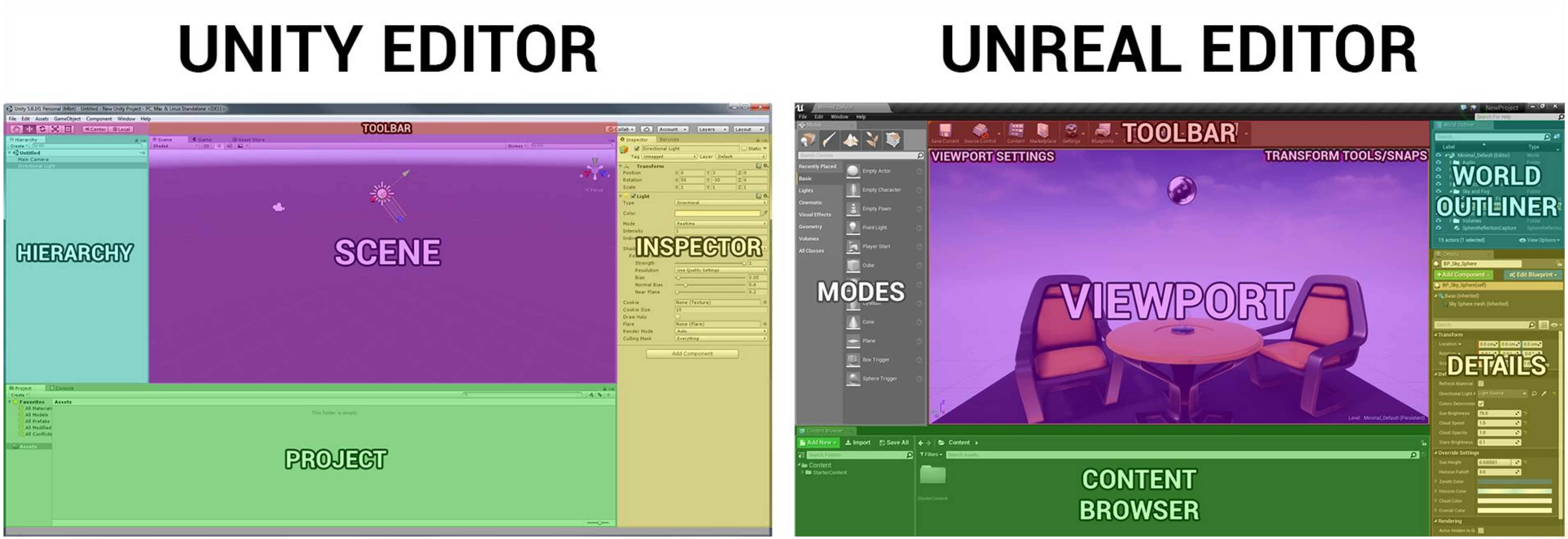
Distributed Programming

Lecture 04 - Unreal Engine and Network Communication

Edirlei Soares de Lima

<edirlei.lima@universidadeeuropeia.pt>

Editor – Unity vs. Unreal



Recommended reading: Unreal Engine 4 For Unity Developers
https://docs.unrealengine.com/en-us/GettingStarted/FromUnity

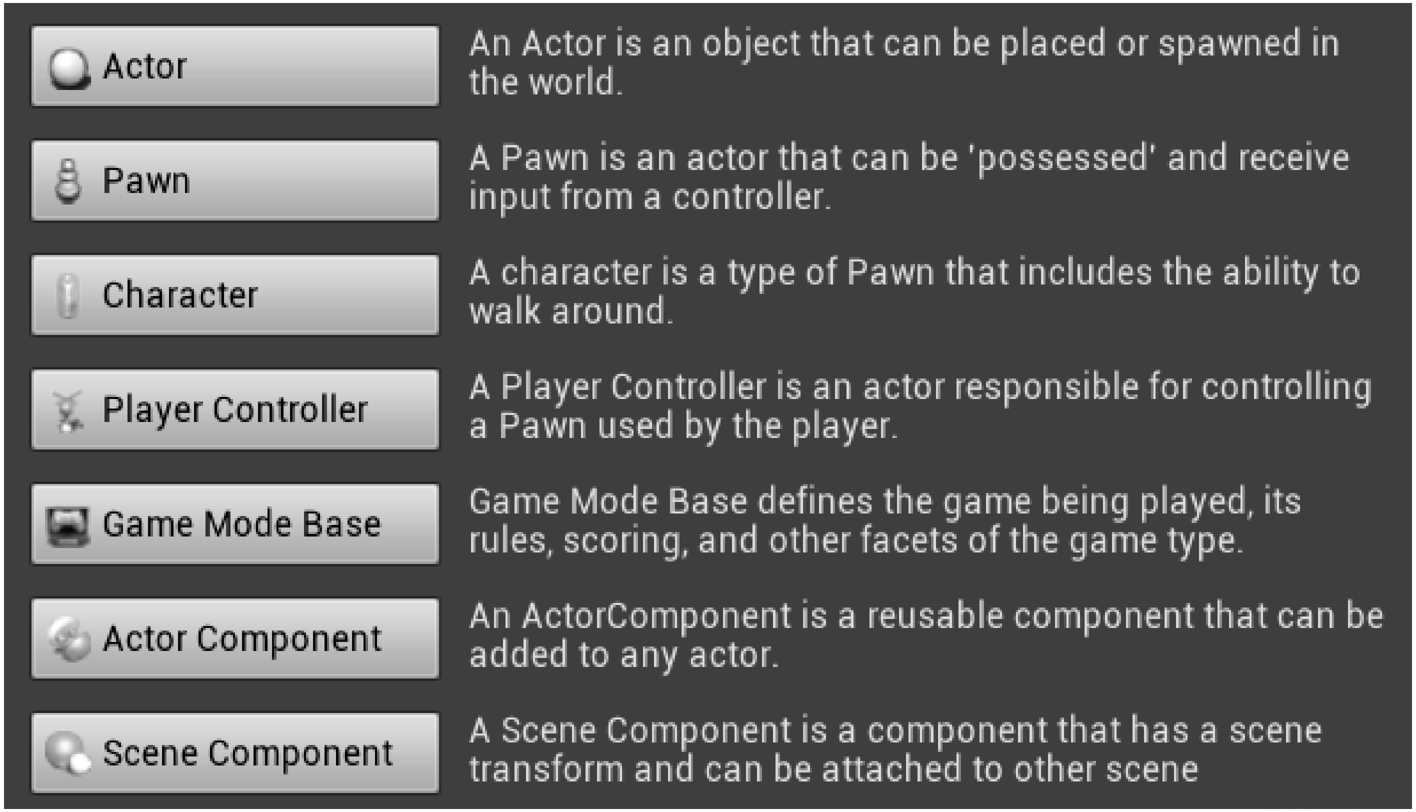
Glossary – Unity vs. Unreal

Category	**Unity**	**Unreal Engine**
Gameplay Types	Component	Component
	GameObject	Actor, Pawn
	Prefab	Blueprint Class
Editor UI	Hierarchy Panel	World Outliner
	Inspector	Details Panel
	Project Browser	Content Browser
	Scene View	Viewport
Meshes	Mesh	Static Mesh
	Skinned Mesh	Skeletal Mesh
Materials	Shader	Material
	Material	Material Instance

Glossary – Unity vs. Unreal

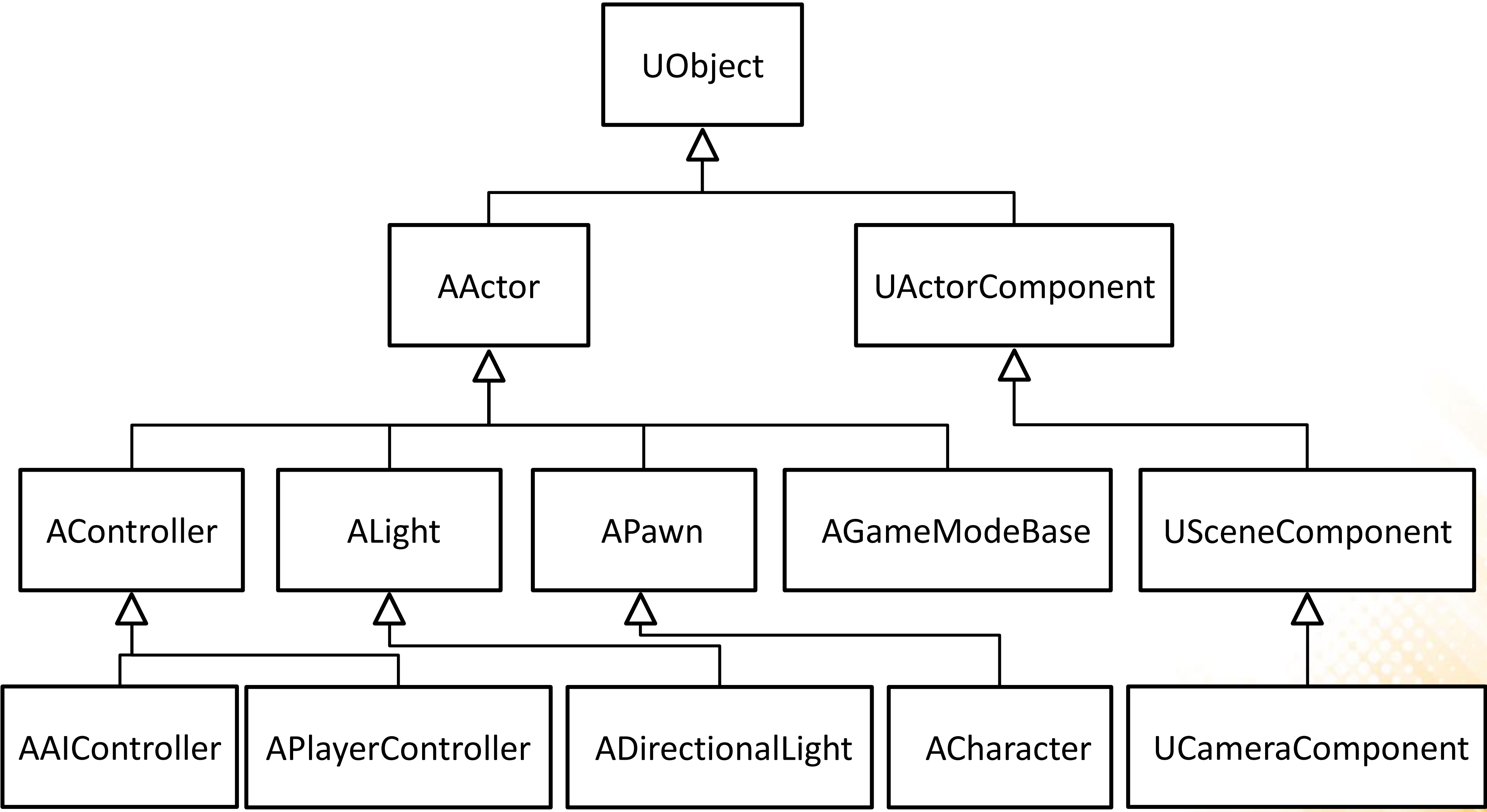
Category	Unity	Unreal Engine
Effects	Particle Effect	Effect, Particle, Cascade
Game UI	UI	UMG
Animation	Animation	Skeletal Animation System
2D	Sprite Editor	Paper2D
Programming	C#	C++
	Script	Blueprint
Physics	Raycast	Line Trace, Shape Trace
	Rigid Body	Collision, Physics

Unreal Engine – Main Classes



Actor	An Actor is an object that can be placed or spawned in the world.
Pawn	A Pawn is an actor that can be 'possessed' and receive input from a controller.
Character	A character is a type of Pawn that includes the ability to walk around.
Player Controller	A Player Controller is an actor responsible for controlling a Pawn used by the player.
Game Mode Base	Game Mode Base defines the game being played, its rules, scoring, and other facets of the game type.
Actor Component	An ActorComponent is a reusable component that can be added to any actor.
Scene Component	A Scene Component is a component that has a scene transform and can be attached to other scene

Unreal Engine – Main Classes

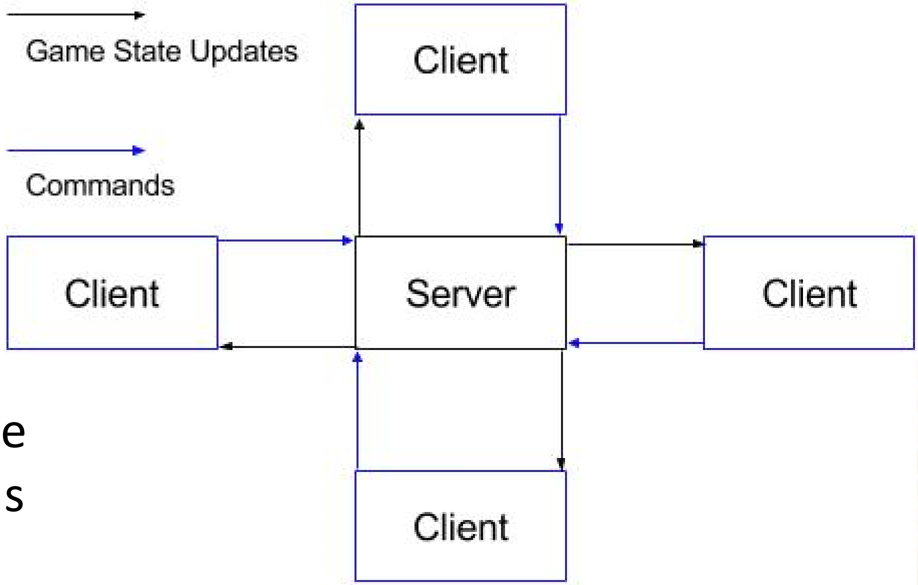


Multiplayer in Unreal Engine

- The Unreal Engine is built with multiplayer gaming in mind.
  - Is very easy to extend a single player experience to multiplayer.
  - Even single-player games, use the networking architecture.
- The engine is based on a **client-server model**.
  - The server is authoritative and makes sure all connected clients are continually updated.
- Actors are the main element that the server uses to keep clients up to date.
  - The server sends information about the actors to clients;
  - Clients have an approximation of each actor and the server maintains the authoritative version.

Multiplayer in Unreal Engine

- **The server is in charge of driving the flow of gameplay.**
  - It handles network connections, notifies clients when gameplay starts and ends, when is time to travel to a new map, along with actor replication updates.
  - Only the server contains a valid copy of the GameMode actor. Clients contain only an approximate copy of the actors, and can use it as a reference to know the general state of the game.



Multiplayer in Unreal Engine

- **Network Modes:**
  - **Standalone**: the server runs on a local machine and not accept clients from remote machines. Is used for single-player games.

  - **Dedicated Server**: the server has no local players and can run more efficiently by discarding sound, graphics, user input, and other player-oriented features. Is used for multiplayer games hosted on a trusted and reliable server where high-performing are needed.

  - **ListenServer**: is a server that hosts a local player, but is open to connections from remote players as well. Is good for games where users can set up and play their own games without a third-party server.

  - **Client**: the machine is a client that can connect to a dedicated or listen server, and therefore will not run server-side logic.

Multiplayer in Unreal Engine

- The core element of the networking process in Unreal Engine is **Actor Replication**.
  - The server maintains a list of actors and updates the client periodically so that the client can have a close approximation of each actor (that is marked to be replicated).

- Actors are updated in two main ways:
  - Property updates;
  - RPCs (Remote Procedure Calls).

- Properties are replicated automatically (any time they change) and RPCs are only replicated when executed.

Property Replication

- Example of property to be replicated: actor's health.

- Each actor maintains a list of properties that can be marked for replication to clients.
  - Whenever the value of the variable changes on the server side, the server sends the client the updated value.
  - Property updates only come from the server (i.e.: the client will never send property updates to the server).
  - Some properties replicate by default (e.g.: Location, Rotation, etc.).

- Actor property replication is reliable.

Property Replication

- **Replicate a property:**
  1. Set the replicated keyword:

```
UPROPERTY(replicated)
float health;
```

  2. Implement the GetLifetimeReplicatedProps function:

```
void MyClass::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>&
                                        OutLifetimeProps) const{
  Super::GetLifetimeReplicatedProps(OutLifetimeProps);
  DOREPLIFETIME(MyClass, health);
}
```

Default replication rule: replicates to all clients

  3. Enable replication in the constructor method:

```
SetReplicates(true);
```

Remote Procedure Calls (RPCs)

- Example of RPC: a function to spawn an explosion to each client at a certain location.

- RPCs are functions that are called locally, but executed remotely on another machine.
  - Primary use: to do unreliable gameplay events that are temporary or cosmetic in nature.
  - E.g.: play sounds, spawn particles, or do other temporary effects that are not crucial to the Actor functioning.

- By default, RPCs are unreliable. To be reliable, a especial keyword (`Reliable`) must be used in the definition of the RPC.

