Fresh Pseudoclasses

It’s time to implement a feature I’ve had in mind for a long time. I’ve called it various things both in my mind and in the Fresh documents: prefabs, presets, prototypes, and templates. Lately I’ve hit on a solution—or the beginnings of one—this is most naturally termed “pseudoclasses.”

First let me outline the basic concept, then I’ll give a series of use cases. Then I’ll expand on the concept to outline a complete solution.

# What is a Pseudoclass?

A pseudoclass is the marriage of a class (that is, a Fresh::ClassInfo) and an ObjectStore. Classes are registered just like other classes, so if you say,

<object class=”MyPseudoClass”>…

You will get an object of whatever *real* class the pseudoclass references. Crucially, however, the object’s initial data state—in effect, the XmlElement that initializes it—will be set to some value determined by the pseudoclass. Thus, any property definitions within the object element will supersede the pseudoclass’s initial state, which in turn supersede’s the object’s C++ value initializations as established by the constructor.

Properties are therefore established in three stages, with each stage superseding the previous.

1. C++ member variable initializers and constructor
2. Pseudoclass’s property values.
3. Object’s XmlElement property values (set in loadFromXml()).

For dynamically-created objects, only the first two stages apply. E.g.:

ParticleEmitter::ptr pEmitter = ObjectManager::instance().createObject( “MyEmitter” );

…delivers a ParticleEmitter (and there’s an important dynamic\_cast missing here) with initial values modified by the MyEmitter pseudoclass.

# Why Pseudoclasses?

Pseudoclasses answer a small set of fairly distinct needs.

## Eliminating “Blank” Classes

A minor goal of pseudoclasses is to eliminate C++ classes like “Coin”:

class Coin : public Treasure

{

FRESH\_DECLARE\_CLASS( Coin, Treasure );

};

inline Coin::Coin( NameRef name )

: Super( name )

{

m\_points = 1;

setTextureByName( "coin01" );

setColor( 0xffffdd00 );

setScale( 0.75f );

}

Class Coin is simply a Treasure with some properties set differently to the normal defaults. A pseudoclass would allow this class to be generated in data only, with slightly less headache involved in FRESH\_IMPLEMENT\_CLASS() bureaucracy and so forth.

## Simplifying Dynamic Data

ParticleEmitters are a pain to setup, typically involving ~20 lines of C++ code which does nothing but set magic numbers. This is a classic case for preferring data over code. And yet there is no real way to offload this complexity to data. You can define a *specific* ParticleEmitter in XML, but this is an actual *instance*, not a type or preset. Certain classes support ParticleEmitter ObjectStores, but again this is implemented on a per-use basis. What we want is to replace this:

ParticleEmitter::ptr pSparkleEmitter = new ParticleEmitter();

pSparkleEmitter->setMaxParticles( 30 );

pSparkleEmitter->setBaseParticleScale( 32.0f );

pSparkleEmitter->setParticleTextureByName( "particle\_gain" );

pSparkleEmitter->blendMode( Fresh::Renderer::BM\_Add );

pSparkleEmitter->setCycleTime( 1.5 );

pSparkleEmitter->addScaleKeyframe( 0.0, 0 );

pSparkleEmitter->addScaleKeyframe( 0.1, 1.0f );

pSparkleEmitter->addScaleKeyframe( 0.7, 1.0f );

pSparkleEmitter->addScaleKeyframe( 1.0, 0.0f );

pSparkleEmitter->setPerParticleColorRange( colorRange.min, colorRange.max );

pSparkleEmitter->setPerFrameColorFlickerRange( Color::Invisible, Color::White );

pSparkleEmitter->setSpawnSpeedRange( 4.0f, 32.0f );

pSparkleEmitter->setVelocityDamping( 0 );

pSparkleEmitter->setParticleGravity( Vector2f( 0.0f, 2000.0f ));

const real spread = 30.0f;

pSparkleEmitter->setSpawnVelocityAngleRange( -90.0f - spread, -90.0f + spread);

pSparkleEmitter->setSpawnScaleRange( 0.25f, 1.0f );

pSparkleEmitter->setMarkForDeletionWhenAllParticlesDead( true );

pSparkleEmitter->spawnBurst();

…with this…

ParticleEmitter::ptr pSparkleEmitter = ObjectManager::instance().createObject( “SparkleEmitter” );

pSparkleEmitter->spawnBurst();

...and elsewhere…

<class name=”SparkleEmitter” extends=”ParticleEmitter”>

<initialParticles>30</initialParticles>

<baseParticleScale>32</baseParticleScale>

… etc.

</class>

One advantage of this system is simplicity of editing, moving code-generated data to data-generated data. Another major advantage is **mutability**. I can dynamically modify a pseudoclass while the game is running, such that new instances of the class receive the new values immediately. (Note that existing instances would go unmodified, because the pseudoclass only acts during object creation.)

