Programming Subsystems

An overview of programming subsystems in Unreal Engine.

Subsystems in **Unreal Engine (UE)** are automatically instanced classes with managed lifetimes. These classes provide easy to use extension points, where the programmers can get Blueprint and Python exposure right away while avoiding the complexity of modifying or overriding engine classes.

Currently supported subsystems lifetimes include:

Subsystem	Inherit From
Engine	`UEngineSubsystem` class
Editor	`UEditorSubsystem` class
GameInstance	`UGameInstanceSubsystem` class
LocalPlayer	`ULocalPlayerSubsystem` class

For example, if you create a class that derives from this base class:

```
1 class UMyGamesSubsystem : public UGameInstanceSubsystem
2
```

Copy full snippet

This results in the following:

1. After `UGameInstance` is created, an instance called `UMyGamesSubsystem` is also created.
2. When `UGameInstance` initializes, `Initialize()` will be called on the subsystem.
3. When `UGameInstance` is shut down, `Deinitialize()` will be called on the subsystem.
4. At this point, the reference to the subsystem is dropped, and the subsystem is garbage-collected if there are no more references to it.

Reasons to Use Subsystems

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

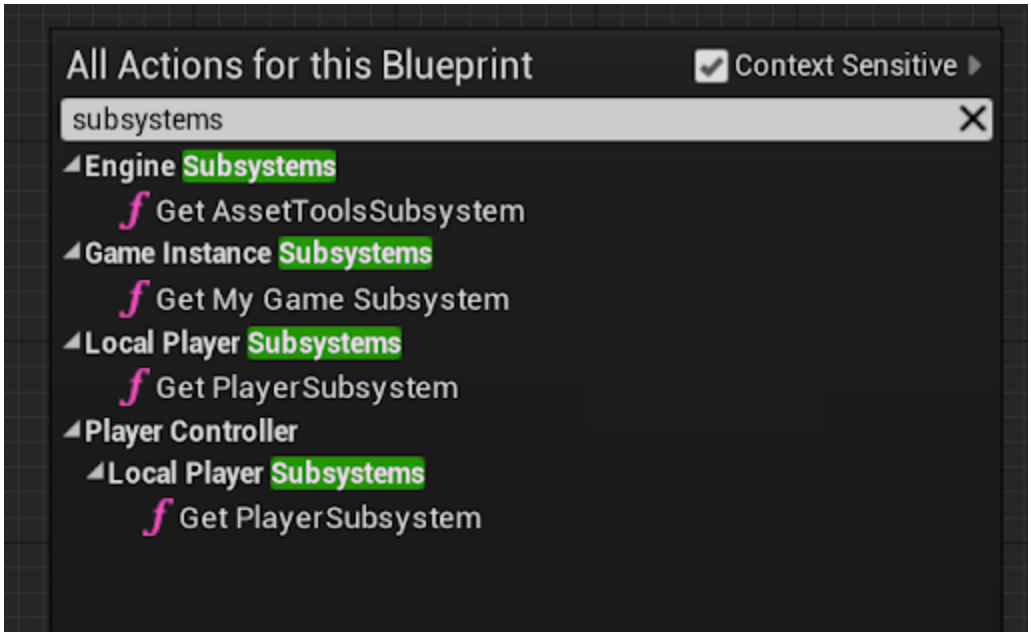
- Subsystems save programming time.
- Subsystems help you avoid overriding engine classes.
- Subsystems help you avoid adding more API on an already busy class.
- Subsystems enable access to Blueprints through user friendly typed nodes.
- Subsystems enable access to Python scripts for editor scripting, or for writing test code.
- Subsystems provide modularity and consistency in the codebase.

Subsystems are particularly useful when creating plugins. You do not need to have instructions about the code needed to make the plugin work. The user can just add the plugin to the game, and you know exactly when the plugin will be instanced and initialized. As a result, you can focus on how to use the API and the functionality provided in UE4.

