Michele Mischitelli

Introduction to Unreal Engine 4

Game engines are no longer used (just) for games...

Recap of my life

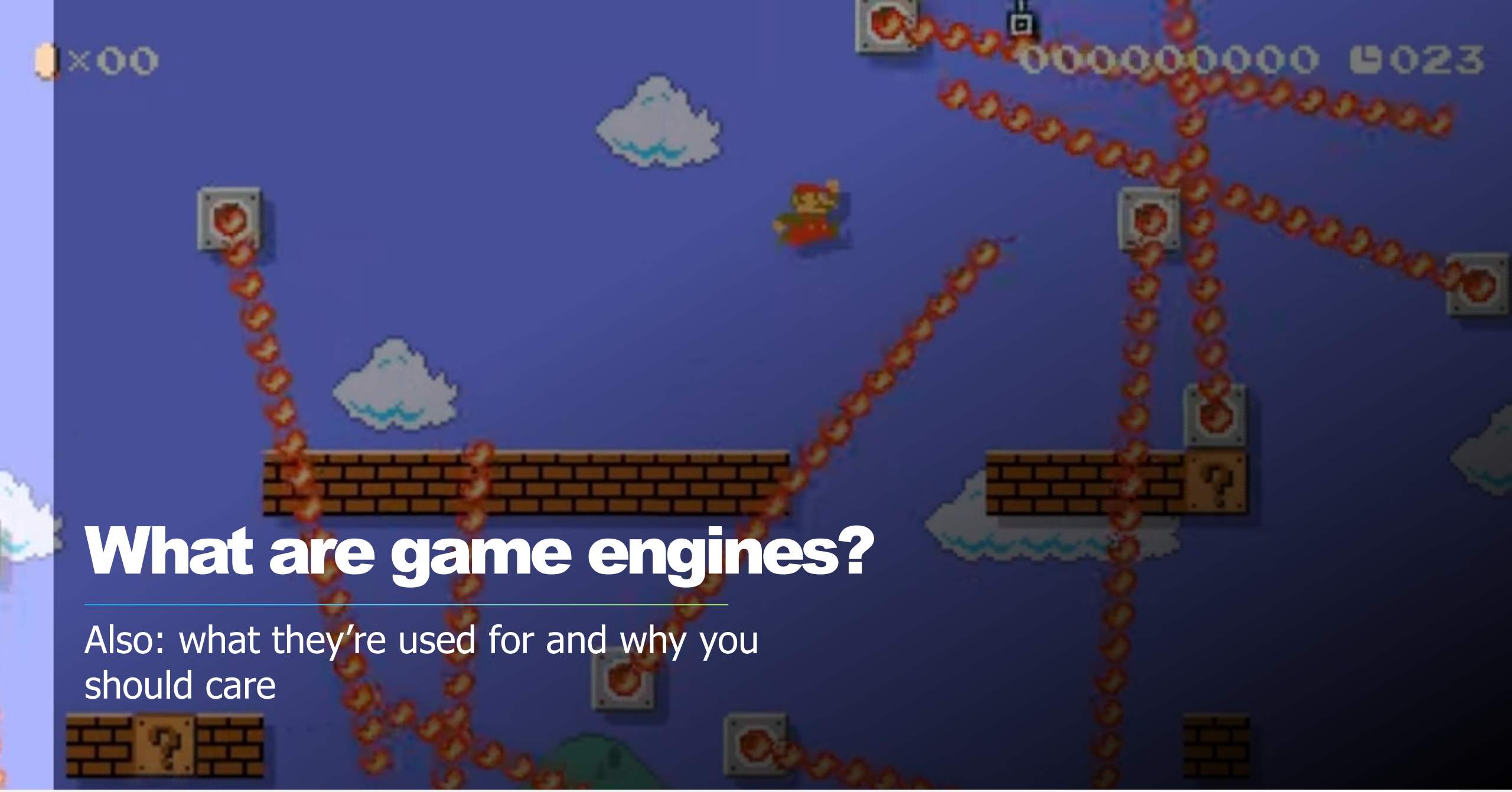
(as developer, of course…)

- 1985: New Coke, Back to the Future, Windows 1.0
- 1996: MSX Basic -> started developing
- 1998: First PC -> AMD K6 200MHz + S3 Virge DX
- 2000: Visual Basic. Backup + Installers
- 2001: ITIS -> Turbo Pascal, Assembler, C
- 2007: CNR Pisa Trainee (OpenGL, Qt, C++)
- 2008: Hypersoft -> TSim-X (C++/C#)
- 2016: DigiCamere -> Web (>_<)
- 2016: Astron -> Astrophotography + Dev (C#)
- 2016: Zuru -> C++ / UE4
- 2019: Scuderia Ferrari -> C++

Who am I…?

Software developer, graphics aficionado, photographer.

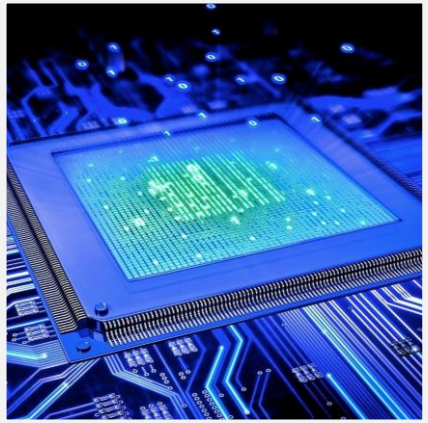
My current themes: cyberpunk, sci-fi, retro-futurism



What are game engines?

Also: what they’re used for and why you should care

Game engines: software frameworks (also IDEs !)



Hardware and OS abstraction layer

We want our game to run on any platform

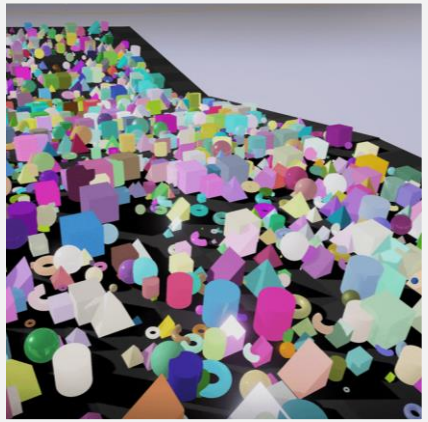
Our engine should be HW and OS independent

Domain engines

Graphics, Physics, Audio and Network are the 4 main sub-engines that compose any game engine

Game logic

Event-driven architecture that allows the various subsystems and actors to interact as result of user input



Runtime objects

Everything that is spawned during the execution of the game

Multiplatform & customizability

Easy to port on other platforms

PC, Mac, Linux, Xbox, PlayStation, Switch, VR…

Tools for devs and designers

Terrain editing, bug reporting, scripts, asset importing

Can be used for different games

RPG that is also an FPS that also makes gamers use vehicles

The engine itself can be sold…

Profits are profits! Good engines are sold to other companies…

Game industry

Game engines are… well, used for games!

GODOT Game engine

DECIMA

FROSTBITE

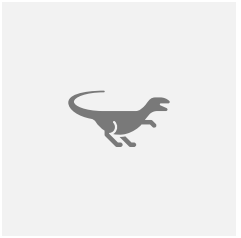
UNREAL ENGINE

CRYENGINE 3

Traditional film production workflow

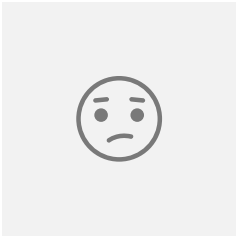
It is like playing an instrument you don't know and hearing the music only weeks after you hit the first note

Want an example?