Remote Procedure Calls (RPCs)

- **Defining an RPC:**
  - To declare a function as an RPC that will be called on the server, but executed on the client:

```
UFUNCTION(Client)
void ClientRPCFunction();
```

  - To declare a function as an RPC that will be called on the client, but executed on the server:

```
UFUNCTION(Server)
void ServerRPCFunction();
```

  - To declare a function as an RPC that will be called from the server, and then executed on the server and on all connected clients:

```
UFUNCTION(NetMulticast)
void MulticastRPCFunction();
```

RPC Validation

- The validation function for an RPC allows the detection of bad parameters or cheating:
  - It can notify the system to disconnect the client who initiated the call.
- Example:

```
UFUNCTION(Server, WithValidation)
void SomeRPCFunction(int32 AddHealth);
```

```
bool SomeRPCFunction_Validate(int32 AddHealth){
  if (AddHealth > MAX_ADD_HEALTH){
    return false;
  }
  return true;
}

void SomeRPCFunction_Implementation(int32 AddHealth){
  Health += AddHealth;
}
```

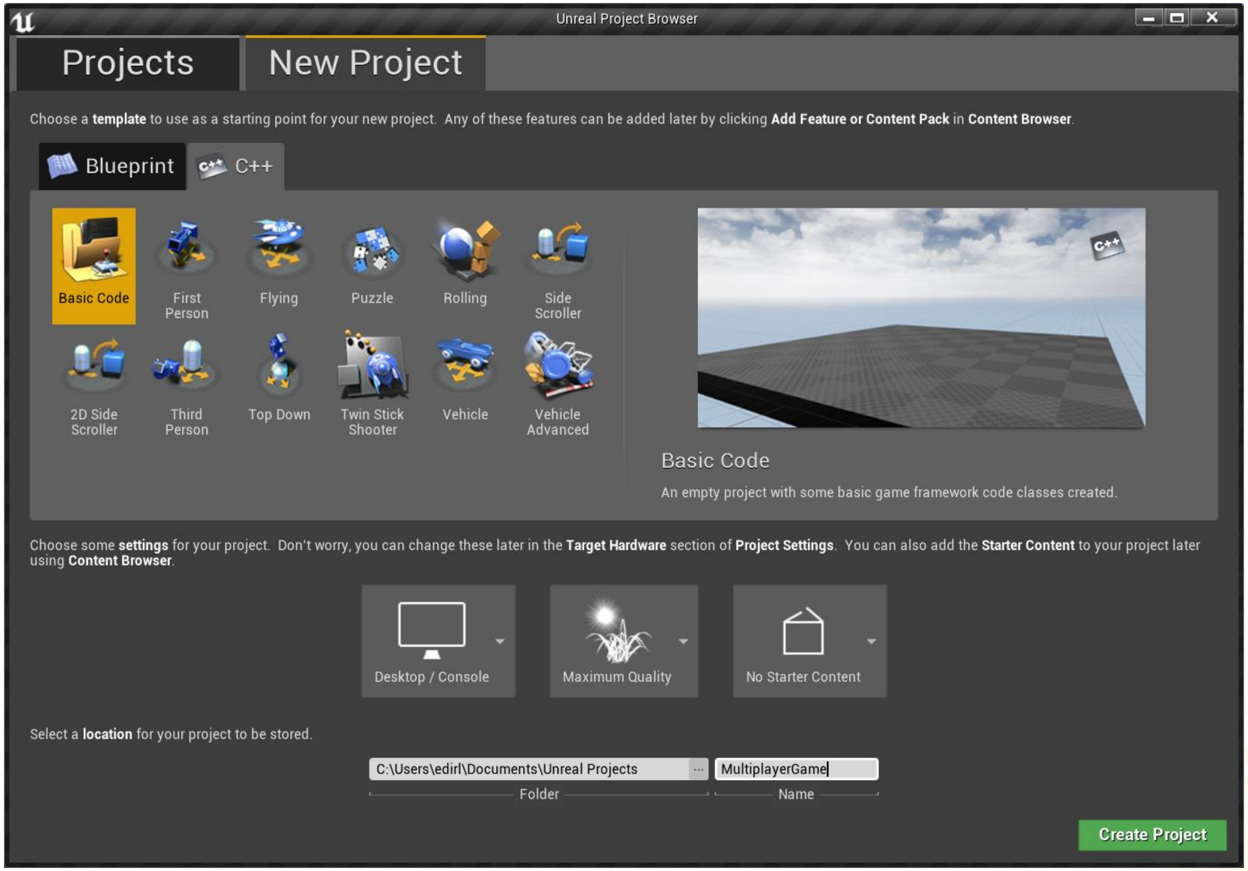
Prototype Game

- **Concept:** a cooperative multiplayer game where players must collect all coins and then go to a specific location to complete the level.

- **Gameplay elements:**
  - Player character (walk, jump, crouch);
  - Collectible coins: after collecting all coins, the player must go the "level complete" area to finish the level.
  - Enemies (zombies): patrol the level and attack the player. If the enemy touches the player, is game over;
  - GUI messages: number of remaining coins, game over, and level completed messages;

Multiplayer Game

- Create a new C++ empty project:



Choose a **template** to use as a starting point for your new project. Any of these features can be added later by clicking **Add Feature or Content Pack** in **Content Browser**.

Basic Code

An empty project with some basic game framework code classes created.

Choose some **settings** for your project. Don't worry, you can change these later in the **Target Hardware** section of **Project Settings**. You can also add the **Starter Content** to your project later using **Content Browser**.

Desktop / Console

Maximum Quality

No Starter Content

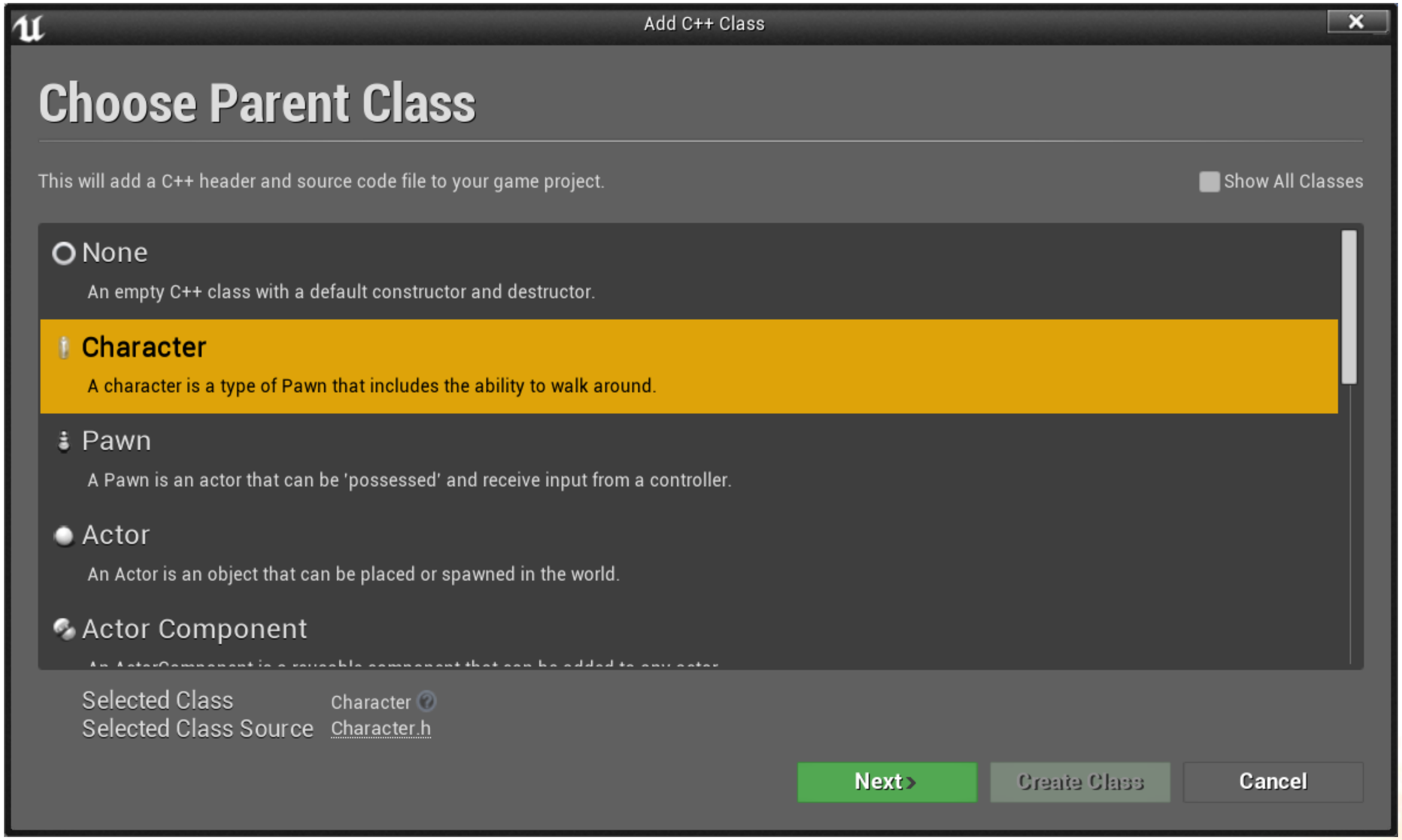
Select a **location** for your project to be stored.

Folder: C:\Users\edirl\Documents\Unreal Projects

Name: MultiplayerGame

Multiplayer Game

- Create a new character class:



Choose Parent Class

This will add a C++ header and source code file to your game project.

□ Show All Classes

○ None
An empty C++ class with a default constructor and destructor.

Character
A character is a type of Pawn that includes the ability to walk around.

Pawn
A Pawn is an actor that can be 'possessed' and receive input from a controller.

Actor
An Actor is an object that can be placed or spawned in the world.

Actor Component
An ActorComponent is a reusable component that can be added to any actor.

Selected Class: Character
Selected Class Source: Character.h

Multiplayer Game

- Implement the character movement:

MyCharacter.h

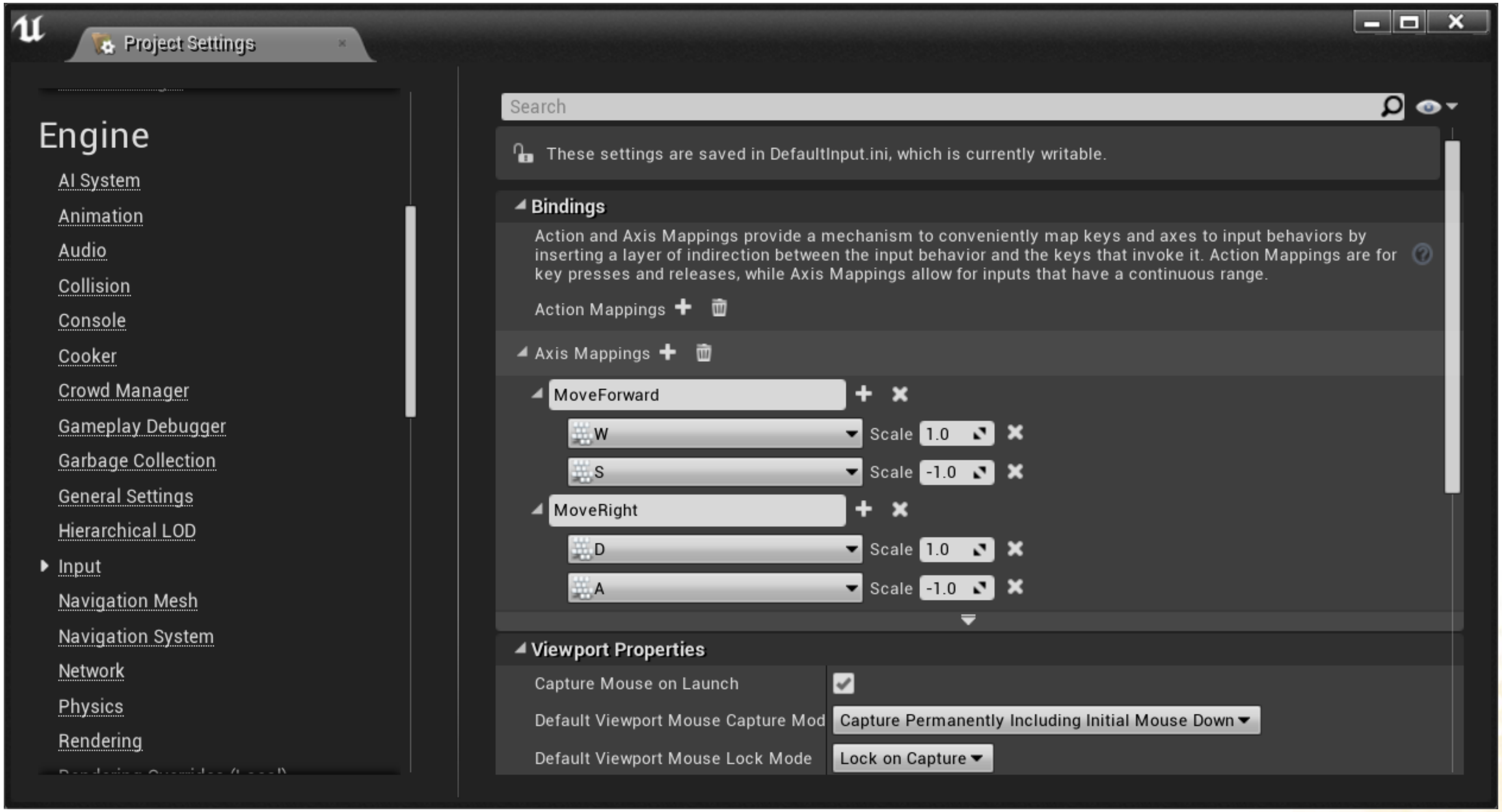
```
protected:
  void MoveForward(float value);
  void MoveRight(float value);
```

MyCharacter.cpp

```
void AMyCharacter::MoveForward(float value){
  AddMovementInput(GetActorForwardVector(), value);
}
void AMyCharacter::MoveRight(float value){
  AddMovementInput(GetActorRightVector(), value);
}
void AMyCharacter::SetupPlayerInputComponent(UInputComponent*
                                             PlayerInputComponent){
  Super::SetupPlayerInputComponent(PlayerInputComponent);
  PlayerInputComponent->BindAxis("MoveForward", this,
                                 &AMyCharacter::MoveForward);
  PlayerInputComponent->BindAxis("MoveRight", this,
                                 &AMyCharacter::MoveRight);
}
```

Multiplayer Game

- Setup the axis keys in the project settings:



These settings are saved in DefaultInput.ini, which is currently writable.

Bindings

Action and Axis Mappings provide a mechanism to conveniently map keys and axes to input behaviors by inserting a layer of indirection between the input behavior and the keys that invoke it. Action Mappings are for key presses and releases, while Axis Mappings allow for inputs that have a continuous range.

Action Mappings

Axis Mappings

MoveForward
W Scale 1.0
S Scale -1.0

MoveRight
D Scale 1.0
A Scale -1.0

Viewport Properties

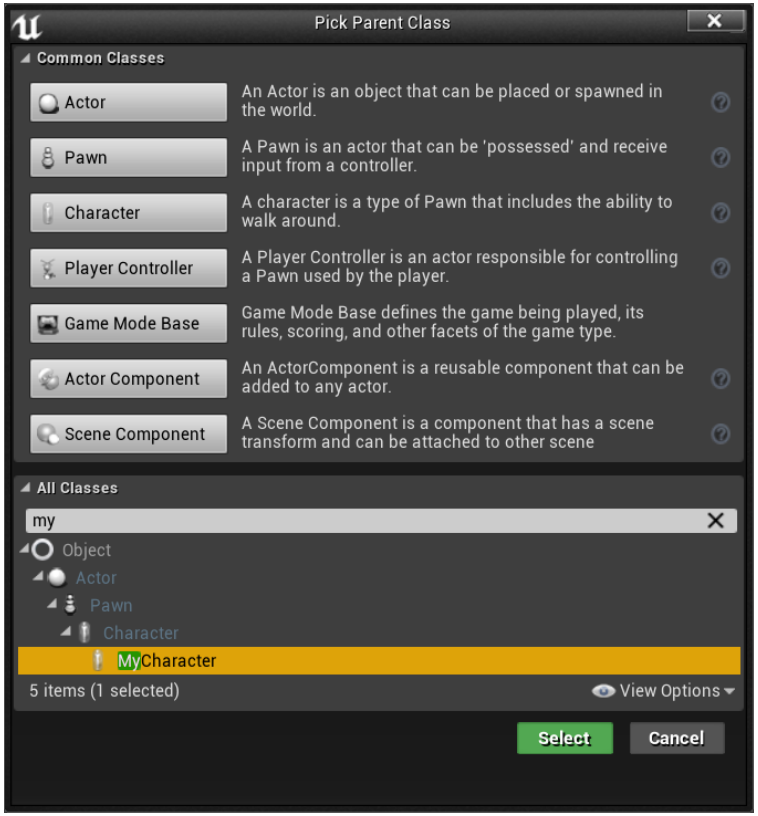
Capture Mouse on Launch ☑

Default Viewport Mouse Capture Mod: Capture Permanently Including Initial Mouse Down

Default Viewport Mouse Lock Mode: Lock on Capture

Multiplayer Game

- Create a blueprint for the character class and test the player movement in the level:



Actor — An Actor is an object that can be placed or spawned in the world.

Pawn — A Pawn is an actor that can be 'possessed' and receive input from a controller.

Character — A character is a type of Pawn that includes the ability to walk around.

Player Controller — A Player Controller is an actor responsible for controlling a Pawn used by the player.

Game Mode Base — Game Mode Base defines the game being played, its rules, scoring, and other facets of the game type.

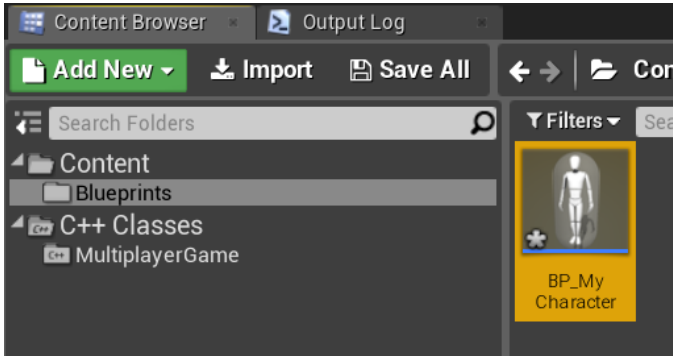
Actor Component — An ActorComponent is a reusable component that can be added to any actor.

