Fresh Assets

# Design

Assets are heavyweight objects. They are slow to construct (generally due to file loading), large in memory, and are often referenced by other objects even before they are constructed. These distinct needs lead to a requirement for additional features beyond the normal object management facilities.

# Use Cases

Assets are referenced just like other objects. For example:

<object class=”Sprite”>

<pTexture>Texture’fancy’</pTexture>

</object>

The asset “fancy”, which is a texture, is wanted by this Sprite. The serialization of the pTexture member of Sprite will inspire a call to getObject(), as with any other reference. But this request is handled differently in a couple of ways.

1. Even though no package is specified for fancy, and even though it is not to be found in this same package, the system can still find it (somehow).
2. Texture “fancy” will not only be referenced by this serialization, it will be created. In a sense, this merely suggests that the getObject() call changes to createOrGetObject() in this case, but it’s more than this: if the object is created, it’s not just created generically as createObject() normally does—rather, it is created based on a previous configuration—what I formerly thought of as a “loader.”

That’s one use case showing two key distinctives associated with assets.

The other use case is on the back end. Unlike other objects, assets are retained even if no object other than the retainer references them, and are released only during special “cleanup” events. In code:

void Application::onCleanupNeeded()

{

releaseUnusedAssets();  
}

These three needs represent the features required to support assets.

# Implementation with Packages

Maybe packages have the ability to pre-load their XML files, storing each node in some form.

Perhaps these “preload objects”, when they are accessed as a result of getObject(), create the real objects that they represent, which is then returned.

Is it possible that all objects should offer this “preload” feature, so that all cross-package citations work this way?

**About searching.** Maybe there is a form of package, or a feature of all packages, that packages can retain their objects and release them when they are (otherwise) unreferenced.

Maybe there is a system of automatic searching of certain packages. Or maybe the system owns an anonymous package (named “”) that is implied (secondary to the current loading package) when an ObjectId has a blank package name.

I’m realizing to my horror that two of the three assets features really just amount to the question of what happens when an object in a manifest references an object that can’t be found. “To my horror” because this whole question and the most obvious solutions take me dangerously close to a system terribly like Unreal’s LinkerLoaders. Enough of that.

We can think through this in very concrete terms. When an object references another object, the system calls getObject<>(). If no package is specified, the currently loading package is checked first; otherwise the indicated package is checked, if it is exists. If getObject<>() can’t find the object, the deserializer hopes the referent will be found by the ObjectLinker later on and simply registers the target pointer for fixup on that basis.

Unless the object is known, however, or is later in the current package, the ObjectLinker will fail to find the object, and so you’ll get a broken link report and a null pointer. That is clearly a valid response and the desired one in many cases. What Assets push is the question of whether we want to expand on that system.

There are two root issues each with various branches.

## Automatic Searching

When getObject<>() receives a package name, there are two broad situations:

1. No package is indicated.
2. A package is indicated.

If a package is indicated, we request it via getPackage<>(). If it exists, all is good. But if it doesn’t, currently we fail out.

**Q:** What should we do when an object requests a specific package that is unknown?

It’s not clear to me that Assets in particular care about the answer to this question. Objects in general might, but taking a more generous approach to the answer (e.g. A: We register the package as one to be sought and loaded later) invites the Unreal Loading Syndrome of endless cross-requests and mis-ordered loads.

What if the package isn’t indicated? There are two categories of interpretation:

1. The user is requesting a search.
2. The user is indicating the “blank named” package (presumably a/the system package).

These are not mutually exclusive: certainly there could be a blank-name system package that gets searched after the current package. Is this the only other package to be searched or is there a (configurable) list? Hard to say.

By analogy with C #include files and library paths, I could use a search strategy such that:

1. The local package is searched.
2. Magic “user packages” are searched, configured arbitrarily by users.
3. Magic “system packages” are searched, configured arbitrarily by Fresh.

But I think this is more complex than needed. After all, the XML files and object systems I’m talking about loading here are generated by editor applications mainly, not by hand. It’s a data file format, not a programming language. This implies that only 1 and 3—and perhaps even a simplification of 3, such as just a single anonymous system package—is adequate.

Something obvious just occurred to me. The “using” or “import” keyword for various languages would perfectly capture #2 in a simple and (fairly) effortless way.

All right then. I’m thinking that my solution will be that there is a nameless system package into which most system things go. For now, assets will go there along with the Renderer and such. An anonymous getObject<>() request will seek the current package first, then the nameless (or secretly named) system package. Fixed up references will use precisely the same logic. Later I can extend this system to support explicit <import url=”etc.” /> additions to the search paths.

