GAME PRODUCTION ENVIRONMENTS

FACHHOCHSCHULE SALZBURG
SOMMER SEMESTER 2023

//CHRISTINA PIBERGER

CHRISTINA PIBERGER

GAME PROGRAMMER

CHRISTINA.PIBERGER@GMAIL.COM

POW WOW ENTERTAINMENT

About me

- B.Sc. - Electrical Engineering and Information Technology
- M.Sc. - Robotics, Cognition, Intelligence

- Worked ~4 years in Software Development:
  - 2D/3D Rendering for Embedded Systems (C/C++)
  - Indoor Navigation for robots

- Currently: Game Programmer at Pow Wow (~2 years)

ABOUT THIS CLASS

We have 4 sessions:

- Thu, 13 April (13:30 - 16:45, 15 min break) - **Today!**
- Fri, 14 April (09:00 - 11:30, 15 min break)
- Thu, 11 May (13:30 - 16:00, 15 min break)
- Fri, 12 May (09:00 - 12:30, **2x**15 min break)

Grading is based on assignment:

- No groups. Every student has to create their own gitlab repo.

- For more details see separate assignment files on Wiki:
  https://wiki.mediacube.at/wiki/index.php?title=Game_Production_Environments_-_SS_2023#Unreal_Engine

Session 1

- Introduction
- Epic Games
- Unreal Editor UI
- Actors & Components
- Blueprints (+ Reflection)

[15 min break at ~15:00]

- Game Framework & Most common classes
- Packaging & Publishing

- Features in Stuntfest
- Look at template projects
- Materials + Landscape Tool

[ends at 16:45]

Session 2

- Input System + Ejection
- Ragdoll + Physics + Anim BP

Session 3

- Main Menu, Loading Levels
- UI, Start/Finish race

Session 4

- Display Highscore List
- Cleaning up project
- TBD

Any topic wishes from you?

EPIC GAMES

History

1994: Tim Sweeney founded Epic Mega Games

1998: Release of Unreal Engine 1 alongside the game “Unreal”

2002: Release of Unreal Engine 2
2006: Release of Unreal Engine 3 alongside “Gears of War”
2012: Sweeney sells 40% of company to Tencent for $330 million.

2014: Release of Unreal Engine 4

- Switch from individual licensing to a subscription model + 5% royalty on gross revenue
- 2015 - Removed subscription fee entirely, **only royalties** remain

2017: Release of Fortnite Battle Royale is a huge success

2022: Release of Unreal Engine 5

(“Standard”) License Today: A 5% royalty is due only if the lifetime gross revenue from a product/game that incorporates Unreal Engine code exceeds $1 million USD
https://www.unrealengine.com/en-US/license

The story behind Fortnite

2011: Fortnite trailer and announcement

https://youtu.be/2GSfjeYVpkQ

2017: Three big titles in parallel development and pre-alpha stage

- **Unreal Tournament 4:** FPS and next title of the Unreal series
- **Paragon:** MOBA to compete with League of Legends
- **Fortnite:** Tower defense + building mechanics to compete with Minecraft

March 2017: Release of **PUBG** the first big successful Battle Royal

September 2017: **Fortnite Battle Royale** was released as free-to-play

- 10 mil. active players in just 2 weeks
- Unreal Tournament 4 and Paragon get cancelled

Today & Future:

- Fortnite is already considered a “social network”
- Tim Sweeney’s vision is to build a metaverse

UNREAL ENGINE

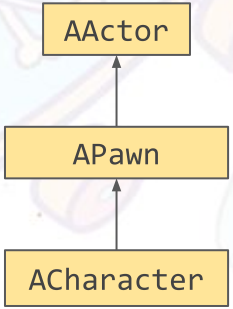
INTRODUCTION

I'm hungry. It's reducing the quality of my posture and aim.

UNREAL ENGINE

ACTORS AND COMPONENTS

Actors



Actor: Any object that can be placed into a level (e.g. PointLight, StaticMeshActor, …)

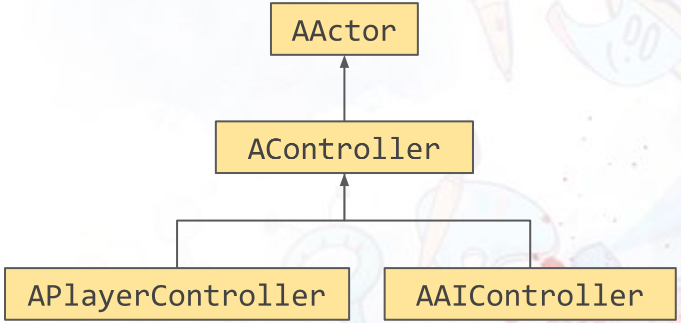
Pawn: Actor that can be controlled by a PlayerController or AIController via “possession”

Character: Pawn with additional functionality (SkeletalMesh + MovementComponent)

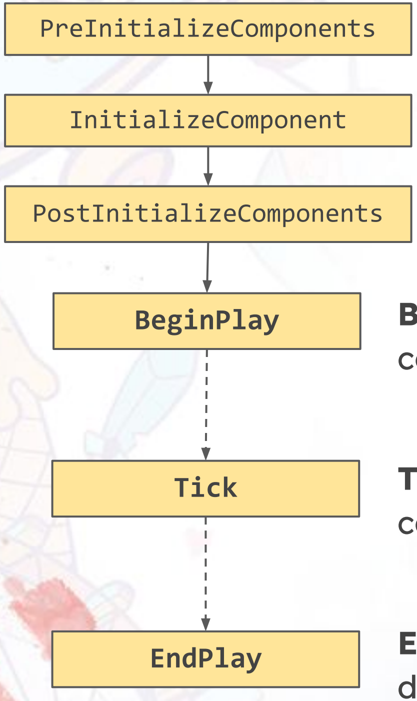
Controllers are non-physical Actors that can possess Pawns to control their actions.

Player Controller: Takes player input and translates it into interactions in the game.

Important function calls are **Possess()** and **Unpossess()**



Actor Lifecycle



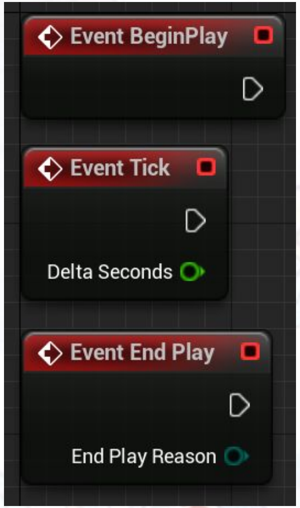
InitializeComponent is called on each component of the Actor.

BeginPlay is called after the Actor's components have been initialized.

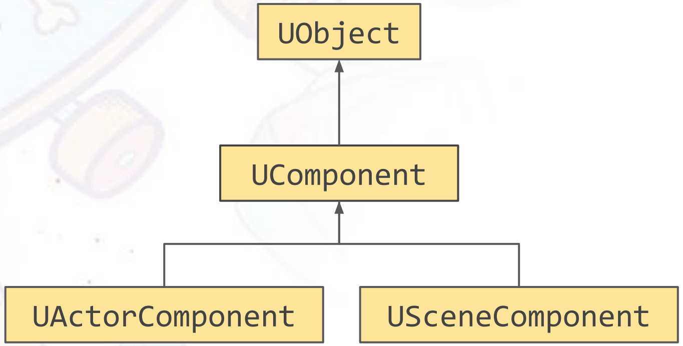
Tick can also be disabled. If enabled it's called every frame or a specified interval.

EndPlay is called right before Actor gets destroyed.

These three can also be overridden in Blueprint:



Components



Components can be added to Actors to extend their functionality.