Digital elements created in post

Placed in scenes already filmed



Set lights don't work with them

Artists are finally able to visualize and choose

Go back to previous stages

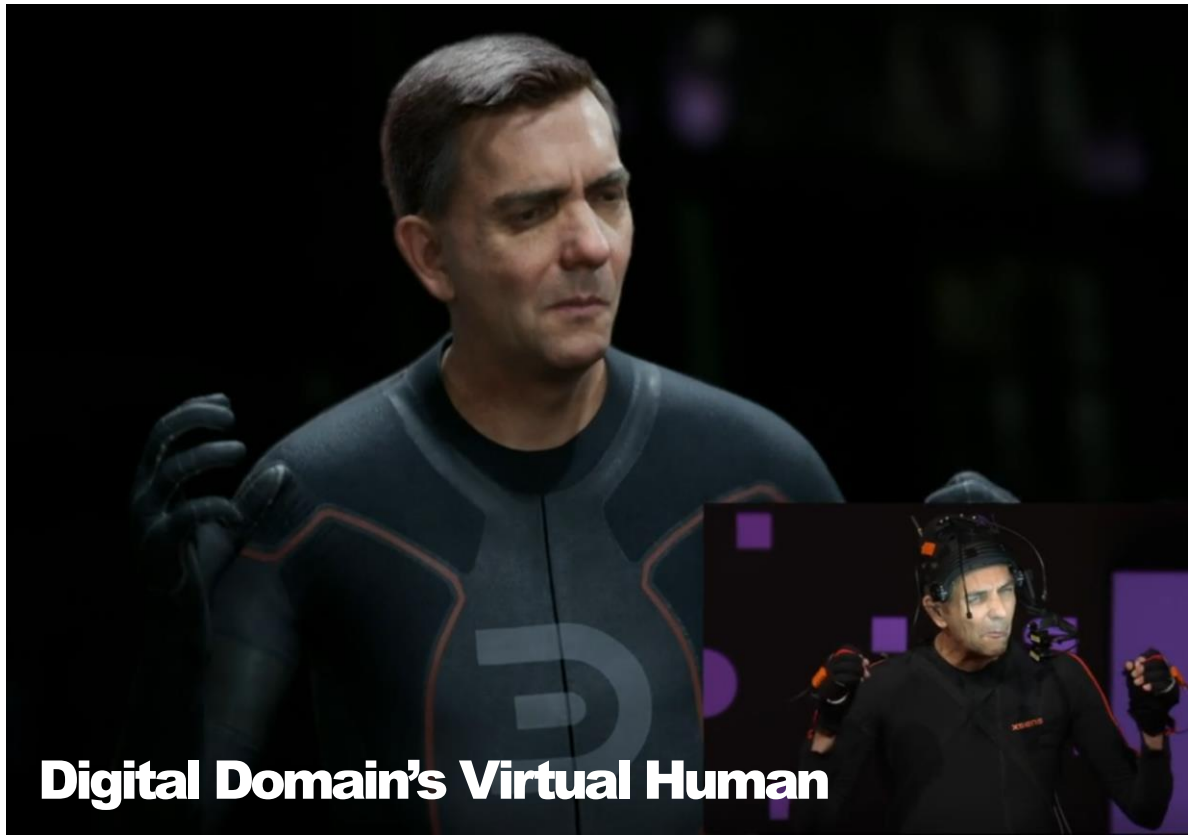
Waste of time and money

Film industry

Tight schedules and lower budget drive interest for RT rendering, while improving workflow

Making film industry more *Agile*

- Encourages a more iterative, non-linear and collaborative process
- Filmmakers collaboratively iterate on visual details on the fly
- Iteration begins much earlier in the production schedule
- High quality imagery can be produced from the outset
- Assets are cross-compatible and usable from pre-vis through final output
- Live production and VFX can occur in parallel



> **"Every hour of pre-production is worth two hours of production"**
>
> *Zach Alexander, founder and COO of Lux Machina*

Constantly reaching for higher fidelity

Drawings and watercolour

Offline rendering

90s: CAD software

Realtime rendering… ?

Saved architects, engineers and designers time and money

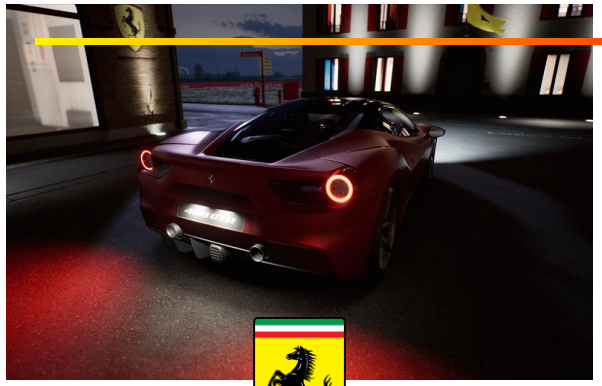
Customers wanted higher quality due to films and CG

Rendering time for animations and stills drive interest for RT

ArchViz

Architecture found in RT rendering a solution to the visualization problem

Differentiation and prototyping



Ferrari and Mackevision created a realistic real-time digital showroom

BMW brings mixed reality to automotive design

Automotive

Car manufacturers use real-time workflows for marketing, design and showrooms

Porsche, together with Nvidia and Epic, revealed a real-time cinematic experience introducing ray-tracing in a game engine

Let's start talking about UE4

One of the most popular and versatile game engine

```cpp
#pragma once

#include "GameFramework/Actor.h"
#include "MyActor.generated.h"


UCLASS()
class AMyActor : public AActor
{
    GENERATED_BODY()

public:

    // Sets default values for this actor's properties
    AMyActor();

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

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

};
```

Unreal Engine 4

C++ development intro

Full access to the engine's source

Can be customized and you can *get inspired*

UE's *Assisted* C++

Alternative to STL and Boost. Epic affirms it's easier to work with

Everything can be done in C++

Even UI, thanks to Slate (but please, don't…)

Constantly updated

A new engine version every 4-5 months with new features and fixes

Two ways of programming in Unreal

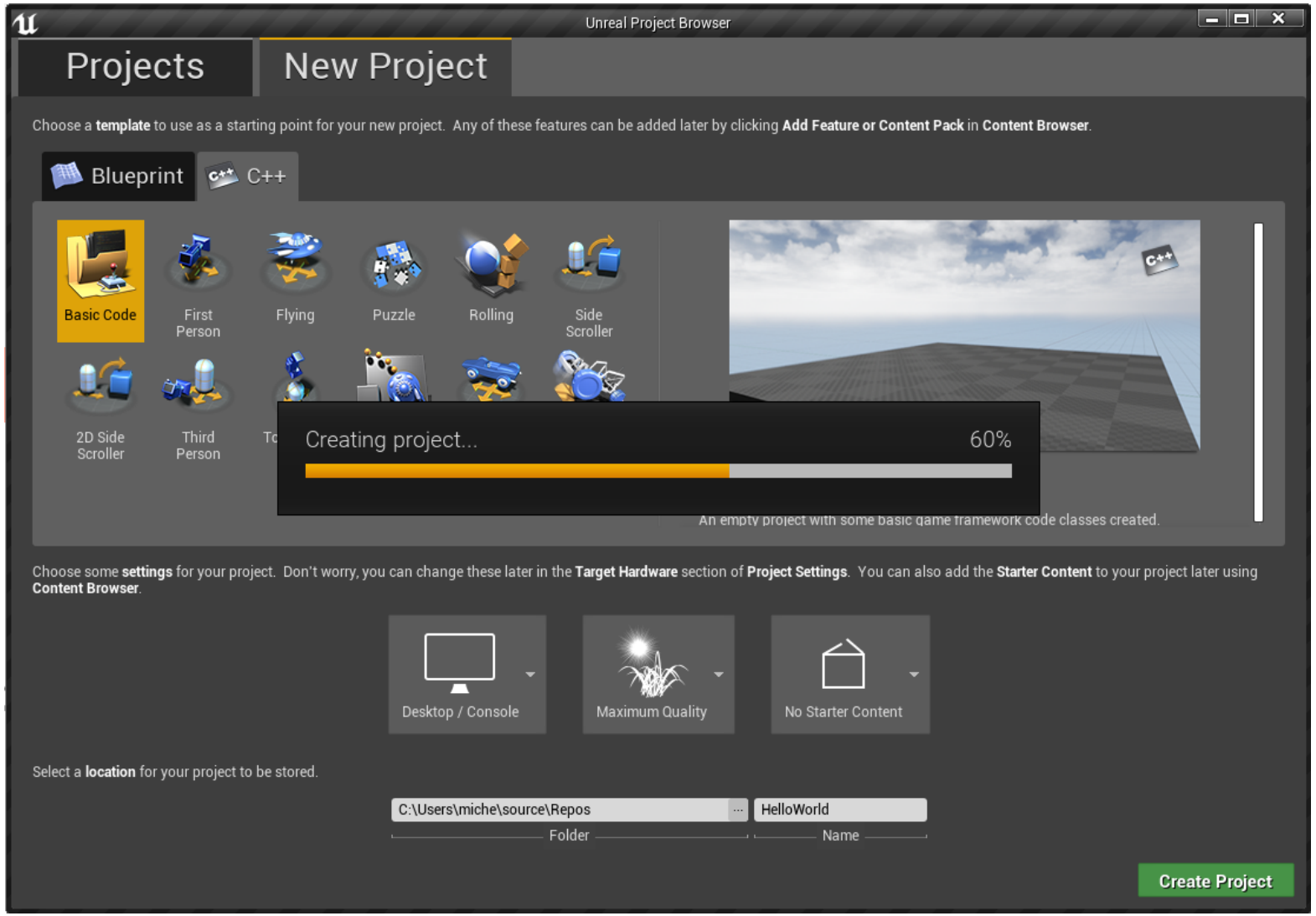
Blueprints

- **PRO**
  - Fast to learn (if unexperienced with c++)
  - Rapid prototyping
  - Mandatory for UI
- **CONS**
  - Slower execution
  - Binary files (hard to work with in teams)
  - Easy to make a mess → Hard to decode
  - No support for merge/diff (although…)

