Fresh Classes

An acute need has arisen in Fresh concerning the way that Pseudoclasses are handled. The purpose of this document is to define and resolve these problems.

# The Problem

The fundamental manifestation of the problem is this.

All CastleElements are actually created based on a pseudoclass. The pseudoclass specifies—really consists of—an XmlElement that serves as an additional serialization load step while the element is being constructed. This process proceeds recursively up from the root of the pseudoclass chain to the pseudoclass specified for the particular object. In a nutshell, a pseudoclass modifies an object at construction time by imposing additional XmlElements on the object after construction. This is an extremely useful feature and works terribly well, especially in combination with the “$^” name notation, which enables an object to reference its parent object in a generic way.

When objects save themselves, they only save a property if the object’s value for that property is different from the “default value” for that property. The default is normally zero (as interpreted by the given type), but may be specified to some other C++-domain value (*not* XML domain) through use of the DVAR macro, or potentially more generally (though uncommonly) through direct manipulation of the property, for example in the object constructor. This policy of avoiding saving properties with their default values is not just a matter of efficiency or file readability (although it does serve those ends). It also implies an important system behavior. Namely, it enforces these rules.

1. If an object has an idiosyncratic value for a property, then that value will be preserved even as the class default value changes.
2. If an object has not specified an idiosyncratic value for a property, then changes to the class default value will propagate automatically to the object.

The problem I have is that pseudoclasses, while acting as classes in some respects, utterly fail to preserve those rules. Concretely, when a CastleElement saves, it saves its mass and creationVerts, among other properties, the values of which it has gained through serializing its pseudoclass’s designated XmlELement. Then when it saves, the CastleElement forgets its pseudoclass, identifying itself as the pseudoclass’s native base class, and redundantly saves these properties. This means that if you later change those pseudoclass values, such as for mass or creationVerts, they will *not* be taken up by CastleElements already residing in a loaded file. In essence, already-saved objects that are configured using pseudoclasses become impervious to changes in the properties specified by those pseudoclasses, when they should instead inherit changes to those properties.

A more minor but related problem is that objects do not recognize pseudoclasses at all. This means that from a level designer’s standpoint (for example), you may have created a “iron\_maiden”, but because an iron\_maiden is really just a pseudoclass wrapper on CastleElement, the object is identified as a CastleElement from creation onward: in the property pane, in saved files, etc. You cannot, for example, use the console to list all instances of class (actually pseudoclass) iron\_maiden. It would be better if you could.

The desire, then, is to support two related features that generally amount to treating pseudoclasses more like (or identically to) ClassInfos.

1. Pseudoclasses should be ClassInfos, at least semantically. An object should be able to say, “this is my ClassInfo,” and actually point to a Pseudoclass. As an extension of that idea, it should be able to say, “this is my class name”, and name a Pseudoclass. hasClass() should fully recognize Pseudoclasses, and so on.
2. When saving, an object should recognize when its property has its default value, even as specified by a Pseudoclass.

# The C++/XML Domain Problem

The difficult aspect of this problem is the difficulty of specifying default values in a uniform way. In essence the problem is that the way that ClassInfos specify default values and the way that Pseudoclasses specify *what amount to* default values are utterly different.

A comparison:

|  |  |  |
| --- | --- | --- |
| **Question** | **Native ClassInfo** | **Pseudoclass** |
| Where is the default value stored? | In the property | Nowhere (implied by pseudoclass chain) |
| In what format is it stored? | As a C++ variable value | As (probably handwritten) text |
| How may values be compared? | Directly (==) | Perhaps through textual normalization and string comparison |
| What value types are likely? | Simple, flat | Complex, with subtrees, keyframes, etc. |

One way of framing the problem is this. If we made Pseudoclass into a kind of (or aspect of) ClassInfo, how would we implement the defaultValue aspect of a property for the pseudoclass? There are at least two aspects to the problem.

1. Normally classes only specify properties that were introduced in that class. This is in keeping both with C++ initializer list policy and with Unreal defaultproperties, though it’s not clear to me whether there’s some deep and inevitable force behind this tradition. Pseudoclasses do nothing *but* specify new values for properties provided in “lower” classes. This is clear breach of semantics.
2. Pseudoclasses specify complex “property values” via XmlElements. Actually, they don’t specify property values so much as recipes for configuring an object after construction. ClassInfo default property values, on the other hand, are focused, C++ type values, often amounting to literals. This is a vast difference. Is it indelible and deep, or a matter of habit?

