Unreal Engine C++

1. Table Of Content

2. Development Environment

2.1. Platform SDK

Unreal Engine 5.6 (see release note)

Windows

- Visual Studio 2022 v17.14 or newer
- Windows SDK 10.0.22621.0 or newer
- .NET 8.0

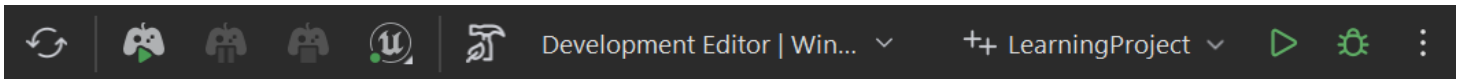
2.2. Visual Studio Setup

Visual Studio Setup

Visual Studio Unreal Extension

2.3. Rider setup

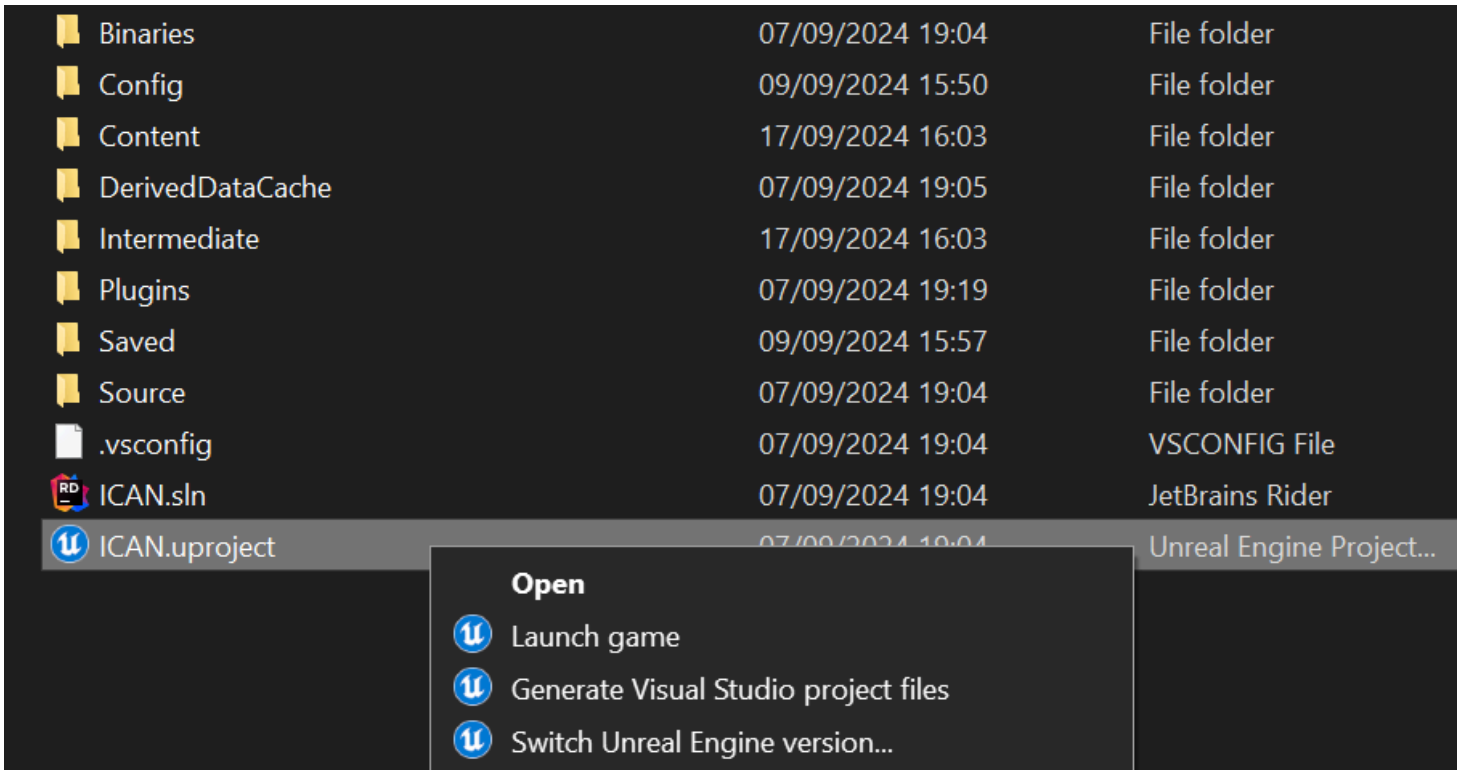
Upon opening the solution with Rider you will be prompt to install RiderLink plugin in the game or directly in the engine.



In rider you have access to all compilation and build feature directly on the top right bar. This includes :

- Generate project file
- Start/Pause/Stop Play In Editor Mode
- Rider Link installation settings
- Build and Rebuild (trigger live coding if unreal editor is open)
- Solution configuration, target and platform
- Project configuration
- Run/Debug/Stop

2.4. Project Folder



Unreal projects contains source files and generated files. Only the source files need to be added to source control. Generated files can be deleted and regenerated at any time.

Source files :

- Config
- Content
- Source
- .uproject

Generated files :

- Binaries
- DerivedDataCache
- Intermediate
- Saved

Visual Studio solution is generated and stored in Intermediates files. Therefore you have to regenerate the solution each time you delete Intermediate folder.