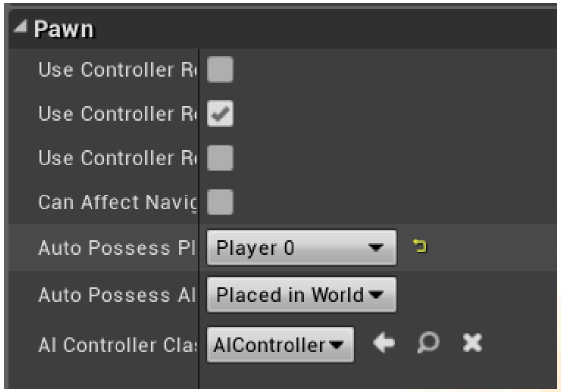
Scene Component — A Scene Component is a component that has a scene transform and can be attached to other scene

my

- Object
  - Actor
    - Pawn
      - Character
        - MyCharacter

5 items (1 selected)





Use Controller R ☐

Use Controller R ☑

Use Controller R ☐

Can Affect Navig ☐

Auto Possess Pl: Player 0

Auto Possess AI: Placed in World

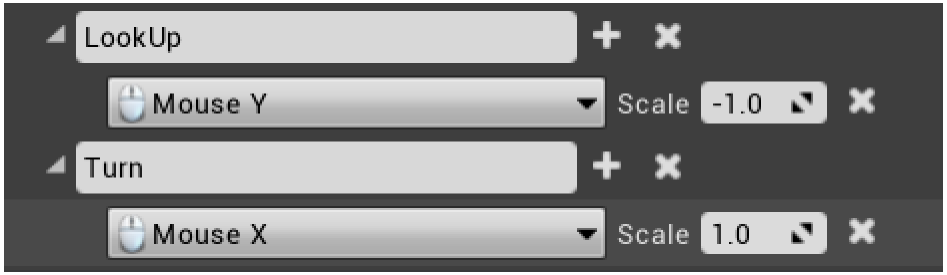
AI Controller Cla: AIController

Multiplayer Game

- Implement the camera movement and setup the axis keys:

MyCharacter.cpp

```
void AMyCharacter::SetupPlayerInputComponent(UInputComponent*
                                            PlayerInputComponent){
  ...

  PlayerInputComponent->BindAxis("LookUp", this,
                           &AMyCharacter::AddControllerPitchInput);
  PlayerInputComponent->BindAxis("Turn", this,
                           &AMyCharacter::AddControllerYawInput);
}
```



LookUp
Mouse Y Scale -1.0
Turn
Mouse X Scale 1.0

Multiplayer Game

- Create a 3rd person camera and a spring arm components in the character class:

MyCharacter.h

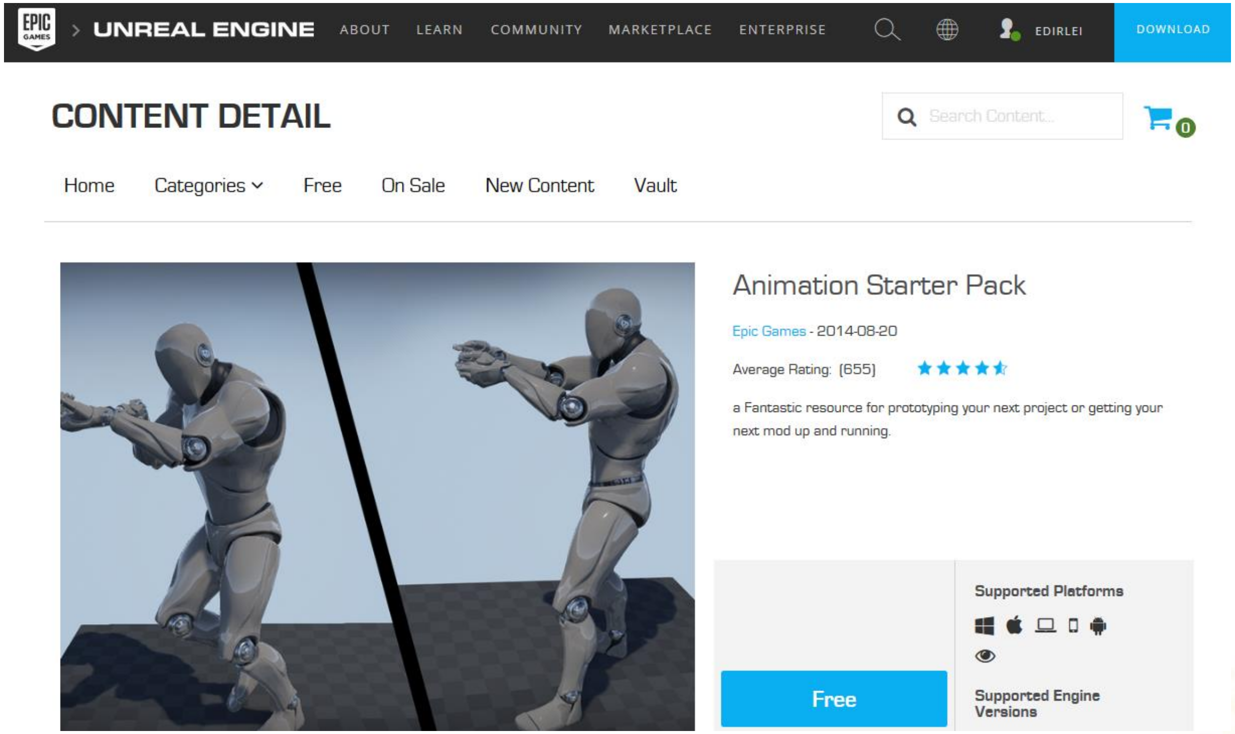
```
protected:
  UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Components")
  class UCameraComponent * CameraComponent;

  UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Components")
  class USpringArmComponent * SpringArmComponent;
```

MyCharacter.cpp

```
AMyCharacter::AMyCharacter(){
  PrimaryActorTick.bCanEverTick = true;
  SpringArmComponent = CreateDefaultSubobject<USpringArmComponent>
                                             ("SpringArm Component");
  SpringArmComponent->bUsePawnControlRotation = true;
  SpringArmComponent->SetupAttachment(RootComponent);

  CameraComponent = CreateDefaultSubobject<UCameraComponent>
                                          ("Camera Component");
  CameraComponent->SetupAttachment(SpringArmComponent);
}
```

Multiplayer Game

- Download and import the Animation Starter Pack:
  - https://www.unrealengine.com/marketplace/animation-starter-pack



CONTENT DETAIL

Animation Starter Pack

Epic Games - 2014-08-20

Average Rating: (655) ★★★★½

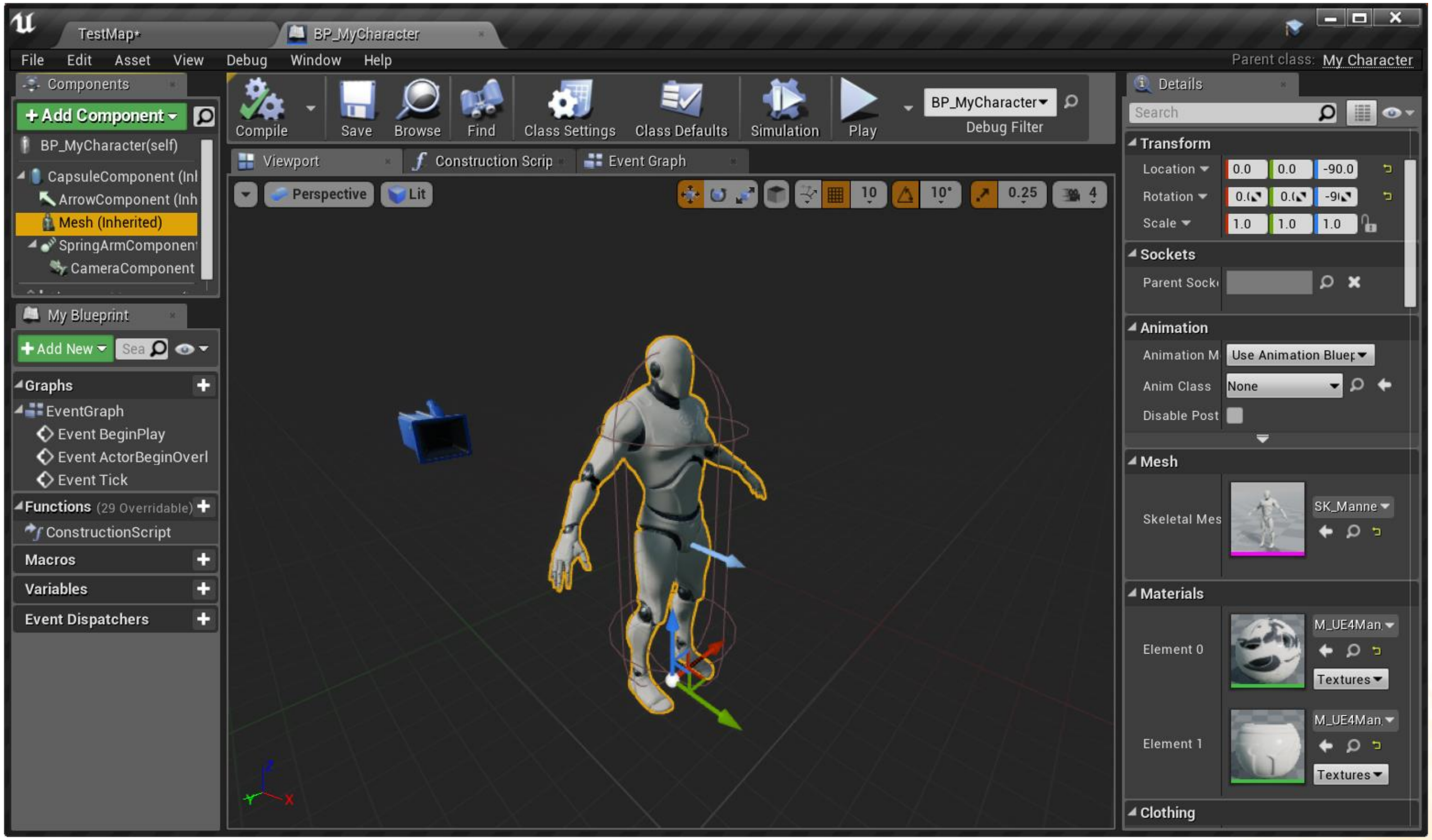
a Fantastic resource for prototyping your next project or getting your next mod up and running.

Supported Platforms

Supported Engine Versions

Multiplayer Game

- Add and setup the model mesh in the character blueprint:



Parent class: My Character

Location 0.0 0.0 -90.0

Rotation 0.(0.(-9(

Scale 1.0 1.0 1.0

Parent Sock

Animation M Use Animation Bluep

Anim Class None

Disable Post

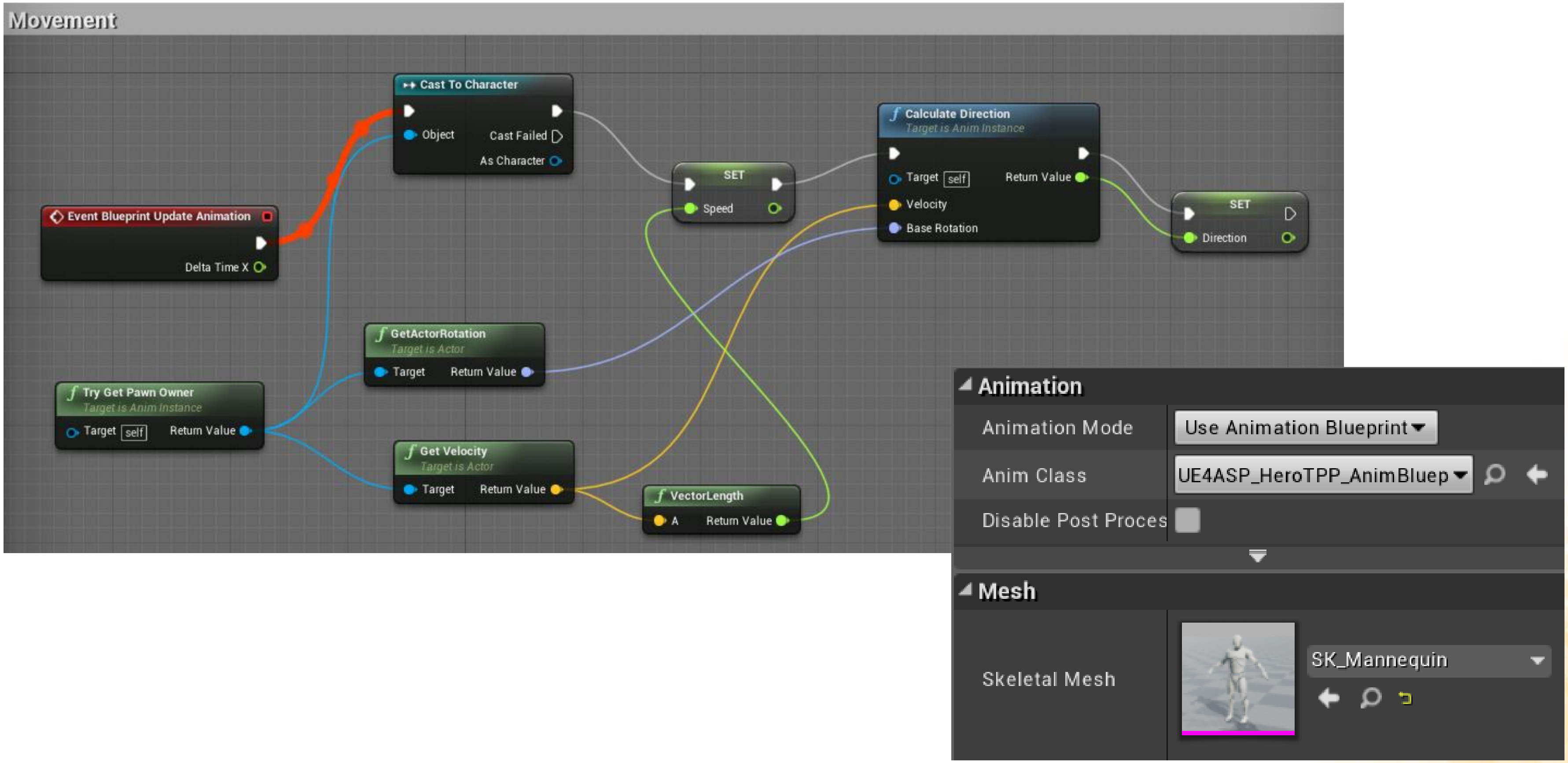
Skeletal Mes SK_Manne

Element 0 M_UE4Man

Element 1 M_UE4Man

Multiplayer Game

- Clear the animation blueprint (UE4ASP_HeroTPP_AnimBlueprint) and setup the animation in the character blueprint.



Animation

Animation Mode: Use Animation Blueprint

Anim Class: UE4ASP_HeroTPP_AnimBluep

Disable Post Proces: □

Mesh

Skeletal Mesh: SK_Mannequin

Multiplayer Game

- Implement the crouch action:

MyCharacter.h

```
protected:
  ...
  void BeginCrouch();
  void EndCrouch();
```

MyCharacter.cpp

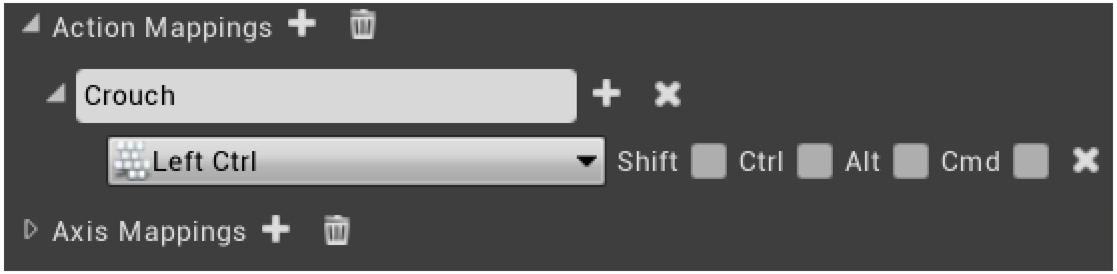
```
void AMyCharacter::BeginCrouch(){
  Crouch();
}

void AMyCharacter::EndCrouch(){
  UnCrouch();
}

AMyCharacter::AMyCharacter(){
  ...
  GetMovementComponent()->GetNavAgentPropertiesRef().bCanCrouch = true;
}
```

Multiplayer Game

- Implement the crouch action:

MyCharacter.cpp

```
void AMyCharacter::SetupPlayerInputComponent(UInputComponent*
                                             PlayerInputComponent){
  ...

  PlayerInputComponent->BindAction("Crouch", IE_Pressed, this,
                                   &AMyCharacter::BeginCrouch);
  PlayerInputComponent->BindAction("Crouch", IE_Released, this,
                                   &AMyCharacter::EndCrouch);
}
```



