A Note on Display Tree Fixup and Notification

Deciding where to execute “late construction” functionality in Fresh has been a longstanding sore spot. (Unreal and Flash both have similar problems, but Fresh is even weaker in this regard.) This document aims to fix these problems.

# Parent Assignment and Notification (onAddedToParent())

The core of the problem essentially lies in a convenience. DisplayObjects are allowed by the rules of the XML-based loading system to neglect to indicate their parent (and stage—but I’m ignoring this question for the moment). This means that during manifest loading, a DisplayObject may be “malformed” in the precise sense that its parent owns it in its child list but the child has no pointer to its parent.

Up til now, the remedy to this problem was:

1. In DisplayObjectContainer::postLoad()…
2. For all children…
3. If the child does not know that I’m its parent…
4. Then set me as the parent, which…
5. Calls the child’s onAddedToParent() function.

The problem with this solution is precisely that the semantics of onAddedToParent() were vague and chaotic. An object wishing to use that notification to perform late construction steps involving, for example, its siblings or its own descendants might find that they had not yet been informed of their parents, and were thus malformed.

I now propose a new solution. The solution is to decouple two actions of:

1. Setting a child’s parent…
2. Calling onAddedToParent()

…with the goal of ensuring clear semantics for onAddedToParent(). Specifically, when onAddedToParent(), you can be assured that:

1. Your parent’s entire subtree is well-formed.
2. All objects “above” you in the Z-order (i.e. siblings later in the list and your own descendants) have already received an onAddedToParent() call.

Enforcing these promises requires some changes to the implementation of tree fixup.

The postLoad() function is not terribly useful for an efficient stage fixup because of its property that it can be called on objects in any order. This means that a “deep” ancestor may get called first, and then a great-grandchild, and then a middle child. Verification of whether the parent has already been set could perhaps enable this to be a correct algorithm, but a highly redundant one. Therefore, postLoad() can be reliably and efficiently used only to set each child’s parent pointer.

1. In DisplayObjectContainer::postLoad()…
2. For all children, set me as the parent.

This leaves the job of calling onAddedToParent() in a front-to-back (in the Painter’s algorithm sense) order (depth first recursion, children traversed in reverse order). Stage::onStageLoaded() recommends itself as the launching off point of this task.

1. In Stage::onStageLoaded(), recursively call DisplayObject::propagateParentNotification() for every child (in reverse order), having copied the child container temporarily for traversal purposes.
2. DisplayObject::propagate…() simply calls onAddedToParent().
3. DisplayObjectContainer::propagate…() calls propagateParentNotification() for each (copied) child in reverse order, then calls the Super version of the function.

A complication emerges when we consider that objects may use the onAddedToParent() function to create and add new subtrees anywhere in their parent’s subtree. How does this solution fare?

Consider, for example, the case where an object creates a subtree and then adds it as a child of a “higher” (in the Z-order) sibling. Well, adding a child in this way involves using addChild\*(). addChild\*() will ensure that the subtree root is treated correctly, because it correctly forms the parent-child relationship by its very nature. And if the parent is well-formed, then there is no problem.

What if we add the subtree to a parent that is not well-formed, or which has malformed children—as for example if we add it to our own parent (a common case)? Well, the subtree root is no problem—it will receive an onAddedToParent() notification and will be well-formed along with the whole tree. The traversal will also be unharmed. That’s because the parent is already in mid-traversal of *a copy of* its children, and so the new child, already notified and well-formed, will not receive a redundant propagateParentNotification() call.

What if we add it to a “later” (i.e. “lower down”) sibling (or descendant thereof)? Here we have a problem. The lower down sibling will receive its propagateParentNotification() call sometime after we’re done. It will then attempt to propagate this call to its descendants. But the well-formed and informed subtree has already received this notification.

Therefore, we will need to add a bool marker to all DisplayObjects causing them to skip the onAddedToParent() call in propagateParentNotification() if they have already received this call from any source.

# Stage Assignment and Notification (onAddedToStage())

Stage assignment and notification has similar issues to parent notification, except that assignment is more indirect because the stage must be propagated through the whole tree.

Stage assignment and notification are complex event under normal circumstances (that is, when not loading a manifest). Consider the case where an elaborate subtree is added to the stage. It’s easy for the root to receive the stage, but it must then propagate the stage through to its descendants.

A requirement that I am not yet enforcing is that *the entire tree should be well-formed before any call to onAddedToStage().* In this case, “well-formed” now refers to an object’s m\_pStage pointer. If the object in fact has a direct line of ancestry back to a stage, but has a null stage pointer, then it is malformed.

Because of the high level of assurance provided by onAddedToStage(), it is the most convenient place for locating “late construction” code. Objects can, theoretically, do anything here: add event listeners on the stage itself, add subtrees anywhere in the tree, etc. Therefore it must behave very reliably and enforce this assurance (that all objects in the tree have stage pointers prior to any onAddedToStage() call) without question.

A similar two-pass solution as used in parent assignment and notification will form the framework for stage assignment and notification.

In Stage::onStageLoaded(), after parent notification, we first assign the stage pointer everywhere.

1. Call DisplayObject::propagateStage() for all children (with a copy, reverse order).
2. DisplayObject::propagateStage() simply assigns the pointer.
3. DisplayObjectContainer::propagateStage() recurses to all children with a copy in reverse order, then calls the Super version.

Now we call onAddedToStage() with DisplayObject::propagateStageNotification(), with identical semantics to propagateParentNotification() above.

In the dynamic case (i.e. when not loading), whenever a child is added (via addChild()), if the parent has a stage, then propagateStage() and propagateStageNotification() are called on the newly added child—one after another. This ensures that not only the child but the whole subtree is both assigned and notified under the correct circumstances.

# Addendum

You know, a lot of these problems would be greatly simplified if m\_stageRootPosition would be implemented more sensibly, so that the stage didn’t have to be referenced so often.