2.5. Compiling

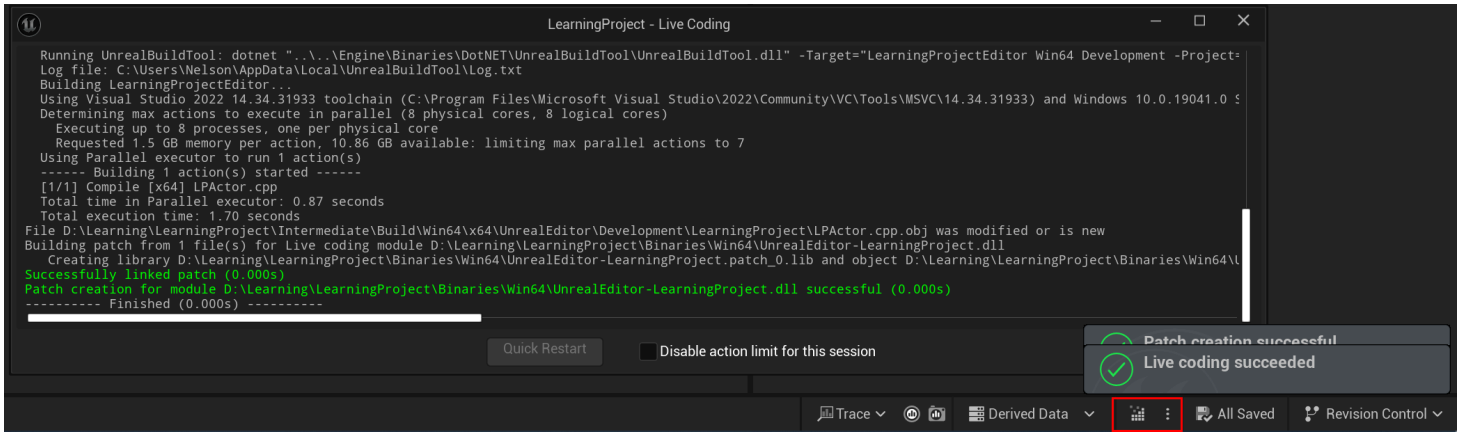
Compiling Game Project

Unreal Engine projects are compiled using UnrealBuildTool(UBT). UBT behaviour is defined by `*.build.cs` and `*.target.cs` files.

2.6. Live Coding

Live Coding Documentation

Live Coding allow to rebuild C++ code and patch associated binaries while the engine is running. The primary use of this feature is to compile C++ modification without closing the editor.



3. Modules

Module Overview

Module Documentation

Gameplay Module Documentation

Modules are the basic structural building block of Unreal Engine's software architecture. They are independent units of code and usually compiled as one compilation unit. A user can think of each module as a library.

Each module

- Should define a clear public interface and keep private logic internal.
- Can depend upon other modules from the engine, game or plugins.
- Can include third-party libraries.
- Can be restricted modules to specific build types (runtime, editor only, etc).
- Can be white/black listed to restrict the platforms they are available for.

Module Directory usually look like this :

- {ModuleName}
  - Private
    - {ModuleName}Module.cpp
    - All private headers
    - All cpp files
  - Public
    - All public headers
  - {ModuleName}.Build.cs

The build.cs file define module dependencies and makes it visible for the Unreal build system. In order to use some module functionalities you need to add the module as a dependency of your module. Usually this is done by listing the wanted module in `PrivateDependencyModuleNames` or `PublicDependencyModuleNames` in the `Build.cs` file of your module.

The Module.cpp file must use IMPLEMENT_MODULE macro to expose your module to the rest of the C++ project.

To control how and when your module loads, add configuration information for your module in your .uproject or .uplugin file. This includes the name, type, compatible platforms, and loading phase of the module.

Adding a module requires to re-run Project Files Generation.

4. Convention

Unreal Engine Coding Standard

4.1. Code Documentation

Unreal Formatting

Java Doc

```
/**
 * Check whether any component of this Actor is overlapping any component of another Actor.
 * @param Other The other Actor to test against
 * @return Whether any component of this Actor is overlapping any component of another Actor.
 */
UFUNCTION(BlueprintCallable, Category="Collision", meta=(UnsafeDuringActorConstruction="true"))
ENGINE_API bool IsOverlappingActor(const AActor* Other) const;
```

4.2. Naming Convention

Unreal Engine uses Upper Camel Case as naming convention.

Unreal Engine also uses prefixes on classes to specify some types of objects :

- **U** : UObject derived classes
- **A** : AActor derived classes
- **F** : Structure and non-uobject classes
- **E** : Enum
- **G** : Global
- **T** : Template
- **I** : Interface

In most scenario projects define their own prefix to avoid collision with engine classes. For example MyProject could use MP as a prefix and can be use to define project classes like AMPCharacter or UMPGameMode.

5. Strings

Strings Documentation

5.1. TCHAR type and TEXT Macro

Unreal Engine uses Unicode as Character Encoding by default. As Unicode is handle differently on each platform, Unreal Engine provide the **TCHAR** type and **TEXT** macro to to ensure text encoding works properly on all platforms.

5.2. FName

FNames are a lightweight type used for efficient string handling. They are case-insensitive, immutable, and cannot be manipulated.

5.2.1. FName Creation and Conversion

```
FName ExampleName = FName(TEXT("ExampleName"));                    // New Name
FName ExampleName = FName(*ExampleString);                                  // Convert String to
FName ExampleName = FName(*(ExampleText.ToString()));    // Convert Text to Name
```

5.2.2. FName Manipulation

```
ExampleName == OtherName; // Slow Comparison
int32 FName::Compare( const FName& Other ) const; // Fast Comparison, both are equals if return
bool IsEqual(const FName& Other, const ENameCase CompareMethod = ENameCase::IgnoreCase, const bo
```

5.3. FText

FText are immutable and cannot be manipulated. They are used for displaying text to the user. They support text localization by providing the following features:

- Creating localized text literals.
- Formatting Text (to generate text from a placeholder pattern).
- Generating text from numbers.
- Generating text from dates and times.
- Generating derived text, such as making text upper or lower case.

5.3.1. FText Creation and Conversion

```
FText();                              // Create empty text
FText::GetEmpty();          // Create empty text

FText::Format(LOCTEXT("ExampleFText", "You currently have {0} health left."), CurrentHealth); //

FText ExampleText = FText::FromString(ExampleString);    // Convert String to Text
FText ExampleText = FText::FromName(ExampleName);                // Convert Name to Text
```

5.3.2. FText Manipulation

```
bool FText::EqualTo(const FText& Other, const ETextComparisonLevel::Type ComparisonLevel) const
int32 FText::CompareTo(const FText& Other, const ETextComparisonLevel::Type ComparisonLevel) co
```

5.4. FString

Unlike FName and FText, FString can be searched, modified, and compared against other strings. However, these manipulations can make FStrings more expensive than the immutable string classes. This is because FString objects store their own character arrays.

