FreshCore

Technical Design

Jeff Wofford, Feb 16, 2013

The goal of the FreshCore library is to provide the most essential functionality for building portable C++ programs of all kinds, including console applications. Much of this functionality is a matter of mere convenience (e.g. case insensitive string comparison), or provides portable platform-dependent features that base C++11 leaves out (e.g. generic path support). The most substantial feature, however, is a robust object reflection and serialization system. The philosophy and design of this feature is the subject of this paper.

FreshCore provides a base class—Object—which is central to providing reflection and automatic serialization features. *Reflection* provides the ability for the properties and methods of objects to be queried, modified, and (in the case of functions) executed at runtime using data-driven means—normally string-based input. Example use cases include interactive consoles, scripting languages, and automatic serialization. *Reflection* also enables the system to create an object of a particular class using only that class’s string name, and the ability to search the system for an object of a given class and name.

*Automatic serialization* is an extension of the reflection feature. It provides Objects with the ability to fully load and save themselves from and to archives (e.g. XML files) without class-specific programming.

Fundamentally, then, FreshCore provides the following basic interfaces.

* createObject( *className* )
* getObject( *className, objectName* )
* Object::load( *archive* ) // No need to override in subclasses.
* Object::save( *archive* ) // No need to override in subclasses.
* Object::call( *function call expression* )
* Object::getPropertyValue( *property* )
* Object::setPropertyValue( *property, propertyValue* )

Concretely, these features are provided by a system of interrelated classes.

* **Object** – the anchor for all these features
* **Property** ­– helps in reading and assigning object properties
* **ClassInfo** – provides reflection information about a given C++ class
* **ObjectManager** – provides object creation and search functionality.

The essential functionality for creating objects by string, finding objects, accessing properties and methods, and saving objects, is fairly trivial. Loading objects, however, proves a more difficult problem calling for specialized solutions.

# The Problem of Linking an Object Graph

The problem of loading a single object from a serialized archive is fairly simple. Most property types can be parsed from string (or other) input using traditional methods. Pointers are slightly more challenging. It is no good saving and loading raw pointer addresses, since on most systems these change freely between runs of the program, much less between installations or machines. Object names (more properly “IDs”, consisting of at least the object class name and name) are stored in place of pointers. When loading a pointer, the string name of the object referent is loaded, then the *getObject()* feature is used to find the object. (This feature is implemented such that an object can safely point even to itself and still receive correct pointer fixup.) So long as the object actually exists, the serialization process is trivial. When loading a single object, if the pointer refers to a non-existent referent, then this is a simple high-level error that can be reported to the (programmer) user and fixed according to its proper cause.

Things become more difficult when loading more than one object at the same time—that is, when loading an entire *object graph.* In this case, objects may refer to each other freely. There is provably no way to ensure that for a given object O, all of O’s referents R1 through Rn are loaded prior to O being loaded. (It is possible for one or more of R1 to Rn—call it S—to mutually refer to O, in which case either O or S must be loaded without the other being present.) Therefore, when loading an *object graph*, a more sophisticated approach must be taken to ensure that all pointers (the edges of the graph) are fixed up correctly before any object in the graph is given over to the application with the assurance that it has been fully constructed.

This raises the concept of a “loading event.” A loading event has a discrete beginning and end. During the event, objects are loaded. These objects may refer freely to themselves, to each other, and to existing objects. Objects are considered “offline” during the event—as being “under construction” in much the same way that a C++ object is considered under construction while still executing its constructor. At the ending of the loading event, however, all objects in the system are considered complete and finalized. This, of course, means that all their pointers (except user-erroneous ones, naturally) have been fully fixed up.

The question, then, is twofold:

* What demarks a loading event?
* What goes on during a loading event to ensure the expectations above?

# A Historical Digression

FreshCore has existed in some form for over ten years. During this time, the serialization and fixup system has relied on a couple of implementations. The latest implementation (Feb 2013) attempts to improve on the prior implementations, and so understanding them clearly is important.

## Version 1: The ObjectManager-Centric System

The initial Fresh system centered on ObjectManager. ObjectManager was the owner of all objects. It was responsible for creating them, finding them, saving and loading them. Serialization features were not fully developed at this time, however, and so saving and loading facilities were minimal.