## Enabling Reuse of Settings

Consider this XML code currently present in hudGame.xml.

<object class="MovieClip" name="gem1">

<position>-127,-4</position>

<scale>1.45</scale>

<isLooping>false</isLooping>

<children>

<object class="Sprite" name="gem1inner">

<pTexture>Texture'gem'</pTexture>

</object>

</children>

<passthrough>

<keyframe t="0" label="celebrate">

<child name="gem1inner">

<scale>1</scale>

<color>ffff0000</color>

</child>

<tween />

</keyframe>

<keyframe t="2">

<child name="gem1inner">

<scale>1.5</scale>

<color>ffffffff</color>

</child>

<tween type="QuadEaseOut" />

</keyframe>

<keyframe t="10">

<child name="gem1inner">

<scale>1</scale>

<color>ffff0000</color>

</child>

<tween />

</keyframe>

</passthrough>

</object>

<object class="MovieClip" name="gem2">

<position>0,-4</position>

<scale>1.45</scale>

<isLooping>false</isLooping>

<children>

<object class="Sprite" name="gem2inner">

<pTexture>Texture'gem'</pTexture>

</object>

</children>

<passthrough>

<keyframe t="0" label="celebrate">

<child name="gem2inner">

<scale>1</scale>

<color>ffff0000</color>

</child>

<tween />

</keyframe>

<keyframe t="2">

<child name="gem2inner">

<scale>1.5</scale>

<color>ffffffff</color>

</child>

<tween type="QuadEaseOut" />

</keyframe>

<keyframe t="10">

<child name="gem2inner">

<scale>1</scale>

<color>ffff0000</color>

</child>

<tween />

</keyframe>

</passthrough>

</object>

<object class="MovieClip" name="gem3">

<position>127,-4</position>

<scale>1.45</scale>

<isLooping>false</isLooping>

<children>

<object class="Sprite" name="gem3inner">

<pTexture>Texture'gem'</pTexture>

</object>

</children>

<passthrough>

<keyframe t="0" label="celebrate">

<child name="gem3inner">

<scale>1</scale>

<color>ffff0000</color>

</child>

<tween />

</keyframe>

<keyframe t="2">

<child name="gem3inner">

<scale>1.5</scale>

<color>ffffffff</color>

</child>

<tween type="QuadEaseOut" />

</keyframe>

<keyframe t="10">

<child name="gem3inner">

<scale>1</scale>

<color>ffff0000</color>

</child>

<tween />

</keyframe>

</passthrough>

</object>

We have here three MovieClips named gem1, gem2, and gem3. Each contains a child named gem?inner where the ? is replaced with the number of the containing gem. The MovieClips have identical keyframes apart from the naming matchups. It is a vast amount of code accomplishing very little—classic redundancy.

We could eliminate this redundancy by rolling these definitions into a C++ class and code. Each “gem” would then be a couple of lines of XML data. But now we’re using code to manage data, which is not a happy move.

The solution is to expand the ability of Fresh’s data to deal with redundancy.

How to do this isn’t entirely clear to me, but what I want in this case is a single pseudoclass that rolls up the whole definition of the MovieClip. The individual gems would then use this pseudoclass.

The difficulty in this particular case is that names would also have to be “relativized” so that each child gets a unique name (possibly aided by namespaces?) despite being defined in a single location. Possibly:

<class name="HudGem" extends="MovieClip">

<scale>1.45</scale>

<isLooping>false</isLooping>

<children>

<object class="Sprite" name="$SELF$::inner">

<pTexture>Texture'gem'</pTexture>

</object>

</children>

<passthrough>

<keyframe t="0" label="celebrate">

<child name="$SELF$::inner">

<scale>1</scale>

<color>ffff0000</color>

</child>

<tween />

</keyframe>

<keyframe t="2">

<child name="$SELF$::inner">

<scale>1.5</scale>

<color>ffffffff</color>

</child>

<tween type="QuadEaseOut" />

</keyframe>

<keyframe t="10">

<child name="$SELF$::inner">

<scale>1</scale>

<color>ffff0000</color>

</child>

<tween />

</keyframe>

</passthrough>

</class>

…where the “$SELF$” symbol automatically replaces the object’s own name. This isn’t a complete solution, however, because creating namespaces by implication isn’t supported yet. However, the above example could use “$SELF$inner” to achieve the same result.