5.4.1. FString Creation and Conversion

```
FString ExampleString = FString(TEXT("This is an example FString.")); // Create New FString

FString ExampleString = ExampleName.ToString();          // Convert Name to String
FString ExampleString = ExampleText.ToString();          // Convert Text to String

FString ExampleString = FString::SanitizeFloat(FloatVariable);             // Convert Float to
FString ExampleString = FString::FromInt(IntVariable);                                 // Convert
FString ExampleString = BoolVariable ? TEXT("true") : TEXT("false");// Convert Bool to String
FString ExampleString = VectorVariable.ToString();                                                 /
FString ExampleString = ObjectPtr->GetName();                                                      /

bool BoolVariable =          ExampleString.ToBool();      // Convert String to Bool
int IntVariable = FCString::Atoi(*ExampleString); // Convert String to Int
float FloatVariable = FCString::Atof(*ExampleString); // Convert String to Float

TCHAR* ExampleBuffer1 = *ExampleString1; // Dereference FString to get underlying TCHAR* buffer

FString ExampleString3 = FString::Printf(TEXT("%s : %02d"), *ExampleString1, ExampleInt); // Bu
```

5.4.2. FString Manipulation

```
ExampleString1 == ExampleString2; // Simple FString comparison
ExampleString1.Equals(ExampleString2, ESearchCase::CaseSensitive); // Advanced FString comparis

bool Contains = ExampleString1.Contains(ExampleString2, ESearchCase::CaseSensitive, ESearchDir:

int32 FirstFoundIndex = ExampleString1.Find(ExampleString2, ESearchCase::CaseSensitive, ESearchD

ExampleString1 = ExampleString1 + ExampleString2; // Concatenate
ExampleString1 += ExampleString2; // Concatenate
```

6. Containers

Containers Documentation

6.1. C++ Native Container

Unreal Engine provide types for most used containers but you can still create arrays using the standard C++ syntax.

```
int Arr[3];                                    // uninitialized array
int Arr[3] = { };            // [0, 0, 0]
int Arr[] = {1, 2, 3};       // [1, 2, 3]
```

6.2. TArray

TArray is responsible for the ownership and organization of a sequence of elements of the same type.

TArray Specification :

- Fast
- Memory Efficient
- Safe
- Homogenous (all element in the array are the same type)
- Own elements
- No shared state (creating an array from another one makes a copy)
- Resizable

6.2.1. Construct, Add and Remove

```
TArray<int32> IntArray; // []
IntArray.Init(10, 5);   // [10, 10 ,10 ,10 ,10]

TArray<float> IntArrayCpy = TArray<float>(IntArray); // Copy constructor

TArray<FString> StrArray; // []

StrArray.Add(TEXT("Hello"));      // ["Hello"], use Copy/Move operation (temp variable)
StrArray.Emplace(TEXT("World"));  // ["Hello","World"], use constructor operation (no temp)

StrArray.AddUnique(TEXT("!"));    // ["Hello", "World", "!"]
StrArray.AddUnique(TEXT("!"));    // Fail;

StrArr.Insert(TEXT("UE"), 1);     // ["Hello", "UE", "World", "!"]

StrArr[3] = TEXT("-");      // ["Hello", "UE", "World", "-"]

FString NativeStrArray[] = {TEXT("from"), TEXT("C++")};
StrArray.Append(StrArray, ARRAY_COUNT(NativeStrArray));
// ["Hello", "UE", "World", "-", "from", "C++"]

StrArr.Remove(TEXT("-")); // ["Hello", "UE", "World", "from", "C++"]

StrArr.RemoveAt(1); // ["Hello", "World", "from", "C++"]

StrArr.SetNum(6); // ["Hello", "World", "from", "C++", "", ""]
StrArr.SetNum(2); // ["Hello", "World"]

StrArr.Empty(); // []
```

6.2.2. Iteration

```
TArray<FString> Words = TArray<FString>({TEXT("Hello"), TEXT("World")});
FString Sentence;

for (auto& Word : Words)
{
  Sentence.Add(Word);
  Sentence += TEXT(" ");
}
```

```
for (int32 Index = 0; Index < Words.Num(); Index++)
{
  Sentence += Words[Index] + TEXT(" ");
}
```

6.2.3. Queries and Sorting

```
TArray<int32> IntArray = TArray<int32>({2, 1, 3, 1});

IntArray.Sort(); // [1, 1, 2, 3]

int32 Last = IntArray.Last(); // 3
int32 Last = IntArray.Last(1); // 2

bool bHas2 = IntArray.Contains(2); // true
bool bHas4 = IntArray.Contains(4); // false

int32 FindIndex;
bool bFound = IntArray.Find(1, FindIndex); // true, FindIndex = 0
bool bFound = IntArray.FindLast(1, FindIndex); // true, FindIndex = 1
bool bFound = IntArray.Find(4, FindIndex); // false, FindIndex = INDEX_NONE = -1
```

6.3. TMap

TMap is a container based on hashing keys. It stores key-value pairs TPair<KeyType, ValueType>. TMap store a given ValueType Element for each unique key. Maps can be constructed using `TMap<KeyType, ValueType> Map;` template.

6.3.1. Construct, Add and Remove

```
TMap<FString, int32> Inventory;

Inventory.Add(TEXT("Weapon"), 3); // {"Weapon": 3}
Inventory.Add(TEXT("Shield"), 1); // {"Weapon": 3, "Shield": 1}

Inventory.Add(TEXT("Weapon"), 2); // {"Weapon": 2, "Shield": 1}

Inventory.Emplace(TEXT("Armor"), 6); // {"Weapon": 2, "Shield": 1, "Armor": 6}

TMap<FString, int32> OtherInventory;
OtherInventory.Add(TEXT("Armor"), 1); // {"Armor": 3}
OtherInventory.Add(TEXT("Ring"), 1); // {"Armor": 3, "Ring": 1}

Inventory.Append(OtherInventory); // {"Weapon": 2, "Shield": 1, "Armor": 3, "Ring": 1}

Inventory[TEXT("Armor")] = 4; // {"Weapon": 2, "Shield": 1, "Armor": 4, "Ring": 1}

Inventory.Remove(TEXT("Weapon")); // {"Shield": 1, "Armor": 4, "Ring": 1}

Inventory.FindAndRemoveChecked(TEXT("Shield")); // {"Armor": 4, "Ring": 1}
Inventory.FindAndRemoveChecked(TEXT("Shield")); // assert

int32 ArmorCount;
bool bArmorFound = Inventory.RemoveAndCopyValue(TEXT("Armor")); // {"Ring": 1}
// bArmorFound == true, ArmorCount = 4

Inventory.Reset(); // {}, keep allocated memory
Inventory.Empty(); // {}, free allocated memory
```