Incidentally I like that the <import> system avoids Unreal’s endless automatic loading of packages.

## Package Preloading (Indexing)

The solution above gets us as far as searching the system package for assets when no package name is indicated. But why should an asset exist in that package if it has never been created? There is a sense in which assets are implicitly created and authoritatively configured merely by being requested. This is the concept currently implemented using the assets.xml and asset-database.xml concepts, along with AssetLoader and AssetLoaderDefaults. Can it be bettered?

Note, first of all, that automatic creation of objects from pre-configured proxies is not normally a desirable feature—not for objects in general. Manifests rarely reference objects outside of the reference anyway. If you reference something like a Renderer, that doesn’t mean you want someone to create it for you—only that you expect someone already has done.

Assets are different precisely because they are heavyweight. It would be desirable, on some level, that all Assets be pre-loaded so that anyone who wants to use them may do so. We want them to be accessible as if they were always loaded. But we don’t want them to be always loaded, because they take up too much memory. So Assets want to “trick” the system by being accessible like a created object, but unloaded like an uncreated object.

The prior solution was to maintain two types of objects—Assets and AssetLoaders. AssetLoaders really are preloaded and act as proxies for Assets. The “trick” is that an AssetLoader has the same name and even answer to the same *class name* as the Asset it creates. This allows users to request Texture’white\_simple’ and for the *AssetLoader*’white\_simple’ to respond.

The notion of a “loader” is decent. There often is real code and data involved in loading that is not involved in the Asset itself, so the concept of a distinct “Loader” class is better than, say, a mere copy of the XmlElement that will later be used for the Asset. So I’m all for keeping loaders.

That leaves two features to yet be achieved.

1. Loaders should be preloaded by the system.
2. When getObject<>() requests an asset, it finds, executes, and returns the result of the loader unless the asset really does already exist.

Can I achieve these without recourse to explicit, detailed new code?

The first part isn’t a particularly heavyweight item and it’s hard to imagine avoiding special code for it. Presumably the Application at least needs to say:

m\_pAssetPackage = createPackage( “assets” ); // Must be “strong”

m\_pAssetPackage->load( “assets.xml” );

addSearchPackage( m\_pAssetPackage );

What of the second part? To do this in an automated way, we would need the general ability for gotten objects to “forward” their “getting” to another object. That is, getObject<> would have to say:

auto pObject = package.find<

typename TypeTraits< return\_t >::proxyType > ( id );

pObject = pObject->getForwardedAccessObject< return\_t >();

That is not a terribly happy bit of code, because it puts a fair amount of complexity into every object that is unlikely to be used except by a very few. What’s the alternative?

You could distort the package. That is, there could be a type of Package that changes the meaning of find(), basically lying about the returned object. This has the benefit of allowing find() to be arbitrarily contorted and knowledgeable about, for example, the need to actually seek Loaders when seeking Assets. This is essentially what AssetManager is now.

I can’t think of a better alternative. (You could also hijack ClassInfo, but I don’t see this being as good as the other two alternatives.)

Well now all told this begins to make a lot of sense. It also lands me not far from the traditional AssetManager. In essence, AssetManager derives from Package. When you create it, it does the things I listed above as needing to be done in the Application constructor (loads itself, registers itself for automatic searching). It overrides the find() function to misdirect requests for Assets in basically the same way that AssetManager does now.

What of AssetLoaders? Well, the AssetDatabase might as well be just another package again, and probably with no unique type. I’m thinking that AssetLoaderDefaults could be replaced by the newer pseudoclass system, but I’ll need to look more closely to confirm that.

So basically I’m looking at a fairly thin reworking of AssetManager in terms of Package.

(Oh, and Package will need to have a “strong reference” option, because AssetManager and LoaderDatabase will certainly use that.)

# “Active” Package

I currently have the notion of the “currentLoadingPackage()”, which is used to grab objects that are created in the process of loading a package.

This is a more generally useful idea however. Any object created when the editor is placing an object, for example, should similarly be added to the edited package (Editor::m\_pEditedPackage).

But the word “loading” doesn’t really convey that idea. Should I change the term to “active” or “capturing” package?

The more serious issue is just to note that the idea of a “capturing” package is pretty scary. It’s a large step in the direction of “pushNamespace()/popNamespace()”, which were one of the ugliest aspects of the old system.

I think it’s different enough, though, that it’s okay.

I think I’d better run with “active” package, because it’s used both for capturing (i.e. in createObject()) and for searching (i.e. in getObject()).