C++

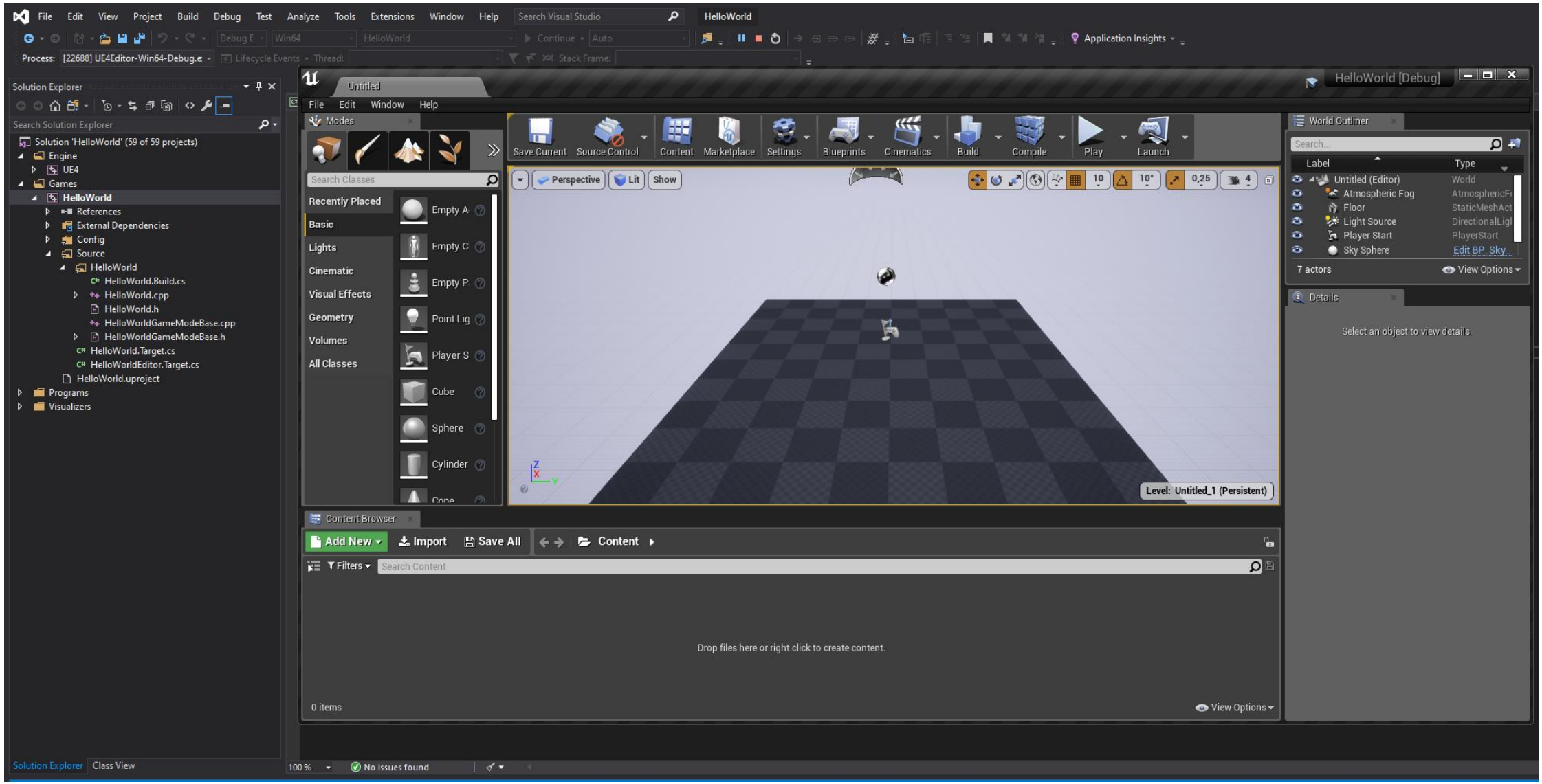
- **PRO**
  - Full access to UE4's source code
  - UE4's assisted C++
  - Fast execution
  - Flexibility
  - Source control support (merge, rebase…)
- **CONS**
  - Hard to learn

Hello, world! – Creating the project



- Launching UE4 brings up this
- Template selector, like VS's File → New → Project
- Many templates, both Blueprint-based and C++
- Can include starter logic and actors to jump start the development
- Starter content also available (materials, textures…)

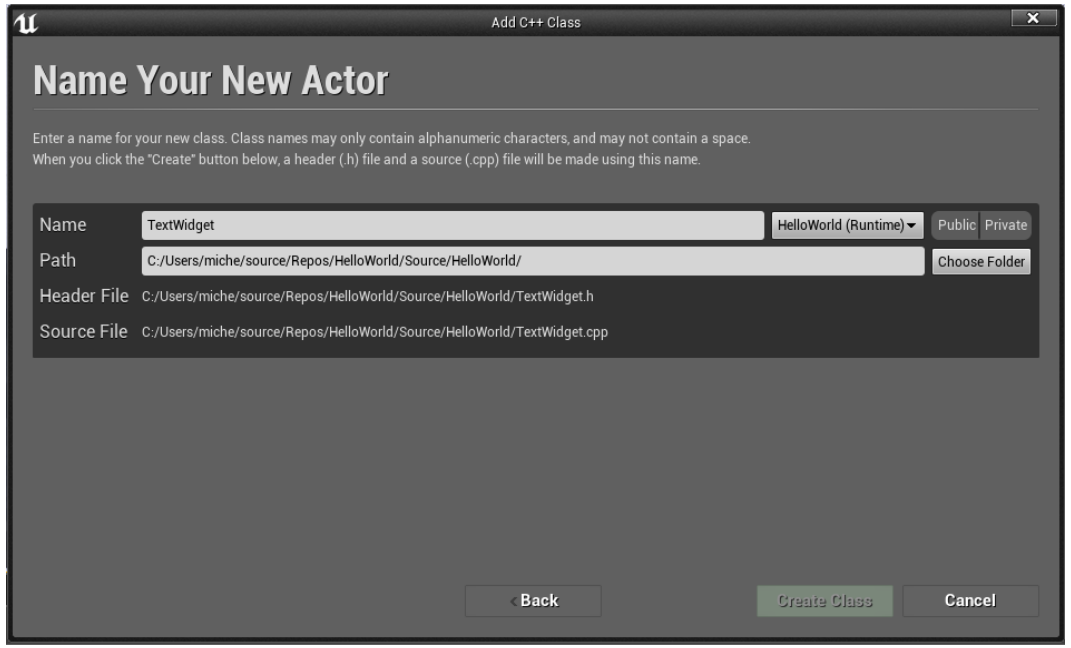
Hello, world! – It lives!

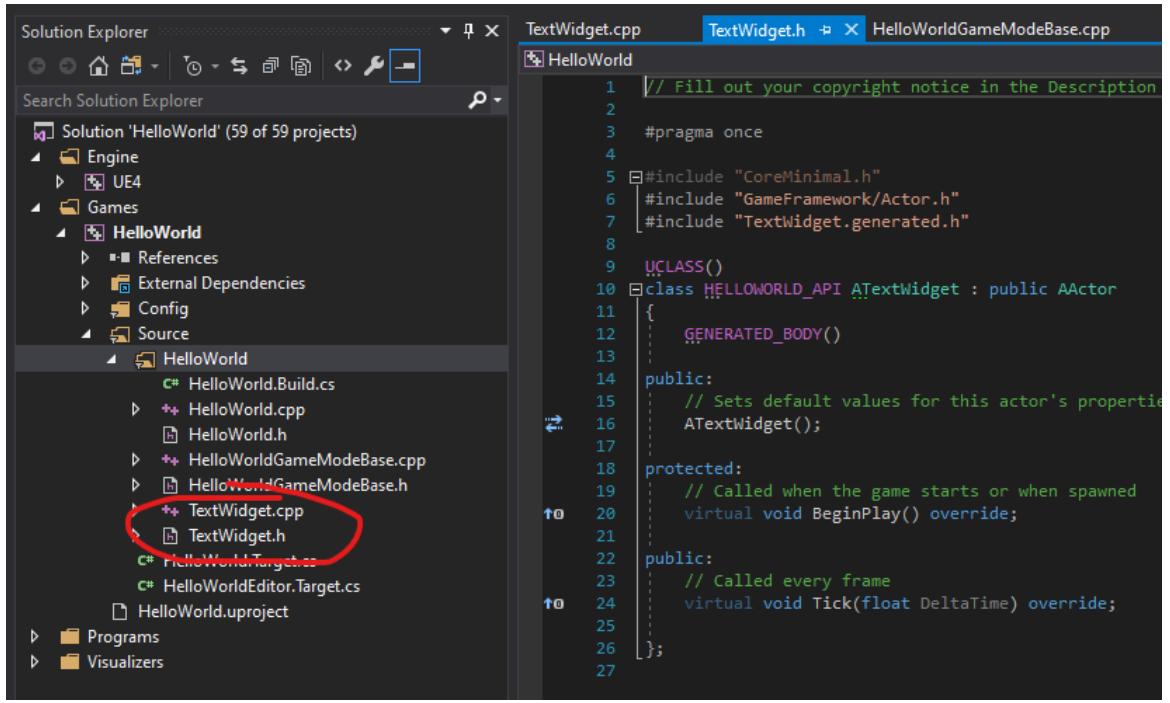


Hello, world! – Creating an actor

Create C++ classes
from within the editor

VS's solution is
updated live





Hello, world! – Actually say hello

- TextWidget.h

```
class UTextRenderComponent;

UCLASS()
class HELLOWORLD_API ATextWidget : public AActor
{
    GENERATED_BODY()

    UPROPERTY()
    UTextRenderComponent* m_TextRenderComp;
```