6.3.2. Query

```
Inventory = TMap<FString, int32>({"Weapon": 2, "Shield": 1, "Armor": 4, "Ring": 1});

int32 InventorySize = Inventory.Num(); // 4

int32 WeaponCount = Inventory[TEXT("Weapon")]; // 2

bool bHasWeapon = Inventory.Contains(TEXT("Weapon")); // true

int32* WeaponValuePtr = Inventory.Find(TEXT("Weapon")); // *ValuePtr == 2
int32* NecklaceValuePtr = Inventory.Find(TEXT("Necklace")); // ValuePtr == nullptr

int32* PantValuePtr = Inventory.FindOrAdd(TEXT("Pants")); // *ValuePtr == 0
// {"Weapon": 2, "Shield": 1, "Armor": 4, "Ring": 1, "Pants": 0}

TArray<FString> InventoryKeys;
Inventory.GenerateKeyArray(InventoryKeys);
// ["Weapon", "Shield", "Armor", "Ring", "Pants"]

TArray<int32> InventoryValues;
Inventory.GenerateValueArray(InventoryValues);
// [2, 1, 4, 1, 0]
```

`FindOrAdd` invalidate previous pointer on memory allocation.

`GenerateKeyArray` and `GenerateValueArray` empty the array before populating the result. Therefore result array size is always equal to map size.

6.3.3. Iteration

```
FString Log;
for (auto& Elem : Inventory)
{
  Log += Elem.Key + TEXT(":") + FString::FromInt(Elem.Value) + TEXT(" ");
}
// Log = Weapon:2 Shield:1 Armor:1 Ring:1
```

6.4. TSet

TSet is a fast container class to store unique elements in a context where order is irrelevant.

```
TSet<FString> SetA, SetB;
// Add some values to set A and set B. See array methods.

TSet<FString> DifferenceSet = SetA.Difference(SetB); // A not in B
TSet<FString> UnionSet = SetA.Union(SetB); // A or B
TSet<FString> IntersectSet = SetA.Intersect(SetB); // A and B
bool bIsIncluded = SetA.Includes(SetB) // Is B entirely included in A
```

7. Debugging

7.1. Logging

Logging to the console is done through `UE_LOG` macro.
The macro requires a log category and a log verbosity level.

```
UE_LOG(LogCategory, LogVerbosity, TEXT("Hello World !"));
```

Log Category define what part of the code created a log.
Generally speaking, each module declare it's own category.
Category are typically used to filter logs.

It is highly advised to create a log category for your game module.

In the relevant header file (usually a separate file) add :

```
DECLARE_LOG_CATEGORY_EXTERN(LogCategoryName, VerbosityLevel, All);
```

In the relevant cpp file, just after include directives, add :

```
DEFINE_LOG_CATEGORY(LogCategoryName);
```

Log Verbosity define how critical the log it.
On top of this it is possible to filter log verbosity for each category.

Log Verbosity Levels :

- Fatal
- Error
- Warning
- Display
- Log
- Verbose
- VeryVerbose

The `UE_LOG` macro also support formatting.

```
FString TestString("Unreal Engine");
int TestInt = 5;
UE_LOG(LogCategory, LogVerbosity, TEXT("Welcome to %s %d !"), *TestString, TestInt);
```

7.2. Draw Debug

The Draw Debug function allow to draw simple shapes for debugging purposes.

```
#include "DrawDebugHelpers.h"

DrawDebugLine(GetWorld(), Start, End, FColor::Green);
DrawDebug**(GetWorld, Start, End, ...);
```

7.3. On Screen Messages

Instead of logging on the console it is sometimes useful to log on Screen

```
#include "Engine/Engine.h"

FString MyDebugString = FString::Printf(TEXT("MyVelocity(%s)"), *MyVelocity.ToCompactString());
GEngine->AddOnScreenDebugMessage(INDEX_NONE, 0.f, FColor::Yellow, MyDebugString, false, FVector
```

7.4. Trace

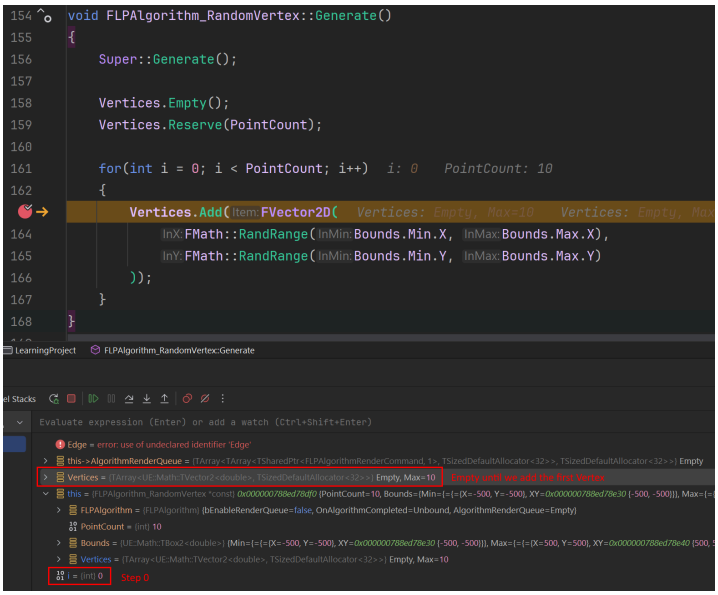
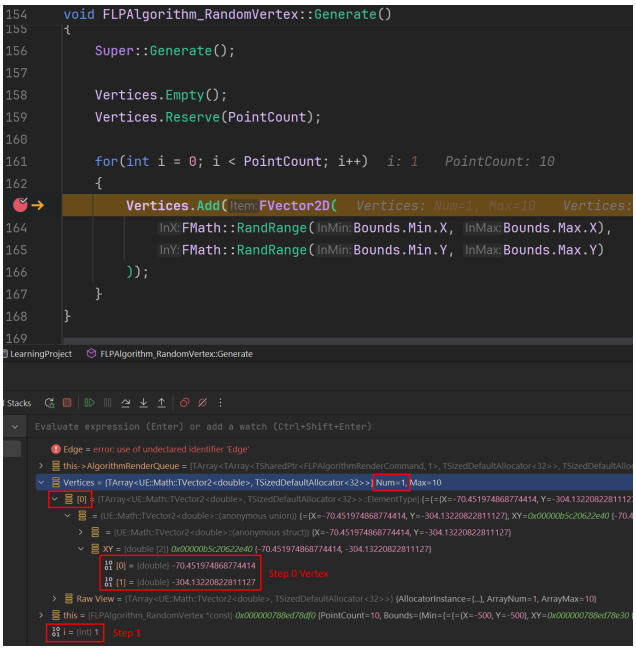
Unreal Engine provide a set of methods to gather physical information along a line segment.
Trace utilities are store in `UKismetSystemLibrary`.

```
UKismetSystemLibrary::LineTraceSingle(...);
```

