawn for any given set. We also have the two float values `KillPoint` and `SpawnPoint` that will be used for spawning logics. Following this, we have an `int` value that will be used to determine within which lane we are spawning coins.

Lastly, we have a new object type we have yet to interact with the `FTimerHandle` object. Unlike our obstacle spawner, we are not going to be using traditional float timers that are incremented within a `Tick()` function. We are instead going to be leveraging the `TimerManager` object that is present in our `GameWorld` (`UWorld`). Through the time manager, we are able to create timer objects that will invoke a provided function after a given time period. The three timer handles here will be used to initialize the various timers we require for this coin spawner.

Coin Spawner function definitions

With our member variables set up, we can now define how the coin spawner will function. As always, ensure the include list at the top of the `coinspawner.cpp` matches these:

```
#include "BountyDash.h"
#include "Engine/TargetPoint.h"
#include "Coin.h"
#include "Floor.h"
#include "CoinSpawner.h"
```

The constructor for the coin spawner is very simple and does not require much of an explanation. The constructor simply initializes `USceneComponent` and assigns it as the root and establishes some default values for the coin spawn variables. Ensure that the `ACoinSpawner` constructor found in the `CoinSpawner.cpp` matches the following:

```
// Sets default values
ACoinSpawner::ACoinSpawner()
{
// Set this actor to call Tick() every frame.  You can turn this off
to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    Root = CreateDefaultSubobject<USceneComponent>(TEXT("Root"));
    RootComponent = Root;

MaxSetCoins = 5;
    MinSetCoins = 3;
    CoinSetTimeInterval = 4.0f;
    CoinTimeInterval = 0.5f;
    MovementTimeInterval = 1.0f;
}
```

We then need to define our `BeginPlay()` function. This function will be responsible for obtaining the information we require from our level as well as initialize the timers we spoke of earlier. Add the following definition for `ACoinSpawner::BeginPlay()` now:

```
// Called when the game starts or when spawned
void ACoinSpawner::BeginPlay()
{
    Super::BeginPlay();

for (TActorIterator<ATargetPoint> TargetIter(GetWorld());
     TargetIter; ++TargetIter)
    {
        SpawnTransforms.Add(*TargetIter);
    }

    for (TActorIterator<AFloor> FloorIter(GetWorld()); FloorIter;
            ++FloorIter)
    {
        if (FloorIter->GetWorld() == GetWorld())
        {
            KillPoint = FloorIter->GetKillPoint();
            SpawnPoint = FloorIter->GetSpawnPoint();
        }
    }

    // Create Timers
    FTimerManager& worldTimeManager = GetWorld()->
      GetTimerManager();

worldTimeManager.SetTimer(CoinSetTimerHandle, this,
&ACoinSpawner::SpawnCoinSet, CoinSetTimeInterval, false);

worldTimeManager.SetTimer(SpawnMoveTimerHandle, this,
&ACoinSpawner::MoveSpawner, MovementTimeInterval, true);
}
```

As you can see, the first section of `BeginPlay()` is identical to that of the `AObstacleSpawner`. Following this, however, we are creating two of the timers we are using for the spawning logic. We first get a handle to the timer manager for this game world via `GetWorld()->GetTimeManager()`. With this `FTimeManager` handle, we are able to set two of the timers we require. We do this via `SetTimer()`. This method takes in an `FTimerHandle` to populate (we provide the handles we declared in the `ACoinSpawner` class definition), an object to call the provided function on, a handle to the function we wish executed upon timer completion, a timer rate (how long it will take for the timer to execute), and if we wish the timer to loop.

Here, we are setting two timers. The first is a coin set timer; this timer will invoke our `SpawnCoinSet` function after a provided time (4.0f by default). We specified that this method is to **not** loop as we will be resetting this timer ourselves. The second timer we are setting is the `Move` timer; this timer will invoke `MoveSpawner()` every second by default. This method will periodically shift the lane in which the coins spawn.

Next, we must define the various methods that will be invoked by our timers. Let's begin with the `SpawnCoinSet` function. Add the following code to `CoinSpawner.cpp` now:

```
void ACoinSpawner::SpawnCoinSet()
{
    NumCoinsToSpawn = FMath::RandRange(MinSetCoins, MaxSetCoins);

    FTimerManager& worldTimeManager = GetWorld()->
GetTimerManager();

    // Swap active timers
worldTimeManager.ClearTimer(CoinSetTimerHandle);

worldTimeManager.SetTimer(CoinTimerHandle, this,
&ACoinSpawner::SpawnCoin, CoinTimeInterval, true);
}
```

The first thing we do is generate the number of coins we will be spawning in this set by calling `FMath::RandRange` and providing the two maxims we declared earlier. As the `SpawnCoinSet()` method was invoked via a timer we now need to clear the coin set timer and active the per coin timer. This is so we can guarantee our spawn coin set timer will only reactivate once all individual coins for a given set have been spawned. We can do this by calling `ClearTimer()` on the `FTimerManager` handle, which will remove the timer from execution. We parse the handle of the timer we wish to clear to the manager (in this case `CoinSetTimerHandle`).