Composition = **has-a** relationship

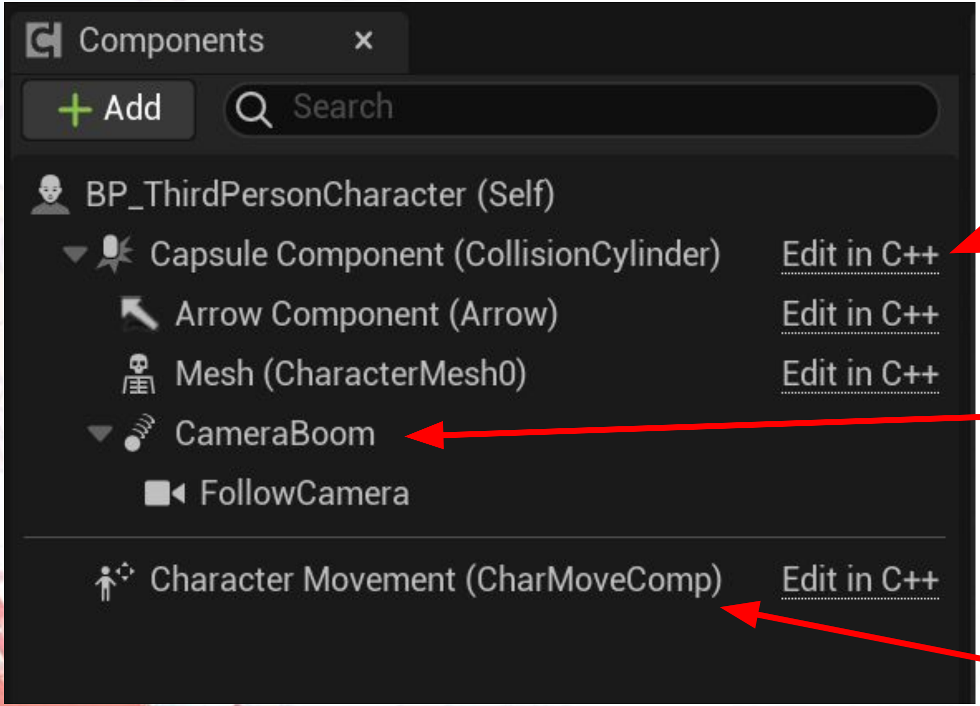
Inheritance = **is-a** relationship

ActorComponent: Non-physical component (e.g. UCharacterMovementComponent)

SceneComponent: Has it’s own transform in world (e.g. UCameraComponent)

Components on BP_ThirdPersonCharacter

(from UE template project)

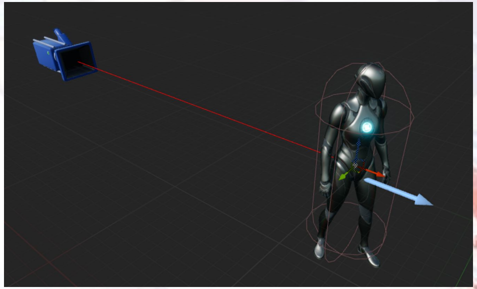


Components ×

+ Add

Search

- BP_ThirdPersonCharacter (Self)
  - Capsule Component (CollisionCylinder) — Edit in C++
    - Arrow Component (Arrow) — Edit in C++
    - Mesh (CharacterMesh0) — Edit in C++
    - CameraBoom
      - FollowCamera
- Character Movement (CharMoveComp) — Edit in C++

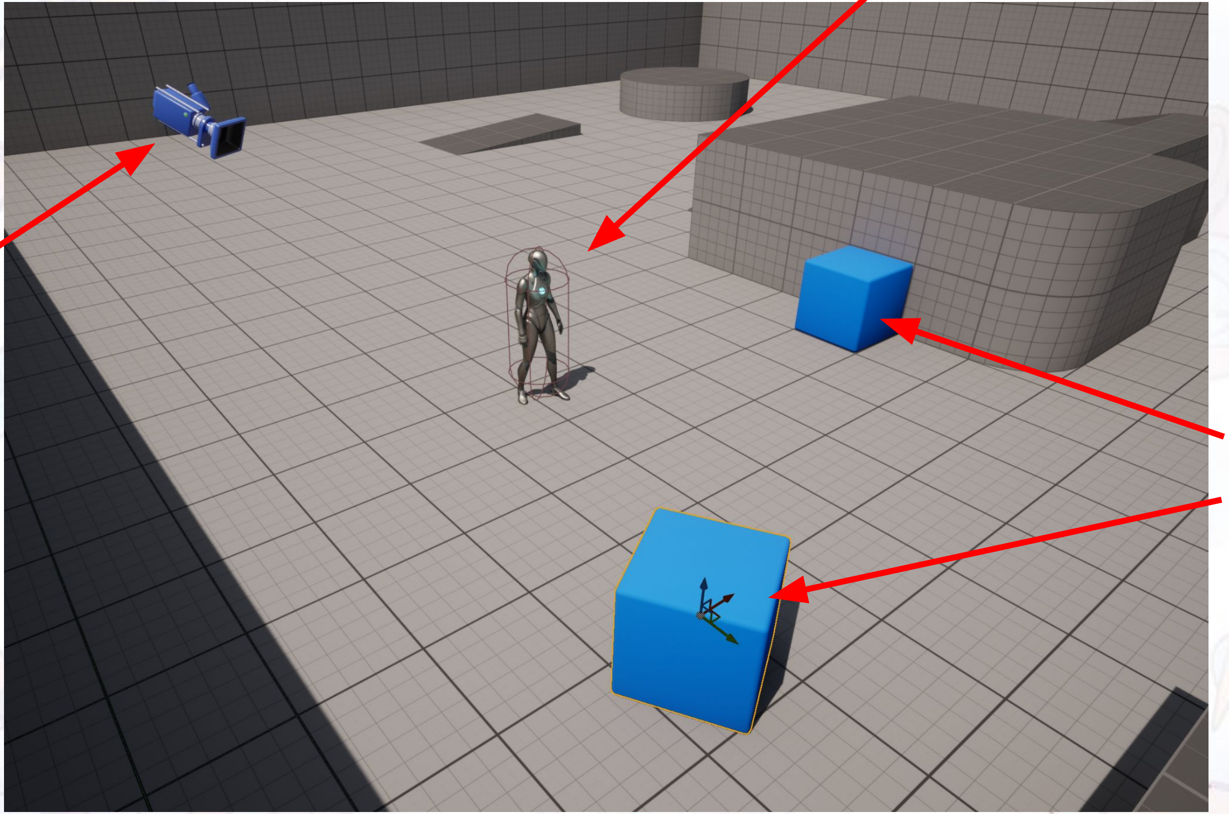


Capsule, Arrow and Mesh are **inherited** SceneComponents.

Spring Arm and Camera Component are SceneComponents and were added directly in the Blueprint and **not inherited** from C++.

Character Movement Component is a ActorComponent and was **inherited.**

Template Scene



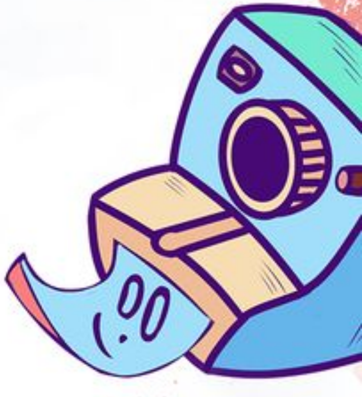
3rd Person Character

Camera Component

StaticMeshActor:
Actor + Static
Mesh Component

UNREAL ENGINE

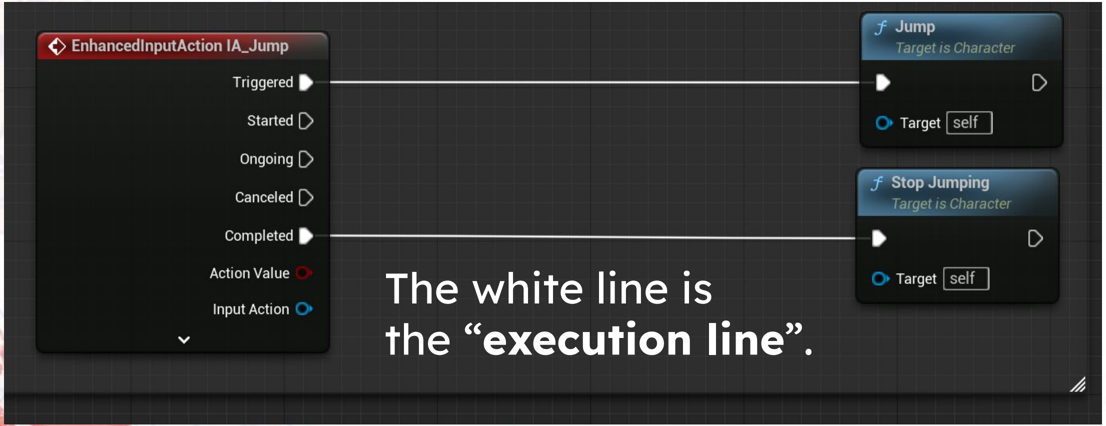
BLUEPRINT



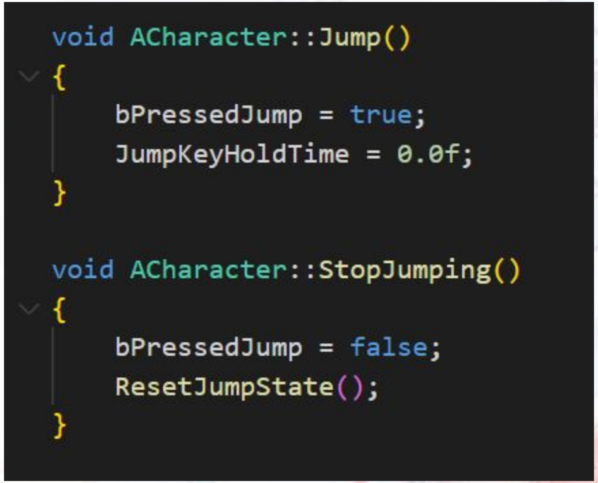
Blueprint

- **Blueprint** is the name of Unreal's Visual Scripting System
- The **Blueprint Editor** is a node-based graph editor

- Blueprint Functions can call C++ functions and vice versa



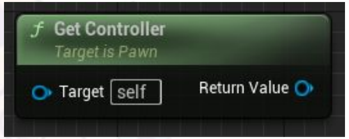
The white line is
the "**execution line**".