This latter question is the key. It implies that one of two general classes of solution will be necessary.

1. Preserve XmlElements as the fundamental mechanism for pseudoclasses. Solve the problem by giving objects the ability to somehow recognize that a particular property value is no different from the “default” as implied by the loading of the chain of Pseudoclass XmlElements.
2. “Bake” Pseudoclass values into raw C++ values, so that the Pseudoclass XmlElement-based system is subsumed into class Property’s defaultValue system. Value comparison then becomes trivial.

There are questions of efficiency below the hood here. Is it more expensive to store an XmlElement or some “baked” version of it? Is baking more expensive or string comparison (if that’s where we end up)? But let’s set aside efficiency concerns for the most part, because they’re not the major concern.

A major question is that of “value depth.” The fundamental difference between default values and XmlElements is that default values tend to be simple literals—at the most complicated, “new Sprite( “whatever” )”—whereas XmlElements tend to consist of whole trees of objects upon objects. It seems impossible that we could “bake” these trees into anything sensible. Yes, I’d say I see a real obstacle here.

But perhaps we’re only concerned with default values for non-pointer types? That would seem reasonable, on the whole. Or at least we could exclude particular designated properties from this system, such as DisplayObjectContainer::m\_children.

The fundamental question, then, is: how do we implement that part of Object::serialize() that asks if( pProp->hasDefaultValue( this )) such that the effect of Pseudoclass XmlElements are preserved.

# Potential Solutions and Aspects of Solutions

## Objects Store initial state

One possible solution would be for each and every object to store a copy of itself (or all applicable properties) as it existed at the end of initialization. The object then compares its properties with its state at construction in order to determine if the object changed.

This solution requires roughly double the memory for all objects, and therefore seems prohibitive.

## Classes construct an inert default object

A more Unreal-like solution might be possible. Each ClassInfo (including pseudoclasses) create an instance of itself, possibly at the time that its first instance is created. (I’m having déjà vu at this moment—I’m sure I’ve thought through this before.) This “default object” is inert (in some sense), and serves only as a standard to which other objects may be compared. Because the default object can serialize Pseudoclass XmlElements (perhaps ignoring certain designated properties), it can serve as an accurate representation of any object of that pseudoclass/class as it should look write after initialization, in the absence of any other property change due to loading.

This is a promising solution, but it does impose some potentially difficult semantic requirements. In particular, it means that all classes would have to support a “minimal constructor” that keeps the object in an inert state (not adding it to management lists and so forth). It might be prudent to store this object in global space, as a static member of ClassInfo or the like; on the other hand, it could be constructed with new. In any case, the semantic requirement that all classes support an “inert form” is difficult because the rule is both strict and vague—not a happy combination. Nonetheless this seems viable, and the behavioral and efficiency qualities of the solution are desirable.

In more detail:

1. Pseudoclass would be rolled into, or subclass from, ClassInfo.
2. ClassInfo would gain a “default object” concept: an Object\* pointing to an instance of the ClassInfo’s native object.
3. Properties would gain a “nodefault” marker that suppresses their being loaded during loadFromXml() only when this function is called for the default object.
4. DVAR would no longer store the default property value anywhere other than the C++ class member variable initialization mechanism. Properties would lose their defaultValue concept.
5. Object::serialize() would consult the value of each property for the ClassInfo’s default property in order to see whether it has the default value (rather than consulting the property’s now-removed hasDefaultValue() function).
6. All objects would gain a special “inert” form of their constructor, which trivially does nothing. “Inert” (“default”) objects are pointed to by nothing except their owner ClassInfo.

Enforcing this last rule is the hard part, but with the proper application of step #3, I think the solution could be workable.

## XmlElement elements are normalized and compared to per-object output

The most “hacky” solution may nonetheless have merit.

The idea is to embrace XmlElements fully, foregoing C++ representations of default values as in the preceding solution in favor of retaining textual representations. There are efficiency issues here—no doubt—but the solution is worth outlining.

