Fresh Object Namespaces

The need for object namespaces was made obvious by problems that occur when one level is loaded after another—even (and perhaps especially) if the second level is the same as the first (that is, built from the same file).

The problem arises because of the behavior of two systems: essentially, deletion and loading.

Naively, to re-start a level you would want to delete, recreate, and reload it. Thus:

delete pLevel;

pLevel = new Level();

load( pLevel );

In Fresh, neither object deletion nor creation are carried out explicitly in this case. So:

pLevel = nullptr;

load( file ); // Automatically generating and setting pLevel

…is roughly equivalent.

And yet it’s not equivalent, because both of these two lines differ markedly in how they behave relative to the original code. That’s a *good* thing—these differences in behavior represent automation of normally tedious and error-prone tasks. Nonetheless, the differences result in this approach to restarting a level failing to give the desired behavior. Specifically:

1. pLevel = nullptr suggests that the level should be destroyed, but it is not likely to be destroyed until the end of the current frame (and even this is not a guarantee).
2. load( file ) does load the file, but it doesn’t necessarily do so with a new, fresh level object. If the file mentions by name the same level that already exists, it will re-load that level rather than recreating it.

Reloading is not equivalent to recreating in all cases because recreating sets all variables back to their defaults, whereas reloading only modifies those specific variables that are explicitly set in the file.

When loading, the “getIfObjectAlreadyExists” semantics prevail, which means that objects mentioned in the file that already exist are reloaded rather than created anew. This is the desired behavior because it enables “tiered loading”, where a series of files—or elements within a file—tweak an existing object by mentioning it again and again with the variables that they’re particularly interested in. This is not a positively essential feature, but I rely on it already and I’m planning to use it even more in the future (e.g. in the Fresh Asset Processor’s transform file).

Therefore, the naïve approach to “deleting” a level and reloading it does not work in Fresh.

What are the solutions?

* **Force deletion.** But this goes against the whole spirit of Fresh’s memory management scheme.
* **Wait for automatic deletion to complete before reloading.** But this wait is of an indeterminate time. Even if the wait would be acceptable in the context of loading new levels (because hitches are perhaps expected), it would not be acceptable in other cases where deletion and reloading might happen mid-game.
* **Manually reset the level.** And all its subobjects? Are you kidding?
* **“Temporarily” (in some sense) disable the getIfObjectAlreadyExists semantics.** The trouble is defining “temporarily.” If temporarily means “throughout level loading,” then you’ve eliminated getIfObjectAlreadyExistscompletely—“temporarily” is a ruse. If you mean, “whenever an object is seen for the first time (only) within a level,” then this sounds like it might be the right idea. But then how do you achieve it?
* **Force a name change on the new level.** This is the simplest solution and it’s refreshingly effective. If we simply force levelX to be called 01levelX on its second load and 02levelX on its third load, then it will not be identified as the same object as the preceding level and so it will be created anew.

The last two approaches begin to hint at something effective. The downside here is that the name change approach only helps with the level object itself. What we want is something that will change the name of every object within the level, too (unless this behavior is explicitly overridden within the file).

Indeed, I encountered the need for changing the name of every object in the file before I encountered the need to change the name of the file object. My solution at the time was to *manually* postfix each object name with a unique tag for that level file. I could avoid the recreate effect for a particular object by leaving the object’s name non-postfixed within multiple levels. In that case the object would “travel” between levels, which is a useful, if not absolutely crucial, facility.

All of this pointed me to a single, fairly simple, and yet probably adequate solution. That is the implementation of “Object Namespaces” within the object management system.

(Note that I call them Object Namespaces to distinguish them from Class [or Type] Namespaces, which I’ve contemplated from time to time for several years.)

The concept is highly—almost completely—analogous to C++ namespaces. Think of it this way. A Namespace is a container of objects and other Namespaces. Every object sits within one namespace, although in an indirect sense it can sit within a whole list of namespaces. To illustrate this in quasi-C++ terms, consider:

namespace world

{

namespace loadedLevel14

{

<object class=”Level” name=”level001”>

<children>

<object class=”Player” name=”thePlayer”/>

</children>

</object>

}

}

This file, which mixes C++ namespace syntax and Fresh’s Object XML syntax, defines two objects:

Level’::world::loadedLevel14::level001’

Player’::world::loadedLevel14::thePlayer’

In reality I only see the need for a single layer of namespace at this point, but I added another layer just to illustrate.

In this example, I’m imagining that the GameWorld’s loading code established the loadedLevel14 namespace with C++ code something like this.