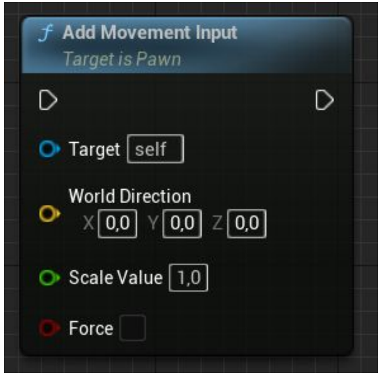
```
void ACharacter::Jump()
{
    bPressedJump = true;
    JumpKeyHoldTime = 0.0f;
}

void ACharacter::StopJumping()
{
    bPressedJump = false;
    ResetJumpState();
}
```

Blueprint: Node Colors

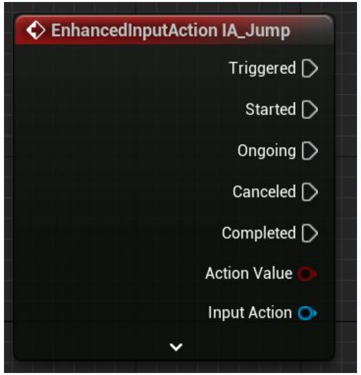
(Pure) Function
Has no execution line. Called when their output is required by an impure node.



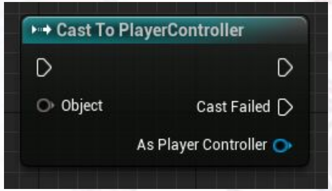
(Impure) Function
Called according to execution line.



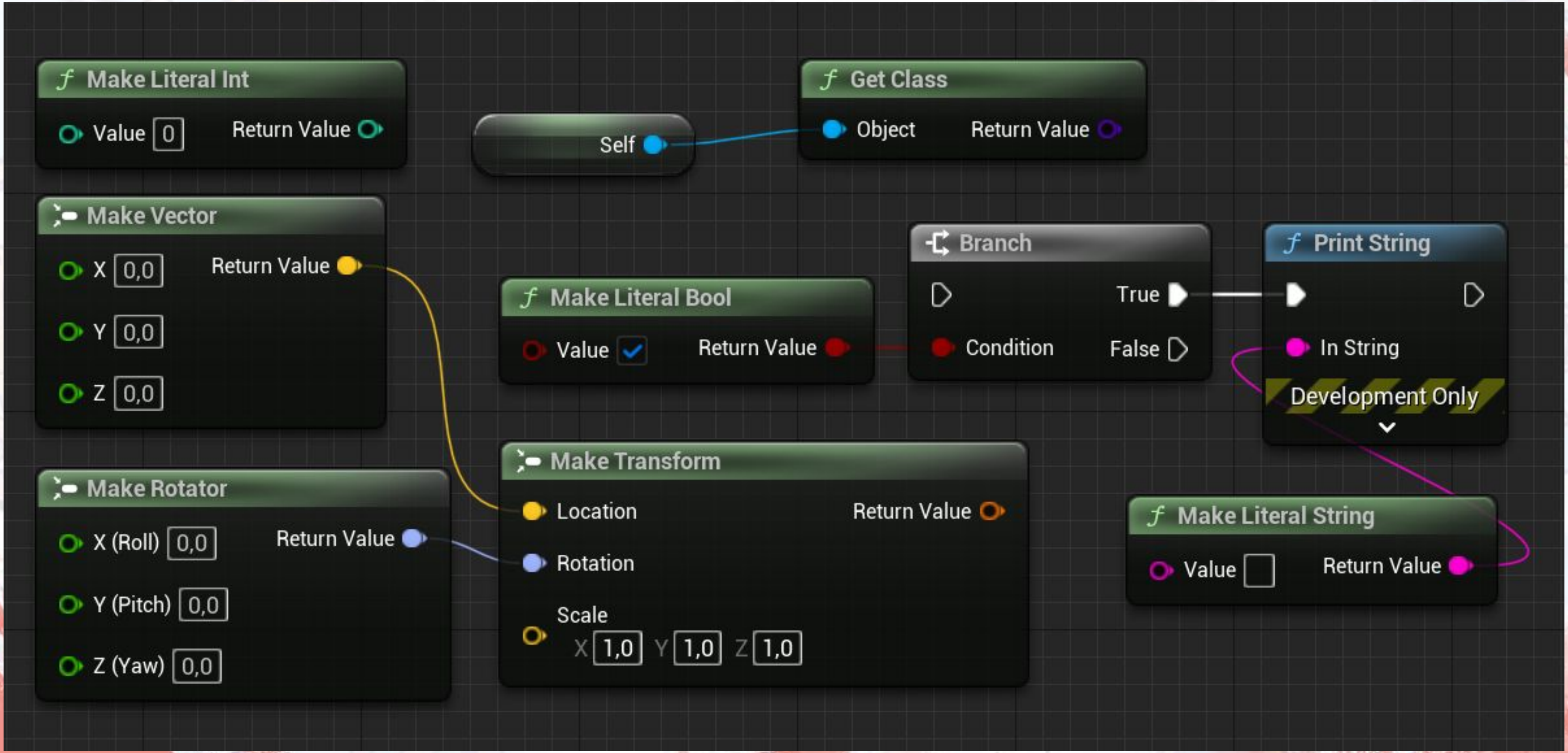
Event
Entry point for execution.



Cast
To convert object pointer into subclass.

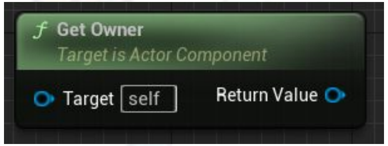


Blueprint: Pin and Wire Colors



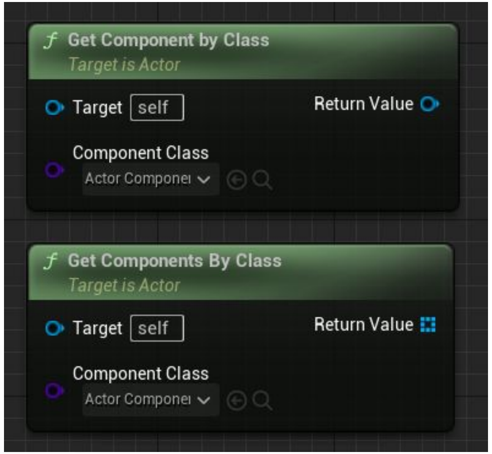
Frequently used BP functions

Actor <-> Component



ƒ Get Owner
Target is Actor Component

Target [self] Return Value



ƒ Get Component by Class
Target is Actor

Target [self] Return Value

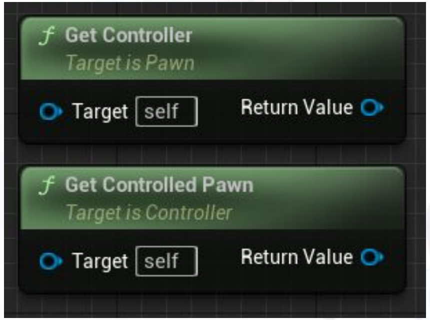
Component Class
Actor Componei

ƒ Get Components By Class
Target is Actor

Target [self] Return Value

Component Class
Actor Componei

Pawn <-> Controller



ƒ Get Controller
Target is Pawn

Target [self] Return Value

ƒ Get Controlled Pawn
Target is Controller

Target [self] Return Value

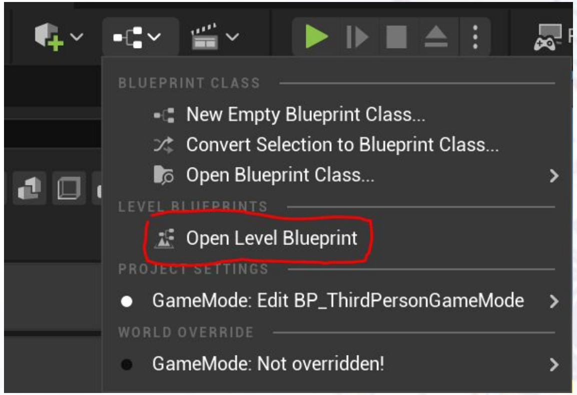
Level Blueprint

- Each level has **one** Level Blueprint

- Parent is: **Level Script Actor**

- It acts as a level-wide global event graph that has references to every Actor in the level