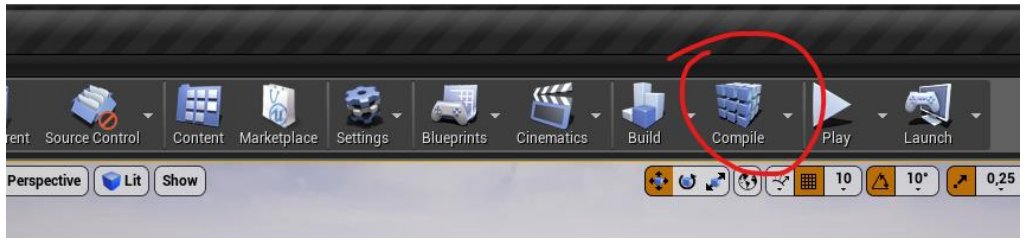
- TextWidget.cpp

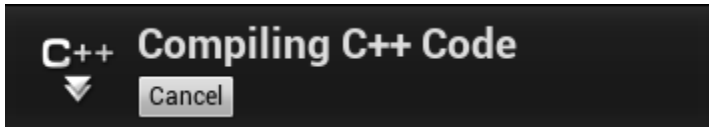
```
// Sets default values
ATextWidget::ATextWidget()
{
    // Set this actor to call Tick() every frame.  You can turn this off to improve performan
    PrimaryActorTick.bCanEverTick = true;

    m_TextRenderComp = CreateDefaultSubobject<UTextRenderComponent>("TextRenderComponent");
    m_TextRenderComp->SetText( Value: "Hello, World!");
    m_TextRenderComp->SetTextRenderColor( Value: FColor::FromHex("FF2800"));
}
```

- HelloWorldGameModeBase.cpp

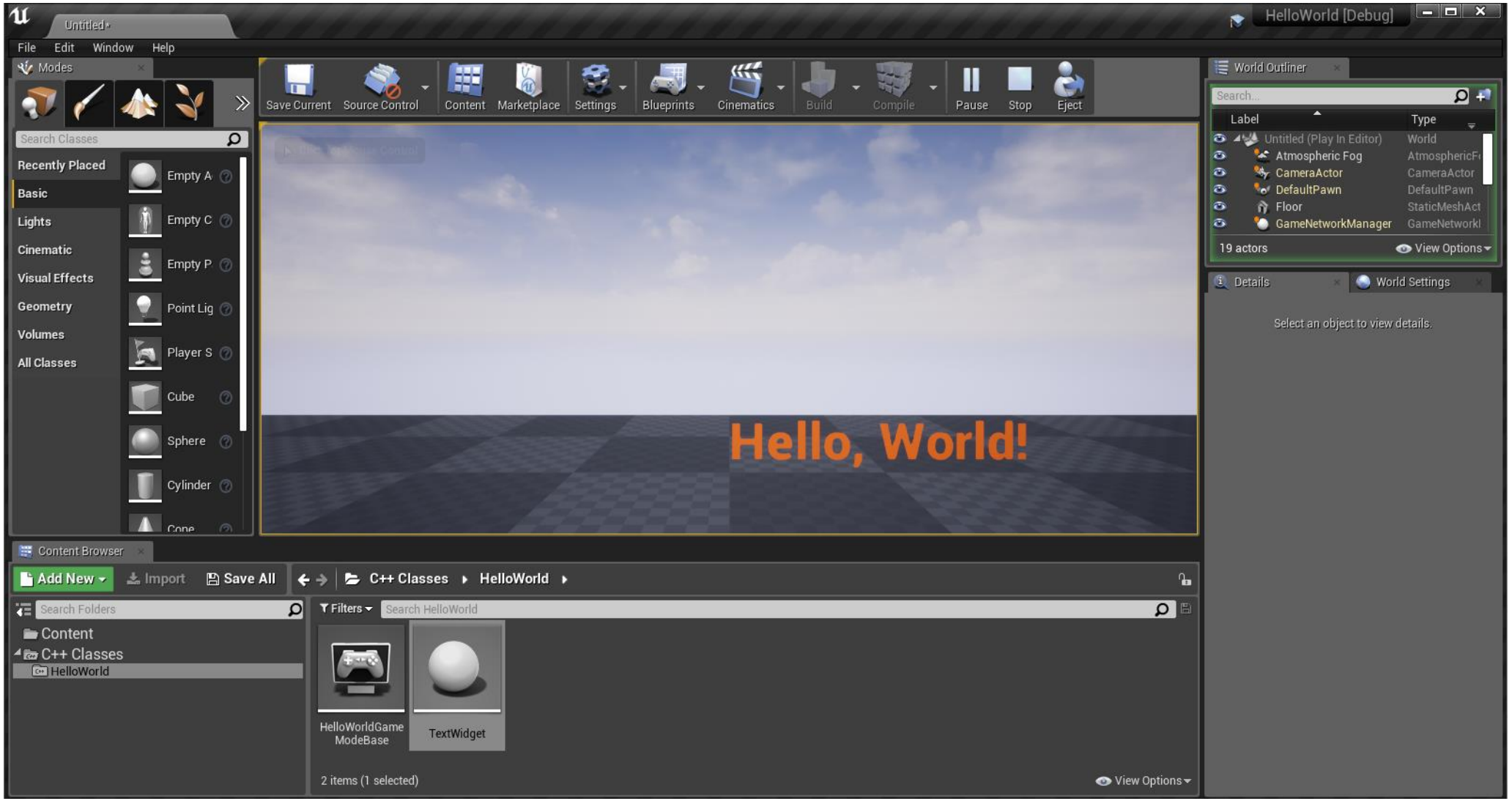
```
void AHelloWorldGameModeBase::StartPlay()
{
    auto actor = GetWorld()->SpawnActor<ATextWidget>();
    actor->SetActorLocationAndRotation({ InX: 200, InY: 0, InZ: 50 }, NewRotation: FQuat( InX: 0, InY: 0, InZ: 1, InW: 0));
}
```

- Click compile without closing UE… meanwhile, VS is still debugging… ;)





Hello, world! – Tadaaan!



Hello, <whatever>

Improving the sample with Unreal-*esque* interactions

TextWidget.h

```
UCLASS()
class HELLOWORLD_API ATextWidget : public AActor
{
    GENERATED_BODY()

    UPROPERTY()
    UTextRenderComponent* m_TextRenderComp;

    UPROPERTY(EditAnywhere, Category="Mischitelli", meta=(DisplayName="Target to salute", AllowPrivateAccess=true))
    FString m_Text;

public:
    ATextWidget();

protected:
    void BeginPlay() override;

#if WITH_EDITOR
    void PostEditChangeProperty(FPropertyChangedEvent& PropertyChangedEvent) override;
#endif  #if WITH_EDITOR

private:
    void _UpdateText(const FString& target);
};
```

Create a new **UPROPERTY** that will hold the customizable text

Define some attributes:
- EditAnywhere
- Category
- meta

Optionally, override the **`PostEditChangeProperty`** method.

Beware! It's declared only in Editor mode!