In a sense, there is no essential trouble in retaining XmlElements for Pseudoclasses and comparing these elements to the elements produced by a serialized object. The fundamental algorithm would appear in Object::serialize().

For each class property P:

save P to text T.

Compare T to the default textual value for P.

If they differ, save T to the stream.

The trouble is that text formatting is highly flexible. Since “the default textual value for P” might easily have been read from a hand-written XML file, there is no guarantee that two values for T and P that differ are actually distinct.

Therefore there is a need to “normalize” the text for the default value for P. Therefore, when loading a pseudoclass, an additional loading step is needed.

1. Load the pseudoclass’s XmlElement.
2. *“Normalize” element.*

But how do we normalize an element? One way is to load each property implied by the element, then save it out again using the same serialization mechanism to be used by Object::serialize(). But this simple statement obscures a complexity: how do we “load each property implied by the element?” In truth, to do this correctly you must call Object::loadFromXml() into an actual object of the same kind.

This means that this solution, at best, requires the previous solution—the idea of a “default object”—in order to work. And then, if that solution works, this solution will necessarily be more expensive both in space and time, because the XML text for a serialized default object (more precisely, for a series of Pseudoclasses as applied to a default object), will unquestionably be much larger and vastly slower than simply working from the binary properties directly.

Therefore, this solution is an unnecessary and inferior layer over the prior solution.

# Solution: Default Objects

This leads us inexorably to the conclusion that the solution of “default objects” is the best (indeed, only reasonable) solution I now see.

I will attempt an implementation.

## Creation of Default Objects

Every non-abstract Object class now offers a static method StaticGetDefaultObject() which creates an “inert” object (via a special constructor) and returns it. This function is called during the construction of ClassInfos during global initialization time, so default objects are effectively created then. This refactor involved a large amount (but not too exhaustive) of refactoring to ensure that classes avoid excessive use of initializer lists, thus making more extensive use of non-static member data initializers and DVAR. It also uncovered all sorts of places where construction or destruction was non-trivial—though none of these proved terribly troublesome. Singletons are perhaps the most awkward case, since now you necessarily have two copies of objects like e.g. Renderer. But the inert one costs nothing more than a relatively cheap constructor and roughly the sizeof() cost of the object in memory.

## Creation of Default Objects for Pseudoclasses

Pseudoclasses will extend the default object concept. They will create an inert object of the same class as their base native class, then loadFromXml() on it with the pseudoclass chain to give it its “shaped” form. This function will have a bool option to suppress the loading of properties that are marked as not-to-be-loaded-for-defaults—of which DisplayObjectContainer::m\_children is the example *par excellence.*

## Ubiquitous Non-Defaults

Some property values, while not being strictly “default” according to the proper definition, are nonetheless so universal that their appearance in serialized data is unnecessary, unwanted, and potentially harmful (in that they resist later changes). Basically, the definition of “default” as enforced by the system, while correct, is too strict to encompass certain types of ubiquitous values. A good example is the default shader for DisplayObjects, which virtually all DisplayObjects share and which isn’t established (and cannot be established) when the default object is created.

The system therefore needs to be able to change default values to the “ubiquitous” form when that is prudent. One natural way to do this would seem to be to use object constructors to check whether their default object has been “doctored”, and if it hasn’t, to doctor it accordingly.

(An alternative approach would be to push back the creation of default objects until the first creation of an instance of the class to which they belong, which is similar in effect to the “doctoring” approach, but which would make constructors in general (1) slightly more expensive and (2) potentially “hitching”.)

A difficulty here is the need to propagate “doctoring” *up* through the class hierarchy. If a DisplayObject marks its default pShaderProgram, it should mark it for all DisplayObject subclasses *except* for those that also mark the pShaderProgram. **TODO**

## Accessing Default Properties

An object can access its default object in a class-specific way by calling StaticGetDefaultObject(). It can access its default object in a class-independent way by calling StaticGetClassInfo().getDefaultObject() (which returns an Object\*; this is null for abstract classes).

When saving an object (in Object::serialize()), the object will ask its property to compare in a type-specific way whether it has the same value as the default object (also passed in). Therefore PropertyAbstract will need a new pure virtual function bool areObjectValuesEqual( const Object\* a, const Object\* b ), which is overridden for type-specificness in the templated version.