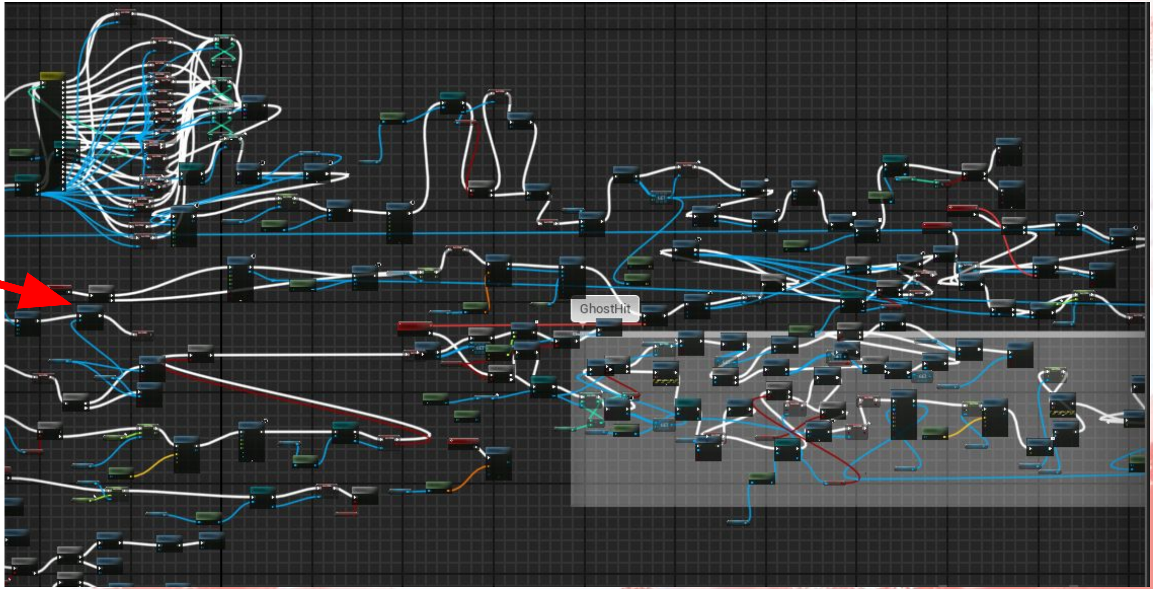
- Should only handle level-specific functionality, as its code is **not re-usable** in other levels


BLUEPRINT CLASS
New Empty Blueprint Class...
Convert Selection to Blueprint Class...
Open Blueprint Class...
LEVEL BLUEPRINTS
Open Level Blueprint
PROJECT SETTINGS
GameMode: Edit BP_ThirdPersonGameMode
WORLD OVERRIDE
GameMode: Not overridden!

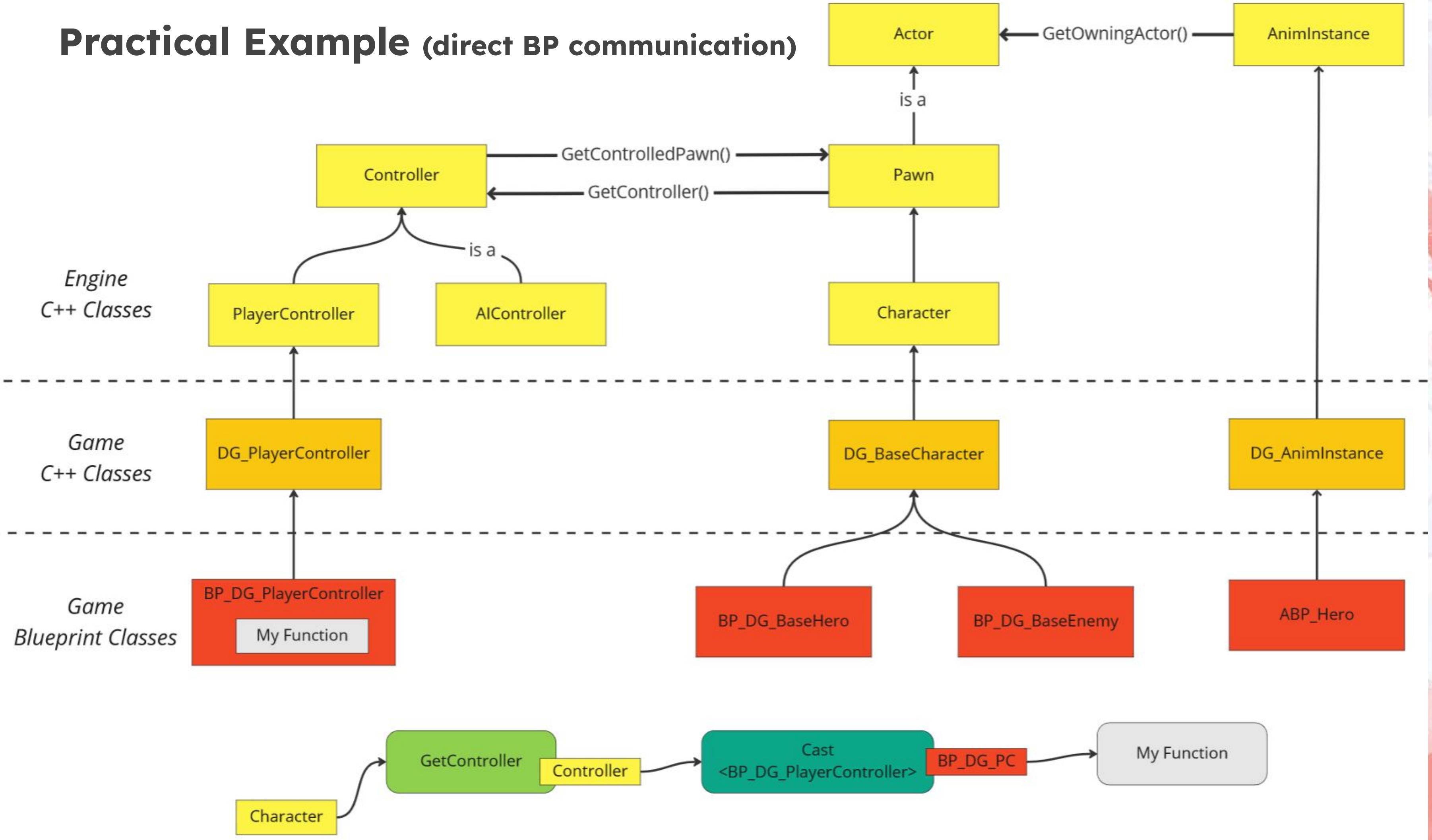
- Don’t get lazy and put everything in the Level BP to avoid Blueprint communication.

C++ vs Blueprint?

- Blueprints are **flexible** (compile without closing the Editor)
  => ideal for rapid prototyping
- Execution is generally **slower** than C++
- Use functions and comments
- Avoid using **Tick** in Blueprint
- Avoid “Spaghetti” Blueprint

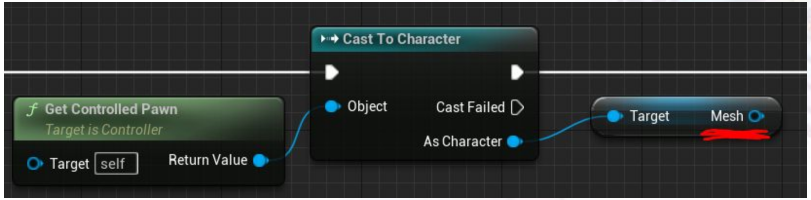


Practical Example (direct BP communication)

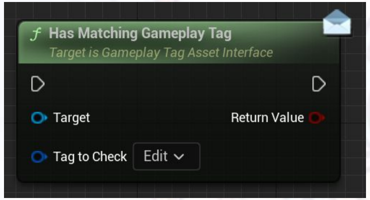


Different types of Blueprint Communication

- Get reference to other object (via casting) and then do a **direct function call** or **access a property**



- Interface call:
  - Get reference to other object and **call interface function**
  - If object **does** implement interface, function will get called
  - If object **does not** implement interface, **nothing happens**


Has Matching Gameplay Tag
Target is Gameplay Tag Asset Interface
Target
Return Value
Tag to Check
Edit

- Event Dispatcher:
  - Objects can **bind** their functions to a sender’s delegate
  - **1-to-many** communication (aka Broadcasting)
  - Implements the well-known **Observer** software design pattern, where the sender does not need to know who the receivers are

UNREAL ENGINE

REFLECTION

Reflection

- Generally, reflection is a mechanism that allows a program to inspect **itself**.
  (in C++ this is done with a lot of “template magic” and exploiting SFINAE)

- In Unreal, reflection **exposes** C++ classes, their functions and properties to the **Unreal Editor**.

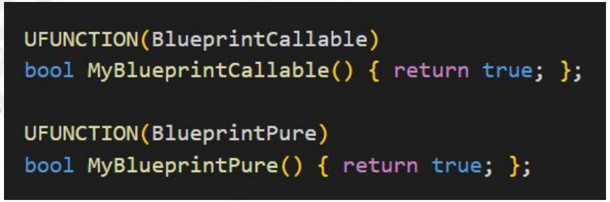
- The necessary code for that is generated at compile time by the **Unreal Header Tool (UHT)**.