# Outstanding Questions

1. What is the lifespan of a pseudoclass? Presumably it is constructed when it’s seen in an XML file (or may be constructed with C++ code), but when does it die? **ANSWER:** Never. They’re not large enough to matter, and they have no real time-performance cost.
2. Can a pseudoclass *extend* another pseudoclass. **ANSWER:** Definitely. This is very useful.
3. Is pseudoclass an actual kind of ClassInfo? That is, does an Object reference its pseudoclass as its class, or the “real” class that the pseudoclass is based on? **HAVE TO THINK ABOUT THIS.** Default answer is “pseudoclass”, because although they have no post-construction behavior, (1) you still might want to filter on the class, differentiating, for example, EvilCoins from Coins using a function like IsA(), and (2) if I end up using a “DefaultObject” concept for minimal object save-serialization, knowing the default *pseudo*Object will help to minimize the save.

# Implementation Thoughts

## Option 1: Pseudoclass as ClassInfo

Pseudoclass is a C++ class derived from ClassInfo. It has an ObjectStore in addition to the usual ClassInfo properties.

ClassInfo’s are normally constructed and registered during global initialization as a result of the automatic construction of global variables and function-static variables. This system is managed with the use of FRESH\_DECLARE\_CLASS and FRESH\_IMPLEMENT\_CLASS macros.

Pseudoclasses extend that same system (they are registered to the same place: ObjectManager::registerObjectFactory()), but are created differently. In loadManifest(), elements with the “class” name are used to initialize a pseudoclass, which is stored on the heap, and a pseudoclass factory, which is registered with the ObjectManager as a typical object factory. Pseudoclass factories are like normal factories except that they call loadFromXml() on the object with the pseudoclass’s XmlElement. Since pseudoclasses can extend each other and thus “chain”, multiple calls to loadFromXml() may be required in addition to the final, normal one. **Problem:** Objects sometimes expect to have loadFromXml() called *once*, and may do “finalizing” things during this function, such that it can’t always reasonably be called twice. **Alternative**: If multiple calls to loadFromXml() is unworkable (actually it’s already part of the system that loadFromXml() *may be* called multiple times, so it should be made to be workable), we could concatenate the pseudoclass XmlElement with any incoming XmlElement. This would involve modifying ObjectManager::createObject(), as well as implementing a general-purpose XmlElement *merge* function—probably far from trivial. The potential behavior of the two systems is quite different.

Objects currently reference their ClassInfo through a static ClassInfo variable tied directly to the Object’s C++ class. This makes it difficult to have the Object reference the pseudoclass as if it were the object’s real “class.” We’d have to add an additional per-instance pointer to each object to bypass this behavior, and initialize the pointer in the object’s constructor. This would be quite a pain in the bum, but not totally unworkable.

## Option 2: Pseudoclass as glorified XmlElement chain

If we reject the idea of having objects reference their pseudoclasses, then pseudoclasses are really more like “presets” than pseudoclasses. They could still inherit from each other, but they needn’t be (and probably shouldn’t be) ClassInfos per se: rather, a tuple of a ClassInfo pointer and an XmlElement, along with a “superpreset”, as it were, which must point to the same ClassInfo. This is a “thinner”, clunkier, but potentially simpler implementation.

It opens the question of whether the syntax for creating an object should be:

* **In Xml:** <object class=”Class or Pseudoclass”>
* **In C++:** ObjectManager::instance().createObject( “Class or Pseudoclass”)

…or…

* **In Xml:** <object class=”Class” preset=”Pseudoclass (optional)”>
* **In C++:** ObjectManager::instance().createObject( “Class”, “Optional Pseudoclass” )

The first form is more convenient, in some sense, but implies—falsely—that pseudoclasses really are classes. It offers more leeway to later solve the other problems with pseudoclasses as classes, however.

The second from is more long-winded but more explicit as to what “pseudoclasses” are doing—basically just providing an additional XmlElement (or series thereof) to load into the object before the last XmlElement.

I’m more frightened of and attracted to the first form. What persuades me to run with it is that it keeps the interface consistent. I don’t need to add yet another optional parameter to createOrGetObject(), for example, along with createAsset() etc. As far as these functions are concerned, and the basic XML syntax, pseudoclasses *are* classes. But the ObjectManager sniffs out the difference.

# The Implementation

Pseudoclass is a C++ class known to ObjectManager. It consists of:

1. a pointer to a ClassInfo (must be non-null)
2. a pointer to an XmlElement (must be non-null, though element may be empty)
3. a pointer to another Pseudoclass (can be null)

Pseudoclasses are created and registered in C++ through an ObjectManager function, but this is normally invoked using a <class> element in XML.

In ObjectManager::createObject(), ObjectManager first looks for a valid factory for the given class name. If it fails to find it, it searches the registry of pseudoclasses. If it finds one, it uses the pseudoclass’s class’s factory.

Given the appropriate factory, createObject() then creates the object. If a pseudoclass is involved, it then asks the pseudoclass to initialize the object. This initialization involves calling the object’s loadFromXml() function, but initialization is recursive up from the bottom of the Pseudoclass chain.

Finally the object’s loadFromXml() function is called as normally (this is an optional call).