- Traces can return single or multiple hits.
- They can be configured to match only with specified trace channel or object types.
- It is also possible to trace shapes along the segment when line trace is not enough.

7.5. Breakpoints

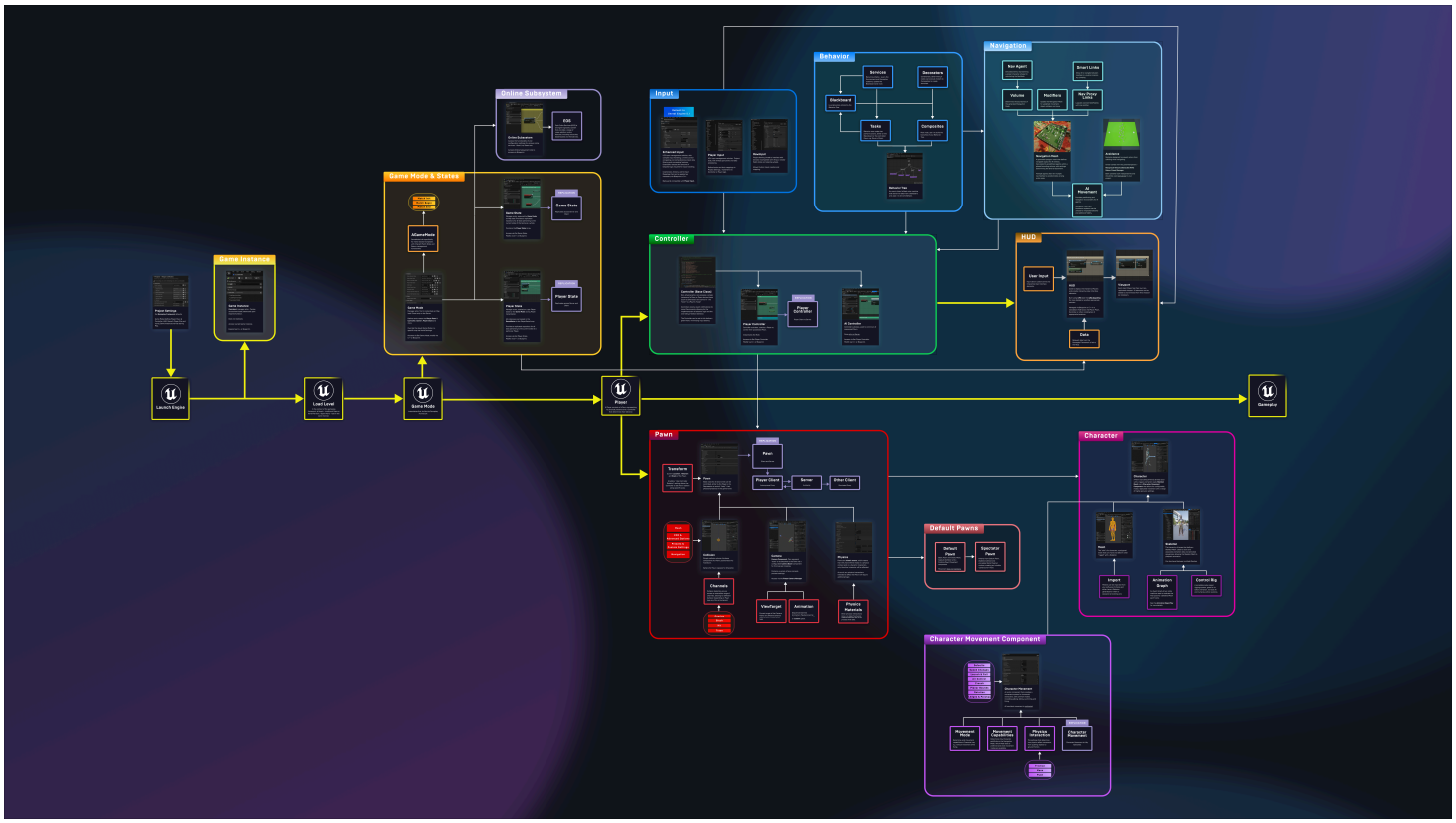
When working with C++ one of the easiest way of debugging is to use Breakpoints.
To do so you just need to start Unreal from your IDE and place a breakpoint.
If you need to keep track of specific symbols remember to add them to the watch list.

Step 0	Step 1
	

When working with C++ it is recommended to always launch Unreal Engine in Debug Mode using your IDE. If anything goes wrong your IDE will pick up the error giving insight on it.