Action Mappings

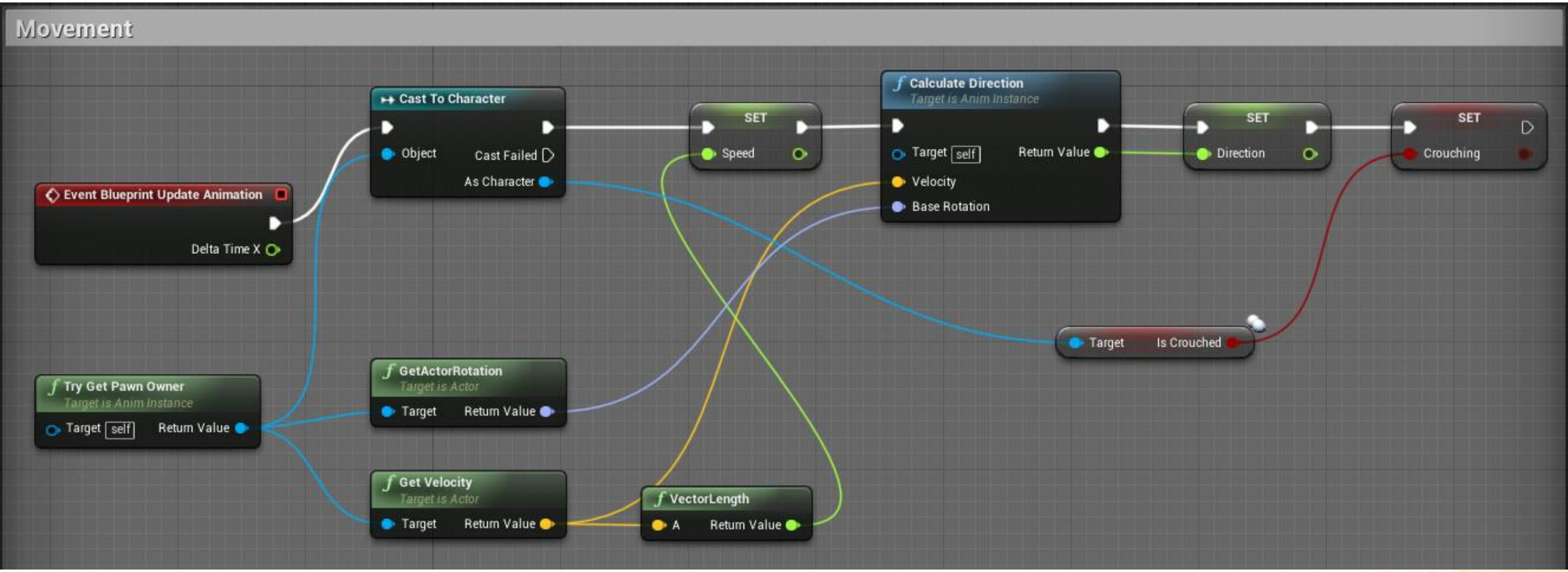
Crouch

Left Ctrl   Shift □ Ctrl □ Alt □ Cmd □

Axis Mappings

Multiplayer Game

- Set the crouch variable in the animation blueprint:

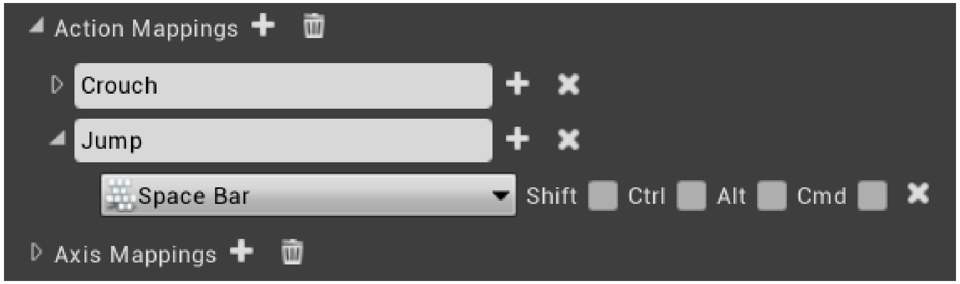


Multiplayer Game

- Implement the jump action:

MyCharacter.cpp

```
void AMyCharacter::SetupPlayerInputComponent(UInputComponent*
                                             PlayerInputComponent){
  ...

  PlayerInputComponent->BindAction("Jump", IE_Pressed, this,
                                   &AMyCharacter::Jump);
}
```



Action Mappings

Crouch

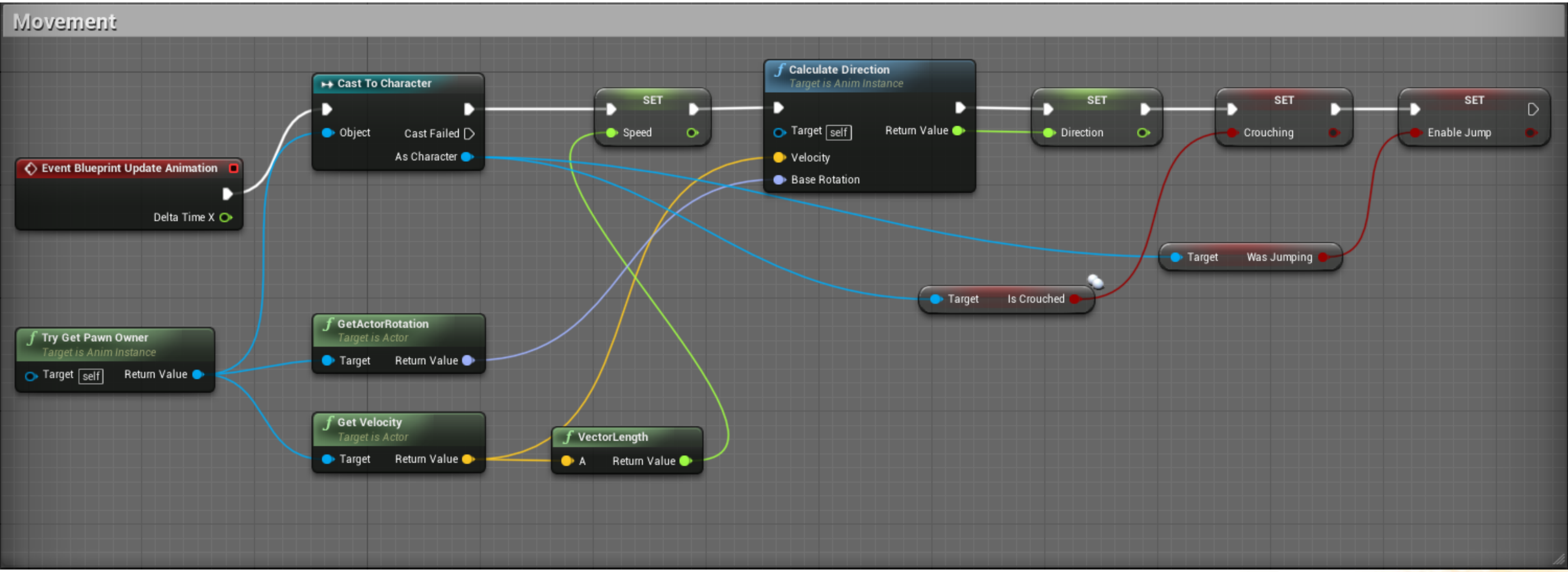
Jump

Space Bar Shift □ Ctrl □ Alt □ Cmd □

Axis Mappings

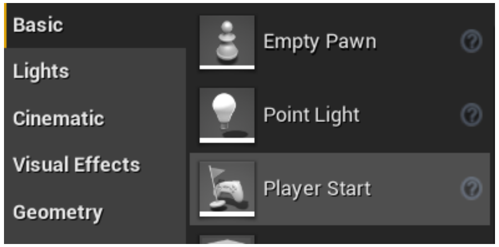
Multiplayer Game

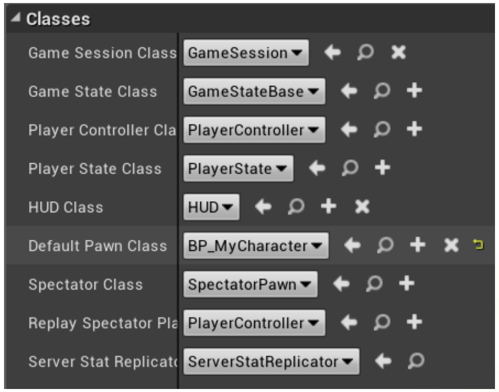
- Implement the jump action:



Multiplayer Game

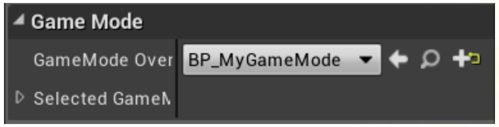
- Setup the game to be played in multiplayer:
  1. Delete the character from the map and add two or more “Player Start” actors to the map.
  2. Create a new Game Mode blueprint and set our character blueprint as default pawn class.
  3. Set the new Game Mode as the Game Mode for the map in the World Settings.
  4. Disable the auto possess option in the character blueprint.





Classes

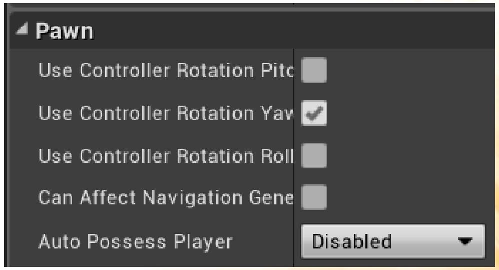
Game Session Class	GameSession
Game State Class	GameStateBase
Player Controller Cla	PlayerController
Player State Class	PlayerState
HUD Class	HUD
Default Pawn Class	BP_MyCharacter
Spectator Class	SpectatorPawn
Replay Spectator Pla	PlayerController
Server Stat Replicat	ServerStatReplicator



Game Mode

GameMode Over: BP_MyGameMode

Selected GameM



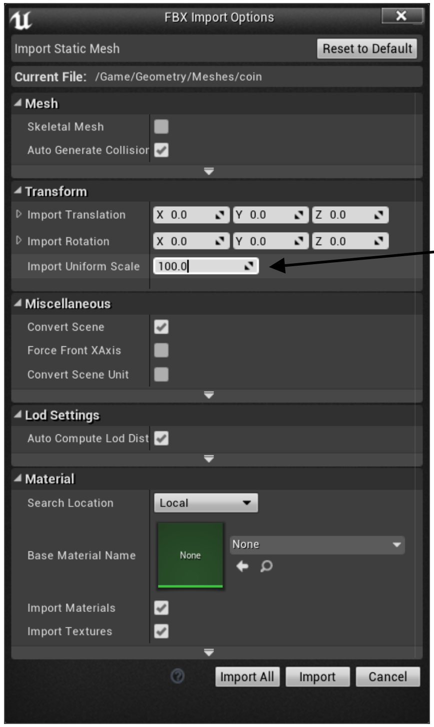
Pawn

Use Controller Rotation Pitc	☐
Use Controller Rotation Yav	☑
Use Controller Rotation Roll	☐
Can Affect Navigation Gene	☐
Auto Possess Player	Disabled

Collectible Coin

- **Low poly coin model:**
  - http://www.inf.puc-rio.br/~elima/dp/coin.fbx

- **Importing the FBX model:** drag and drop



FBX Import Options

Import Static Mesh

Current File: /Game/Geometry/Meshes/coin

Mesh

Skeletal Mesh □

Auto Generate Collisior ☑

Transform

Import Translation X 0.0 Y 0.0 Z 0.0

Import Rotation X 0.0 Y 0.0 Z 0.0

Import Uniform Scale 100.0

Miscellaneous

Convert Scene ☑

Force Front XAxis □

Convert Scene Unit □

Lod Settings

Auto Compute Lod Dist ☑

Material

Search Location Local

Base Material Name None None

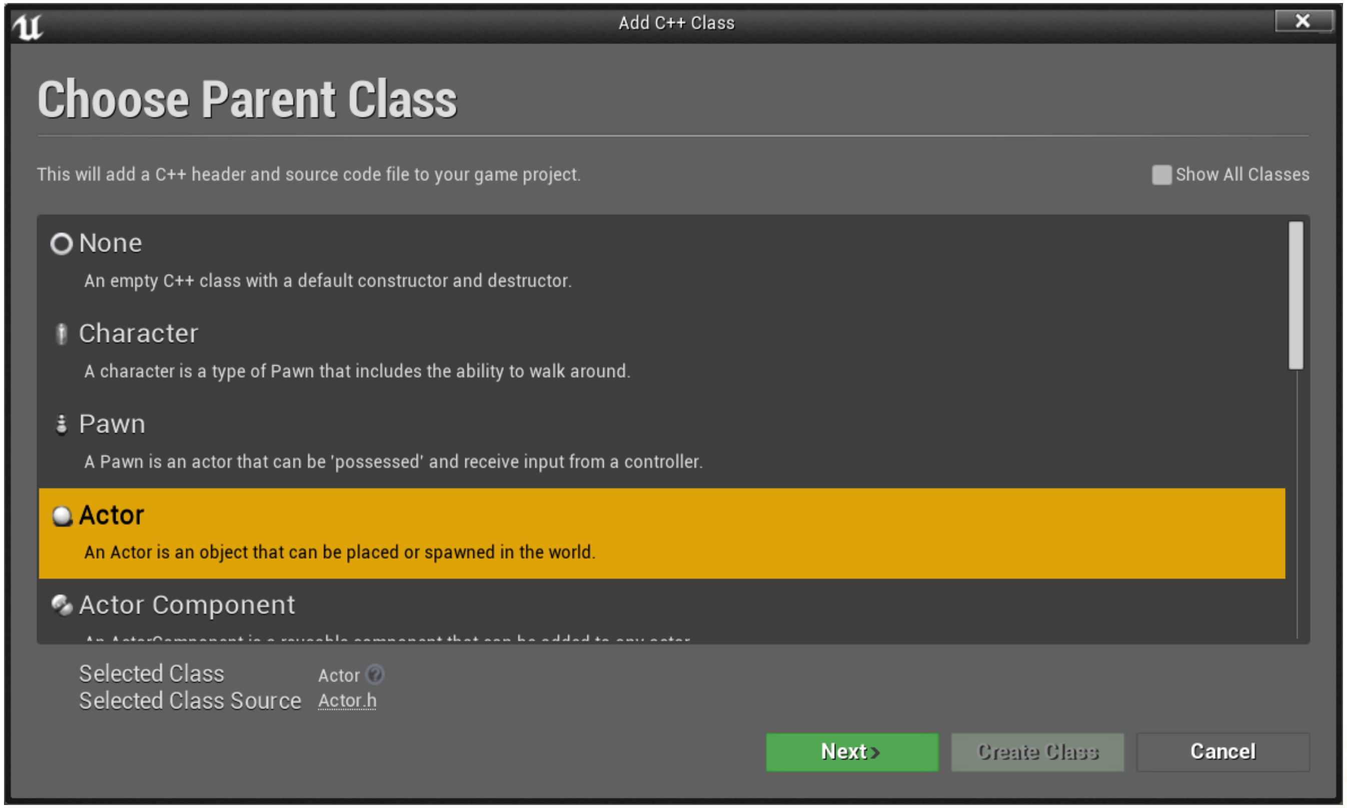
Import Materials ☑

Import Textures ☑

Import Uniform Scale = 100.0

Collectible Coin

- **Create a new C++ class:** CollectibleCoin



Choose Parent Class

This will add a C++ header and source code file to your game project.

□ Show All Classes

○ None

An empty C++ class with a default constructor and destructor.

Character

A character is a type of Pawn that includes the ability to walk around.

Pawn

A Pawn is an actor that can be 'possessed' and receive input from a controller.

● Actor

An Actor is an object that can be placed or spawned in the world.

Actor Component

An ActorComponent is a reusable component that can be added to any actor.

Selected Class Actor

Selected Class Source Actor.h

CollectibleCoin.h

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "CollectibleCoin.generated.h"

UCLASS()
class MYFIRSTGAME_API ACollectibleCoin : public AActor
{
	GENERATED_BODY()
public:
	// Sets default values for this actor's properties
	ACollectibleCoin();

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

public:
	// Called every frame
	virtual void Tick(float DeltaTime) override;
};
```

CollectibleCoin.cpp

```cpp
#include "CollectibleCoin.h"

// Sets default values
ACollectibleCoin::ACollectibleCoin()
{
  // Set this actor to call Tick() every frame.
  PrimaryActorTick.bCanEverTick = true;
}

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

// Called every frame
void ACollectibleCoin::Tick(float DeltaTime)
{
  Super::Tick(DeltaTime);
}
```

Collectible Coin

- Next step: define the structure of the collectible coin:

CollectibleCoin.h

```
...

#include "Components/SphereComponent.h"
#include "CollectibleCoin.generated.h"

...

protected:

  UPROPERTY(VisibleAnywhere, Category = "Components")
  UStaticMeshComponent* MeshComponent;

  UPROPERTY(VisibleAnywhere, Category = "Components")
  USphereComponent* SphereComponent;

  virtual void BeginPlay() override;
  ...
```

Collectible Coin

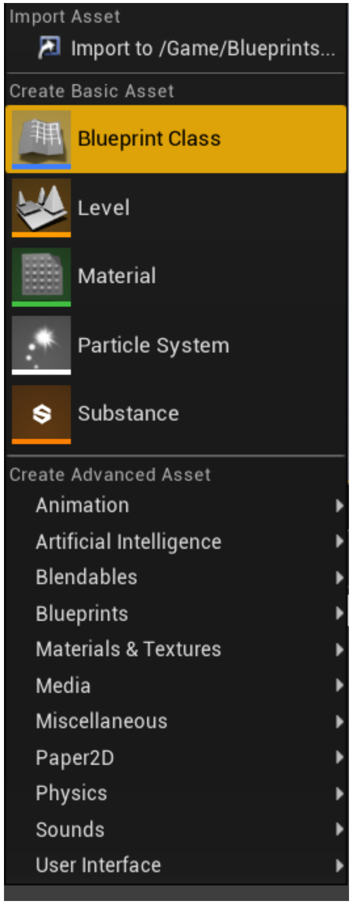
- Next step: define the structure of the collectible coin:

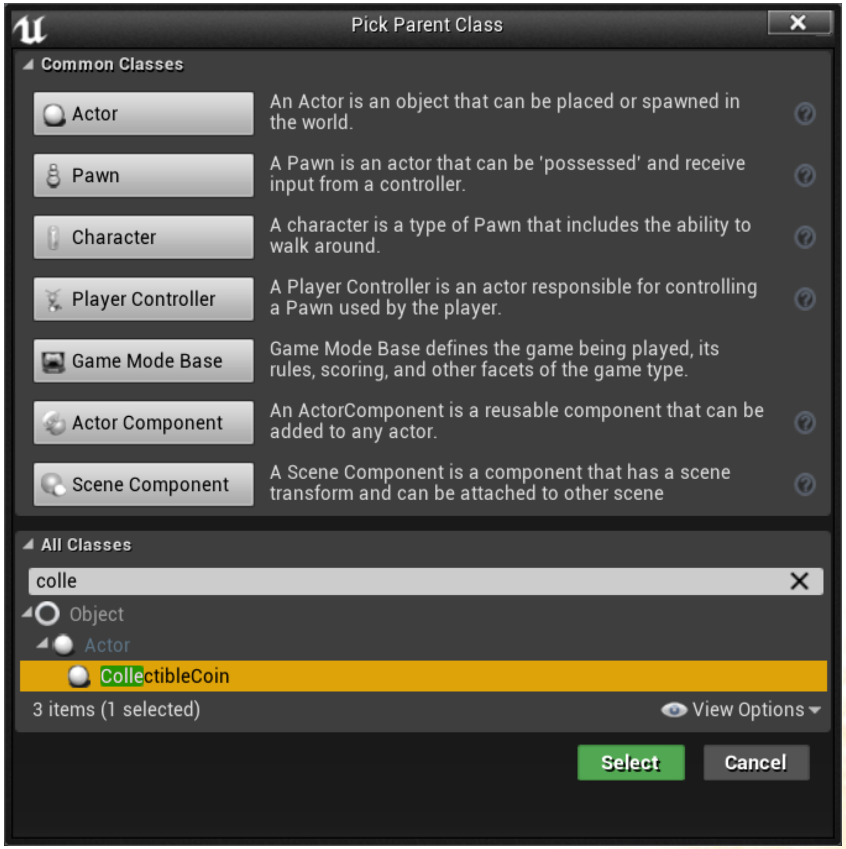
CollectibleCoin.cpp

```
...