Hello, <whatever>

TextWidget.cpp

```
ATextWidget::ATextWidget()
{
    m_Text = "World";

    m_TextRenderComp = CreateDefaultSubobject<UTextRenderComponent>("TextRenderComponent");
    m_TextRenderComp->SetText(TEXT("Placeholder"));
    m_TextRenderComp->SetTextRenderColor(FColor::FromHex("FF2800"));

    SetRootComponent(m_TextRenderComp);
}

void ATextWidget::BeginPlay()
{
    Super::BeginPlay();
    _UpdateText(m_Text);
}

#if WITH_EDITOR
void ATextWidget::PostEditChangeProperty(FPropertyChangedEvent& e)
{
    Super::PostEditChangeProperty(e);

    const FName kPropName = (e.Property != nullptr) ? e.Property->GetFName() : NAME_None;
    if (kPropName == GET_MEMBER_NAME_CHECKED(ATextWidget, m_Text))
    {
        auto* valuePtr = e.Property->ContainerPtrToValuePtr<FString>(this);
        if (valuePtr)
        {
            _UpdateText(*valuePtr);

            /* This is equivalent to the above line! */
            //_UpdateText(m_Text);
        }
    }
}
#endif #if WITH_EDITOR

void ATextWidget::_UpdateText(const FString& target)
{
    m_TextRenderComp->SetText(FString::Printf(TEXT("Hello, %s!"), *target));
}
```

1) Define a default value that **`m_Text`** will hold

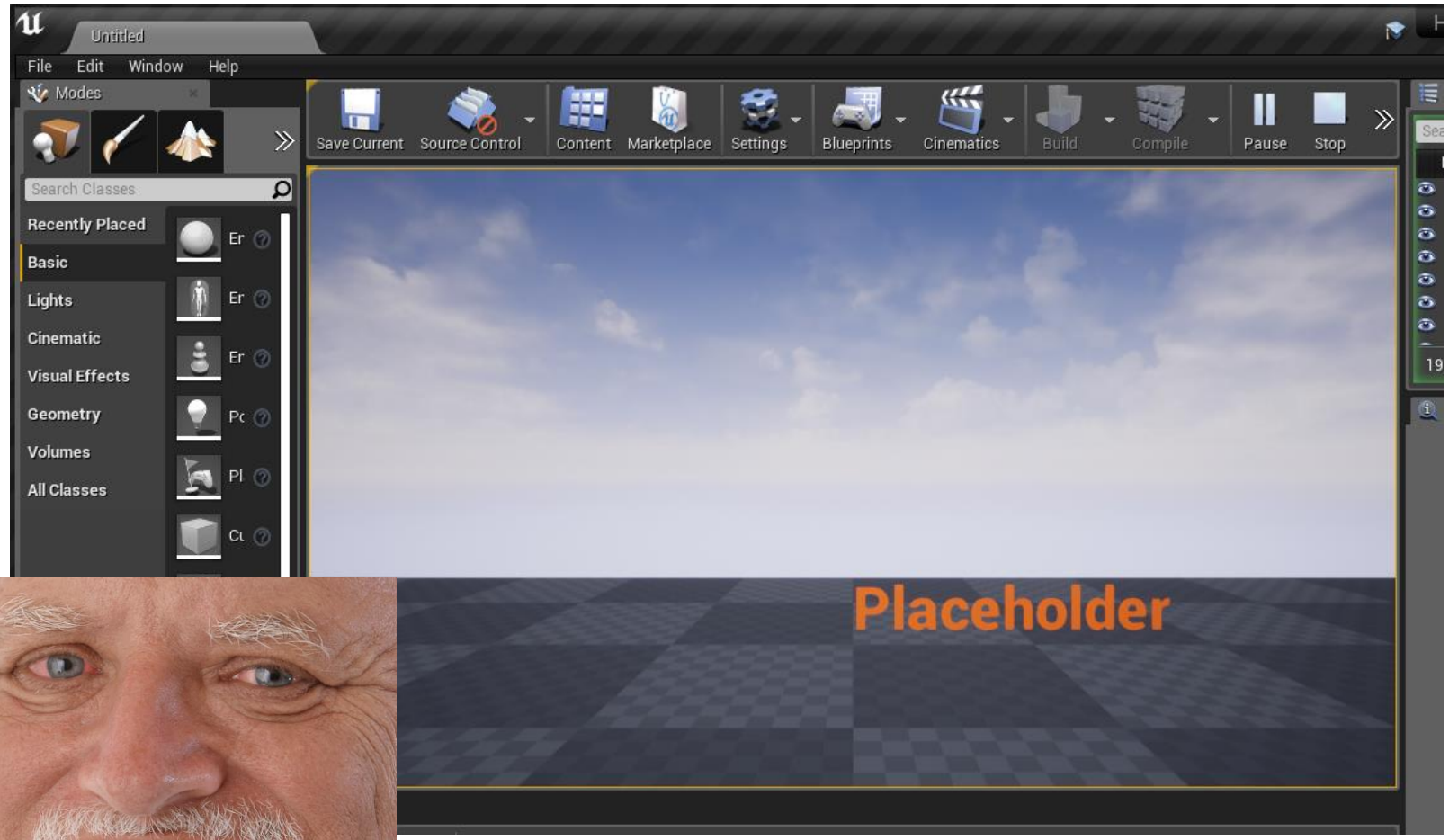
2) To understand exactly what happens, let's give the text render component a placeholder text

3) Move the default text assignment from the CTOR to the **`BeginPlay`** method

4) Define the **`PostEditChangeProperty`**. It acts very much like **`PropertyChanged`** (C#/XAML)

5) Introduce a utility method to update the text render component

Hello, Placeholder … my old friend



Were you expecting **Hello, World!** to show up…?

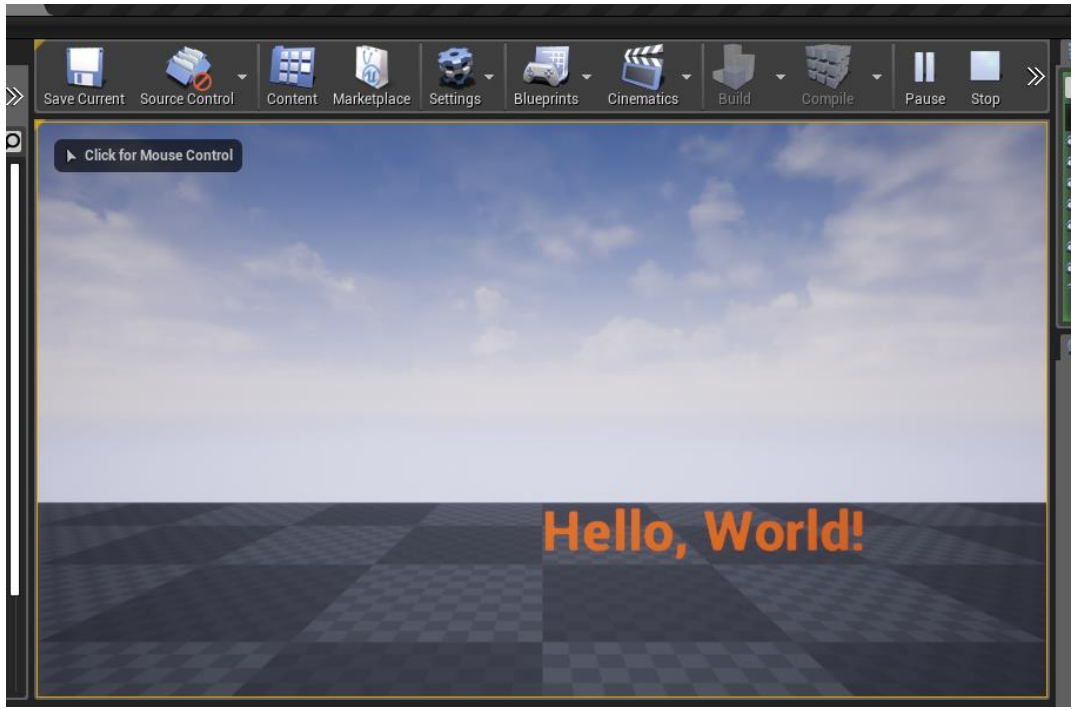
You are right. I made a mistake…

`ATextWidget` it's alright. The problem is somewhere else…

What are we really missing here…? What piece of code is apparently not getting executed…?

