# Optimizing ListView Item Rendering

The previous post on this blog entitled All About WinJS Promises explored the relationships that are inherent in promises, bringing us to the point where we could decipher some of the more mysterious pieces of promise-heavy code. To delve into the subject more deeply, the [HTML ListView optimizing performance sample](http://code.msdn.microsoft.com/windowsapps/ListView-performance-39fb71f0) (specifically scenario 1), provides a number of additional cases of intricately interrelated promises. In this post, then, we’ll look at the different stages of using item rendering functions to optimize ListView performance, and in the process deepen our understanding of promises still further.

To be clear, item rendering in a ListView control happens through either a declarative template defined in HTML or through a custom JavaScript function, where the latter is called for every item in the list that needs to be rendered. The benefit of using a rendering function is that it lets you customize the rendering on a per-item basis and enables various optimizations which you cannot achieve with a declarative template.

## Basic Renderers

You might expect that an item rendering function would just be given an item from the ListView’s data source, and in response it would just create the HTML elements needed for that particular item and return the root element. This is essentially what happens, but there are two added considerations. First, the item data itself might be loaded asynchronously, so it makes sense to tie element creation to the availability of that data. Furthermore, the process of rendering the item itself might involve other asynchronous work, such as loading images from remote URIs or reading data in other files identified in the item data. The different levels of optimization, in fact, allow for an arbitrary amount of async work between the request of the item’s elements and actual delivery of those elements.

So instead of giving the rendering function the item data directly, the ListView supplies a promise for that data. And instead of the function returning the item’s root element, it returns a promise for that element. This is what we see in the **simpleRenderer** function of the sample:

function simpleRenderer(itemPromise) {  
 return itemPromise.then(function (item) {  
 var element = document.createElement("div");  
 element.className = "itemTempl";  
 element.innerHTML = "<img src='" + item.data.thumbnail +  
 "' alt='Databound image' /><div class='content'>" + item.data.title + "</div>";  
 return element;  
 });  
}

This code attaches a completed handler to the **itemPromise**, so that handler is invoked when the item data is available, creating the elements in response. The return value—the promise from **itemPromise.then**—will be fulfilled when that completed handler returns, and its fulfillment value is **element**.

With this, the ListView can join many item rendering promises together if it needs to wait until a whole page of items has been built up. It does this, in fact, to intelligently manage how it builds up different pages, building up the page of visible items first, then two offscreen pages forward and back where users are most likely to pan to next. In addition, having all these promises in place means that the ListView can easily cancel the rendering of unfinished items if the user pans away and avoid unnecessary element creation.

If the rendering function itself needs to perform additional async work, then it can just chain those intermediate promises together, returning the promise from the last **then** in the chain. For example:

function someRenderer(itemPromise) {  
 return itemPromise.then(function (item) {  
 return doSomeWorkAsync(item.data);  
 }).then(function (results) {  
 return doMoreWorkAsync(results1);  
 }).then(function (results2) {  
 var element = document.createElement("div");  
 // Use results2 to configure the element  
 return element;  
 });  
}

## Placeholder Renderers

The next stage of ListView optimization involves what’s called a *placeholder renderer*, which separates building up the element into two stages. Here, the renderer returns an object that contains two properties:

* **element**  The top-level element in the item’s structure that’s enough to define its size and shape and is not dependent on the item data. This allows the ListView to quickly determine how those items are laid out in the list without having to build up all the child elements.
* **renderComplete** A promise that’s fulfilled when the remainder of the element’s contents are constructed, that is, returns the promise from whatever chain you have starting with **itemPromise.then** as before. The completed handler in this case, however, need not return a promise if no async work is involved.

All that’s happening here is that the item construction process allows the ListView to ask for the outer placeholder separately from the full item. The ListView is smart enough to check whether your renderer returns a promise (the basic case as before) or an object with **element** and **renderComplete** properties (more advanced cases). Thus the equivalent placeholder renderer for the previous **simpleRenderer** is as follows (this is also in the sample):

function placeholderRenderer(itemPromise) {  
 // create a basic template for the item which doesn't depend on the data  
 var element = document.createElement("div");  
 element.className = "itemTempl";  
 element.innerHTML = "<div class='content'>...</div>";  
  
 // return the element as the placeholder, and a callback to update it when data is available  
 return {  
 element: element,  
  
 // specifies a promise that will be completed when rendering is complete  
 // itemPromise will complete when the data is available  
 renderComplete: itemPromise.then(function (item) {  
 // mutate the element to include the data  
 element.querySelector(".content").innerText = item.data.title;  
 element.insertAdjacentHTML("afterBegin", "<img src='" +  
 item.data.thumbnail + "' alt='Databound image' />");  
 })  
 };  
}

Note that the **element.innerHTML** assignment can even be moved inside **renderComplete** because the *itemTempl* class in css/scenario1.css specifies the width and height of the item directly. The reason why it’s included in the **element** property is because it provides the default “…” text in the placeholder. You could just as easily use an **img** element that refers to a small in-package resource that’s shared across all the items (and thus renders quickly).