- As users, we need to add these macros to our declarations:
  `UENUM()   UCLASS()   USTRUCT()   UFUNCTION()   UPROPERTY()`

- UHT creates <filename>.generated.h - has to be included in header file

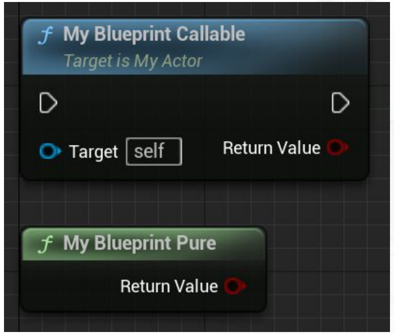
UFUNCTION

C++ functions that can be called from BP:

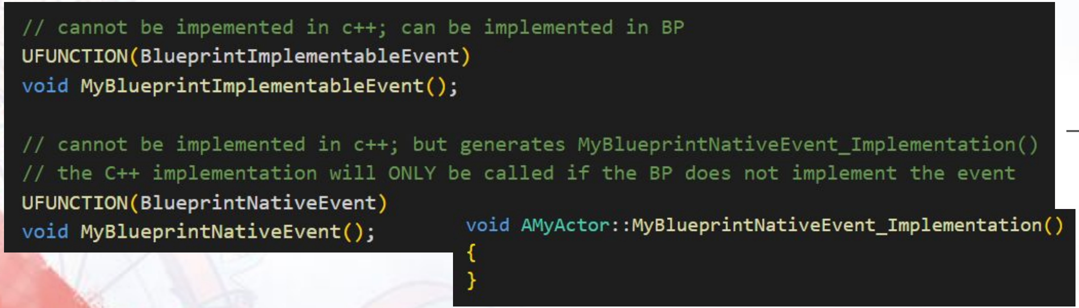


```cpp
UFUNCTION(BlueprintCallable)
bool MyBlueprintCallable() { return true; };

UFUNCTION(BlueprintPure)
bool MyBlueprintPure() { return true; };
```

generates →



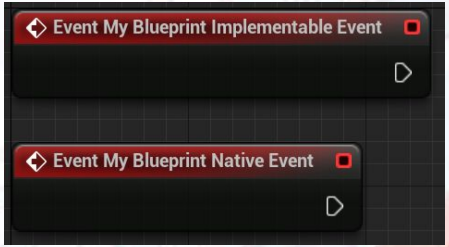
Functions that can be implemented in BP (called from C++):



```cpp
// cannot be impemented in c++; can be implemented in BP
UFUNCTION(BlueprintImplementableEvent)
void MyBlueprintImplementableEvent();

// cannot be implemented in c++; but generates MyBlueprintNativeEvent_Implementation()
// the C++ implementation will ONLY be called if the BP does not implement the event
UFUNCTION(BlueprintNativeEvent)
void MyBlueprintNativeEvent();
```

```cpp
void AMyActor::MyBlueprintNativeEvent_Implementation()
{
}
```

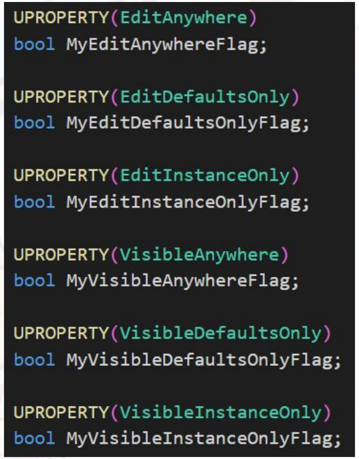
generates →


Event My Blueprint Implementable Event
Event My Blueprint Native Event

List of `UFUNCTION()` specifiers: https://docs.unrealengine.com/5.1/en-US/ufunctions-in-unreal-engine/

UPROPERTY

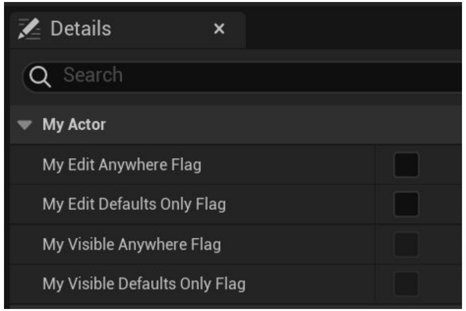
Exposes variable to the Unreal Editor + includes it in Unreal's memory management system.



```
UPROPERTY(EditAnywhere)
bool MyEditAnywhereFlag;

UPROPERTY(EditDefaultsOnly)
bool MyEditDefaultsOnlyFlag;

UPROPERTY(EditInstanceOnly)
bool MyEditInstanceOnlyFlag;

UPROPERTY(VisibleAnywhere)
bool MyVisibleAnywhereFlag;

UPROPERTY(VisibleDefaultsOnly)
bool MyVisibleDefaultsOnlyFlag;

UPROPERTY(VisibleInstanceOnly)
bool MyVisibleInstanceOnlyFlag;
```

generates →

Class Defaults in BP:



Details ×

Search

My Actor

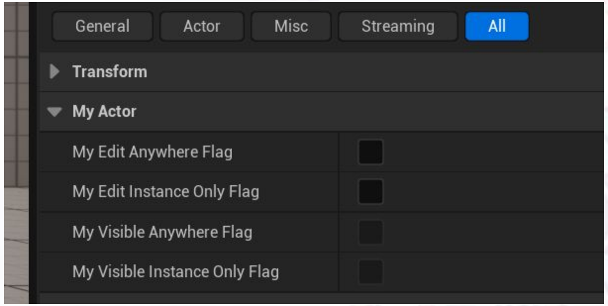
My Edit Anywhere Flag □

My Edit Defaults Only Flag □

My Visible Anywhere Flag □

My Visible Defaults Only Flag □

Actor placed in world:



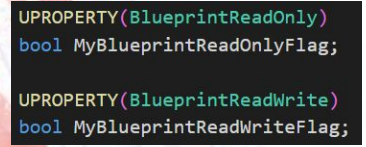
My Actor

My Edit Anywhere Flag □

My Edit Instance Only Flag □

My Visible Anywhere Flag □

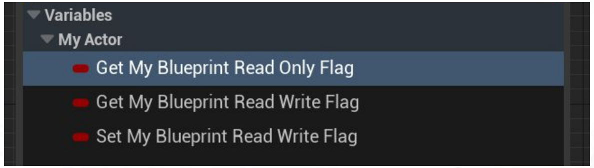
My Visible Instance Only Flag □



```
UPROPERTY(BlueprintReadOnly)
bool MyBlueprintReadOnlyFlag;

UPROPERTY(BlueprintReadWrite)
bool MyBlueprintReadWriteFlag;
```

generates →

In child BP:



Variables

My Actor

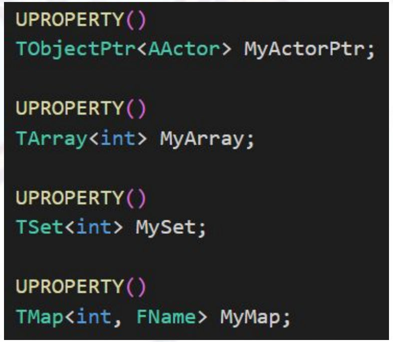
Get My Blueprint Read Only Flag

Get My Blueprint Read Write Flag

Set My Blueprint Read Write Flag

List of UPROPERTY( ) specifiers: https://docs.unrealengine.com/5.1/en-US/unreal-engine-uproperties/

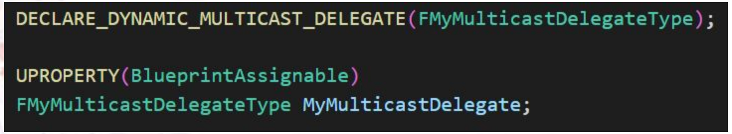
UPROPERTY



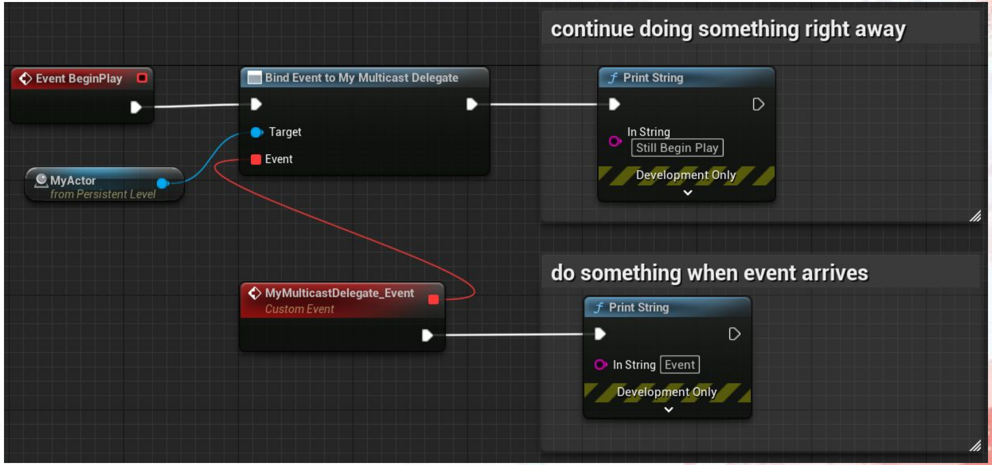
```
UPROPERTY()
TObjectPtr<AActor> MyActorPtr;

UPROPERTY()
TArray<int> MyArray;

UPROPERTY()
TSet<int> MySet;

UPROPERTY()
TMap<int, FName> MyMap;
```