Hello, bugs

```
void AHelloWorldGameModeBase::StartPlay()
{
    // Don't forget to call the Super on UE-declared virtual methods!
    Super::StartPlay();

    auto actor = GetWorld()->SpawnActor<ATextWidget>();
    actor->SetActorLocationAndRotation({ 200,0,50 }, FQuat(0, 0, 1, 0));
}
```



`StartPlay` signals the game has started playing

It sets an internal flag in the current world to true: **`bBegunPlay`**

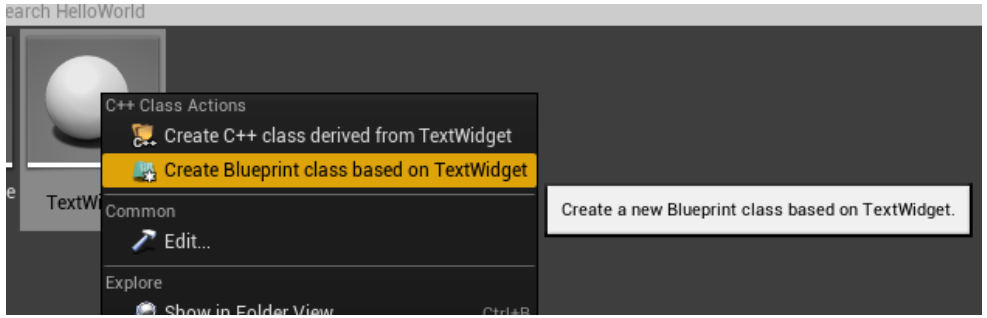
If that flag is false, **`BeginPlay`** events on objects won't get called

To fix the bug, it's sufficient to forward the method call on the parent's

Alternatively…

```
void AHelloWorldGameModeBase::HandleStartingNewPlayer_Implementation(APlayerController*
{
    auto actor = GetWorld()->SpawnActor<ATextWidget>();
    actor->SetActorLocationAndRotation({ 200,0,50 }, FQuat(0, 0, 1, 0));
}
```

Hello, blueprints

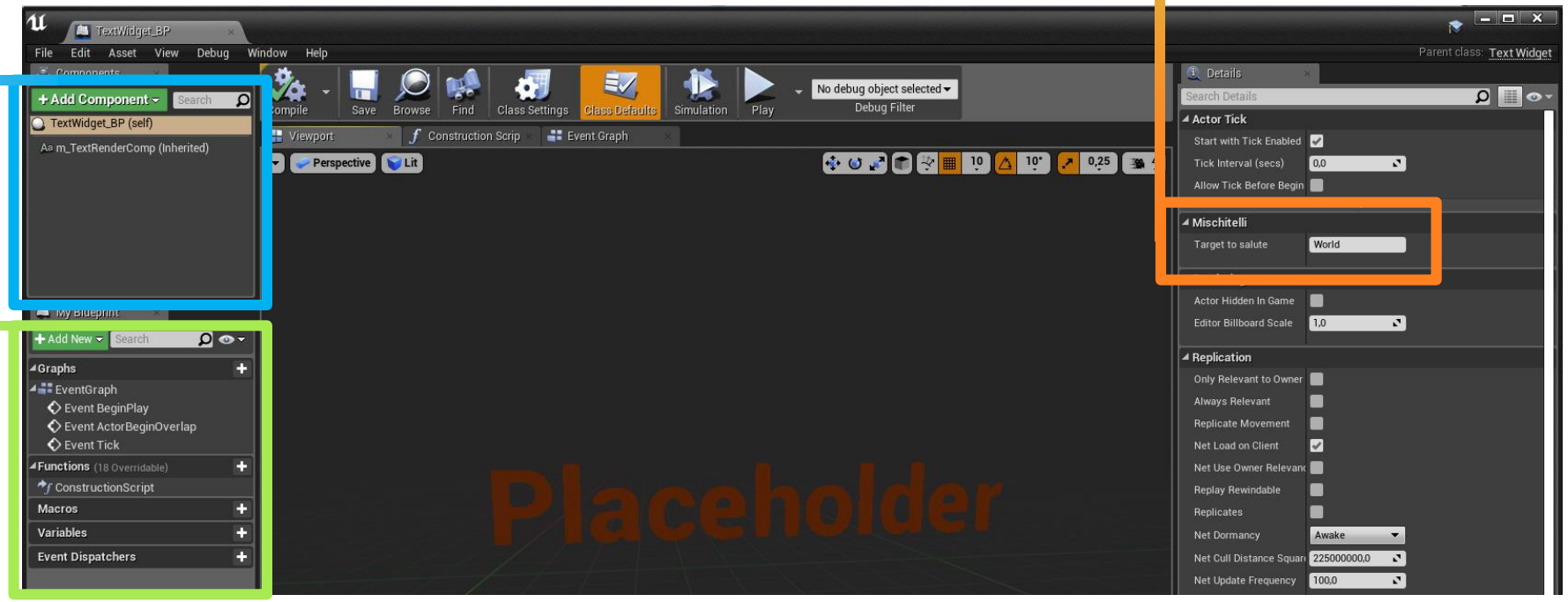


Here's where our `m_Text` property appears...
... and it's editable!

Let's change it to **`Goofy`**!

You can add more components here

Add events, functions, variables to this Actor



Hello, UClass*

```
void AHelloWorldGameModeBase::StartPlay()
{
    // Don't forget to call the Super on UE-declared virtual methods!
    Super::StartPlay();

    auto actor = GetWorld()->SpawnActor<ATextWidget>();
    actor->SetActorLocation
}
```

We're spawning a simple C++ class…

How do we spawn the blueprint associated to this? How does the **`SpawnActor`** method works…?

It's getting the **`StaticClass`** from T

```
/** Templated version of SpawnActor that allows you to specify a class type via the template type */
template< class T >
T* SpawnActor( const FActorSpawnParameters& SpawnParameters = FActorSpawnParameters() )
{
    return CastChecked<T>(SpawnActor(T::StaticClass(), NULL, NULL, SpawnParameters),ECastCheckedType::NullAllowed);
}
```

This really gets complex and involves talking about UE4's **reflection system**… NOPE

It should be enough knowing that Unreal classes are described by this. **`SpawnActor`** needs to know which **`UClass`** to spawn… so either determines it by itself like above, or we pass it to an overload…

Hello, moar blueprints

Back to the **ATextWidget_BP**. It acts like a *specialization* of our C++ class...

- HelloWorldGameModeBase.h \ .cpp

```
UPROPERTY(EditAnywhere, Category = "Mischitelli", meta = (DisplayName="TextWidget Class", AllowPrivateAccess = true))
TSubclassOf<ATextWidget> m_TextWidgetClass;
```

```
if (m_TextWidgetClass != nullptr)
{
    auto actor = GetWorld()->SpawnActor<ATextWidget>(m_TextWidgetClass);
    actor->SetActorLocationAndRotation({ 200,0,50 }, FQuat(0, 0, 1, 0));
}
else {
    GLog->Log(ELogVerbosity::Error, TEXT("TextWidgetClass is null! Can't spawn the actor..."));
}
```

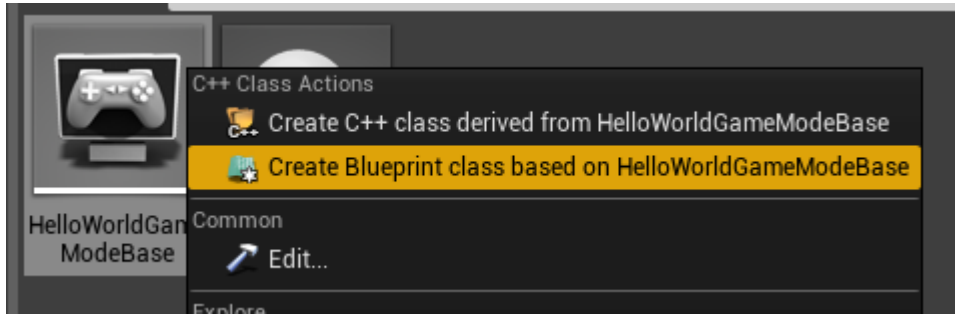
Create a new **UPROPERTY** in our **HelloWorldGameModeBase**

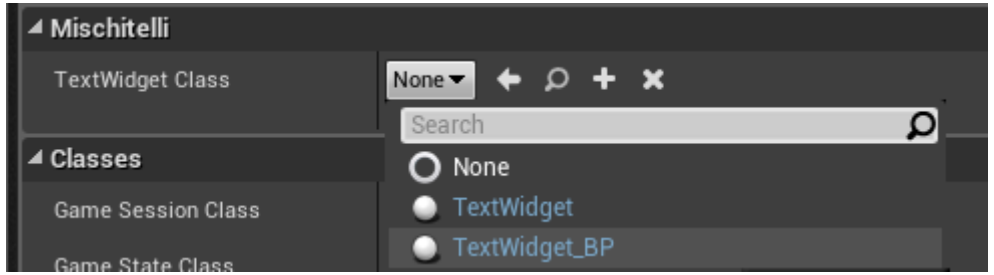
Modify the spawn method adding the newly created property