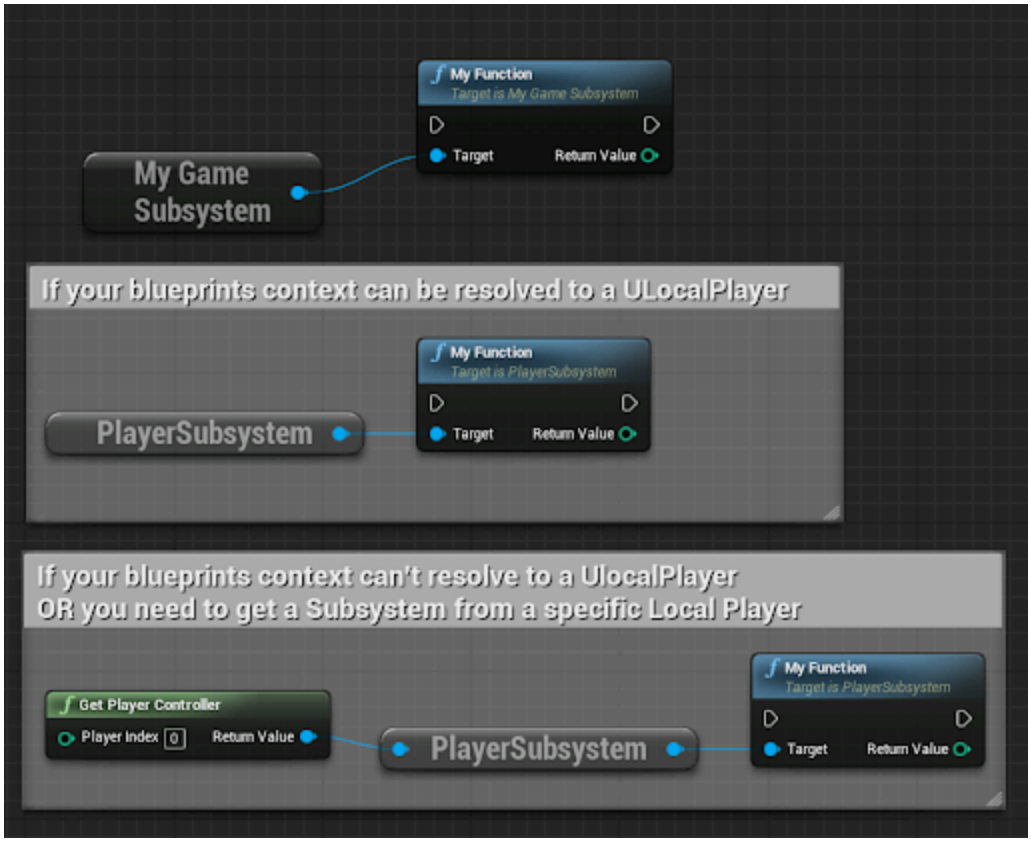
Accessing Subsystems with Blueprints

Subsystems are automatically exposed to Blueprints, with smart nodes that understand context, and that do not require casting. You are in control of what API is available to Blueprints with the standard `UFUNCTION()` markup and rules.

If you right-click in a Blueprint graph to display the context menu and search for "subsystems," you see something similar to the image below. There are categories for each major type and individual entries for each specific subsystem.



If you add the nodes from above, you get results like the following.



Accessing Subsystems with Python

If you are using Python, you can use built-in accessors to access subsystems, as shown in the example below.

```
my_engine_subsystem = unreal.get_engine_subsystem(unreal.MyEngineSubsystem)
my_editor_subsystem = unreal.get_editor_subsystem(unreal.MyEditorSubsystem)

```

Copy full snippet

> Python is currently an experimental feature.

Subsystem Lifetimes in Detail

Engine Subsystems

```
class UMyEngineSubsystem : public UEngineSubsystem { ... };

```

Copy full snippet

When the Engine Subsystem's module loads, the subsystem will `Initialize()` after the module's `Startup()` function has returned, and the subsystem will `Deinitialize()` after the module's `Shutdown()` function has returned.

These subsystems are accessed through GEngine as shown below.

```
UMyEngineSubsystem* MySubsystem = GEngine->GetEngineSubsystem<UMyEngineSubsystem>();

```

Copy full snippet

Editor Subsystems

```
class UMyEditorSubsystem : public UEditorSubsystem { ... };

```

Copy full snippet

When the Editor Subsystem's module loads, the subsystem will `Initialize()` after the module's `Startup()` function has returned, and the subsystem will `Deinitialize()` after the module's `Shutdown()` function has returned.

These subsystems are accessed through GEditor as shown below.

```
UMyEditorSubsystem* MySubsystem = GEditor->GetEditorSubsystem<UMyEditorSubsystem>();

```

Copy full snippet

GameInstance Subsystems

```
class UMyGameSubsystem : public UGameInstanceSubsystem { ... };

```

Copy full snippet

These subsystems can be accessed through UGameInstance as shown below.

```
UGameInstance* GameInstance = ...;
UMyGameSubsystem* MySubsystem = GameInstance->GetSubsystem<UMyGameSubsystem>();

```

Copy full snippet

LocalPlayer Subsystems

```
class UMyPlayerSubsystem : public ULocalPlayerSubsystem { ... };
```

```
2
```

Copy full snippet

These subsystems can be accessed through ULocalPlayer as shown below.

```
1 ULocalPlayer* LocalPlayer = ...;
2 UMyPlayerSubsystem * MySubsystem = LocalPlayer-
  >GetSubsystem<UMyPlayerSubsystem>();
3
```

Copy full snippet

Subsystems Example

In the following example, we want to add a stats system to the game to track the number of gathered resources.

We could derive from `UGameInstance`, and make `UMyGamesGameInstance`, then add the `IncrementResourceStat()` function to it. However, we know that eventually, the team will want to add other stats as well as stat aggregators and saving/loading of stats, and so on. Therefore, you decide to put all of that in a class, such as `UMyGamesStatsSubsystem`.

Again, we could make `UMyGamesGameInstance` and add a member of our `UMyGamesStatsSubsystem` type. Then we can add an accessor to it, and hook up the Initialize and Deinitialize functions. However, there are a couple of problems with this.

- There is not a game-specific derivative of `UGameInstance`.
- `UMyGamesGameInstance` exists, but it already has a large number of functions, and it is less optimal to add more to it.

There are plenty of good reasons to derive from `UGameInstance` in a sufficiently complicated game. However, when you have subsystems you do not need to use it. Best of all, using a subsystem requires less coding than the alternatives.

So, the code we finally use is shown in the example below.

```
1 UCLASS()
2 class UMyGamesStatsSubsystem : public UGameInstanceSubsystem
```

```
{
GENERATED_BODY()
public:
// Begin USubsystem
virtual void Initialize(FSubsystemCollectionBase& Collection) override;
virtual void Deinitialize() override;
// End USubsystem

void IncrementResourceStat();
private:
// All my variables
};
```

Copy full snippet