Member variables of non-primitive data types should **always** be a UPROPERTY (to register for garbage collection)

Dynamic multicast delegates are declared and exposed to BP like this:



```
DECLARE_DYNAMIC_MULTICAST_DELEGATE(FMyMulticastDelegateType);

UPROPERTY(BlueprintAssignable)
FMyMulticastDelegateType MyMulticastDelegate;
```



List of UPROPERTY() specifiers: https://docs.unrealengine.com/5.1/en-US/unreal-engine-uproperties/

UNREAL ENGINE

GAME FRAMEWORK &
MOST COMMON CLASSES

Game Mode, Game State, Player State

The **Game Mode** defines the **rules** of game

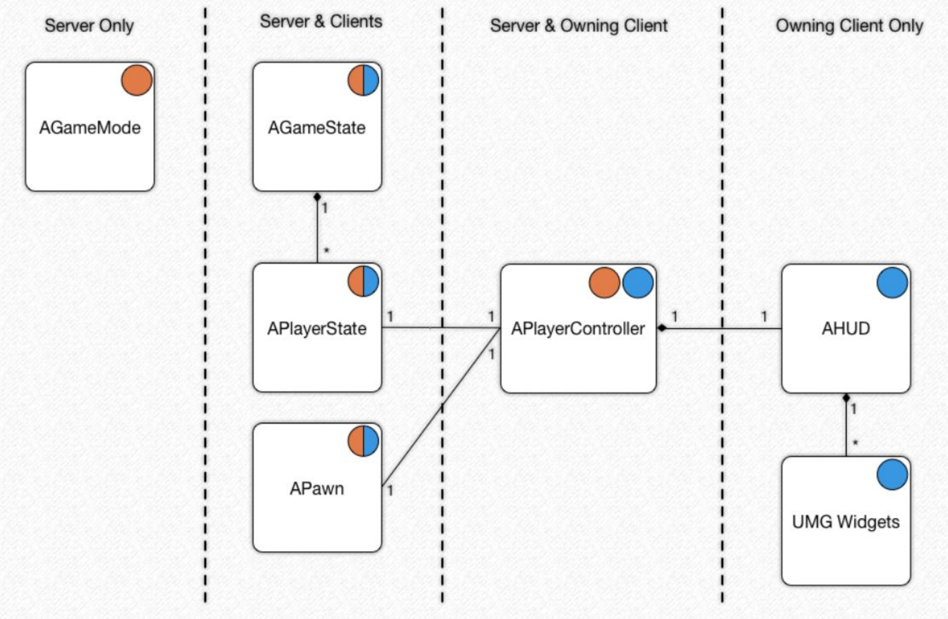
- Handles spawning the players
- **Exists only on the server**
- e.g. evaluates win-lose conditions, how many lives each player starts with

The **Game State** manages information that is relevant for **all** connected players

- e.g. remaining time until round ends, track scores of all players

The **Player State** manages information that is relevant for **one specific** player

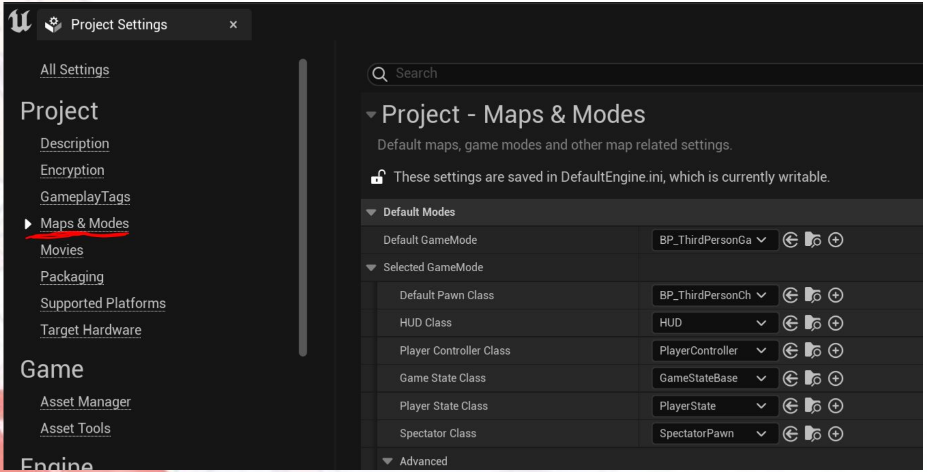
- e.g. username, remaining lives



Note that none of these classes will survive a level change! They share lifetime of UWorld.

Game Mode, Game State, Player State

Project default classes can be set in
Project Settings -> Maps & Modes

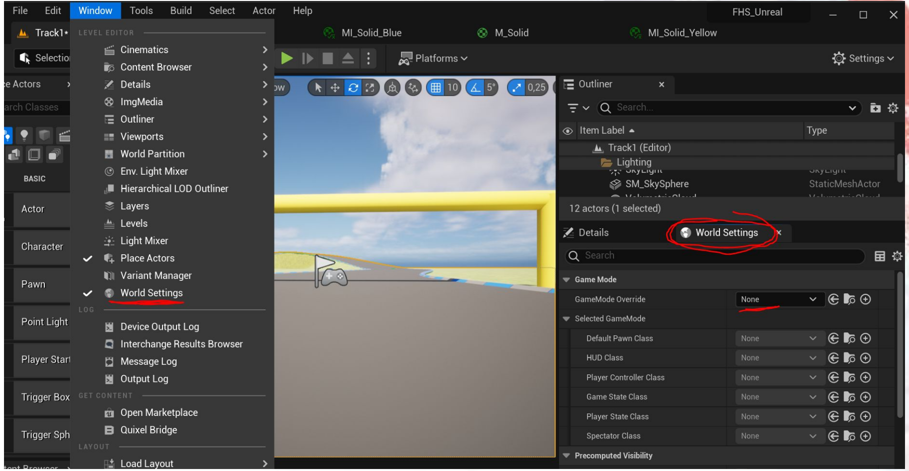


Project - Maps & Modes

Default maps, game modes and other map related settings.

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

Default Modes	
Default GameMode	BP_ThirdPersonGa
Selected GameMode	
Default Pawn Class	BP_ThirdPersonCh
HUD Class	HUD
Player Controller Class	PlayerController
Game State Class	GameStateBase
Player State Class	PlayerState
Spectator Class	SpectatorPawn
Advanced	



Game Mode	
GameMode Override	None
Selected GameMode	
Default Pawn Class	None
HUD Class	None
Player Controller Class	None
Game State Class	None
Player State Class	None
Spectator Class	None
Precomputed Visibility	

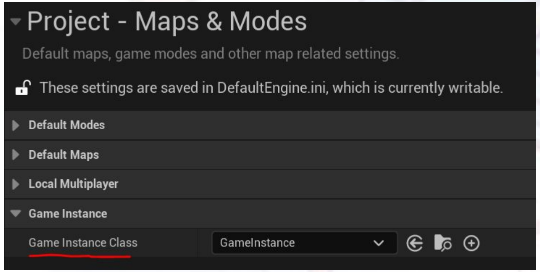
Each world can specify a override in the **World Settings.**

=> Levels can have **different Game Modes**

Game Instance, Local Player, World

Game Instance:

- High-level manager object that persists as long as the game is running
- Ideal place to **store data across maps**
- Can be set in ProjectSettings -> Maps & Modes
- It creates World, LocalPlayer, and GameMode



Project - Maps & Modes

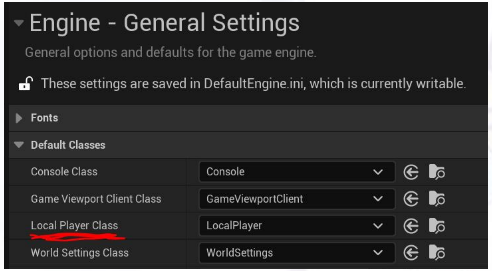
Default maps, game modes and other map related settings.