8. Gameplay Framework

Gameplay Framework Documentation



- UObject : Base class for game objects
- AActor : Game object that can be placed into a level
- UActorComponent : Building block of actors
- UBlueprintFunctionLibrary : Expose functions to Blueprints
- APawn : Actor that can be controlled by player or AI
- ACharacter : Extended Pawn for player control
- AController : Actor that can possess and control Pawn actions
- UWorld : Root object representing a map
- UGameMode : Define base classes to use and specify game rules
- UGameInstance : Persistent class throughout the lifetime of the application
- UCameraComponent : Player view point
- UGameplayStatics : Common game functionalities

9. Blueprint Function Libraries

Blueprint Function Libraries Documentation

Blueprint Function Libraries are a collection of static functions that provide utility functionality not tied to a particular gameplay object. These libraries can be grouped into logical function sets or contain utility functions that provide access to many different functional.

Creating a Blueprint Function Library is very similar to exposing functions to Blueprints using the UFUNCTION() macro. Instead of deriving from an Actor or directly from UObject all Blueprint Libraries inherit from UBlueprintFunctionLibrary. They should also contain only static methods.

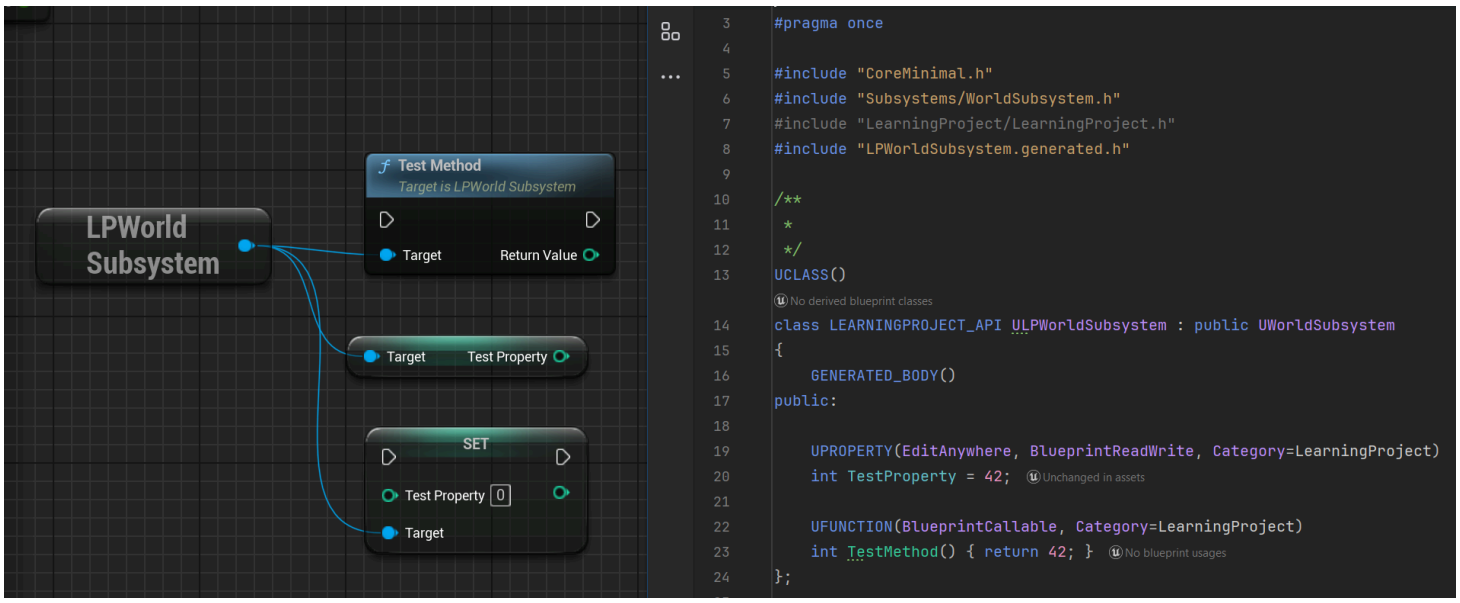
```
UCLASS()
class UMyBlueprintFunctionLibrary : public UBlueprintFunctionLibrary
{
    GENERATED_BODY()

    UFUNCTION(BlueprintCallable, Category="MyCategory")
    static void LogSomething();
}
```

```
void UMyBlueprintFunctionLibrary::LogSomething()
{
    UE_LOG(LogTemp, Log, TEXT("Something"));
}
```

10. Subsystems

Subsystems Documentation



Subsystems are automatically instanced classes with managed lifetimes. These classes provide easy to use extension points, where the programmers can get Blueprint exposure right away while avoiding the complexity of modifying or overriding engine classes.

There are several reasons to use programming subsystems, including the following

- Save programming time.
- Avoid overriding engine classes.
- Avoid adding more API on an already busy class.
- Exposed to blueprint through user friendly typed nodes.
- Provide modularity and consistency in the codebase.

Unreal define multiple type of subsystem with different lifecycle. For example :

- `UEngineSubsystem` : Initialize after engine module `Startup()`. Deinitialize after engine module `Shutdown()`.
- `UEditorSubsystem` : Initialize after editor module `Startup()`. Deinitialize after editor module `Shutdown()`.
- `UGameInstanceSubsystem` : Same as UGameInstance lifecycle. Can be access through UGameInstance directly.
- `ULocalPlayerSubsystem` : Same as ULocalPlayer lifecycle. Can be access through ULocalPlayer directly.

11. UObject and Refection System

UObject Documentation

UObject features :

- Garbage collection
- Reference updating
- Reflection
- Serialization
- Automatic updating of default property changes
- Automatic property initialization
- Automatic editor integration
- Type information available at runtime
- Network replication

11.1. Header

```
#pragma once
#include "ClassName.generated.h"

UCLASS([specifier, specifier, ...], [meta(key=value, key=value, ...)])
class MYPROJECT_API  ClassName : public ParentName
{
    GENERATED_BODY()
public:
    UPROPERTY([specifier, specifier, ...], [meta(key=value, key=value, ...)])
    PropertyType Property;

    UFUNCTION([specifier1=setting1, specifier2, ...], [meta(key1="value1", key2, ...)])
    ReturnType FunctionName([Parameter1, Parameter2, ..., ParameterN1=DefaultValueN1, ParameterN
}
```

`#pragma once` is used to tell the compiler to include this file only once.

`ClassName.generated.h` contains unreal generated code for the objects declared in the header file. This is expected to be the last include so any other include have to be specified above this line.