Create a BP based on **HelloWorldGameModeBase**

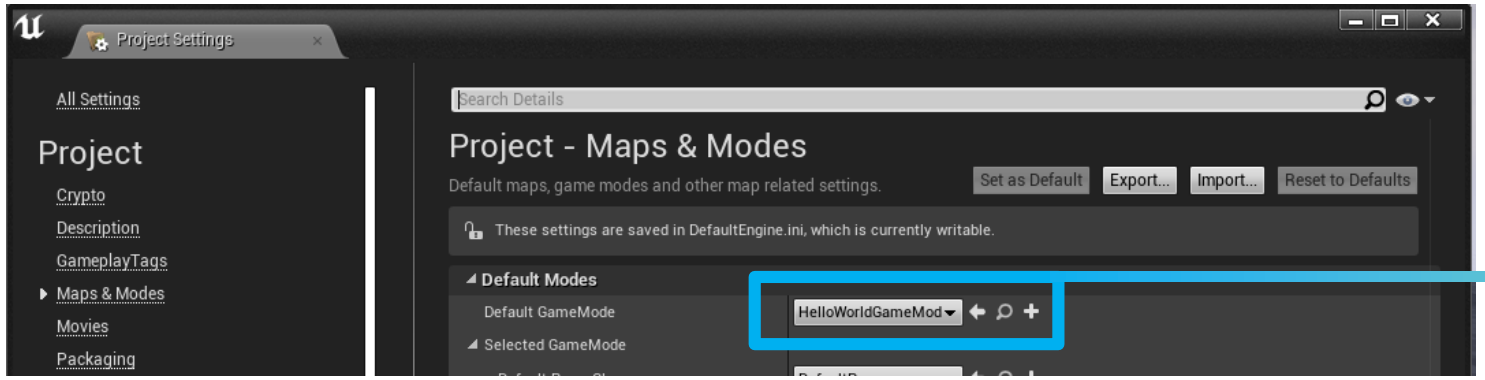
We can finally specify which class to use to spawn **ATextWidget**

- HelloWorldGameModeBase_BP



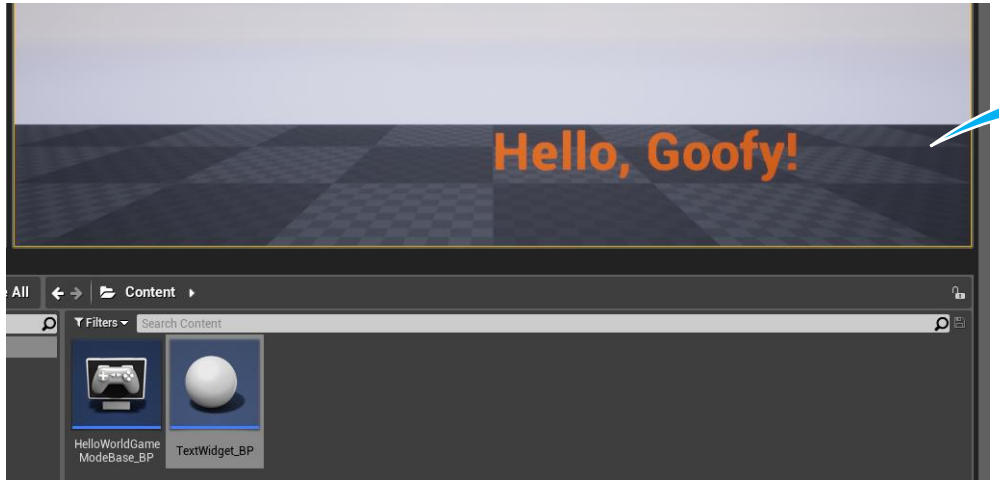


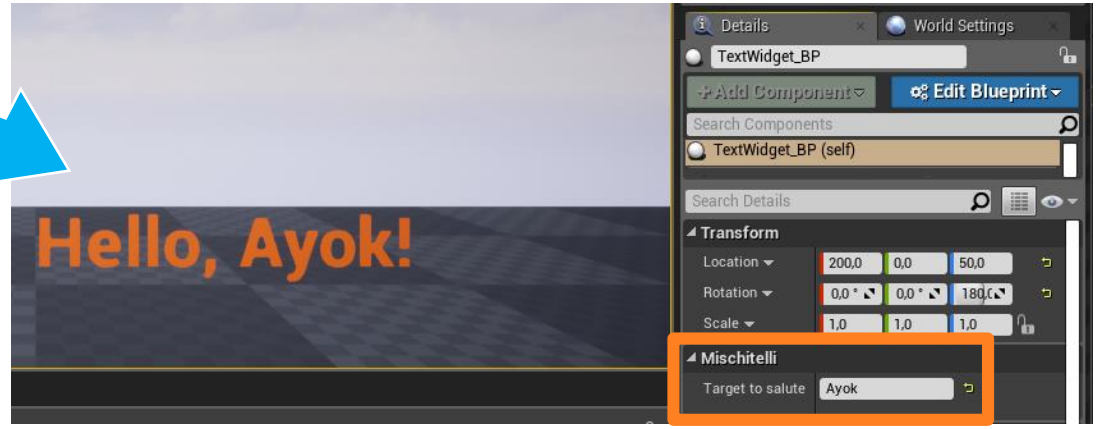
Hello, Goofy!



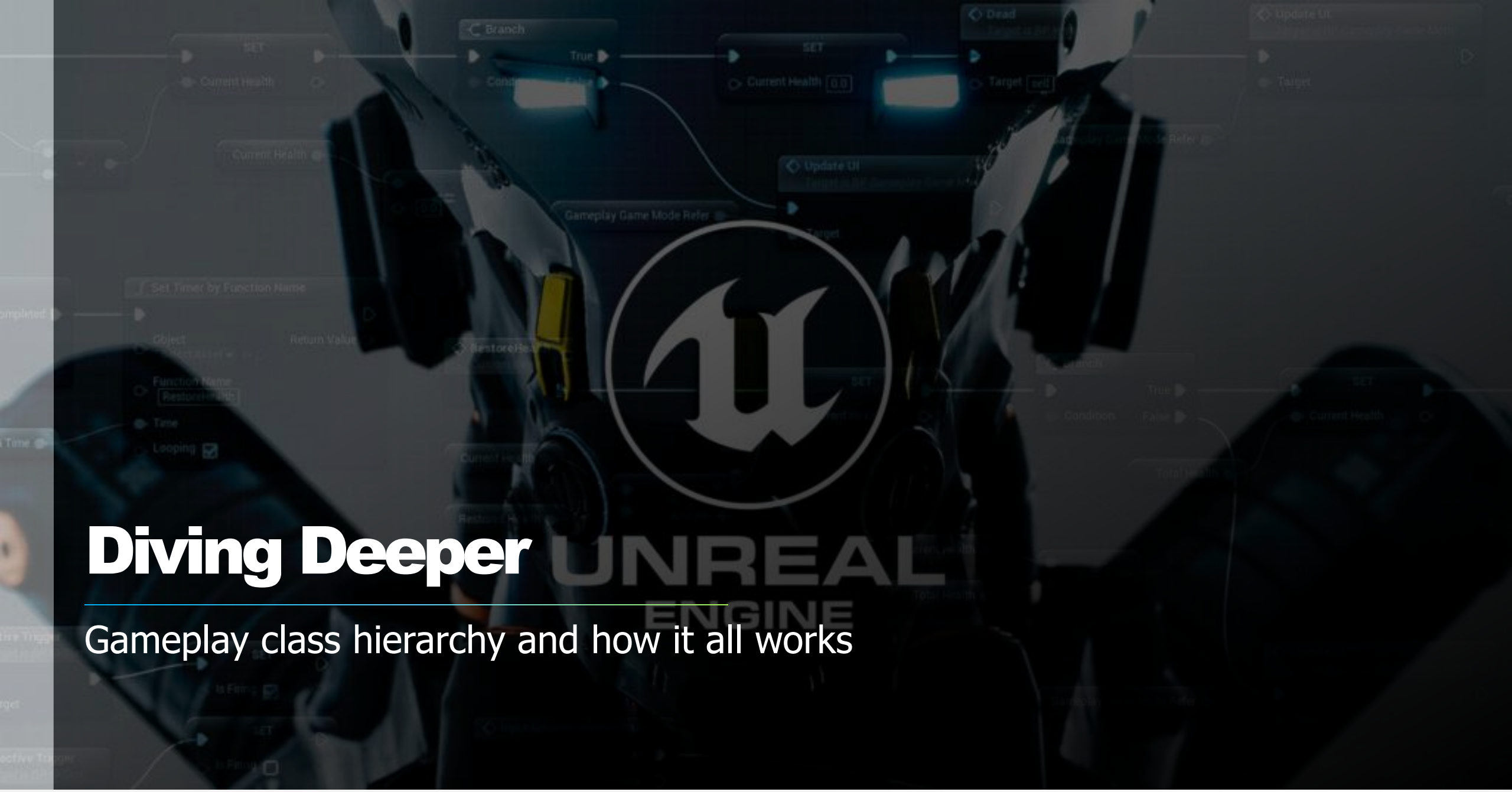
Update project settings with the new **`HelloWorldGameModeBase_BP`**

Again, we need to tell UE4 which flavour of this class we'd like to use. In this case it's different because it's a special case…