ACollectibleCoin::ACollectibleCoin()
{
  PrimaryActorTick.bCanEverTick = true;

  MeshComponent = CreateDefaultSubobject<UStaticMeshComponent>
                  ("Mesh Component");
  RootComponent = MeshComponent;
  SphereComponent = CreateDefaultSubobject<USphereComponent>
                    ("Sphere Component");
  SphereComponent->SetupAttachment(MeshComponent);
}

...
```

Collectible Coin

- **Next step:** create a Blueprint Class for the collectible coin:





Actor — An Actor is an object that can be placed or spawned in the world.

Pawn — A Pawn is an actor that can be 'possessed' and receive input from a controller.

Character — A character is a type of Pawn that includes the ability to walk around.

Player Controller — A Player Controller is an actor responsible for controlling a Pawn used by the player.

Game Mode Base — Game Mode Base defines the game being played, its rules, scoring, and other facets of the game type.

Actor Component — An ActorComponent is a reusable component that can be added to any actor.

Scene Component — A Scene Component is a component that has a scene transform and can be attached to other scene

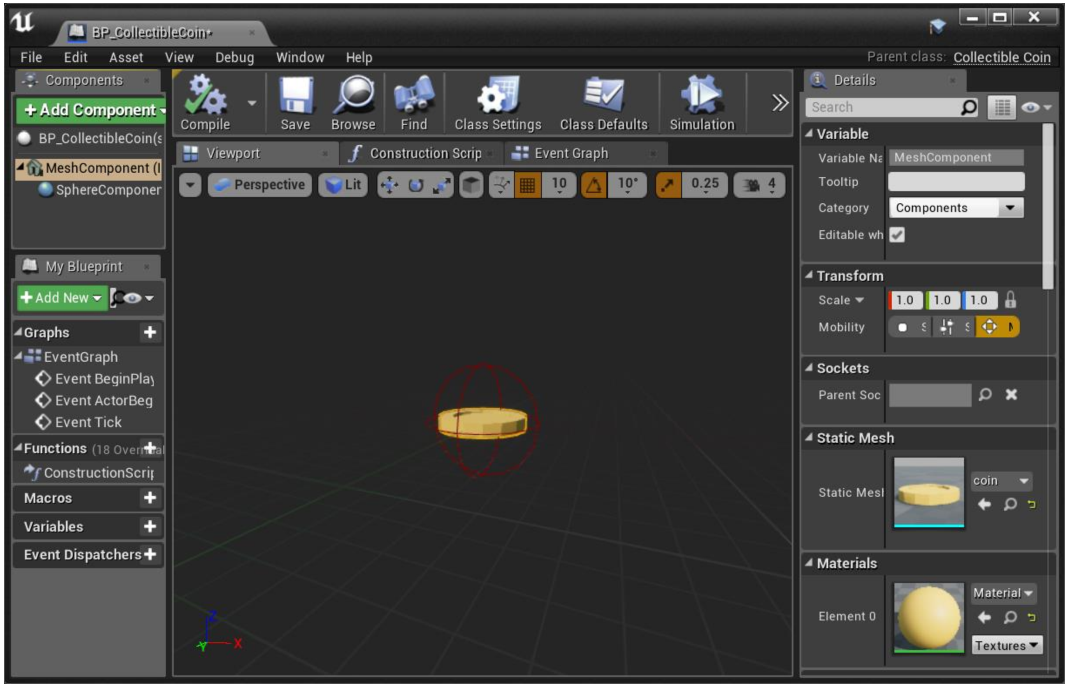
colle

Object
Actor
CollectibleCoin

3 items (1 selected)

Collectible Coin

- In the Blueprint editor, select the mesh of the coin:



Variable Na: MeshComponent
Tooltip:
Category: Components
Editable wh: ☑
Scale: 1.0 1.0 1.0
Mobility
Parent Soc:
Static Mesh: coin
Element 0: Material

- Then, compile the blueprint and place it in the level.

Collectible Coin

- Rotating the coin in the game:

CollectibleCoin.h

```
...

public:
  UPROPERTY(EditAnywhere, Category = "Gameplay")
  float RotationSpeed;

...
```

CollectibleCoin.cpp

```
...

void ACollectibleCoin::Tick(float DeltaTime)
{
  Super::Tick(DeltaTime);
  AddActorLocalRotation(FRotator(RotationSpeed * DeltaTime, 0, 0));
}

...
```

Collectible Coin

- Destroying the coin when the player collides:

CollectibleCoin.h

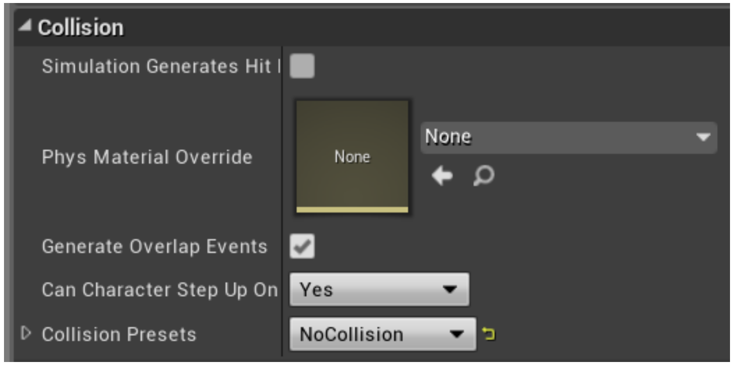
```
...

public:
  ...
  virtual void NotifyActorBeginOverlap(AActor* OtherActor) override;

...
```

CollectibleCoin.cpp

```
...

void ACollectibleCoin::NotifyActorBeginOverlap(AActor* OtherActor)
{
  Super::NotifyActorBeginOverlap(OtherActor);
  if (dynamic_cast<AMyCharacter*>(OtherActor) != nullptr){
    Destroy(this);
  }
}
...
```

Collectible Coin

- Setup the collision properties in the blueprint:
  - MeshComponent:



Collision

Simulation Generates Hit I □

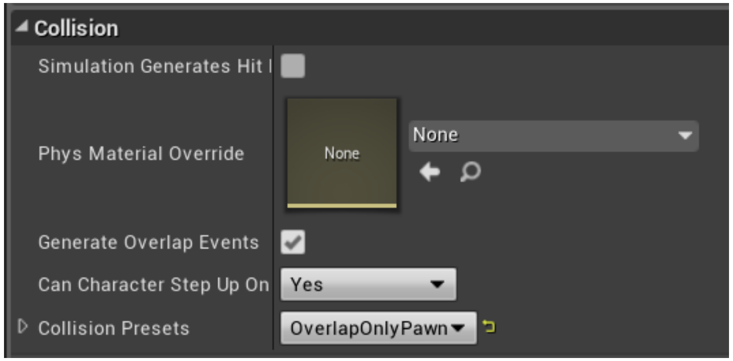
Phys Material Override: None | None

Generate Overlap Events ☑

Can Character Step Up On: Yes

Collision Presets: NoCollision

  - SphereComponent:



Collision

Simulation Generates Hit I □

Phys Material Override: None | None

Generate Overlap Events ☑

Can Character Step Up On: Yes

Collision Presets: OverlapOnlyPawn

UProperty Specifiers

Property Tag	Effect
VisibleAnywhere	Indicates that this property is visible in all property windows, but cannot be edited.
EditAnywhere	Indicates that this property can be edited by property windows, on archetypes and instances.
EditDefaultsOnly	Indicates that this property can be edited by property windows, but only on archetypes.
BlueprintReadOnly	This property can be read by Blueprints, but not modified.
BlueprintReadWrite	This property can be read or written from a Blueprint.
EditInstanceOnly	Indicates that this property can be edited by property windows, but only on instances, not on archetypes.

https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Reference/Properties/Specifiers

UFunction Specifiers

Function Specifier	Effect
BlueprintCallable	The function can be executed in a Blueprint or Level Blueprint graph.
BlueprintImplementableEvent	The function can be implemented in a Blueprint or Level Blueprint graph.
BlueprintNativeEvent	The function is designed to be overridden by a Blueprint, but also has a default native implementation.
CallInEditor	The function can be called in the Editor on selected instances via a button in the Details Panel.
ServiceRequest	The function is an RPC (Remote Procedure Call) service request.
ServiceResponse	This function is an RPC service response.

https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Reference/Properties/Specifiers

Collectible Coin

- Spawning a particle system when the player collects the coin:

CollectibleCoin.h

```
...

protected:
  ...

  UPROPERTY(EditDefaultsOnly, Category = "Effects")
  UParticleSystem* CollectEffects;

  void PlayEffects();

  ...
```

Collectible Coin

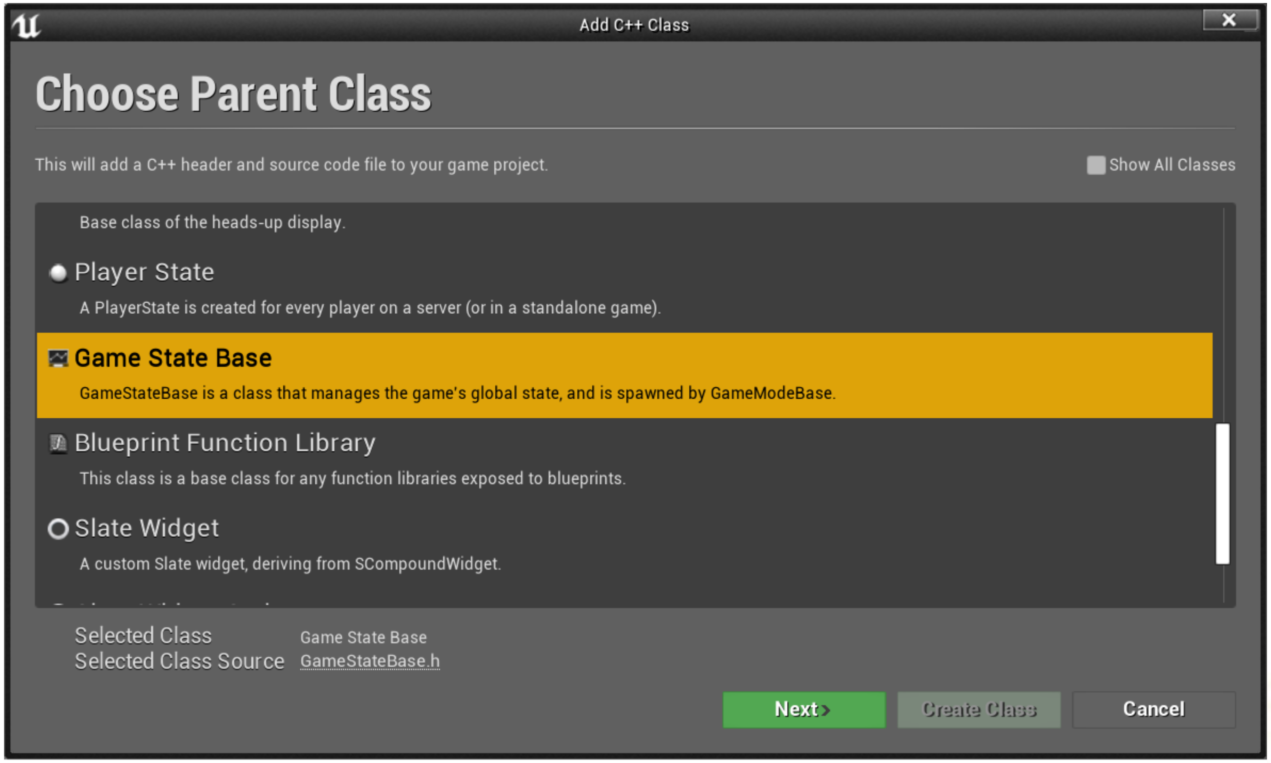
- Spawning a particle system when the player collects the coin:

CollectibleCoin.cpp

```
#include "Kismet/GameplayStatics.h"
...

void ACollectibleCoin::NotifyActorBeginOverlap(AActor* OtherActor)
{
  Super::NotifyActorBeginOverlap(OtherActor);
  if (dynamic_cast<AMyCharacter*>(OtherActor) != nullptr){
    Destroy(this);
    PlayEffects();
  }
}

void ACollectibleCoin::PlayEffects()
{
  UGameplayStatics::SpawnEmitterAtLocation(this, CollectEffects,
                                           GetActorLocation());
}
```

Collectible Coin

- Counting the number of coins remaining in the level:
  - Create new C++ Game State class:



Choose Parent Class

This will add a C++ header and source code file to your game project.

Base class of the heads-up display.

Player State

A PlayerState is created for every player on a server (or in a standalone game).

Game State Base

GameStateBase is a class that manages the game's global state, and is spawned by GameModeBase.

Blueprint Function Library

This class is a base class for any function libraries exposed to blueprints.

Slate Widget

A custom Slate widget, deriving from SCompoundWidget.

Selected Class: Game State Base

Selected Class Source: GameStateBase.h

Collectible Coin

- Counting the number of coins remaining in the level:

MyGameStateBase.h

```
public:
  UFUNCTION(BlueprintCallable)
  int CountCoinsInLevel();
```

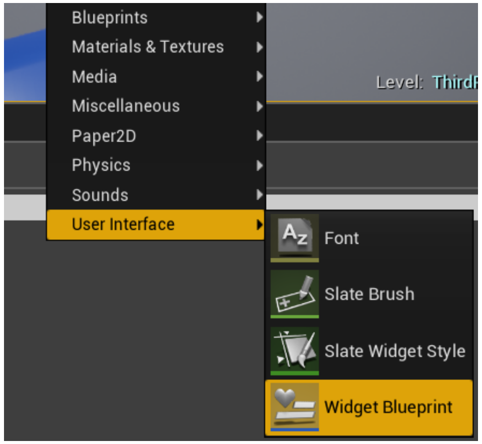
MyGameStateBase.cpp

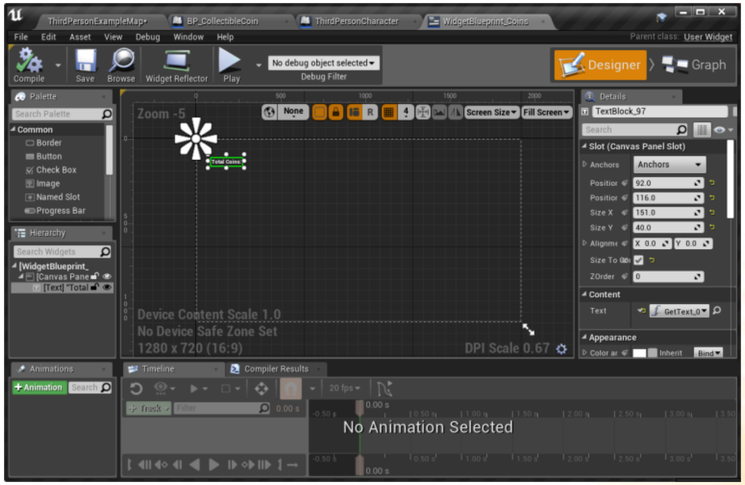
```
int AMyGameStateBase::CountCoinsInLevel()
{
  TArray<AActor*> FoundCoins;
  UGameplayStatics::GetAllActorsOfClass(GetWorld(),
                       ACollectibleCoin::StaticClass(), FoundCoins);
  return FoundCoins.Num();
}
```

Collectible Coin

- Displaying the information in the game UI with a Widget Blueprint:
  1. First, show number of coins remaining in the level;
  2. After collecting all coins in the level, show the message “All coins collected!!!”.

- **Step 1:** create a Widget Blueprint





Parent class: User Widget

No debug object selected

TextBlock_97

Slot (Canvas Panel Slot)

Anchors: Anchors
Positior: 92.0
Positior: 116.0
Size X: 151.0
Size Y: 40.0
Alignme: X 0.0 Y 0.0
Size To Co: ☑
ZOrder: 0

Content

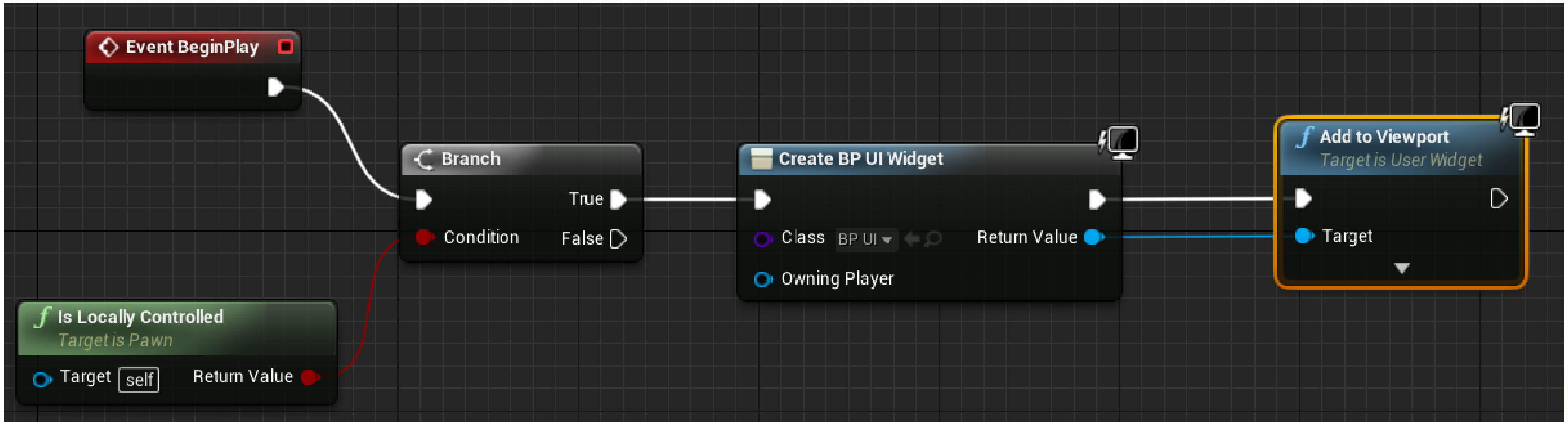
Text: GetText_0

Appearance

Color ar: Inherit Bind

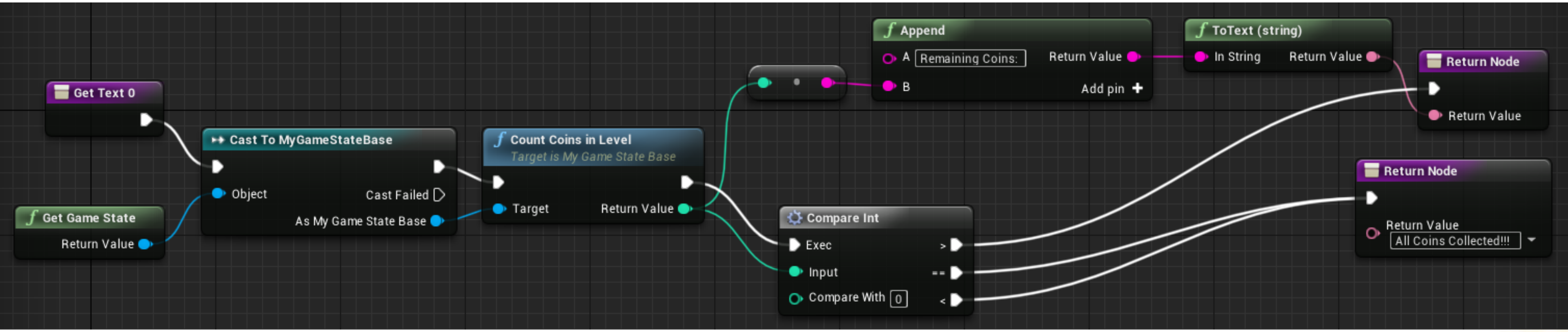
Collectible Coin

- **Step 2:** instantiate the Widget Blueprint in the BeginPlay event of the MyCharacter blueprint.



Collectible Coin

- **Step 3:** bind the text value and create the Widget Blueprint logic.



Level Complete

- **Level complete area:** alter collecting all coins, the player can go to this area to complete the level.

LevelCompleteArea.h