`MYPROJECT_API` is used to expose the class to other modules.

`GENERATED_BODY` macro is required for all UCLASS and USTRUCT and setup boilerplate code required by the engine.

`UCLASS` macro is used to expose UObject derived class to the Unreal Reflection System.

11.2. Creating Objects

`NewObject` is used to create a new object instance. There are multiple signatures in `UObjectGlobals.h`. Unreal provides default values for all parameters so they are optional.

Common parameters :

- `UObject* Outer` : new object owner
- `FName Name` : new object unique name, see `MakeUniqueObjectName`
- `EObjectFlags Flags` : new object flags, see `EObjectFlags`

```
T* Object = NewObject<T>();
T* Object = NewObject<T>(Outer, Name, Flags);
```

`CreateDefaultSubobject` is used in Objects constructor to instantiate subobjects. It is typically used to instantiate actor's components.

```
T* Subobject = CreateDefaultSubobject<T>(Name);
```

11.3. Updating Objects

UObject does not provide update function by default. In order to have access to a Tick function the UObject also need to inherit from `FTickableGameObject`.

11.4. Deleting Objects

Object destruction is handled automatically by the garbage collection system when an Object is no longer referenced.
When the garbage collector runs, unreferenced Objects found are removed from memory. In addition, the function MarkAsGarbage() can be called directly on an Object to fully delete it on the next garbage collection pass.

11.5. UCLASS

UCLASS Documentation

`UCLASS` macro is used to expose UObject derived class to the Unreal Reflection System. It gives the UObject a reference to a UCLASS that describes its Unreal-based type. Each UCLASS maintains one Object called the Class Default Object(CDO).

The CDO is essentially a default 'template' Object, generated by the class constructor and unmodified thereafter. CDO also hold default variable values. It is created early in Engine initialization therefore the C++ class constructor should not use objects nor contains any game code.

Common UClass specifiers :

- *Category* : Display in editor panels. Nested categories can be specified using `|` operator.
- *Blueprintable* : Allow to create derived blueprint classes
- *BlueprintType* : Can be used as variable type in blueprints

UClass also provides additional class manipulation and checks :

```
UClass* Class; // any uclass is accepted
TSubclassOf<UBaseClass> Class; // only child classes of BaseClass are accepted
Object.GetClass(); // Return object UClass pointer
```

11.6. UPROPERTY

UPROPERTY Documentation

```
UPROPERTY([specifier, specifier, ...], [meta(key=value, key=value, ...)])
PropertyType Property;
```

UPROPERTY is used to expose variables to Unreal Reflection System.

Common UPROPERTY specifiers :

- *Category* : Display in editor panels. Nested categories can be specified using | operator.
- *BlueprintReadOnly* : Property can be read from blueprints
- *BlueprintReadWrite* : Property can be read and write from blueprints
- *EditAnywhere* : Property can be edited from archetype and instance property window
- *EditDefaultsOnly* : Property can only be edited from archetype property window
- *EditInstanceOnly* : Property can only be edited from instance property window
- *VisibleAnywhere* : Property is displayed on archetype and instance property window
- *VisibleDefaultsOnly* : Property is only displayed on archetype property window
- *VisibleInstanceOnly* : Property is only displayed on instance property window
- *AllowPrivateAccess* : Meta specifier used to give blueprint access to a private or protected variable. This is often used on components.

11.7. UFUNCTION

UFUNCTION Documentation

```
UFUNCTION([specifier1=setting1, specifier2, ...], [meta(key1="value1", key2, ...)])
ReturnType FunctionName([Parameter1, Parameter2, ..., ParameterN1=DefaultValueN1, ParameterN2=De
```

Any Function of an UObject or UBlueprintFunctionLibrary can be expose to Unreal Reflection System using UFUNCTION macro.

Common UFUNCTION specifiers :

- *Category* : Display in editor panels. Nested categories can be specified using | operator.
- *BlueprintCallable* : Function can be call from blueprint graphs.***printPure* : Use when the function does not modify owning object. True by default on const function. In blueprint graphs it remove the execution pin.
- *CallInEditor* : Function can be called from object instance detail panel.

11.8. USTRUCT

USTRUCT Documentation

UStruct are really similar to UObject and most of the features provided by UObject are applied to UStruct as well. Main difference is that UStruct are not garbage collected because they are considered value type.

```
USTRUCT([specifier, specifier, ...], [meta(key=value, key=value, ...)])
struct MYPROJECT_API FStructName
{
    GENERATED_BODY()
};
```

USTRUCT is used to expose structures to Unreal Reflection System.

Common USTRUCT specifiers :

- *Category* : Display in editor panels. Nested categories can be specified using | operator.
- *BlueprintType* : Can be used as variable type in blueprints

11.9. UENUM

```
    UENUM([specifier, specifier, ...], [meta(key=value, key=value, ...)])
    enum class EThing : uint8
    {
        Thing1,
        Thing2
    }
```

UENUM is used to expose enums to Unreal Reflection System. In order to be exposed to Blueprints, enumerations have to be based on uint8.

Common UENUM specifiers:

- *Category* : Display in editor panels. Nested categories can be specified using | operator.
- *BlueprintType* : Can be used as variable type in blueprints

```
    // use enum as a property
    UPROPERTY()
    EThing MyProperty;
```

Enumeration syntax has evolved over time. Therefore you might encounter namespace enum and TEnumAsByte<> properties but these syntaxes are obsoletes.

11.9.1. ENUM CLASS FLAGS

```
enum class EFlags
{
    None = 0x00,
    Flag1 = 0x01,
    Flag2 = 0x02,
    Flag3 = 0x04
};

ENUM_CLASS_FLAGS(EFlags)
```

ENUM_CLASS_FLAGS(EnumType) macro automatically define all of the bitwise operators.

12. Components

Components Documentation

Components are a special type of Object that Actors can attach to themselves as sub-objects. Components are useful for sharing and reusing common behaviours.

In unreal, most of the features are ultimately handle by components like rendering mesh, handling collision, setting up camera, etc. Actors can then gather one or multiple components to bring these functionalities in the world.

`UActorComponent` is the base class for all Components.