ObjectManager::instance().beginNamespace( “loadedLevel” + nLoads++ );

load( “level001” );

ObjectManager::instance().endNamespace();

Therefore, a subsequent call to GameWorld’s loading code will create additional objects. Structurally, the overall result will be:

namespace world

{

namespace loadedLevel14

{

<object class=”Level” name=”level001”>

<children>

<object class=”Player” name=”thePlayer”/>

</children>

</object>

}

namespace loadedLevel15

{

<object class=”Level” name=”level001”>

<children>

<object class=”Player” name=”thePlayer”/>

</children>

</object>

}

}

And we then have these objects:

Level’::world::loadedLevel14::level001’

Player’::world::loadedLevel14::thePlayer’

Level’::world::loadedLevel15::level001’

Player’::world::loadedLevel15::thePlayer’

Because Player’::world::loadedLevel15::thePlayer’ does not have the same *qualified name* as Player’::world::loadedLevel14::thePlayer’, it will not be reckoned as the same object and will therefore be created and loaded anew.

Within an XML file (or other string-object-name contexts), objects can specify their own names as well as other objects’ names with or without qualification, with syntax identical to C++. E.g.

<object class=”Level” name=”level001”>

<pPlayer>**Player’thePlayer’**</pPlayer>

<children>

<object class=”Player” name=”thePlayer”/>

</children>

</object>

Here we have an unqualified name, and so the ObjectManager will *search* the namespace hierarchy for the correct player, starting with the *current namespace.* Devising the precise behavior of this search is one of the main purposes of writing this document.

The file could also use a qualified name.

<object class=”Level” name=”level001”>

<pPlayer>**Player’world::loadedLevel14::thePlayer’**</pPlayer>

<children>

<object class=”Player” name=”thePlayer”/>

</children>

</object>

Now the referenced player will be the one in loadedLevel14, even if that is not the most local player.

As in C++, we can also qualify to global scope (the default scope when no namespace is specified, and the one which, in a sense, Fresh has always previously used).

<object class=”Level” name=”level001”>

<pPlayer>**Player’::thePlayer’**</pPlayer>

<children>

<object class=”Player” name=”thePlayer”/>

</children>

</object>

Now if there’s a player in the global scope, the level will point to it. Otherwise, pointer fixup will fail, a warning will be issued (from the existing ObjectLinker mechanism), and pPlayer will remain null.

# The Search Mechanism

Given an object name, how do we find the object referenced by that name?

The search mechanism operates on a chain of object names known as a *qualified name*. In string form, the chain looks like this:

world::loadedLevel14::level001

…which is equivalent to a list of strings where the most-abstracted namespace is at the front of the list and the object’s actual name is at the back.

Because the global namespace has an empty name, the above qualified name might be equivalent to:

::world::loadedLevel14::level001

The difference, however, is that the first example is not *fully* qualified, whereas the second is completely qualified. The second necessarily shows a complete branch of namespaces from the root to a leaf, whereas the first may (or may not) only represent a sub-branch of the namespace tree.

The search mechanism must know the *current* namespace. Given the qualified name (namechain) and the *current* namespace, the algorithm proceeds.

Let *n* be the *current* namespace.

begin repeat

if n is null, return the null object

if the qualified name is recognized in *n*,

return the referent

else, let n be n’s parent namespace

repeat

But what does it mean, “if the name is recognized in n”?

let *p* be the first name in *namechain*

if the size of namechain is 1,

Lookup and return *p* in *n*’s list of objects

else

if *n* has the name *p*

pop the first name from *namechain*

repeat using the new *n* and *namechain*

else return failure

By Jove I think it’ll work.

# Object Creation

Objects are themselves given the unqualified form of their name. This conserves space and avoids redundancy between Namespaces and object names. An object is always created in the “current” namespace. By default this is the global namespace—that is, the ObjectManager itself.

Each object holds a pointer to its immediately-containing namespace. An object can reproduce its fully qualified name by traversing the chain of namespaces back toward the global, appending Namespace names and delimiters as it visits each one.

# Controlling Object Namespace

I’ve lived with the above system for a few weeks and found it to be generally useful and effective. There is, however, one problem that causes rather than solves. The problem is that Assets tend to be created in the level-load-specific namespace, because they are often created in the course of loading a level. This causes them not to be shared across loads even when they could be. The solution is to force all Assets to always sit in the global namespace. Fair enough.

There is an opposite problem. The level-load-specific namespace is only “current” while the level is actually loaded. But what if an object is created in the middle of playing a level? In that case, the object will be situated in the global namespace even though it is “really” part of the level. I can’t remember why at the moment, but this is a problem in practice.

The solution to both of these problems is to allow the user to more finely control where a new object is created. The current beginNamespace()/endNamespace() solution isn’t flexible enough because it necessarily pushes up and down through the namespace tree. We want to be able to jump through the tree, either by providing a specific namespace object through the createObject(), or by indicating the precise namespace by qualifying the string name of the object.

This latter point connects to a missing bit of functionality, which is that an object with a qualified (fully or partially) name should be able to “jump” through the namespace tree. This works for referencing objects but not (I think) for creating them.

So there are two solutions here: (1) Assets always force themselves into the global namespace. (2) Objects that create other objects in mid-level should generally give the new object the creating object’s own namespace.

To implement this, I need to implement parsing of object names during object registration (Namespace::registerObject()). I may also need to provide an Object accessor to give the string fully-qualified namespace name for that object’s namespace.