Asynchronous Asset Loading

Methods for loading and unloading assets during runtime.

There are several methods in **Unreal Engine (UE)** that simplify the process of asynchronously loading asset data. These methods work identically in development and with cooked data on devices, so you do not need to maintain two code paths for loading data on demand. There are two general methods you can use to refer to and load data on demand:

FSoftObjectPaths and TSoftObjectPtr

The easiest way to have an artist or designer reference an asset is to create a `UProperty` of a hard pointer and give it a category. In UE, if you have a hard `UObject` pointer property referencing an asset, that asset will be loaded when the object containing the property is loaded (either by being placed in a map, or referenced from something like a gameinfo). If you are not careful, you can end up loading 100% of your assets at game startup time. If you want artists/designers to be able to make references to specific assets using the same UI as a hard pointer, without always loading the referenced asset, use a `FSoftObjectPath` or `TSoftObjectPtr`.

A `FSoftObjectPath` is a simple struct that contains a string with the full name of an asset. If you make a property of that type in your class, it shows up in the editor as if it was a `UObject *` property. It also properly handles cooking and redirects, so if you have a `SoftObjectPath`,

it is guaranteed to work properly on a device. A `TSoftObjectPtr` is basically a `TWeakObjectPtr` that wraps around a `FSoftObjectPath`, and will template to a specific class so you can restrict the editor UI to only allow selecting certain classes. If the referenced asset exists in memory, `TSoftObjectPtr.Get()` will return it. If it does not, you can call `ToSoftObjectPath()` to find out the asset that it refers to, load that using the method described below, then call `TSoftObjectPtr.Get()` again to dereference it.

`TSoftObjectPtrs` and Soft Object Paths are great if an artist or designer is setting up references manually, but if you want to do something like a query to find an asset satisfying certain requirements, without loading all of those assets, you want to use the asset registry and object libraries.

The Asset Registry and Object Libraries

The asset registry is a system that stores metadata about assets and allows searches and queries about those assets. It is used by the editor to display the information in the **Content Browser**, but you can also use it from gameplay code to query metadata about gameplay assets that are not currently loaded. To make data about an asset searchable, you need to add the `AssetRegistrySearchable` tag to the property. Queries to the asset registry return objects of type `FAssetData`, which contains information about the object as well as a map of key-value pairs containing the properties marked as searchable.

The easiest way to work with groups of unloaded assets is with an `ObjectLibrary`. An `ObjectLibrary` is an object that contains a list of either loaded objects or `FAssetData` for unloaded objects, that inherit from a shared base class. You load an object library by giving it a path to search, and it will add all assets in that path. This can be very useful, because you can designate parts of your content folder for different types, and artists/designers can add new assets without having to edit some manual master list. Here is an example of how you would load AssetData off disk using an object library:

```
1 if (!ObjectLibrary)
2 {
3 ObjectLibrary = UObjectLibrary::CreateLibrary(BaseClass, false, GIsEditor);
4 ObjectLibrary->AddToRoot();
5 }
6 ObjectLibrary-
  >LoadAssetDataFromPath(TEXT("/Game/PathWithAllObjectsOfSameType");
```

```
if (bFullyLoad)
{
ObjectLibrary->LoadAssetsFromAssetData();
}

```

Copy full snippet

In this example, it creates a new object library, associates a base class, then loads all asset data in a given path. It then optionally loads the actual assets. You want to fully load the assets either if the assets are small, or if you are cooking and need to make sure that they all get cooked. As long as you perform an asset registry query during cooking, and load the assets returned, your object library will work identically with cooked data on a device as in development. Once you have the asset data contained in an `ObjectLibrary`, you can do queries and selectively load certain assets. Here is an example of how you would do a query:

```
TArray<FAssetData> AssetDatas;
ObjectLibrary->GetAssetDataList(AssetDatas);

for (int32 i = 0; i < AssetDatas.Num(); ++i)
{
FAssetData& AssetData = AssetDatas[i];

const FString* FoundTypeNameString =
AssetData.TagsAndValues.Find(GET_MEMBER_NAME_CHECKED(UAssetObject,TypeName));

if (FoundTypeNameString && FoundTypeNameString->Contains(TEXT("FooType")))
{
return AssetData;
}
}

```

Copy full snippet

In this example, it searches the object library for anything that has a `TypeName` field containing `"FooType"`, then returns the first it finds. Once you have that `AssetData`, you can call `ToStringReference()` to convert that to an `FSoftObjectPath`, which you can then load asynchronously using the next system:

StreamableManager and Asynchronous Loading

Now that you have a `FSoftObjectPath` that refers to an asset on disk, how do you actually load it asynchronously? `FStreamableManager` is the easiest way to do this. First, you will need to create an `FStreamableManager`, I would suggest putting it in some sort of global game singleton object, such as one specified using `GameSingletonClassName` in `DefaultEngine.ini`. Then, you can pass your `FSoftObjectPath` into it and start a load. `SynchronousLoad` will do a simple, blocking load and return the object. This method may be fine for smaller objects, but it could potentially stall your main thread for too long. In that case, you will need to use `RequestAsyncLoad`, which will asynchronously load a group of assets and call a delegate after completion. Here is an example:

```
1  void UGameCheatManager::GrantItems()
2  {
3  TArray<FSoftObjectPath> ItemsToStream;
4  FStreamableManager& Streamable = UGameGlobals::Get().StreamableManager;
5  for(int32 i = 0; i < ItemList.Num(); ++i)
6  {
7  ItemsToStream.AddUnique(ItemList[i].ToStringReference());
8  }
9  Streamable.RequestAsyncLoad(ItemsToStream,
   FStreamableDelegate::CreateUObject(this,
   &UGameCheatManager::GrantItemsDeferred));
10 }
11
12 void UGameCheatManager::GrantItemsDeferred()
13 {
14 for(int32 i = 0; i < ItemList.Num(); ++i)
15 {
16 UGameItemData* ItemData = ItemList[i].Get();
17 if(ItemData)
18 {
19 MyPC->GrantItem(ItemData);
20 }
21 }
22 }
23
```

In this example, `ItemList` is a `TArray< TSoftObjectPtr<UGameItem> >` that was modified by designers in the editor. The code iterates that list, converts them to `StringReferences`, and queues up a load of them. When all of those items are loaded (or fail to load because they are missing), it calls the passed in delegate. That delegate then iterates the same list of items, dereferences them, and gives them to a player. `StreamableManager` keeps hard references to any assets it loads until the delegate is called, so you can safely know that none of the objects you wanted to asynchronously load will be garbage collected before the delegate is called. It releases those references after the delegate is called, so you need to hard reference them somewhere else if you want to ensure they will stay around.

You can use the same method to asynchronously load a `FAssetData`, just call `ToStringReference` on them, add them to an array, and call `RequestAsyncLoad` with a delegate. The delegate can be anything you want, so you can pass along payload information if you want. By combining the methods described above, you should be able to set up a system that allows efficient loading of any asset in your game. It will take some time to convert your gameplay code that does direct memory accesses to handle asynchronous loading, but afterwards, your game will have far fewer stalls and much lower memory usage.