Delaying Element Initialization for Collapsed Controls

by R. Stacy Smyth

The Problem

Many of the time delays associated with getting a control from XAML to the screen are automatically eliminated for collapsed controls: most of the bindings are never evaluated, the measure and arrange passes never occur, and of course rendering does not occur.

The delays that remain for collapsed controls are the delays associated with initialization: InitializeComponent() is called at the top of each control constructor, and this results in the XAML for that control being loaded, parsed, and used to construct the visual tree of the control, whether that control will ever be displayed or not. What can be even more costly is that InitializeComponent() connects the event handlers specified in the XAML for the control to the appropriate events, and this results in the event handlers for the Initialized and Loaded events being called as well — once again, before it has ever been determined if the control will ever be made visible.

The objective of this paper, and the accompanying project which implements a solution, is to find a way to eliminate these time delays in the case of collapsed controls.

There are two parts to the problem:

1. Specifying delayed initialization: using XAML to indicate which specific controls should have their initialization delayed.
2. Implementing delayed initialization: giving control classes the ability to delay the initialization of the controls in those classes.

(This ordering may seem unusual. When adding a new capability, we're usually concerned first about how to make it work at all, then interested in how to turn it on and off as almost an afterthought with several standard solutions. But this problem has some unique difficulties. You'll see why below.)

Specifying Delayed Initialization

We don't want our solution to cause sweeping changes all over the program by causing delayed initialization *everywhere*. Changing how and when control initialization is performed is fairly deep stuff, and we need to be able to turn on this ability on in one place at a time so that we can evaluate the results. This means that each class needs to *default* to performing as it always has — by calling InitializeComponent immediately — but we need some sort of property or setting to tell the some of the controls to wait until visibility has been evaluated.

Furthermore, we need to add this ability to *lots* of different control classes, and we don't want to add code to all of those different classes unless we absolutely have to.

This combination of requirements makes attached behaviors sound like the perfect solution — an attached behavior would both add the property for saying we want to delay initialization, and change the behavior of the controls without directly adding code to the control classes.

Unfortunately, the attached behavior solution — at least in its purest form — immediately hits a couple of insurmountable obstacles:

1. The call to InitializeComponent() is hard-coded at the beginning of each constructor, before any events associated with the object are raised, so there is categorically nothing that we can do with clever event handling (the heart of attached behaviors) that can keep InitializeComponent() from being called.

2. Inside InitializeComponent() a field named _contentLoaded determines whether the method body is actually executed. This is a private bool, so once again there is nothing we can do from an attached behavior to hijack this field so as to prevent InitializeComponent() from executing.

As such, we already know that the solution is going to require at least minimal coding changes to the classes for which we would like to enable delayed initialization — we just can't do it all from attached behaviors, which rely on event handling.

All right then, if we're going to make this feature functional by actually adding code to each enabled class (we'll come back to that later — it'll turn out to be a single line of code), the question then becomes what mechanism we should use to specify which specific control instances we want to have delay their initialization.

We definitely want to be able to use XAML to indicate which controls to delay initialization for, so that leaves us with either adding a dependency property to each class, or using an attached property.

At this point we hit another, quite major snag: we cannot evaluate an attached property at the top of the control constructor to decide whether to do immediate or delayed initialization, because at the time the constructor is executed, these properties haven't been evaluated and assigned yet! The same goes for regular dependency properties that are part of the class definition of the control — their values aren't assigned to the control until after the constructor has finished executing. Which seems to leave us completely stuck.

My solution to this part of the problem works, but I won't claim it's pretty, and I welcome suggestions to improve it. Namely, we don't attach the property indicating delayed initialization to the control we are interested in, but rather to the control's parent (or other ancestor) — because the parent's properties will already have been evaluated and assigned by the time the child controls that we're interested in have their constructors called.

To specify which child control(s) we would like to have delayed initialization for, the obvious thing to do is to set the value of the attached property to the name (or names) of the control(s) we would like to affect. But this, too, has a problem: it's useless to set the name of the control we want to affect as the value of the attached property, because — you guessed it — the "name" property of the child control hasn't been assigned yet at the time the constructor is executed. So the control we would like to affect doesn't know its own name by the time we would like to use its name to make the decision of whether we will delay initialization.