```
UCLASS()
class MYFIRSTGAME_API ALevelCompleteArea : public AActor
{
  ...

protected:
  UPROPERTY(VisibleAnywhere, Category = "Components")
  class UBoxComponent* BoxComponent;

  UFUNCTION()
  void HandleBeginOverlap(UPrimitiveComponent* OverlappedComponent,
                AActor* OtherActor, UPrimitiveComponent* OtherComp,
                int32 OtherBodyIndex, bool bFromSweep, const
                FHitResult & SweepResult);
};
```

Level Complete

- **Level complete area:** alter collecting all coins, the player can go to this area to complete the level.

LevelCompleteArea.cpp

```
ALevelCompleteArea::ALevelCompleteArea()
{
  BoxComponent = CreateDefaultSubobject<UBoxComponent>("BoxComponent");
  BoxComponent->SetBoxExtent(FVector(200.0f, 200.0f, 200.0f));
  BoxComponent->SetCollisionEnabled(ECollisionEnabled::QueryOnly);
  BoxComponent->SetCollisionResponseToAllChannels(ECR_Ignore);
  BoxComponent->SetCollisionResponseToChannel(ECC_Pawn, ECR_Overlap);
  RootComponent = BoxComponent;

  BoxComponent->OnComponentBeginOverlap.AddDynamic(this,
                             &ALevelCompleteArea::HandleBeginOverlap);

}
```

Level Complete

- **Level complete area:** alter collecting all coins, the player can go to this area to complete the level.

LevelCompleteArea.cpp

```
void ALevelCompleteArea::HandleBeginOverlap(UPrimitiveComponent*
                 OverlappedComponent, AActor* OtherActor,
                 UPrimitiveComponent* OtherComp, int32 OtherBodyIndex,
                 bool bFromSweep, const FHitResult & SweepResult){
  AMyCharacter* character = Cast<AMyCharacter>(OtherActor);
  AMyGameStateBase* gamestate = Cast<AMyGameStateBase>(
                                          GetWorld()->GetGameState());
  if ((character) && (gamestate)){
    if (gamestate->CountCoinsInLevel() == 0){
      gamestate->MulticastOnLevelComplete(character, true);
    }
  }
}
```

Level Complete

- **Level complete area:** alter collecting all coins, the player can go to this area to complete the level.

MyGameStateBase.h

```
public:
   ...

   UFUNCTION(NetMulticast, Reliable)
   void MulticastOnLevelComplete(APawn* character, bool succeeded);
```

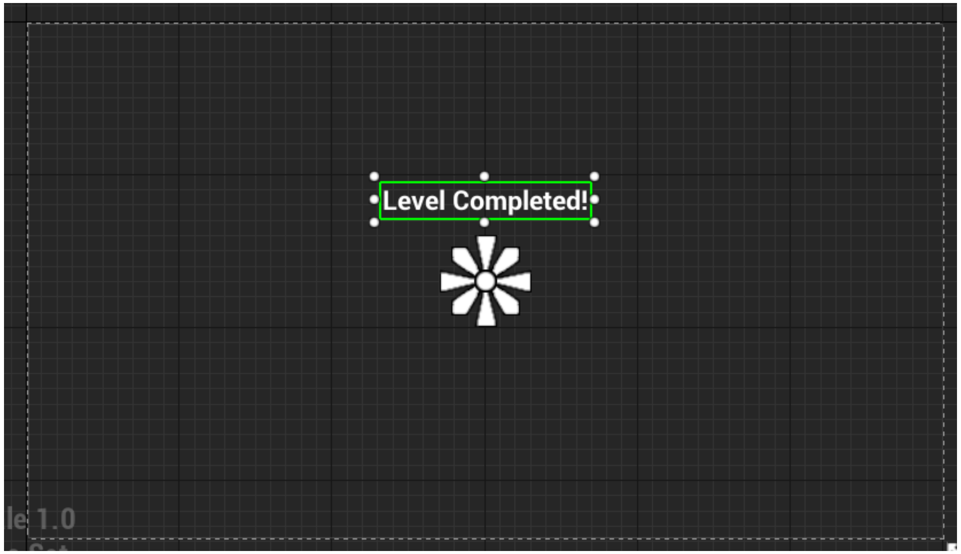
MyGameStateBase.cpp

```
void AMyGameStateBase::MulticastOnLevelComplete_Implementation(APawn*
                               character, bool succeeded)
{

   ...

}
```

Level Complete

- **<u>Level complete area:</u>** alter collecting all coins, the player can go to this area to complete the level.

1. Create a Widget Blueprint with a “Level Completed!” message;

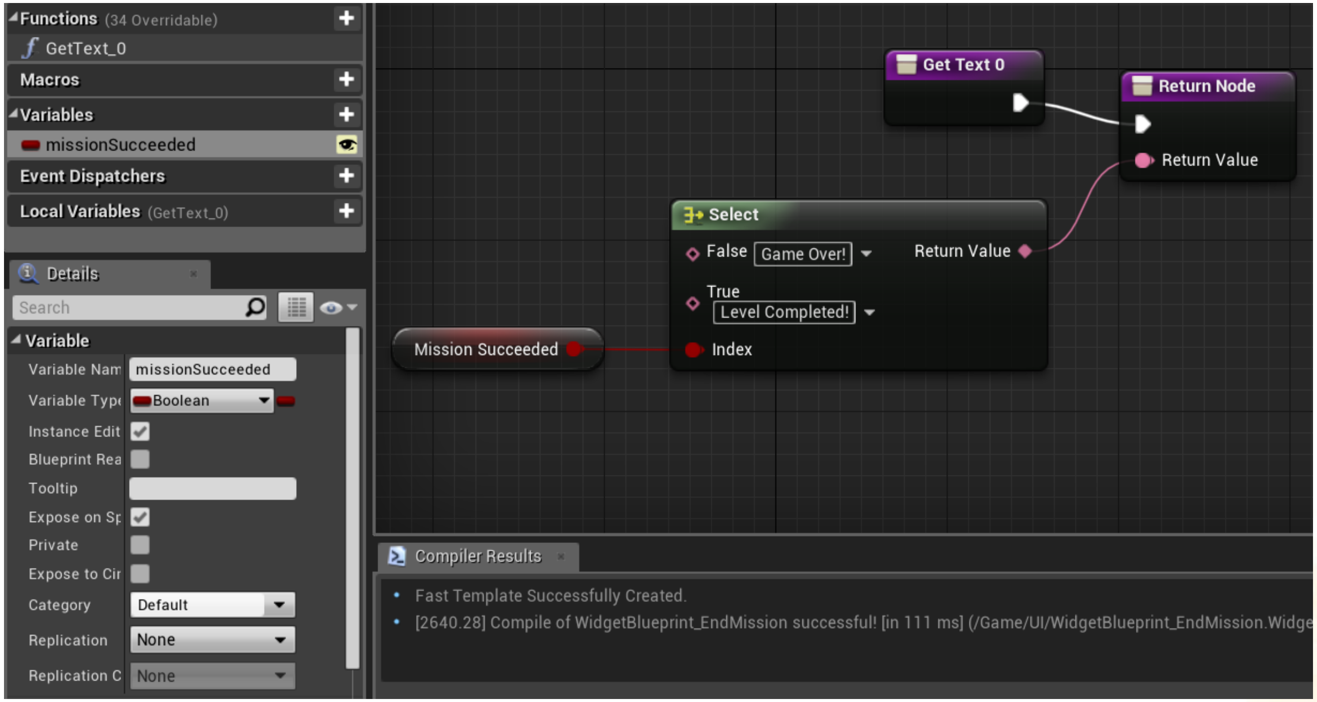


Level Completed!

Level Complete

- **<u>Level complete area:</u>** alter collecting all coins, the player can go to this area to complete the level.

2. In the Widget, create a new boolean variable to represent succeeded value and a blueprint to bind the correct message based on variable value;



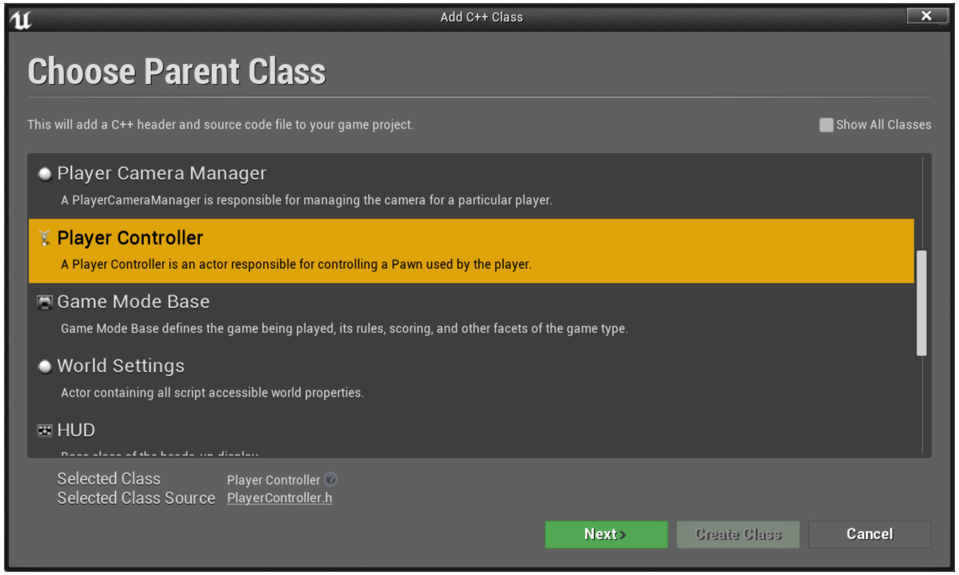
Variable Nam missionSucceeded
Variable Type Boolean
Instance Edit ☑
Blueprint Rea □
Tooltip
Expose on Sp ☑
Private □
Expose to Cir □
Category Default
Replication None
Replication C None

False: Game Over!
True: Level Completed!

```
• Fast Template Successfully Created.
• [2640.28] Compile of WidgetBlueprint_EndMission successful! [in 111 ms] (/Game/UI/WidgetBlueprint_EndMission.Widge
```

Level Complete

- **<u>Level complete area:</u>** alter collecting all coins, the player can go to this area to complete the level.
  - Now we need a class to create the widget that exist only once on the clients: Game Controller.
  - Create the widget in a new Game Controller class;



Add C++ Class

Choose Parent Class

This will add a C++ header and source code file to your game project.

☐ Show All Classes

Player Camera Manager
A PlayerCameraManager is responsible for managing the camera for a particular player.

Player Controller
A Player Controller is an actor responsible for controlling a Pawn used by the player.

Game Mode Base
Game Mode Base defines the game being played, its rules, scoring, and other facets of the game type.

World Settings
Actor containing all script accessible world properties.

HUD
Base class of the heads-up display.

Selected Class: Player Controller
Selected Class Source: PlayerController.h

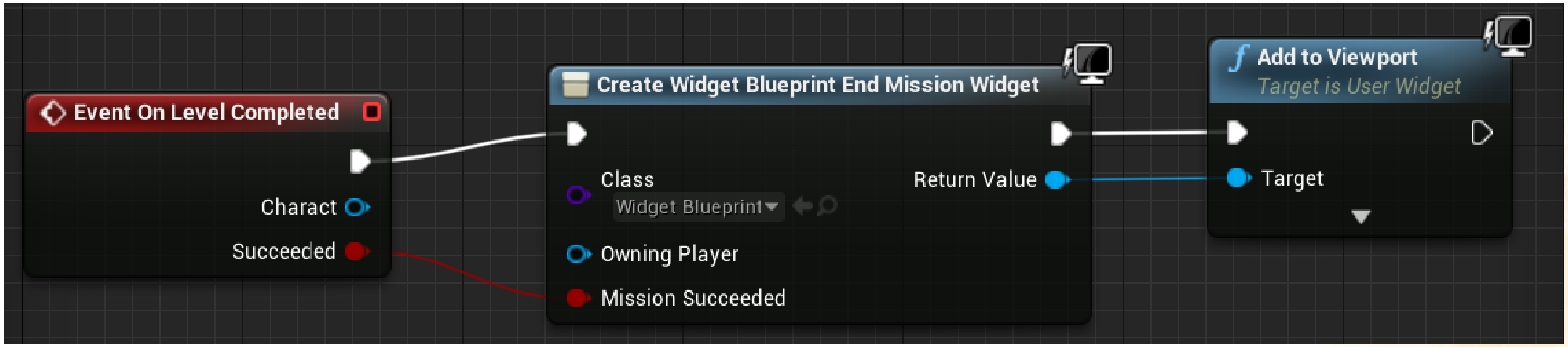
Level Complete

- **Level complete area:** alter collecting all coins, the player can go to this area to complete the level.
  - Create and expose a OnLevelCompleted function to blueprint implementation:

MyPlayerController.h

```
public:
  UFUNCTION(BlueprintImplementableEvent, Category = "Gameplay Events")
  void OnLevelCompleted(APawn* charact, bool succeeded);
```

  - Create a blueprint for the new player controller and implement the event:

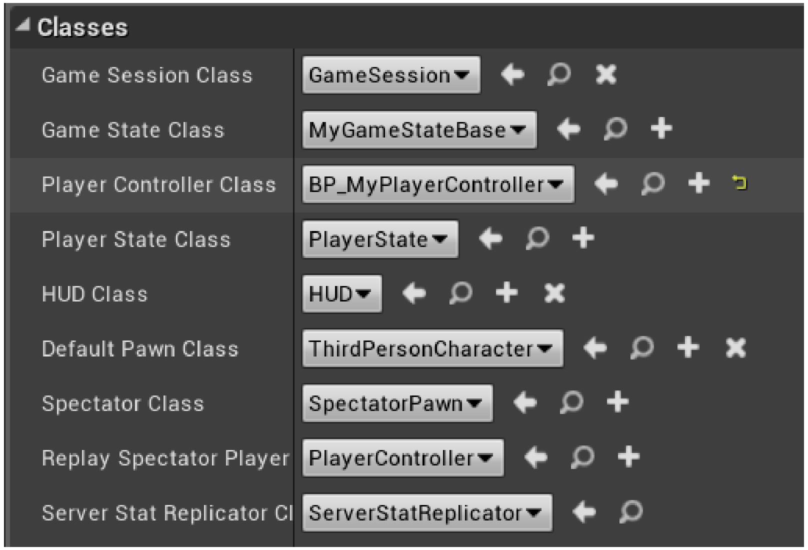


MyGameStateBase.cpp