Following this, we set another timer via the `SetTimer` method we used earlier. This time we are setting the individual spawn coin timer. We have set this timer to loop as we wish the timer continue to spawn coins until all coins have been spawned. Following this, we can define the `MoveSpawner` function. This function is very simple and simply changes the `TargetLoc` integer we declared earlier. This integer will be used for a look up into the `SpawnTransforms` array. Add the following function definition to the `CoinSpawner.cpp`:

```
void ACoinSpawner::MoveSpawner()
{
    TargetLoc = FMath::Rand() % SpawnTransforms.Num();
}
```

We are nearly done with the coin spawner. The last thing we need to do is define the SpawnCoin() function. This function is very similar to AObstacleSpawn::SpawnObstacle(). It appears as follows:

```
void ACoinSpawner::SpawnCoin()
{
    FActorSpawnParameters spawnParams;

    FTransform coinTransform = SpawnTransforms[TargetLoc]->
GetTransform();

coinTransform.SetLocation(FVector(SpawnPoint, coinTransform.
GetLocation().Y, coinTransform.GetLocation().Z));

ACoin* spawnedCoin = GetWorld()->SpawnActor<ACoin>(CoinObject,
coinTransform, spawnParams);

    if (spawnedCoin)
    {
USphereComponent* coinSphere = Cast<USphereComponent>(spawnedCoin->
GetComponentByClass(USphereComponent::StaticClass()));

        if (coinSphere)
        {
        float offset = coinSphere->
GetUnscaledSphereRadius();

spawnedCoin->AddActorLocalOffset(FVector(0.0f, 0.0f, offset));
        }

        NumCoinsToSpawn--;
    }

    if (NumCoinsToSpawn <= 0)
    {
        FTimerManager& worldTimeManager = GetWorld()->
GetTimerManager();

worldTimeManager.SetTimer(CoinSetTimerHandle, this,
&ACoinSpawner::SpawnCoinSet, CoinSetTimeInterval, false);

        worldTimeManager.ClearTimer(CoinTimerHandle);
    }
}
```

The first section of this function is exactly the same as the spawn method used in the `AObstacleSpawner`. The most noticeable difference is the inclusion of the timer swap logic and the end of the function. Each time a coin has spawned, we decremented the `NumCoinsToSpawn` variable. Once this value has reached 0 or lower, we swap our timers again. This swap is very similar to the one we performed in the `SpawnCoinSet()` method, yet this time we are clearing the individual coin timer and resetting the coin set timer.

Testing what we have so far!

Ok now that we have our floor, our coins, our obstacles, and our character set up, let's add the coin spawner to the level to complete the world object set! Do this by navigating back to the C++ classes folder of **Content** browser and dragging an `ACoinSpawner` object into the world. Select the new `CoinSpawner` in **World Outlier** and address the details panel. Within the details panel, set the **Coin Object** property under the **Coin Spawner** section to be `CoinBP`. With that in place, press **Play** and you will see rocks and coins hurtling at the player at 10 m/s!

Creating the interactions between the world objects

Finally, we have created the game world for `BountyDash`! Now, it is time to make it all interact with one another. We have already done this somewhat by having the coin adjust height when it is spawned on top of an obstacle object. We must scrutinize how we are going to interact with the player when it runs into the objects around him. First, let's detail how the player is going to be colliding with the obstacles.

Pushing the Character

When the player runs into a rock, we want the player avatar to be pushed back into the game world down the x-axis. This will be quite simple to implement; we can introduce the same offset backwards to the player that we do for all the other objects if he is currently colliding with a rock obstacle! This is very easily done. Navigate to the `BountyDashCharacter.cpp` in code; we are going to be defining the collision function.

ABountyDashCharacter Collision functions

The first collision function we will be defining is `MyOnComponentOverlap()`. The purpose of this function is to detect if the player has collided with an obstacle from a reasonably head on approach. We do this so that when the player is skipping between lanes there is a margin of allowance for moving over the back of an obstacle. If so, we are going to be enacting on the collision. Add the following code to `ABounty DashCharacter::myOnComponentOverlap`:

```
if (OtherActor->GetClass()->IsChildOf(AObstacle::StaticClass()))
{
FVector vecBetween = OtherActor->GetActorLocation() -
GetActorLocation();

float AngleBetween = FMath::Acos(FVector::DotProduct(vecBetween.
GetSafeNormal(), GetActorForwardVector().GetSafeNormal()));

    AngleBetween *= (180 / PI);

    if (AngleBetween < 60.0f)
    {
        bBeingPushed = true;
    }
}
```

We check that the offending actor is either of type `AObstacle` or a child of the class. If so, we get a vector between the character and the obstacle. We determine if the character is heading toward the obstacle within a margin by checking the angle between the direction of the vector between the two objects and the forward vector of the character. We do this by obtaining a dot product between the two vectors via the `FVector::DotProduct` function.