We can even modify the string without recompiling thanks to the **`PostEditChangeProperty`** we overrode previously



Diving Deeper

Gameplay class hierarchy and how it all works

Gameplay Classes

Unreal Objects: `UObject`

- Reflection of properties and methods
- Serialization of properties
- Garbage collection
- Networking support for properties and methods

Actors: `AActor`

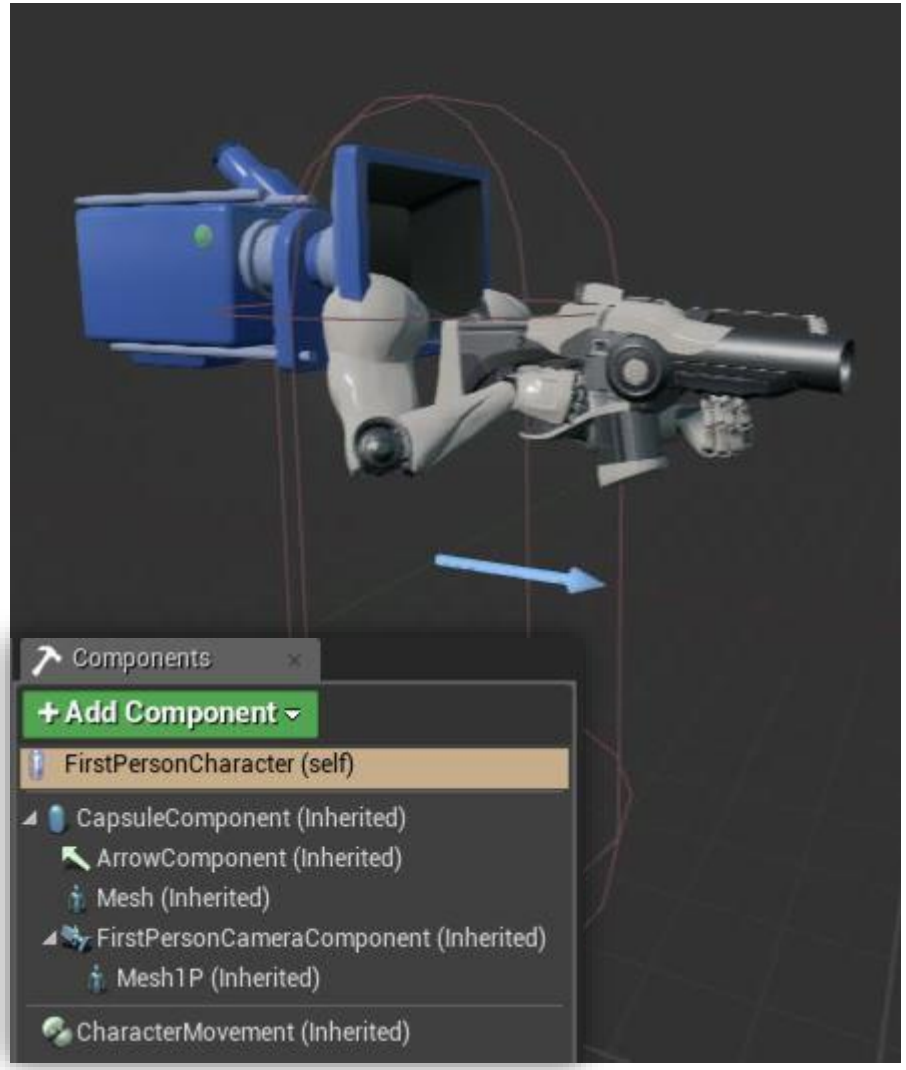
- Inherits from **`UObject`**, core to gameplay experience
- Objects that can be *placed*
- Composed of **`UActorComponent`**s
- Network replication

Components: `UActorComponent`

- Define their own behaviour
- Functionality that is shared across actors
- Actors are given high-level goals → components perform tasks that support those

Structs: `UStruct`

- No need to inherit from a particular class
- Just mark it with **`USTRUCT()`**
- Not Garbage Collected
- PODs + reflection + networking + blueprint



Unreal Reflection System intro

UCLASS

Tells UE4 to generate reflection data for a class.

Blueprintable → can be extended by a BP

UPROPERTY

Allows replication, BP interaction, serialization, GC (reference count).

EditAnywhere → editable in property window on archetypes and instances

```
#include "MyObject.generated.h"

UCLASS(Blueprintable)
class UMyObject : public UObject
{
    GENERATED_BODY()

public:
    UMyObject();

    UPROPERTY(BlueprintReadOnly, EditAnywhere)
    float ExampleProperty;

    UFUNCTION(BlueprintCallable)
    void ExampleFunction();
};
```

GENERATED_BODY

This is replaced by hundreds of lines of boilerplate code

UFUNCTION

BP interaction, RPC in networked scenarios

BlueprintCallable → can be called from BP

Memory Management and Garbage Collection

```
UCLASS()
class AMyActor : public AActor
{
    GENERATED_BODY()

public:
    UPROPERTY()
    UNativeType* m_SafeObject;

    UNativeType* m_DoomedObject;

    AMyActor(const FObjectInitializer& ObjectInitializer)
        : Super(ObjectInitializer)
    {
        m_SafeObject = NewObject<UNativeType>();
        m_DoomedObject = NewObject<UNativeType>();
    }
};
```

```
if (actor->m_SafeObject != nullptr)
{
    // Use SafeObject
}
```

```
class FMyNormalClass : public FGCObject
{
public:
    UObject* SafeObject;

    FMyNormalClass(UObject* Object)
        : SafeObject(Object)
    { }

    void AddReferencedObjects(FReferenceCollector& Collector) override {
        Collector.AddReferencedObject(SafeObject);
    }
};
```

Root set → list of objects that the GC will not garbage collect

Objects are not GC/ed as long as there is a path of reference from an object in the root set to the object in question

If no such path exists, objects are said to be unreachable and will be GC/ed the next time the GC runs

What counts as reference? Pointers stored in **`UPROPERTY`**

Actors are automatically part of the root set and have to be manually destroyed: **`actor->Destroy()`**

After calling **`Destroy()`**, actors are marked as **Pending Kill** and will be actually removed from memory during the next GC clean-up

When **`UObject`** are GC/ed **`UPROPERTY`** are set to **`nullptr`**

It is possible to manage **`UObjects`** inside non-**`UObjects`** by inheriting from **`FGCObject`**

Numeric types and strings

Signed/Unsigned integers

- `int8 / uint8`
- `int16 / uint16`
- `int32 / uint32`
- `int64 / uint64`

Floating point

- `float`
- `double`

```
TNumericLimits<T>::Min()
TNumericLimits<T>::Max()
TNumericLimits<T>::Lowest() //on fp –Max()
```

`FString`

- Mutable string (like std::string)
- `FString str = TEXT("Hello, world!");`

`FText`

- Like above, but for localized text
- `FText txt = NSLOCTEXT("ns", "key", "default");`

`FName`

- Commonly recurring string, stored as identifier to save memory. Also faster during comparisons
- `nameA.Index == nameB.Index`

`TCHAR` – *do not confuse with TChar<T>, FChar…*

- Used to store chars independent of the character set used
- UE4 strings use TCHAR arrays (wchar_t / char)
- Raw data can be accessed using the dereference operator

Containers

`TArray<V, Allocator>`

- Much like **`std::vector`** with more functionality
- Elements are GC/ed if **`TArray`** is marked as **`UPROPERTY`**
- Custom allocator (**`FHeapAllocator`**)

`TArrayView<V>`

- Templated, fixed-sized view of another array
- Stores internally a pointer to the array's first element, as well as the array's size
- Abstraction that tells the developer you're not supposed to add/remove elements to the array
- Original array can still be altered through Algo::Sort, Reverse

`TSet<V, KeyFuncs, Allocator>`

- Addition, removal, finding are O(1)
- Uses a sparse array for elements
- Links elements into a hash through the use of buckets
- **`KeyFuncs`** specify how elements are compared and searched

`TMap<K, V, Allocator, KeyFuncs>`

- Implemented using **`TSet`** with custom **`KeyFuncs`**
- Much like **`std::map`**
- Key-value pairs: **`TPair<K, V>`**
- Any type for key as long as it has a **`GetTypeHash`**
- Custom allocator (**`TSetAllocator`**) that includes:
  - Sparse array allocator: **`TArray`** (elems) + **`TBitArray`** (allocated)
  - Hash allocator (**`FHeapAllocator`**)
  - How many hash buckets the map should use
- **`TMultiMap`**: supports storing multiple identical keys

Michele Mischitelli

Thank you

linkedin.com/in/michelemischitelli
twitter.com/michelemischit1
michelemischitelli@outlook.com
mmischitelli.github.io