## Version 2: The Namespace-Centric System

During the *House of Shadows* project (2012) I relied more and more heavily on object serialization. I implemented the “manifest” concept that allowed a set of objects in a single file to be loaded in simultaneously. (Saving, though a simpler feature to implement, was not needed as seriously and not developed as fully.) While using manifests I discovered that I needed a facility for “modularizing” or “grouping” objects, so that a set of related objects could be brought in or flushed out more or less simultaneously. A particularly compelling use-case for this feature related to restarting an already-playing level. In that case, the level objects already existed, but I would (for various reasons) re-load the level rather than deleting it completely and then loading it from scratch. But re-loading the level (loading over the existing level), though it had some advantages, tended to be problematic. I needed a way to load the same level—the same manifest—but feeding the level’s objects into a new “group”. Once the new group was fully loaded, I could then delete the old group. But what kind of “group” was I to use?

A related problem that arose during HOS was associated with the use of Pseudoclasses. Pseudoclasses provide a sort of “prototype” system whereby a class with particular property settings (as opposed to the default settings) can be instanced and correctly initialized. Pseudoclasses’ property values often involved the instantiation of sub-objects, and these sub-objects needed both unique names of their own and specialized references to other sub-objects within the object. How were these unique names and name references to be provided? If you created two instances of the “Ghost” Pseudoclass, for example, and both Ghosts wanted a subobject “ghost\_head”, then both ghosts would end up sharing a single object named “ghost\_head”—not a desirable situation.

These two problems—the need to load an already-existing level into a fresh “group” and the need to differentiate sub-objects of Pseudoclasses—along with a few others motivated me to create the Namespace system, which served as the core of the second version of the serialization system.

Namespaces were based closely on the C++ notion of namespaces while vaguely associating (too vaguely, as it has turned out) with the concept of “packages” found in such languages as Java, C#, and ActionScript. Despite the obvious differences between my needs and C++’s namespaces (I needed to group and differentiate *objects*, whereas C++ groups and differentiates *types* and other language constructs), Namespaces seemed to solve my serialization problems well.

In the Namespace-centric version of the serialization system, most of ObjectManager’s features were factored into a new Namespace class. Just as an ObjectManager had held a list of all objects, now each Namespace held a list of all the objects in the namespace. Just as ObjectManager would allow you to search for any object by name, each Namespace would allow you to search it for any object by name. But because Namespaces were themselves both named and arranged in a hierarchical tree, complex object naming qualification could be employed to differentiate objects.

So, for example, when loading a *House of Shadows* level, each particular *load* of that level was placed into a discrete namespace. This ensured that each object in that load was created a second time and not confused with any prior version of the object. Yet because each object could qualify its name or references if it wished, objects within a manifest could push themselves into a particular namespace (not only the namespace directly associated with loading the level), and could refer to objects both within their own namespace and outside of it.

In this system, ObjectManager became a subclass of Namespace with a few additional features. It also served as the root of the Namespace tree (with a blank name—like the global namespace in C++—e.g., “::anObject”). Each Object also had a pointer to the namespace to which it belonged. And another class—Namechain—was used to parse, store, and calculate on qualified object names (that is, those having “::” and thus indicating a particular path through the namespace tree).

The resulting system was reasonably successful, but it suffered from what is perhaps the obvious problem: overcomplication. ObjectManager.h became a misery of convoluted code and twisted class relationships. Thankfully, namespace bugs were few, but when they appeared they were terrifyingly difficult to debug.

## The Need for a Version 3

The beginning of the *Crush the Castle Revamp* project saw major strides in the architecture of Fresh, driven largely by the implementation of the Fresh editor system. The Fresh editor put stress on certain aspects of Fresh that had long been vague in my mind. For example, most Fresh applications rely on a display tree rooted in class Stage. I had previously assumed that Stage should be a singleton. But as I implemented the editor, it became clear that class Editor should be a Stage, and yet might operate on a Stage. The singleton idea had to be eliminated, leading to improvements in the ways that object fixup between parent and stage and object initialization at the end of loading were implemented. Similarly, Fresh had always had the idea that an Application would load a stage file based on a configured file path. Now the Editor also needed to load stage files, but it also had to create and save them. It became more clear to me that both the Application class (actually ApplicationStaged) and the Editor class itself needed to behave more like traditional GUI applications operating on a Document-View paradigm, complete with such document-centered operations as New, New from Template, Save, Save As, Close, and so forth.