We then plug the result of this calculation into `FMath::Acos`. We do this as the dot product will return a ratio value that when parsed through an `arccos` function will return the angle between the two vectors in radians. To change this value to degrees, we multiply it by (*180 / PI*) as PI radians = 180 degrees. If the angle between vectors is less than `60.0f`, we can assume that the collision is fairly direct, so we then inform the character of a collision with the obstacle by setting the `bBeingPushed` Boolean to `true`. As we just referenced the `AObstacle` type, ensure you add `#include "Obstacle.h"` to the include list of this `.cpp`.

Next, we define the `MyOnActorEndOverlap()` function. This one is much less complicated, we will simply check that the leaving actor is of type `AObstacle`; and if so, we will enact on the end of the overlap, add the following code to `ABountyDashCharacter::myOnComponentEndOverlap`:

```
if (OtherActor->GetClass()->IsChildOf(AObstacle::StaticClass() ))
{
bBeingPushed = false;
}
```

As you can see, we set the `bBeingPushed` value to false when the obstacle leaves the bounds of the character f capsule component collider. Now, we must add some code to the `Tick()` function to ensure the character's position updates when he is being pushed by an obstacle.

Pushing the character back

In `ABountyDashCharacter::Tick()`, we are going to be checking if the `bBeingPushed` value is set to true; if so, we will be offsetting the character's location down the x-axis at the same speed as the obstacles. Add the following code to `ABountyDashCharacter::Tick()`:

```
if (bBeingPushed)
{
    float moveSpeed = GetCustomGameMode<ABountyDashGameMode>
     (GetWorld())->GetInvGameSpeed();
    AddActorLocalOffset(FVector(moveSpeed, 0.0f, 0.0f));
}
```

As you can see, if the character is currently being pushed we will add an offset to the character's position down the x-axis. The effect this will create is the player will feel as though the character has stopped moving! The idea is to then convince that a wall of death has caught up with them and it is game over. But, we will be programming the wall of invisible death in the next chapter.

Picking up coins

The next thing we have to implement is the collecting of coins! The collision functionality for this is much simpler than that defined for the obstacle collision. We will simply detect if the player has overlapped with a coin. If so, we will inform the player to score up. We will also have to properly define that function, so the player score increases and the game mode is informed of the new score!

Coin collision

Let's start with the coin collision functionality. Add the following code to `ACoin::MyOnActorOverlap()` found in `Coin.cpp`:

```
if (otherActor->GetClass()->IsChildOf(ABountyDashCharacter::StaticCla
ss()))
{
ABountyDashCharacter* myChar = Cast<ABountyDashCharacter>(otherActor);

myChar->ScoreUp();

GetWorld()->DestroyActor(this);
}
```

We are simply going to check if the offending actor is of type `ABountyDashCharacter`. If so, we are going to inform the character to score up. We then inform the game world to destroy the coin actor. Next, we have to define the functionality for `ABountyDashCharacter::ScoreUp()`. The purpose of this function is to increment the player's internal score count as well as informing the `Game Mode` of that increment. Add the following to code to `ABountyDashCharacter::ScoreUp()`:

```
Score++;
GetCustomGameMode<ABountyDashGameMode>(GetWorld())-
>CharScoreUp(Score);
```

This will increment the score then inform the game mode to score up thus invoking the functionality we previously implemented.

Summary

We implemented a rudimentary endless runner! Compile the changes we just made and run the game! You be pushed back by obstacles and you will be able to pick up coins! We will continue developing bounty dash in the next chapter as it is far from finished at the moment. We are yet to add in sound effects, fog, so we can't see our objects popping in, particle effects, respawning logic for the character and don't forget the invisible wall of death!

Congratulations on making it this far through the book. We are almost half way through our journey with Unreal Engine 4! In this chapter, we learned many important C++ techniques such as getting world data within objects and reading component data from objects. We wrote our very first C++ character, and in doing, so we learned how to tie the different layers of UE together with the use of polymorphism, `UClasses`, and template functions! We will be learning even more about Unreal and C++ in the next chapter when we finish Bounty Dash and add the layer of polish that brings the Unreal Factor!