

Referencing Assets

Control how an asset is referenced and loaded into memory.



Unreal Engine (UE) provides a number of mechanisms to control how an asset is referenced and by extension loaded into memory. You can think of references in two ways: a hard reference where object A refers to object B and causes object B to be loaded when object A is loaded; and a soft reference where object A refers to object B via an indirect mechanism such as the string form of the path to the object. The first two sections following cover hard references with the remaining sections exploring soft references.

Direct Property Reference

This is the most common case for asset references and is exposed via the `UPROPERTY` macro. Your gameplay class exposes a `UPROPERTY` that lets your designers specify a particular asset on either the archetype via Blueprint inheritance or via an instance placed in the world. For instance, the code below is from the `AStrategyBuilding` contained in the **StrategyGame** sample and allows the designer to select which sound is played when a building of a type is constructed.

```
1 /** construction start sound stinger */  
2
```

```
3 UPROPERTY(EditDefaultsOnly, Category=Building)
4
5 USoundCue* ConstructionStartStinger;
6
```

 Copy full snippet

This property can only be set as part of the default properties for an object (the `EditDefaultsOnly` keyword controls this). The designer creates a new Blueprint class that extends from `AStrategyBuilding`. Then the sound the designer wants can be saved for that Blueprint. Whenever that designer created Blueprint is loaded, the sound that it references as part of that UPROPERTY is automatically loaded too.

Construction Time Reference

The second type of hard reference you will encounter is when the programmer knows the exact asset that needs to be loaded for a given property and sets that property as part of the object's construction. This is done using a special class, `ConstructorHelpers`, which finds objects and classes for an object during the construction phase. Again from the StrategyGame sample, here's a snippet of the HUD assigning assets that it wants to use as part of its rendering.

```
1 /** gray health bar texture */
2
3 UPROPERTY()
4
5 class UTexture2D* BarFillTexture;
6
7 AStrategyHUD::AStrategyHUD(const FObjectInitializer& ObjectInitializer) :
8 Super(ObjectInitializer)
9 {
10 static ConstructorHelpers::FObjectFinder<UTexture2D>
    BarFillObj(TEXT("/Game/UI/HUD/BarFill"));
11
12 ...
13
14 BarFillTexture = BarFillObj.Object;
15
16 ...
```

```
17
18 }
19
```

 Copy full snippet

In the constructor above, the `ConstructorHelpers` class attempts to find the asset in memory and loads it if not found. Note the full path to the asset is used to specify what to load. If the asset doesn't exist or can't be loaded due to an error, the property will be set to `nullptr`. When this happens, the code that tries to access the texture will crash. It's better to assert that the asset loaded correctly if the later code assumes the reference is valid.

The declaration of the UPROPERTY appears the same as the previous hard reference example. They function the same way with the only difference being how they are initially set. One consideration about hard references is that as objects are loaded and instantiated the hard referenced assets are loaded too. Careful consideration needs to happen or your memory footprint can balloon due to many assets being loaded at once. If you want to defer that loading or determine what to load at runtime, then the following sections help with that.

Indirect Property Reference

One easy way to control when an asset is loaded is to use a `TSoftObjectPtr`. To the designer, they can work with it just like the direct property reference. However, instead of a direct pointer reference, the property is stored as a string with template code to enable safe checking of whether the asset has been loaded yet or not. Use the `IsPending()` method to check if the asset is ready to be accessed or not. Note that using `TSoftObjectPtr` requires you to manually load the asset when you want to use it. You can use the templated `LoadObject<>()` method, `StaticLoadObject()`, or the `FStreamingManager` to load your object (see [Asynchronous Asset Loading](#) for more information). The first two methods load the asset synchronously which can cause frame rate spikes, so should only be used if you know it will not affect the gameplay.

```
1 UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category=Building)
2 TSoftObjectPtr<UStaticMesh> BaseMesh;
3
4 UStaticMesh* GetLazyLoadedMesh()
5 {
```

```

6  if (BaseMesh.IsPending())
7  {
8  const FSoftObjectPath& AssetRef = BaseMesh.ToStringReference();
9  BaseMesh = Cast< UStaticMesh>(Streamable.SynchronousLoad(AssetRef));
10 }
11 return BaseMesh.Get();
12 }
13

```

 Copy full snippet

The code above uses a `TSoftObjectPtr` of `UStaticMesh` to lazy load the mesh at runtime. The asset is checked to see if the object has been loaded or not. If it has not, a synchronous load happens using the `FStreamingManager`. Otherwise, the `UStaticMesh` pointer within the `TSoftObjectPtr` is returned to the caller.

If you want to defer loading a UClass, you use the same approach as the `TSoftObjectPtr` substituting the class specific version `TSoftClassPtr` template type. This functions the same as referring to a specific asset, but instead refers to the UClass for the asset instead of an instance.

Find/Load Object

So far these examples have all been UPROPERTY based. However, what if you want to build a string at runtime and use that to get a reference to an object? There are two options that you can use. If you want to use the UObject only if it has already been loaded or created, then the right choice is `FindObject<>()`. If you want to load the object if it is not already loaded, then using `LoadObject<>()` is the right choice. Note that `LoadObject<>()` does the equivalent of `FindObject` under the covers, so it's not necessary to try to find the object first and then load it. Here are some examples of using each function.

```

1  AFunctionalTest* TestToRun = FindObject<AFunctionalTest>(TestsOuter,
    *TestName);
2  GridTexture = LoadObject<UTexture2D>(NULL,
    TEXT("/Engine/EngineMaterials/DefaultWhiteGrid.DefaultWhiteGrid"), NULL,
    LOAD_None, NULL);
3

```

 Copy full snippet


There is a specialization of `LoadObject` available when loading a UClass. This is just an easier way to load the class and provides an automatic verification of the type. The code snippet below illustrates this.

```
1 DefaultPreviewPawnClass = LoadClass<APawn>(NULL, *PreviewPawnName, NULL,  
  LOAD_None, NULL);  
2
```

 Copy full snippet

Is the same as

```
1 DefaultPreviewPawnClass = LoadObject<UClass>(NULL, *PreviewPawnName, NULL,  
  LOAD_None, NULL);  
2  
3 if (!DefaultPreviewPawnClass->IsChildOf(APawn::StaticClass()))  
4 {  
5   DefaultPreviewPawnClass = nullptr;  
6 }
```

 Copy full snippet