```
void AMyGameStateBase::MulticastOnLevelComplete_Implementation(APawn*
                                          character, bool succeeded){
  if (succeeded){
    for (FConstPawnIterator it = GetWorld()->GetPawnIterator();
         it; it++){
      APawn* pawn = it->Get();
      if (pawn && pawn->IsLocallyControlled()) {
        pawn->DisableInput(nullptr);
      }
    }
    for (FConstPlayerControllerIterator it = GetWorld()->
                           GetPlayerControllerIterator(); it; it++){
      AMyPlayerController* pController =
                           Cast<AMyPlayerController>(it->Get());
      if ((pController) && (pController->IsLocalController())) {
        pController->OnLevelCompleted(character, succeeded);
      }
    }
  }
  else{
    if (character){
      character->DisableInput(nullptr);
      AMyPlayerController* pController =
               Cast<AMyPlayerController>(character->GetController());
      if ((pController) && (pController->IsLocalController())) {
        pController->OnLevelCompleted(character, succeeded);
  }  }  }  }
```

Level Complete

- **Level complete area:** alter collecting all coins, the player can go to this area to complete the level.
  - Set the new Player Controller class in the blueprint instance of the Game Mode:



Classes

Game Session Class	GameSession
Game State Class	MyGameStateBase
Player Controller Class	BP_MyPlayerController
Player State Class	PlayerState
HUD Class	HUD
Default Pawn Class	ThirdPersonCharacter
Spectator Class	SpectatorPawn
Replay Spectator Player	PlayerController
Server Stat Replicator Cl	ServerStatReplicator

Level Complete

- **Level complete area:** if the player goes to the level complete area without collecting all coins, a sound notification is played.

LevelCompleteArea.h

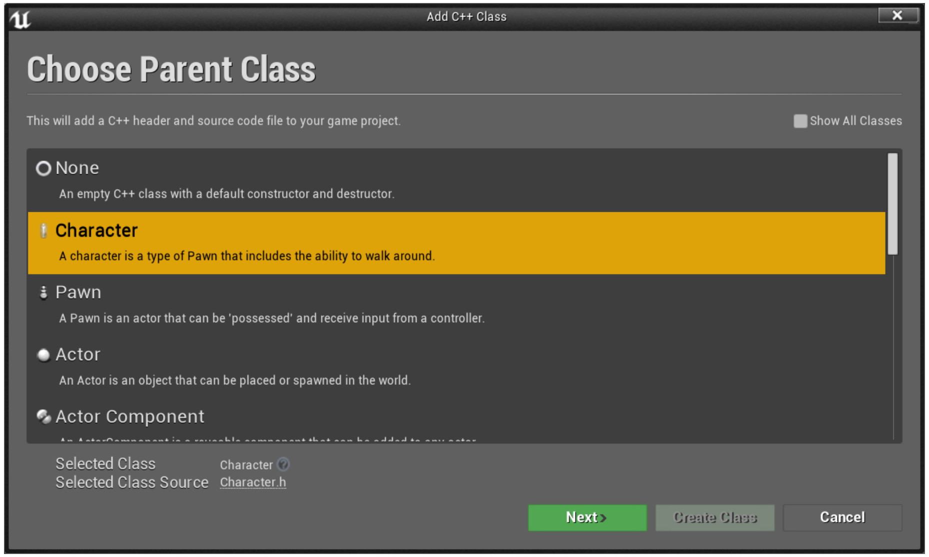
```
protected:
  ...

  UPROPERTY(EditDefaultsOnly, Category = "Sounds")
  USoundBase* LevelNotCompletedSound;
```

LevelCompleteArea.cpp

```
void ALevelCompleteArea::HandleBeginOverlap(...){
  ...
  if ((character)&& (gamestate)){
    if (character->GetTotalCoins()== gamestate->GetTotalLevelCoins()){
      gamestate->MulticastOnLevelComplete(character, true);
    }
    else{
      UGameplayStatics::PlaySound2D(this, LevelNotCompletedSound);
    }
  }
}
```

Enemies

- **Next step:** create an enemy with AI that walks between waypoints. When the enemy sees the player, he follows and attacks the player.
  - Create new C++ class for the enemy: base class Character



Add C++ Class

Choose Parent Class

This will add a C++ header and source code file to your game project.

□ Show All Classes

○ None
An empty C++ class with a default constructor and destructor.

Character
A character is a type of Pawn that includes the ability to walk around.

Pawn
A Pawn is an actor that can be 'possessed' and receive input from a controller.

Actor
An Actor is an object that can be placed or spawned in the world.

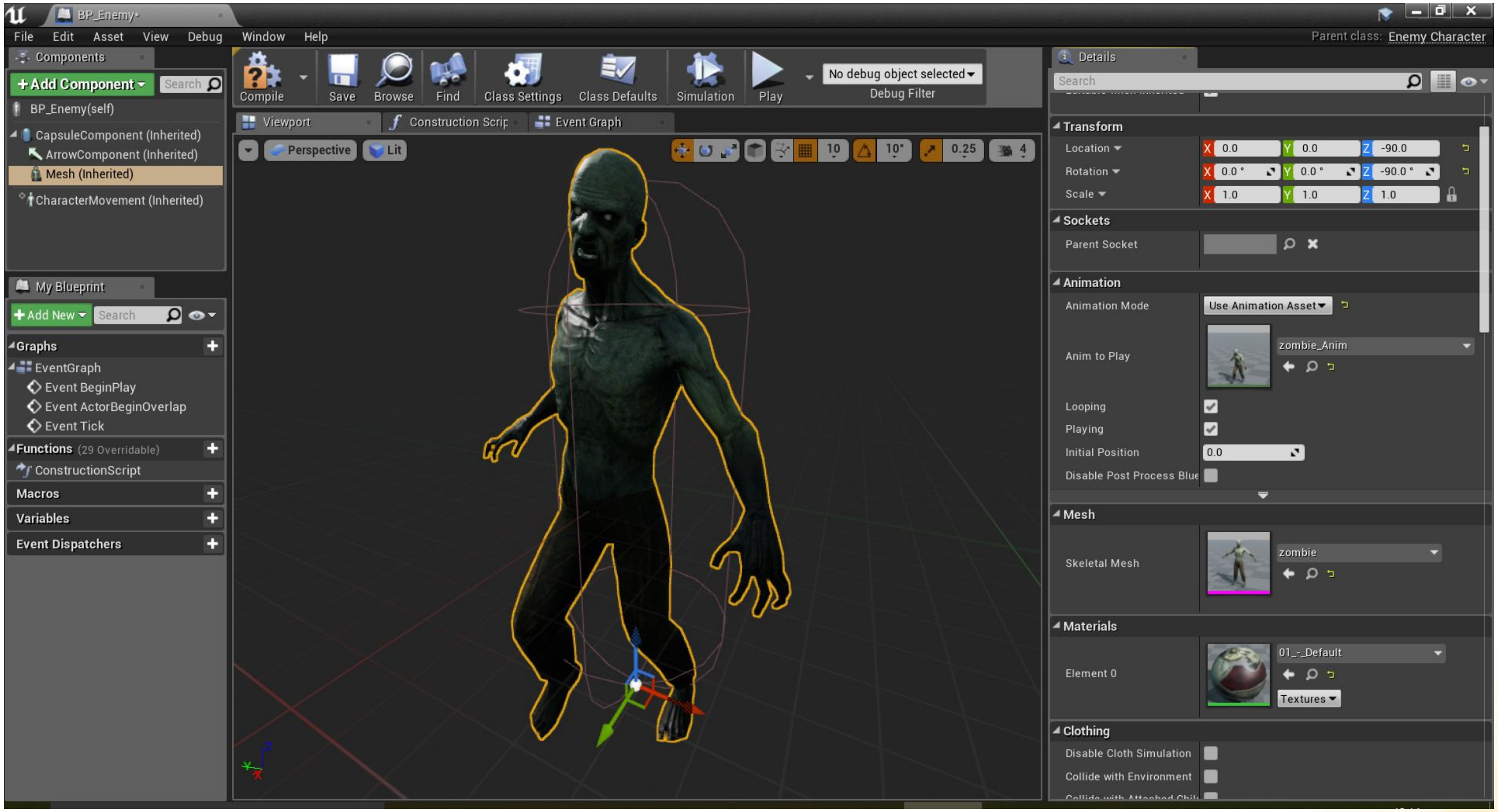
Actor Component

Selected Class Character
Selected Class Source Character.h

  - Download and import the enemy model:
    - http://www.inf.puc-rio.br/~elima/dp/zombie.zip

Enemies

- **Next step:** create an enemy with AI.
  - Create and setup a blueprint for the new C++ enemy class:



Parent class: Enemy Character

Transform
Location: X 0.0, Y 0.0, Z -90.0
Rotation: X 0.0 °, Y 0.0 °, Z -90.0 °
Scale: X 1.0, Y 1.0, Z 1.0

Sockets
Parent Socket:

Animation
Animation Mode: Use Animation Asset
Anim to Play: zombie_Anim
Looping: ☑
Playing: ☑
Initial Position: 0.0
Disable Post Process Blue: □

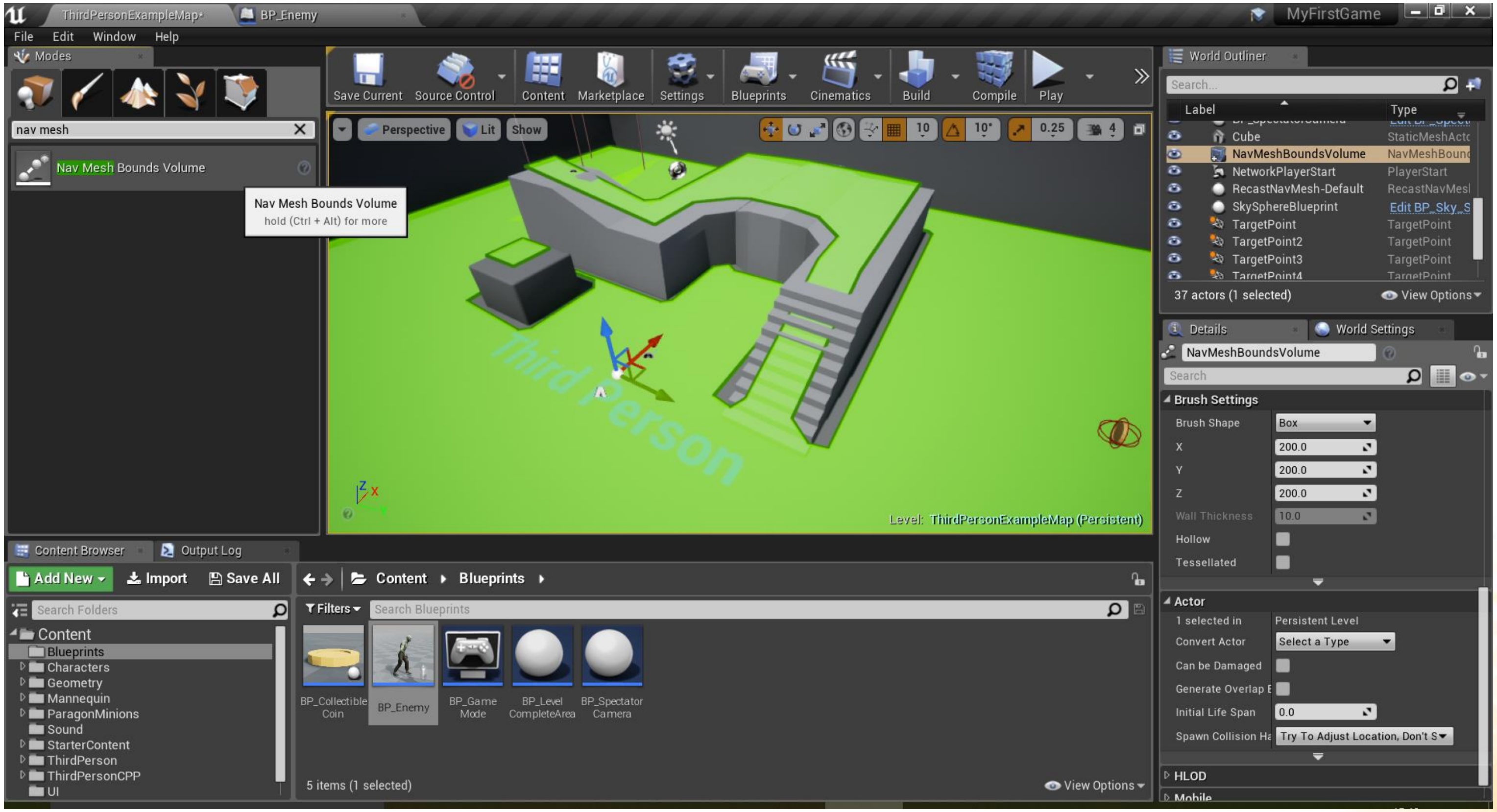
Mesh
Skeletal Mesh: zombie

Materials
Element 0: 01_-_Default
Textures

Clothing
Disable Cloth Simulation: □
Collide with Environment: □

Enemies

- **Next step:** create an enemy with AI.
  - Add a Nav Mesh Bounds Volume and resize it so that it fits all of the walkable space in the level (press P to show the Nav Mesh):

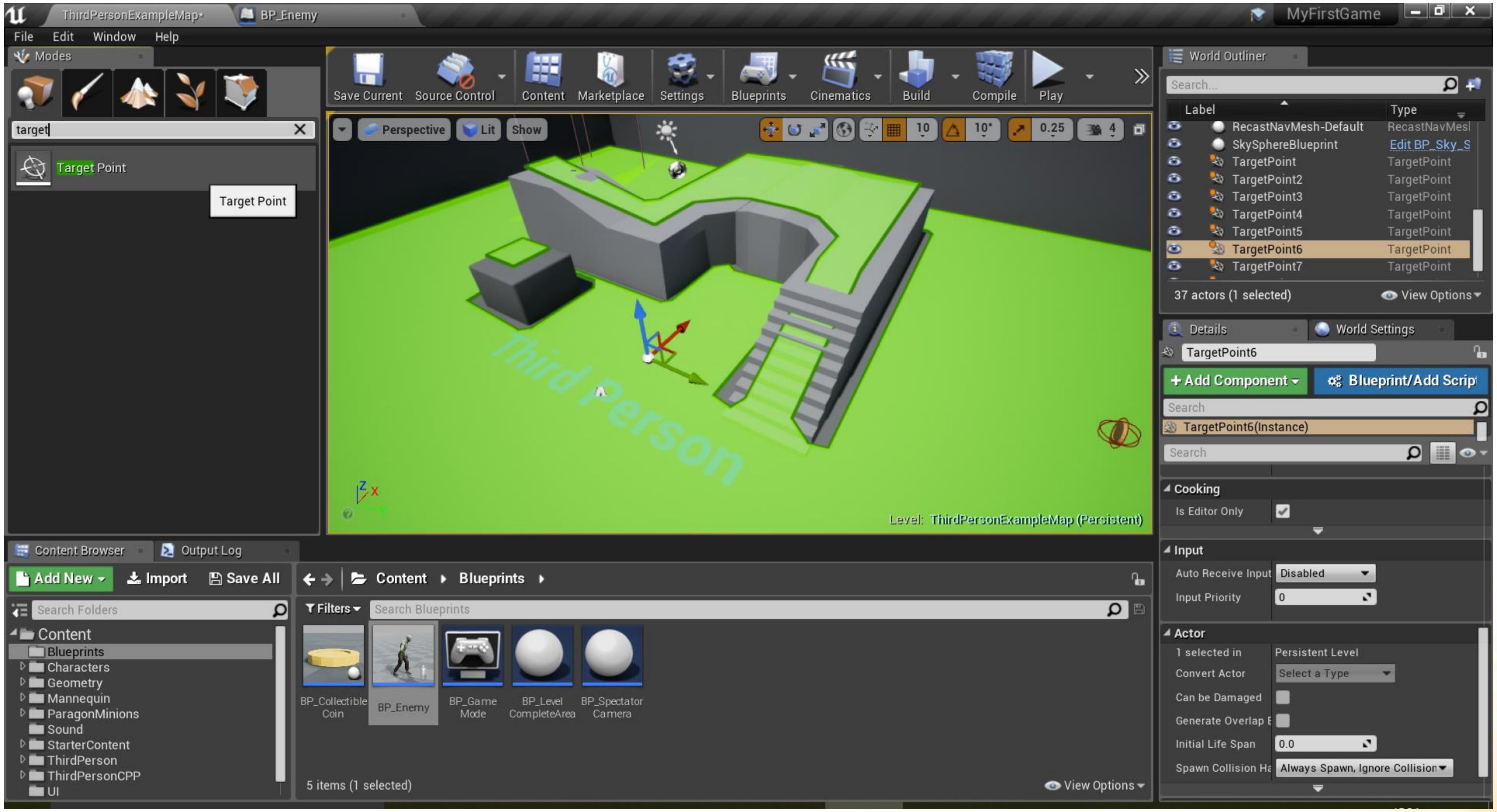


Brush Shape: Box
X: 200.0
Y: 200.0
Z: 200.0
Wall Thickness: 10.0
Hollow: □
Tessellated: □

1 selected in: Persistent Level
Convert Actor: Select a Type
Can be Damaged: □
Generate Overlap E: □
Initial Life Span: 0.0
Spawn Collision Ha: Try To Adjust Location, Don't S

Enemies

- **Next step:** create an enemy with AI.
  - Place some waypoints (Target Point) in the level:



Is Editor Only ☑

Auto Receive Input: Disabled

Input Priority: 0

1 selected in: Persistent Level

Convert Actor: Select a Type

Can be Damaged □

Generate Overlap E □

Initial Life Span: 0.0

Spawn Collision Ha: Always Spawn, Ignore Collision

Enemies

- **Next step:** create an enemy with AI.

EnemyCharacter.h

```
...

protected:
  ...