`USceneComponent` inherit from UActorComponent and define a Transform. Therefore it is the base class for all components that need to have a location in the world.

`UPrimitiveComponent` inherit from USceneComponent and define interfaces required for rendering. Therefore it is the base class for all components that have a visual representation.

12.1. Register

In order for a component to work properly they first need to be registered.
For components created as Actor subobjects this happen automatically but for runtime created components `RegisterComponent` function have to be call manually. They can also be unregister using `UnregisterComponent`.

Component `OnRegister` and `OnUnregister` can also be override if needed.

Actors also provide some method to control its component registration like `RegisterAllComponents`, `UnregisterAllComponents`, `PreRegisterAllComponents`, `PostRegisterAllComponents`.

12.2. Updating

Components does not register for updating by default. Like for actors, component have a `PrimaryComponentTick` structure property that is used to define component ticking behaviour. Once tick enable, the `TickComponent` method can be override to add update logic on the component.

12.3. Hierarchy

`SceneComponent` are attach to each other in order to describe spatial relationship between them. The method `SetupAttachment` is used on unregistered components, for example in an actor constructor. Otherwise use the `AttachToComponent` function.

12.4. Render

Components that creates a render state, like `UPrimitiveComponent`, can be marked as dirty using `MarkRenderStateDirty` method. Dirty components will have their render data updated at the end of the frame.

12.5. Expose to Blueprint

`BlueprintSpawnableComponent` specifier needs to be added into `UCLASS` macro if you need to spawn the component from blueprint.

12.6. Example Header

```
#pragma once

#include "CoreMinimal.h"
#include "Components/SceneComponent.h"
#include "LPComponent.generated.h"

UCLASS(Blueprintable, BlueprintType, meta=(BlueprintSpawnableComponent))
class LEARNINGPROJECT_API ULPComponent : public USceneComponent
{
    GENERATED_BODY()

public:
    ULPComponent()
    {
        PrimaryComponentTick.bCanEverTick = true;
        PrimaryComponentTick.bStartWithTickEnabled = true;
    }

protected:
    virtual void BeginPlay() override;

public:
    virtual void TickComponent(float DeltaTime, ELevelTick TickType, FActorComponentTickFunctio
    virtual void OnRegister() override;
    virtual void OnUnregister() override;
};
```

13. Actors

An Actor is any object that can be placed into a Level, such as a camera, Static Mesh, or player start location. In C++ actors inherit from `AActor` class.

Actors features :

- Support 3D transforms.
- Can be created (spawned) and destroyed through gameplay code.
- Actors also act as a container for components.

Unreal define a lot of premade actors you can use to build game world or gameplay. For example you can use :

- AStaticMeshActor : Add a static mesh to game world.
- ASkeletalMeshActor : Add a skeletal mesh to game world.
- ACameraActor : Define a player view point.

13.1. Create Actor

`SpawnActor` function is used to spawn a new actor. You can check all the templated and not templated signatures in `UWorld` class.

```
AActor* NewActor = World->SpawnActor(AActor::StaticClass(), ...);
AActor* NewActor = World->SpawnActor<AActor>(...);
```

13.2. Declare Actors

`Constructor` sets actor default property values and initialize subobjects. This is part of the process to create the CDO.

`RootComponent` define the top component of the component tree of the actor. It is typically assigned in the constructor.

`PrimaryActorTick` structure describe how the actor is ticking. Is is usually configured in the constructor but can be called anywhere if actor tick behaviour needs to be changed.

13.3. Actor Lifecycle

`BeginPlay` is called on world start and actor spawned.

`Tick` is called every frame and provide the Delta Time between frames.

`BeginDestroy` mark the actor for destroy. The destroy itself is handle by the engine and can be delayed if needed.

`EndPlay` is call when game world stop.

13.4. Example

```
#pragma once

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

UCLASS(Blueprintable, BlueprintType, Category=LearningProject)
class LEARNINGPROJECT_API ALPActor : public AActor
{
    GENERATED_BODY()

public:
    ALPActor()
    {
        SceneComponent1 = CreateDefaultSubobject<USceneComponent>("SceneComponent1");

        RootComponent = SceneComponent1;

        SceneComponent2 = CreateDefaultSubobject<USceneComponent>("SceneComponent2");
        SceneComponent2->SetupAttachment(SceneComponent1)
    }

    virtual void Tick(float DeltaTime) override;

protected:
    virtual void BeginPlay() override
    {
        SceneComponent3 = NewObject<USceneComponent>(this, TEXT("SceneComponent3"));
        SceneComponent3->AttachToComponent(SceneComponent1, FAttachmentTransformRules::KeepRelat
        SceneComponent3->RegisterComponent();
    }

    virtual void EndPlay(const EEndPlayReason::Type EndPlayReason) override;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=LearninProject, meta=(AllowPrivateAcces
    TObjectPtr<USceneComponent> SceneComponent1;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=LearninProject, meta=(AllowPrivateAcces
    TObjectPtr<USceneComponent> SceneComponent2;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=LearninProject, meta=(AllowPrivateAcces
```

```
    TObjectPtr<USceneComponent> SceneComponent3;
};
```

14. Interfaces

```
#pragma once

#include "CoreMinimal.h"
#include "UObject/Interface.h"
#include "LPInterface.generated.h"

UINTERFACE(Blueprintable)
class LEARNINGPROJECT_API ULPInterface : public UInterface
{
    GENERATED_BODY()
};

class LEARNINGPROJECT_API ILPInterface
{
    GENERATED_BODY()

public:
    // Cpp only interface function
    virtual void PureVirtualFunctionCppOnly() const = 0;

    // Can be implemented in blueprints
    UFUNCTION(BlueprintImplementableEvent, BlueprintCallable, Category=LearningProject)
    void BlueprintImplementableEventFunction();

    // Can be implemented in blueprints and have a default cpp implementation
    UFUNCTION(BlueprintNativeEvent, BlueprintCallable, Category=LearningProject)
    void BlueprintNativeEventFunction();
    virtual void BlueprintFunction_Implementation()
    {
        // Default cpp implementation
    }
};
```

15. References

15.1. Unreal Engine Root Documentations

Unreal Engine Programming Documentation

Unreal C++ API

Unreal Tutorials