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

- Default Modes
- Default Maps
- Local Multiplayer
- Game Instance
  - Game Instance Class: GameInstance

Local Player:

- It stays alive across maps
- LocalPlayer triggers the spawn of PlayerController
- Multiple can be created for local multiplayer (splitscreen/coop)



Engine - General Settings

General options and defaults for the game engine.

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

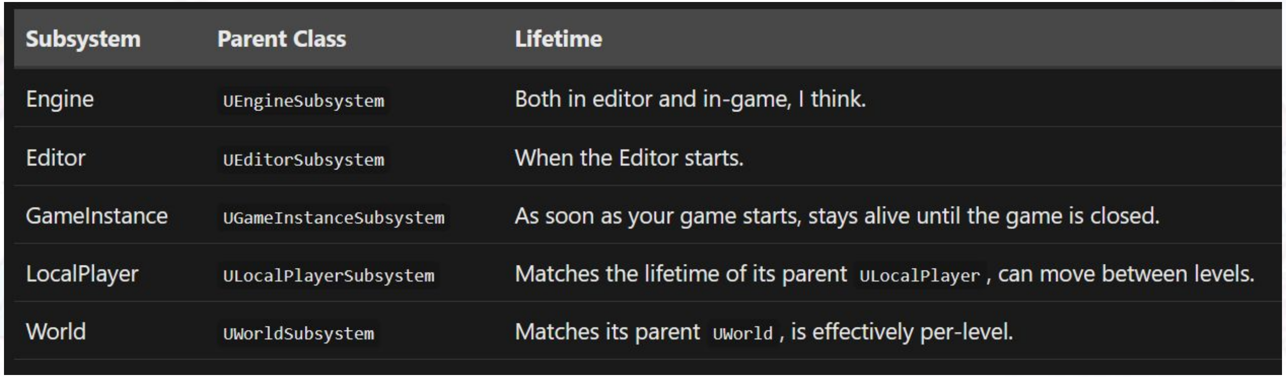
- Fonts
- Default Classes
  - Console Class: Console
  - Game Viewport Client Class: GameViewportClient
  - Local Player Class: LocalPlayer
  - World Settings Class: WorldSettings

World:

- Top-level object representing a map => does not live across maps
- GameMode, GameState and PlayerState share lifetime with World.

Subsystems

Singleton “manager” classes that share lifetime with their parent system.



Subsystem	Parent Class	Lifetime
Engine	`UEngineSubsystem`	Both in editor and in-game, I think.
Editor	`UEditorSubsystem`	When the Editor starts.
GameInstance	`UGameInstanceSubsystem`	As soon as your game starts, stays alive until the game is closed.
LocalPlayer	`ULocalPlayerSubsystem`	Matches the lifetime of its parent `ULocalPlayer` , can move between levels.
World	`UWorldSubsystem`	Matches its parent `UWorld` , is effectively per-level.

Some examples:

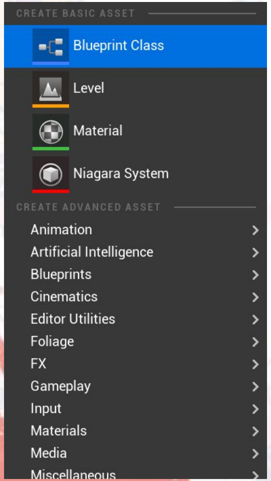
- Unreal’s **Enhanced Input System** is a LocalPlayerSubsystem
- To track score over multiple maps, one could use a GameInstanceSubsystem.
- In a RPG a “quest manager” could be a WorldSubsystem.

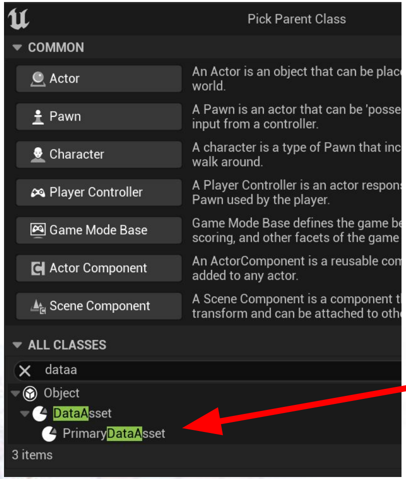
Data Assets

Asset specifically designed to:

- store data + be easily serializable

Allows for **data-driven game design**
(not to be confused with “data-oriented” design - the concept behind ECS)





An Actor is an object that can be plac
world.

A Pawn is an actor that can be 'posse
input from a controller.

A character is a type of Pawn that inc
walk around.

A Player Controller is an actor respon
Pawn used by the player.

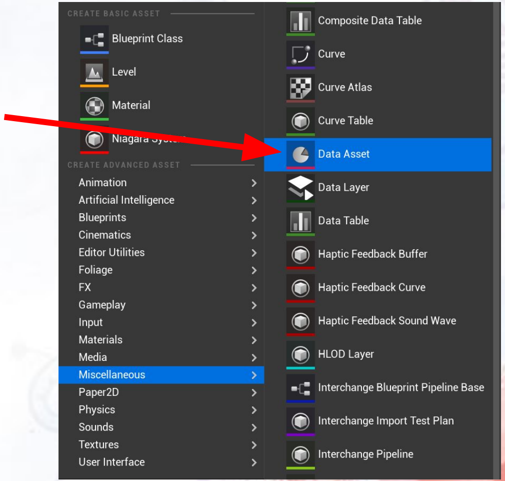
Game Mode Base defines the game be
scoring, and other facets of the game

An ActorComponent is a reusable com
added to any actor.

A Scene Component is a component t
transform and can be attached to oth

Creates a child **Blueprint class** derived from Data Asset

Creates a Data Asset **instance**



Data Assets: How to setup in practice

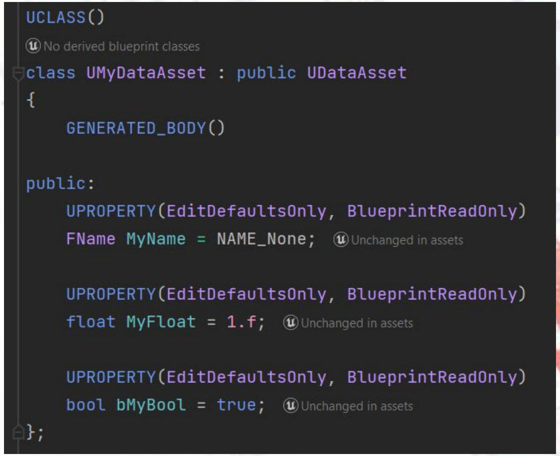
In C++:

- Create a new class that **derives from UDataAsset** and add properties there
- Add a UObjectPtr<UDataAsset> in the class that should use your Data Asset (e.g. MyActor)

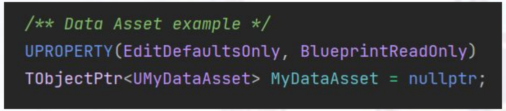
In the Editor:

- Create a **Data Asset instance**
- Assign the Data Asset in your BP class
- Access properties from data asset via the TObjectPtr (has to be BlueprintReadable)

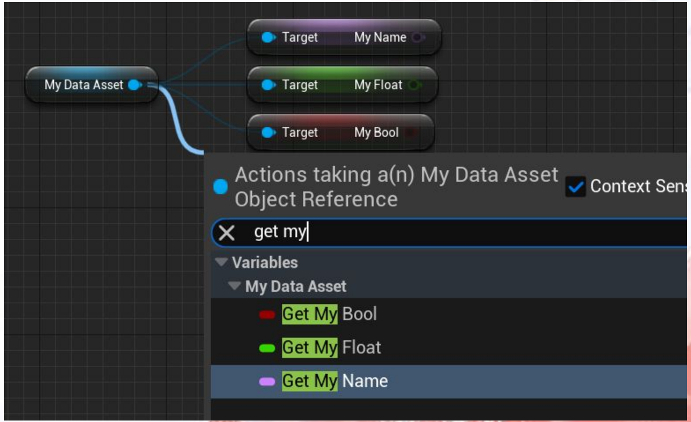
(see example in MyActor + MyDataAsset)



```
UCLASS()
No derived blueprint classes
class UMyDataAsset : public UDataAsset
{
    GENERATED_BODY()

public:
    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
    FName MyName = NAME_None;  Unchanged in assets

    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
    float MyFloat = 1.f;  Unchanged in assets

    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
    bool bMyBool = true;  Unchanged in assets
};
```



```
/** Data Asset example */
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
TObjectPtr<UMyDataAsset> MyDataAsset = nullptr;
```



UNREAL ENGINE

PACKAGING AND PUBLISHING

What is **building**?

- Generally, building is the process of converting source code files into standalone software artifact(s).
- In C++, building is to **compile** the source code (.cpp/.h) into obj-files and then **linking** them into an executable (.exe), a dynamic-load library (.dll) or a static library (.lib).

What is **cooking**?