And it gets worse: The visual tree for the child control hasn't been constructed yet (avoiding doing this is one of our big objectives), much less connected to the overall logical tree, so the "Parent" property is not yet available. We can't check the parent to see if it has an attached property set, because there is no way to find out who our intended parent is.

So what good does it do to set an attached property on the parent of the control if we can't *find* the parent to check the property? Well, there is a way.

Specifically, we create an OnChanged handler for the attached property on the parent and use it to set a static field indicating that delayed initialization is desired for whatever is being constructed.

Wait, though: that sounds like a total failure. We've just permanently turned on delayed initialization for whatever is constructed after the element with the attached property, right? And that's not what we wanted at all.

We correct this by turning our attached property into an attached behavior: we have our OnChanged handler for the attached property add an event handler for the "Initialized" event of the element to which the property is attached. This handler resets the static field to indicate we no longer want delayed initialization. This has the effect of turning on the static "do delayed initialization if possible" flag while the descendants of the propertied element are being constructed, and turning it back off afterwards. That way, the descendants of the ancestor element are the only controls affected.

Affecting all of the delay-capable descendants of the propertied element is better than nothing, but it still seems a bit scattershot — there may be child controls for which we want delayed initialization and child controls for which we don't. There's one more thing we can easily do to tighten our focus: the child controls being constructed don't have names yet, but they do already have types. Instead of just setting the property (I'll call it “SkipType”) to "True", indicating a desire for delayed initialization on all descendants, we set SkipType to the Type of the control we want to affect — or to System.Object if we want to affect every descendant that is capable of delayed initialization. If we grant that the number of delayable control classes within a parent element is going to be small, this gives us the level of control necessary to turn this feature on and off with adequate precision.

If we ever run into a situation where we want to turn on delayed initialization for one control but off for a direct sibling that is of the same class (unlikely), we could always wrap the control that we want to delay in a 0-thickness Border and set the attached property on that.

So now, at last, we have a way of indicating to a control whether we want it to delay initialization until after it has determined whether it is visible.

Implementing Delayed Initialization

Making this feature *work* is actually simpler than specifying which controls we want to have it apply to. In one way or another, we want the following logic to be performed at the top of our control constructors:

If the control isn't of the type that we want to delay initialization for, we simply call InitializeComponent() on the control. If the control is of the desired class, though, we set an event handler for IsVisibleChanged, and call InitializeComponent() from there. InitializeComponent(), in turn, parses the XAML, builds the visual tree for the control, and wires up the control's event handlers. The IsVisibleChanged handler won't be called until the visibility of the control is changed, and for collapsed controls that's going to be never. Which is the whole point.

There's one more thing that IsVisibleChanged needs to do: in between when the control is constructed and when IsVisibleChanged is called, the framework goes ahead and raises the Initialized and Loaded events for the control, even though the control has no visual tree and certainly is neither initialized nor loaded. Fortunately for us, this does not result in the handlers for these events being prematurely called: the usual place for these handlers to be specified is in the XAML for the control, and of course this XAML won't have been parsed before our IsVisibleChanged is called!

So we've dodged having these handlers be incorrectly and prematurely called by the framework. But we still need to call them from IsVisibleChanged, now that the control has actually been initialized and loaded.

Unfortunately, there is no way to raise these events from within the IsVisibleChanged handler — they can only be raised from within the control class, and we'd really like for all of this code we've been talking about to not be copied into each class that requires delayed initialization.

We solve the "where do we put the code" problem by moving all of this additional code into a static class called InitializationOptions — that's where we put the static field we mentioned earlier,too — and we do both the wiring up of IsVisibleChanged and the decision-making about delayed initialization though a call to InitializationOptions.Initialize(), which we call from the constructor of our control class instead of calling InitializeComponent.

We solve the "how do we raise Loaded and Initialized" problem by overloading InitializationOptions.Initialize() so that it can accept two additional parameters if desired — the handlers for the Initialized and Loaded events on the control. We can't raise the events these handlers are associated with, but there's nothing to keep InitializationOptions.IsVisibleChanged from calling the handlers directly, right after calling InitializeComponent().