  TArray<AActor*> Waypoints;

  class AAIController* AIController;

  TScriptDelegate<FWeakObjectPtr> MovementCompleteDelegate;

  class ATargetPoint* GetRandomWaypoint();

  UFUNCTION()
  void AIMoveCompleted(FAIRequestID RequestID,
                       EPathFollowingResult::Type Result);

  ...
```

Enemies

- **Next step:** create an enemy with AI.

EnemyCharacter.cpp

```cpp
#include "Engine/TargetPoint.h"
#include "AIController.h"

void AEnemyCharacter::BeginPlay()
{
  Super::BeginPlay();
  UGameplayStatics::GetAllActorsOfClass(GetWorld(),
                              ATargetPoint::StaticClass(), Waypoints);
  AIController = Cast<AAIController>(GetController());
  this->bUseControllerRotationYaw = false;

  if ((Waypoints.Num() > 0)&&(AIController)){
    MovementCompleteDelegate.BindUFunction(this, "AIMoveCompleted");
    AIController->ReceiveMoveCompleted.Add(MovementCompleteDelegate);

    AIController->MoveToActor(GetRandomWaypoint());
  }
}
```

Important: the AIModule must be included as a public dependency.

Enemies

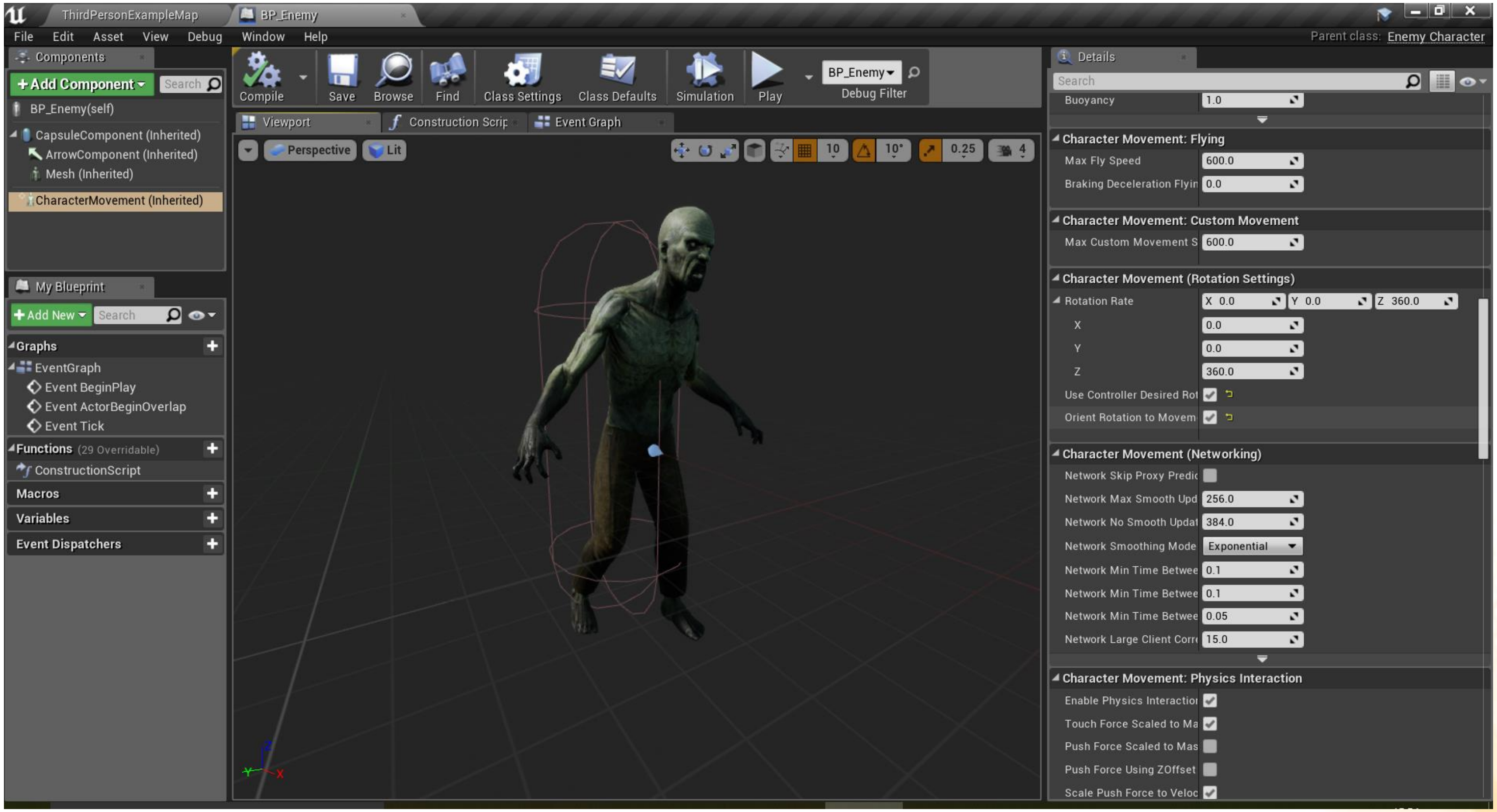
- **Next step:** create an enemy with AI.

EnemyCharacter.cpp

```
ATargetPoint* AEnemyCharacter::GetRandomWaypoint()
{
  int index = FMath::RandRange(0, Waypoints.Num() - 1);
  return Cast<ATargetPoint>(Waypoints[index]);
}


void AEnemyCharacter::AIMoveCompleted(FAIRequestID RequestID,
                                  EPathFollowingResult::Type Result){
  if (Result == EPathFollowingResult::Success)
  {
    if ((Waypoints.Num() > 0) && (AIController))
    {
      AIController->MoveToActor(GetRandomWaypoint());
    }
  }
}
```

Enemies

- **Next step:** create an enemy with AI.
  - Adjust max speed and rotation settings in the CharacterMovement component:



Buoyancy 1.0

Character Movement: Flying

Max Fly Speed 600.0

Braking Deceleration Flyin 0.0

Character Movement: Custom Movement

Max Custom Movement S 600.0

Character Movement (Rotation Settings)

Rotation Rate X 0.0 Y 0.0 Z 360.0

X 0.0

Y 0.0

Z 360.0

Use Controller Desired Rot ☑

Orient Rotation to Movem ☑

Character Movement (Networking)

Network Skip Proxy Predic □

Network Max Smooth Upd 256.0

Network No Smooth Updat 384.0

Network Smoothing Mode Exponential

Network Min Time Betwee 0.1

Network Min Time Betwee 0.1

Network Min Time Betwee 0.05

Network Large Client Corr 15.0

Character Movement: Physics Interaction

Enable Physics Interactior ☑

Touch Force Scaled to Ma ☑

Push Force Scaled to Mas □

Push Force Using ZOffset □

Scale Push Force to Veloc ☑

Enemies

- **Next step:** create an enemy with AI.
  - Add a SensingComponent to allow the enemy to see the player:

EnemyCharacter.h

```
protected:

   ...

   UPROPERTY(VisibleAnywhere, Category = "Components")
   class UPawnSensingComponent* SensingComponent;

   AActor* target;

   UFUNCTION()
   void SeePlayer(APawn *pawn);

   ...
```

Enemies

- **Next step:** create an enemy with AI.
  - Add a SensingComponent to allow the enemy to see the player:

EnemyCharacter.cpp

```
AEnemyCharacter::AEnemyCharacter()
{
  SensingComponent = CreateDefaultSubobject<UPawnSensingComponent>
                                                 ("SensingComponent");
  SensingComponent->OnSeePawn.AddDynamic(this,
                                         &AEnemyCharacter::SeePlayer);
  SensingComponent->SetSensingUpdatesEnabled(true);
}
void AEnemyCharacter::SeePlayer(APawn *pawn)
{
  if ((pawn) && (AIController) && (!target)) {
    target = pawn;
    this->GetMesh()->GlobalAnimRateScale = 2.5f;
    this->GetCharacterMovement()->MaxWalkSpeed = 150.0f;
    AIController->MoveToActor(pawn);
  }
}
```

Enemies

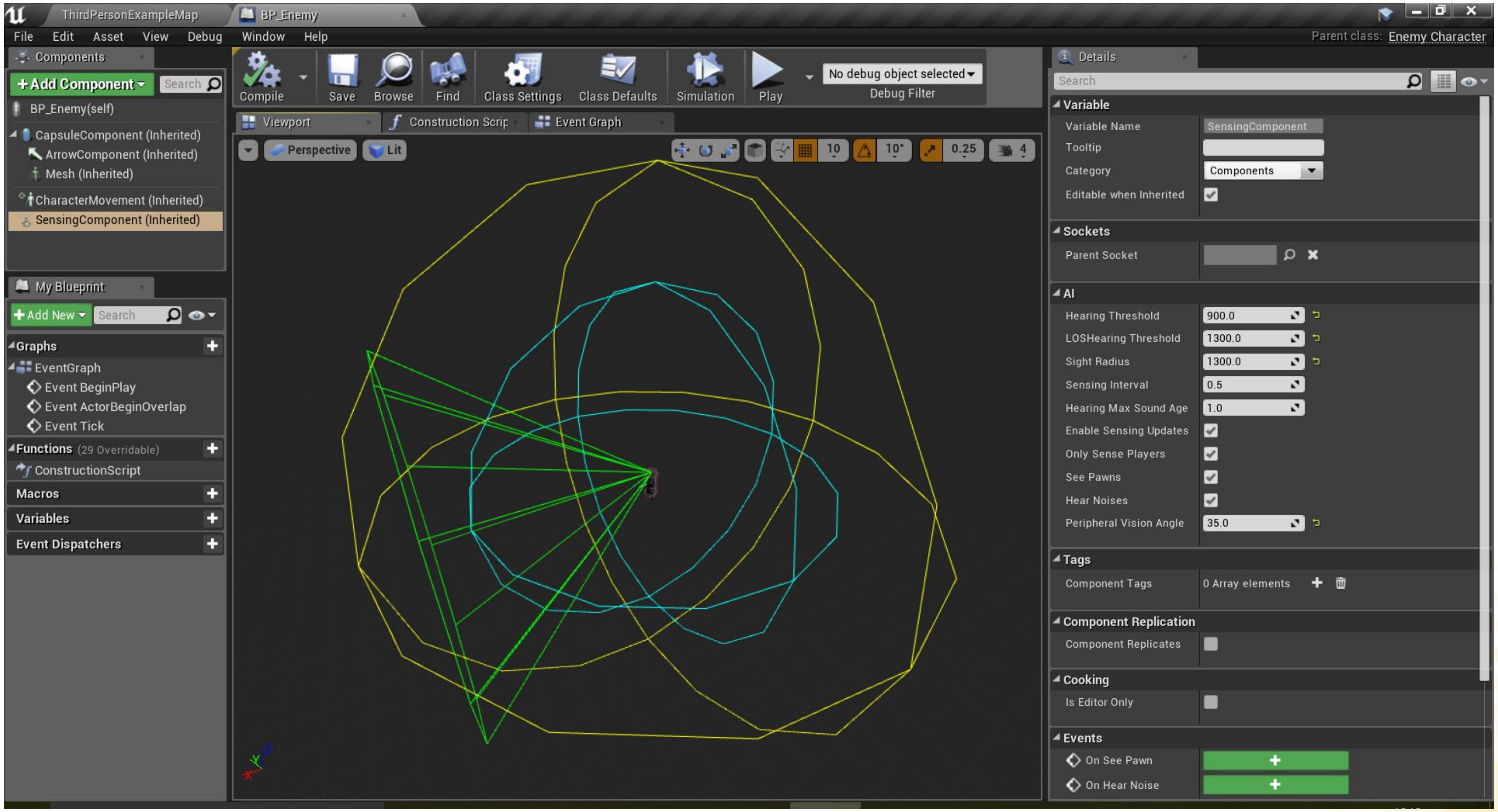
- **Next step:** create an enemy with AI.
  - Add a SensingComponent to allow the enemy to see the player:

EnemyCharacter.cpp

```
void AEnemyCharacter::Tick(float DeltaTime)
{
  Super::Tick(DeltaTime);
  if (target)
  {
    if (FVector::Dist(GetActorLocation(), target->GetActorLocation())
        > SensingComponent->SightRadius)
    {
      this->GetMesh()->GlobalAnimRateScale = 1.0f;
      this->GetCharacterMovement()->MaxWalkSpeed = 50;
      target = nullptr;
      AIController->MoveToActor(GetRandomWaypoint());
    }
  }
}
```

Enemies

- **Next step:** create an enemy with AI.
  - Adjust the sight properties in the enemy blueprint:



Variable	
Variable Name	SensingComponent
Tooltip	
Category	Components
Editable when Inherited	☑
Sockets	
Parent Socket	
AI	
Hearing Threshold	900.0
LOSHearing Threshold	1300.0
Sight Radius	1300.0
Sensing Interval	0.5
Hearing Max Sound Age	1.0
Enable Sensing Updates	☑
Only Sense Players	☑
See Pawns	☑
Hear Noises	☑
Peripheral Vision Angle	35.0
Tags	
Component Tags	0 Array elements
Component Replication	
Component Replicates	□
Cooking	
Is Editor Only	□
Events	
On See Pawn	+
On Hear Noise	+

Enemies

- **Next step:** create an enemy with AI.
  - If the enemy gets to the player position, show the game over message:

```
void AEnemyCharacter::AIMoveCompleted(FAIRequestID RequestID,
                                   EPathFollowingResult::Type Result){
  if (Result == EPathFollowingResult::Success){
    if (target){
      AMyCharacter* character = Cast<AMyCharacter>(target);
      AMyGameStateBase* gamestate = Cast<AMyGameStateBase>(GetWorld()
                                                    ->GetGameState());
      if ((character) && (gamestate)){
        gamestate->MulticastOnLevelComplete(character, false);
      }
      target = nullptr;
    }
    if ((Waypoints.Num() > 0) && (AIController)){
      AIController->MoveToActor(GetRandomWaypoint());
    }
  }
}
```

Enemies

- **Next step:** create an enemy with AI.
  - We need to manually synchronize the animation speed with all clients.
  - Solution: create a variable to represent a chasing state and replicate it to all clients:

EnemyCharacter.h

```
protected:
  ...
  UPROPERTY(Replicated)
  bool isChasing;
```

EnemyCharacter.cpp

```
#include "UnrealNetwork.h"

void AEnemyCharacter::GetLifetimeReplicatedProps(TArray
<FLifetimeProperty>& OutLifetimeProps) const
{
  Super::GetLifetimeReplicatedProps(OutLifetimeProps);
  DOREPLIFETIME(AEnemyCharacter, isChasing);
}
```

Enemies

EnemyCharacter.cpp

```
AEnemyCharacter::AEnemyCharacter(){

  ...

  isChasing = false;
  SetReplicates(true);
}

void AEnemyCharacter::SeePlayer(APawn *pawn)
{
  if ((pawn) && (AIController) && (!target))
  {
    ...

    isChasing = true;
  }
}
```

Enemies

EnemyCharacter.cpp

```
void AEnemyCharacter::Tick(float DeltaTime){
  Super::Tick(DeltaTime);
  if ((target)&& (Role == ROLE_Authority)){
    if (FVector::Dist(GetActorLocation(),
       target->GetActorLocation()) > SensingComponent->SightRadius){
       ...
       isChasing = false;
    }
  }
  if ((isChasing)&& (this->GetMesh()->GlobalAnimRateScale != 2.5f)){
    this->GetMesh()->GlobalAnimRateScale = 2.5f;
    this->GetCharacterMovement()->MaxWalkSpeed = 150.0f;
  }
  else if (this->GetMesh()->GlobalAnimRateScale != 1.0f){
    this->GetMesh()->GlobalAnimRateScale = 1.0f;
    this->GetCharacterMovement()->MaxWalkSpeed = 50.0f;
  }
}
```

Exercise 1

1) Continue the implementation of the game:

   a) Play a sound effect when a coin is collected.

   b) Count the number of remaining coins only on the server and synchronize the events of “collecting coins” on all clients.

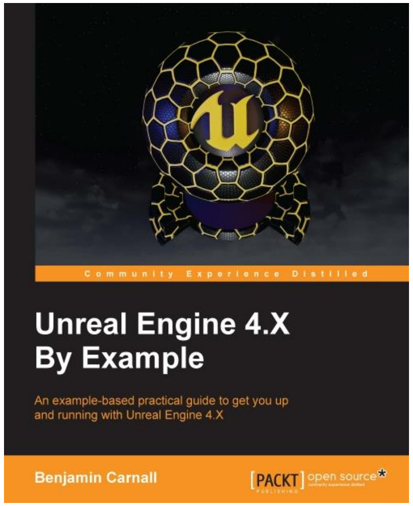
      - In our current implementation, the coins are being counted locally by the clients. If there is small desynchronization in the position of players, the whole game will get desynchronized.

   c) Count the number of coins collected by each player.

   d) Improve the level by adding more coins, more enemies, more waypoints, and adjusting the position of the coins, enemies and waypoints. In addition, balance the gameplay by adjusting the speed of the enemies according to the speed of the player.

Further Reading

- Carnall, B. (2016). **Unreal Engine 4.X By Example**. Packt Publishing. ISBN: 978-1785885532.



Community Experience Distilled
Unreal Engine 4.X By Example
An example-based practical guide to get you up and running with Unreal Engine 4.X
Benjamin Carnall
[PACKT] open source

- **Web Resources:**
  - Introduction to C++ Programming in UE4 - https://docs.unrealengine.com/en-US/Programming/Introduction
  - Coding Standard - https://docs.unrealengine.com/en-US/Programming/Development/CodingStandard
  - Gameplay Programming - https://docs.unrealengine.com/en-us/Programming/UnrealArchitecture
  - Networking and Multiplayer in Unreal Engine - https://docs.unrealengine.com/en-us/Gameplay/Networking
  - Network Guide - https://wiki.unrealengine.com/index.php?title=4_13%2B_Network_Guide