At the same, the Editor also needed more prosaic features such as an undo-redo history and copy and paste functionality. *Persistence*, and therefore *serialization*, features were clearly needed. How would these be provided using the existing Namespace/manifest paradigm?

Working on the Editor revealed broader problems with the Namespace/manifest paradigm. A great deal of highly finicky and arbitrary code had to be employed in both the ApplicationStaged and the Editor in order to manage Namespaces. Namespaces had always been entirely flexible, something like a file system. You could create a namespace (like creating a folder), push into a namespace (like pushd), pop from a namespace (like popd), and jump to a namespace (like cd). Once in a namespace, creating an individual object or loading a manifest would place the new objects into, or relative to, the current namespace. The intent was for namespaces to act like the namespace X { … namespace Y { … } … } paradigm in C++. But this paradigm proved unwieldy.

In essence, the trouble was that namespace usage was under-constrained. In practice, namespaces *wanted* to be closely associated—indeed, almost *directly* associated—with manifests. Yet because of their flexibility, it was difficult to reliably associate a particular namespace with a particular manifest.

Another limitation with namespaces was the assumption that a given object would always live within the namespace into which it was born. This assumption was invalid in the implementation of pasting, however. When pasting a manifest into an existing object graph, the new objects would naturally sit in a temporary “paste” namespace, but of course they could not remain there. They needed to rejoin the namespace of the objects already in the edited scene—assuming, of course, that those objects happened to all belong to the same namespace. But without the ability to move objects from namespace to namespace, pasting proved far too difficult to implement.

All in all, the Namespace system proved too complex, too unwieldy, and (paradoxically) too inflexible to conveniently serve the full range of features now required by the Editor. I actually *did* implement all these features using the Namespace system, and this experience helped me to see what an improvement would look like.

# A Package-Centric Object Management System

I said at the outset that the needs of an object graph loading-and-linking system lead to two inevitable questions:

* What demarks a loading event?
* What goes on during a loading event to ensure the expectations above?

The goal of FreshCore’s Package-centric object management system is to answer these questions in the most direct, simple, and convenient way while providing all the necessary functionality.

## What is a Package?

A package is a named collection of objects. Packages are arranged in a flat system—that is, packages may not contain each other. Packages offer a limited set of directly useful functionality, including the loading and saving of manifests that are directly (but not strictly permanently) associated with the package.

* createObject()
* findObject()
* openFile()
* saveFile()
* moveObjectToOtherPackage()
* mergeWithOtherPackage()

Packages may be freely created by a user. They may also be deleted, but not by the user: a package dies automatically when (1) it is empty of objects and (2) it has no direct references itself.

Because packages are intrinsically simpler than Namespaces (at least in that they are arranged in a flat structure), they have greater opportunity to provide a larger set of small, useful features, such as saving and loading, or merging with other packages.

## No “Default Package”

Another crucial improvement of the package system over the namespace system is that there is *no concept of a “default” or “current” package.* Whenever you create an object, you *must* specify into which package it is to go. This is to avoid the nightmare concept of a “currently active object creation destination” exemplified by the “gotoNamespace() - pushNamespace() - popNamespace()” functions of the Namespace-centric system.

That said, there are conveniently accessible “temp” and “system” packages that are readily and reliable usable for objects that don’t need a permanent home or are part of Fresh itself.

## What Else is Involved?

The Package system doesn’t merely add Packages to FreshCore or replace Namespaces with Packages. It involves a major refactor of all the major classes.

### ObjectManager

…disappears or becomes a shadow of its former self, useful only for global object searches, class searches, and package creation and search. Some previously common long-winded functions such as:

DisplayObject::ptr pObject = dynamic\_freshptr\_cast< DisplayObject::ptr >( ObjectManager::instance().createObject( “Sprite”, … ));

…are now greatly simplified to more closely resemble the std::make\_shared<> model:

DisplayObject::ptr pObject = createObject< DisplayObject >( “Sprite”, … );

or better yet:

auto pObject = createObject< DisplayObject >( “Sprite”, … );

Note that in these examples, a Sprite class is created, not actually a DisplayObject per se. Leaving off the class name as the first parameter allows you to create the precise class specified in the function template parameter:

auto pObject = createObject< Sprite >( … );

Much nicer.