The only additional wrinkle is that the InitializationOptions class needs some way to keep track of these handlers in between the call to Initialize() and the call to IsVisibleChanged(). We do this by having Initialize() set a pair of attached properties on the control itself.

And that wraps up how to defer initialization until a control is actually displayed, or is at least changed to something other than collapsed — we need to call InitializeComponent() for Hidden controls too, since we need sizing information from them.

Effectiveness

The sample program that accompanies this document has two modes, selectable by changing which visual tree is commented in/out in MainWindow.xaml.

In "Functionality Test Mode", the app uses delayed initialization for a single custom control. This is useful for stepping through the code in the debugger and understanding what's going on.

In "Speed Test" mode, the app uses delayed initialization to create a substantial block of Collapsed, non-trivial custom controls.

Both modes display a dialog showing how many milliseconds were spent during loading, and in both modes it is possible to comment out the attached property specifying delayed initialization, thereby allowing a comparison of load times.

The result of multiple timing runs on my particular system was that creating the window with no children in the StackPanel took 46 ms; creating the block of collapsed controls using standard initalization took 187 ms; and creating the block of collapsed controls using delayed initialization took 62 ms. If you subtract out the time to create an empty window (46 ms), that leaves 141 ms dedicated to creating the controls the normal way, and 16 ms dedicated to creating the controls using deferred initialization.

In other words, creating this block of collapsed controls using delayed initialization was approximately ***Nine Times as Fast*** as conventional initialization.

This example does not have any code in the handlers for the Initialized and Loaded events of the sample control class. If the control did any work in the handlers for Initialized or Loaded, **all** of that work would be done using standard initialization, and **none** of it would be done using delayed initalization. In other words, the difference in speed would expand from 9-to-1 to being arbitrarily large, depending on the work done in the two event handlers.

Limitations

As cool as this system is, there are several situations it can't handle:

1. Binding Visibility with a relative source. RelativeSource requires a logical tree to be relative to — and a control using delayed initalization doesn't get plugged into the logical tree until after visibility is evaluated. So this isn't going to work.
2. "OnChanged" handlers for properties that are set from XAML outside of the control, before the control is made visible. Setting the properties is fine, but until the control is shown, it doesn't parse its XAML, and it doesn't wire up its event handlers. If you need this functionality, set up the event handlers in the constructor for the control — that way they won't be dependent on when the XAML for the control gets loaded.
3. Doing stuff in the constructor that relies on the visual tree that we're not building because we're not calling InitializeComponent. Any such code needs to get moved to a handler for Initialized or Loaded.
4. Non-custom controls. Since you need to add code (one line) to the affected control classes, this technique is only for custom control classes, not the pre-defined, framework-provided classes.

The Sample Program

- Nearly all of the meat of how delayed initialization works is in InitializationOptions.cs.
- The code for the sample control that uses delayed initialization is in SimpleControl.xaml and SimpleControl.cs.
- To change the mode in which the sample program operates, read the comments in MainWindow.xaml and follow the instructions. You can run the program with or without delayed initialization, and with or without enough delayed-initialization controls to make a perceptible difference in speed.

How to Use Delayed Initialization in Your Own Code

If you want to modify a class so that it can use delayed initialization, do the following:

1. Add the InitializationOptions.cs file from my sample project to your project.

2. Replace the line in the constructor that reads

```
InitializeComponent();
```

with the line

```
InitializationOptions.Initialize(this
[,MyControl_Initialized][,MyControl_Loaded]);
```

where MyControl_Initialized and MyControl_Loaded are optional parameters specifying the control's handlers for the Initialized and Loaded events, respectively.

3. In the XAML that creates the control, on an *ancestor* of the control — **not on the control itself** — set the following property:

```
initopt:InitializationOptions.SkipType="{x:Type sys:Object}"
```

By default, this will turn on delayed initialization for all delayed-initialization-capable descendants of the control with this property. If you want to turn on delayed initialization for only the control of one class, specify that class instead of sys:Object.