- Unreal Engine stores content assets in particular formats such as PNG for texture data or WAV for audio. These might not be supported by the target platform.
- The process of converting content from the internal format to the platform-specific format is referred to as **cooking**. (https://docs.unrealengine.com/5.1/en-US/cooking-content-in-unreal-engine/)

What is **packaging**?

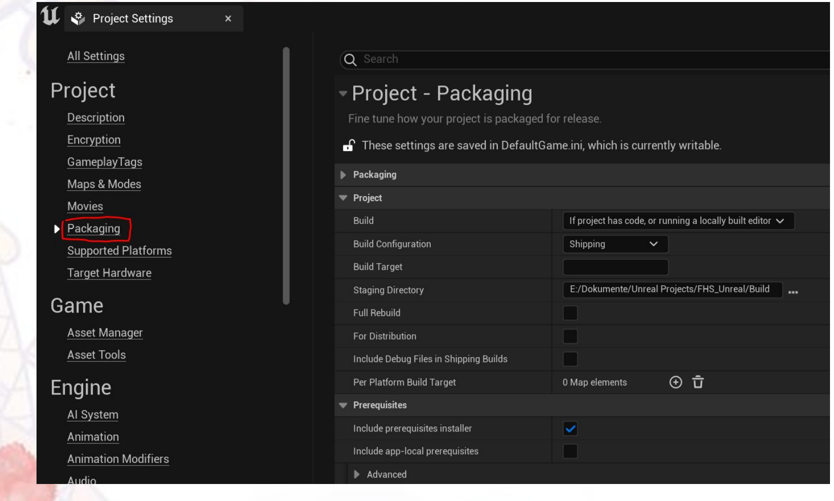
- Building and Cooking the project for a specific target platform (e.g. Windows)
- Output is the “package”: a distributable set of files, such as an installer or the game as executable.

What is **deploying**?

- The process of making software available to be used on a system by users and other programs. (e.g. uploading your game on Steam where people can download it)

BuildCookRun

Packaging is done by the **Unreal Automation Tool** (UAT) with a particular command called **BuildCookRun.**



Project - Packaging

Fine tune how your project is packaged for release.

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

Packaging

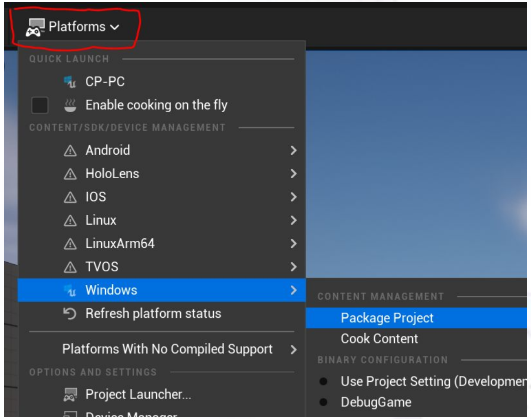
Project

Build	If project has code, or running a locally built editor
Build Configuration	Shipping
Build Target	
Staging Directory	E:/Dokumente/Unreal Projects/FHS_Unreal/Build
Full Rebuild	□
For Distribution	□
Include Debug Files in Shipping Builds	□
Per Platform Build Target	0 Map elements

Prerequisites

Include prerequisites installer	☑
Include app-local prerequisites	□

Advanced

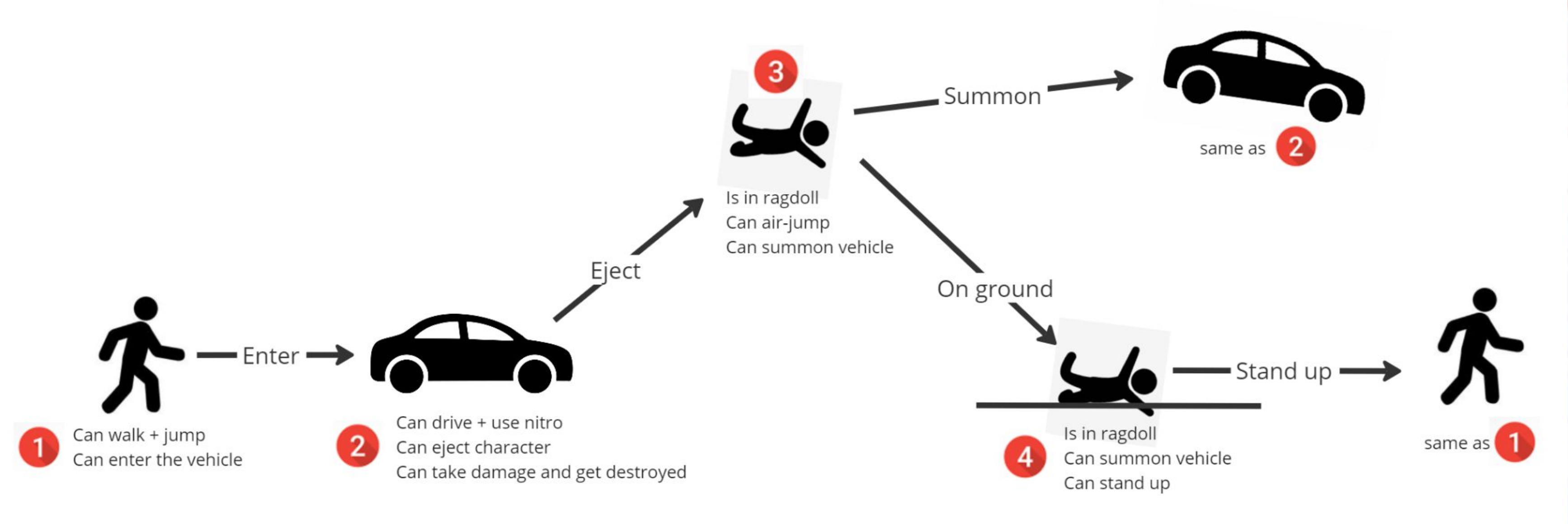


```
"%ENGINE_DIR%\Engine\Build\BatchFiles\RunUAT.bat" BuildCookRun
-project="%PROJECT_DIR%\%PROJECT_NAME%.uproject" -noP4 -platform=Win64
-config=%BUILD_CONFIGURATION% -build -cook -stage -pak -allmaps -CrashReporter -archive
-archivedirectory="%ARCHIVE_DIRECTORY%"
```

STUNTFEST™ WORLD TOUR

LET'S TAKE A LOOK AT STUNTFEST

Key Gameplay Features of Stuntfest



- Different Game Modes (race, distance jump, …)

Resources & References

- Epic’s Documentation: https://docs.unrealengine.com/5.1/en-US/

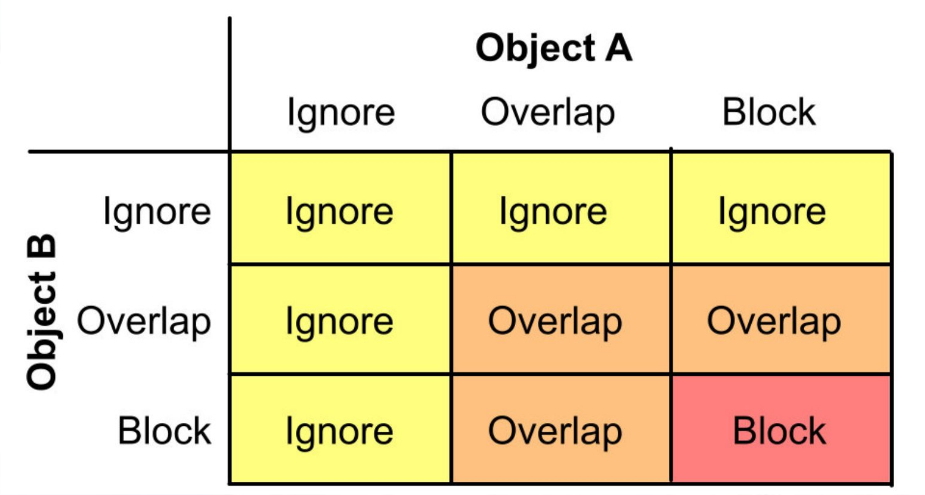
- Blog by Ben Humphrey: https://benui.ca/unreal/
- Blog by Tom Looman: https://www.tomlooman.com/?post_type=post
- Blog by Nuno Afonso: http://www.nafonso.com/

- Network Compendium by Cedric 'eXi' Neukirchen: https://cedric-neukirchen.net/Downloads/Compendium/UE4_Network_Compendium_by_Cedric_eXi_Neukirchen.pdf

- John Coogan - Why Epic Games Took 25 Years to make Fortnite: https://youtu.be/vNbrhLf36Uo

Bonus Slide: Collisions between two moving objects

- Each object has its own collision profile
- The response is the **least blocking** one, like so:



		Object A		
		Ignore	Overlap	Block
Object B	Ignore	Ignore	Ignore	Ignore
	Overlap	Ignore	Overlap	Overlap
	Block	Ignore	Overlap	Block