### ObjectId

Becomes more robust. Namechain is replaced by a single member of ObjectId: *packageName*. When used as a reference this may be left blank, where it takes on the value of the same package as the referrer.

When searching for an object with a blank package name in the reference, if the object is not found in the current package, it may be sought in the system package. This allows objects to continue to reference “Texture’white\_simple’” and have it be found in “Texture’::white\_simple’”.

## A Single-Package Loading Event

Loading a single package is now simple, using essentially the same logic as a current loadManifest() call:

1. All objects in the package are loaded, along with pseudoclasses.
2. A root object—the first object in the manifest, if unspecified—is identified and retained. (Otherwise the package objects would instantly die after loading is complete.)
3. Object linking is completed, and incomplete references are reported as warnings.
4. All objects in the package are asked to postLoad() (or “postFixup()”), allowing them to perform initialization directly responding to pointer fixups.

## Multi-Package Loading Events

Why didn’t I move to a package-based system much earlier? One of my motivations to avoid a system like this was to avoid one of Unreal’s most serious architectural problems. Although the Unreal Engine 1 and 2 loading system was brilliantly elegant in design, it proved to load with incredible sloth when moved to console systems. A whole new system—a pretty deeply hacky system called “linear loading”—had to be rigged up quickly for the PS2 version of Unreal Tournament (if my memory serves correctly). I wanted to avoid this performance problem.

Unreal’s loading system allows objects in one package to force the loading of individual objects in other packages. The system is recursive: whenever a reference is found, the new package is loaded, searched (via a table) for the referent object, and the referent object is loaded. This object may, of course, in turn request other objects either in the same package, the prior package, or yet a third package. And so on recursively throughout the reference graph, which may span any number of packages and objects. The system is tolerably fast in terms of CPU time, fairly slow on a hard disk, but intolerable on a DVD where seek times are long.

The Fresh “manifest” concept which emerged at the start of the Namespace-centric version of the system embodies the idea that when a set of objects is loaded, the whole set is loaded sequentially and with no fuss. The intent of this policy is to avoid the pick-and-choose fragmentation that bedeviled the Unreal system. The Fresh Package system likewise inherits this idea.

Yet it is at least worth considering the option of allowing objects in a package to reference objects in other, as yet unloaded, packages. The implementation of this feature is trivial, but the various philosophical options for how the system should be limited offer various tradeoffs.

1. Objects cannot reference unloaded objects. This is the simplest and most restrictive policy.
2. References to yet-unloaded objects causes the unloaded objects’ whole package(s) to load (with recursive extension).
3. References to yet-unloaded objects causes the object to be loaded singly from the package (with recursive extension). This is the Unreal policy.

The choice is really between #1 and #2. Is there a use-case for #2 that is compelling enough to offset the additional complexity and performance risk?

Actually, the use case already exists in a different form. The “<includes>” element acts something like this feature, enabling a manifest to embed another manifest within it. But this is not quite the same thing, because in that case, the <included> elements are actually taken as being inside the embedding manifest, not simply referred to from it.

Still, this model does suggest a use case. A reusable set of optional, non-system objects (not Assets, for example), might be referenced by objects in a package. This would allow centralization of “utility” or “reusable” objects within certain packages. Actually, the Asset system could be reworked to function this way, although the existing system has advantages of its own, including per-object loading and pre-loaded metadata.

Hm. Yes. Apart from Assets, I’m having a hard time thinking of “utility objects” that would be invoked from some levels but not from others, and that would not be equally well encapsulated in the form of <include>.

So for now I don’t see a compelling reason to allow chained, multi-package load events.

## Explicit Multiple-Package Loading

A fairly common use-case can cause multiple packages to be loaded at the same time even if no automated system for chaining package loads is provided. An object can explicitly load a second package during its postLoad() function. This use case appeared during HOS development and had to be quashed due to the difficulty of supporting it with the Namespace-oriented system. The use case was that the player needed to load a PlayerCharacter (a particular art set for the player, such as the male or female versions) as it was constructed, and the particular chosen PlayerCharacter was stored in a named manifest file.

The package system supports this kind of explicit loading effortlessly. Since each package load is autonomous, multiple packages can load in a nested fashion. This can result in redundant fixup requests (e.g., the inner package calls for fixup at the end of its loading process, and then the outer package immediately does so again), but not to any ill effect.