The next stage now, the *recycling placeholder* renderer, doesn’t add anything new where promises are concerned. It simply adds awareness of a second parameter called **recycled** that the ListView can provide to your rendering function when the ListView’s **loadingBehavior** is set to "randomaccess". If **recycled** is given, you can just clean out the element, return it as the placeholder, and then fill in the data values within the **renderComplete** promise as before. If it’s not provided (as when the ListView is first created or when **loadingBehavior** is "incremental"), you’ll create the element anew. Here’s the code from the sample for that variation:

function recyclingPlaceholderRenderer(itemPromise, recycled) {  
 var element, img, label;  
 if (!recycled) {  
 // create a basic template for the item which doesn't depend on the data  
 element = document.createElement("div");  
 element.className = "itemTempl";  
 element.innerHTML = "<img alt='Databound image' style='visibility:hidden;'/>" +  
 "<div class='content'>...</div>";  
 }  
 else {  
 // clean up the recycled element so that we can re-use it   
 element = recycled;  
 label = element.querySelector(".content");  
 label.innerHTML = "...";  
 img = element.querySelector("img");  
 img.style.visibility = "hidden";  
 }  
 return {  
 element: element,  
 renderComplete: itemPromise.then(function (item) {  
 // mutate the element to include the data  
 if (!label) {  
 label = element.querySelector(".content");  
 img = element.querySelector("img");  
 }  
 label.innerText = item.data.title;  
 img.src = item.data.thumbnail;  
 img.style.visibility = "visible";  
 })  
 };  
}

In **renderComplete**, be sure to check for the existence of elements that you don’t create for a new placeholder, such as **label**, and create them here if needed.

If you’d like to clean out recycled items, you can also provide a function to the ListView’s **resetItem** property that would contain the same code as shown above for that case. The same is true for the **resetGroupHeader** property, because you can use template functions for group headers as well as items. We haven’t spoken of these as much because group headers are far fewer and don’t typically have the same performance implications. Nevertheless, the capability is there.

## Multistage Renderers

This brings us to the penultimate optimization, the *multistage renderer*, which extends the recycling placeholder renderer to delay-load images and other media until the rest of the item is wholly present in the DOM. It also delays effects like animations until the item is actually on screen. All of this involves more async operations, so you can expect there are more promises involved.

The hooks for this are provided as members on the **item** result from **itemPromise**: a property called **ready** (a promise) and two methods, **loadImage** and **isOnScreen**, both of which return promises. That is, you’ll find these members on the item that’s passed to your first completed handler:

renderComplete: itemPromise.then(function (item) {  
 // item.ready, item.loadImage, and item.isOnScreen available  
 })

Here’s how you use them:

* **ready** Return this promise from the first completed handler in your chain. This promise is fulfilled when the full structure of the element has been rendered and is visible. This means you can chain another **then** with a completed handler in which you do other post-visibility work like loading images.
* **loadImage** Downloads an image from a URI and displays it in the given **img** element, returning a promise that’s fulfilled with . You attach a completed handler to this promise, which itself returns the promise from **isOnScreen**.
* **isOnScreen** Returns a promise whose fulfillment value is a Boolean indicating whether the item is visible or not. In present implementations, this is a known value so the promise is fulfilled synchronously. By wrapping it in a promise, though, it can be used in an overall chain.

We see all this in the sample’s **multistageRenderer** function, where completion of the image load is used to start a fade-in animation. Here I’m just showing what’s returned from the **renderComplete** promise:

renderComplete: itemPromise.then(function (item) {  
 // mutate the element to update only the title  
 if (!label) { label = element.querySelector(".content"); }  
 label.innerText = item.data.title;  
  
 // use the item.ready promise to delay the more expensive work  
 return item.ready;  
 // use the ability to chain promises, to enable work to be cancelled  
 }).then(function (item) {  
 //use the image loader to queue the loading of the image  
 if (!img) { img = element.querySelector("img"); }  
 return item.loadImage(item.data.thumbnail, img).then(function () {  
 //once loaded check if the item is visible  
 return item.isOnScreen();  
 });  
 }).then(function (onscreen) {  
 if (!onscreen) {  
 //if the item is not visible, then don't animate its opacity  
 img.style.opacity = 1;  
 } else {  
 //if the item is visible then animate the opacity of the image  
 WinJS.UI.Animation.fadeIn(img);  
 }  
 })

Even though there’s a lot going on, we still just have a basic promise chain here. The first async operation in the renderer updates simple parts of the item element, such as text. It then returns the promise in **item.ready**. When that promise is fulfilled—or, more accurately, if that promise is fulfilled—you can use the item’s async **loadImage** method to kick off an image download, returning the **item.isOnScreen** promise from that completed handler, such that the **onscreen** visibility flag gets to the final completed handler in the chain. When and if that **isOnScreen** promise is fulfilled, you can perform those operations that are relevant only to a visible item.

I emphasize the if part because it’s very likely that the user will be panning around within the ListView while all this is happening. Having all these promises chained together again makes it possible for the ListView to cancel the async operations any time these items are scrolled out of view and/or off any buffered pages. Suffice it to say that the ListView control has gone through a *lot* of performance testing!

It’s also important to remind ourselves that we’re using then throughout all these chains because we’re still returning a promise from the rendering function within the **renderComplete** property. We’re never the end of the chain in these renderers, so we’ll never use **done** at the end.

## Thumbnail Batching

The last optimization is truly the *coup de grace* for the ListView control. In the function called **batchRenderer** we find this structure for **renderComplete** (most code omitted):

renderComplete: itemPromise.then(function (i) {  
 item = i;  
 // ...  
 return item.ready;  
 }).then(function () {  
 return item.loadImage(item.data.thumbnail);  
 }).then(thumbnailBatch()  
 ).then(function (newimg) {  
 img = newimg;  
 element.insertBefore(img, element.firstElementChild);  
 return item.isOnScreen();  
 }).then(function (onscreen) {  
 //...  
 })

This is almost the same as the **multistageRenderer** except for the insertion of this call to some function called **thumbnailBatch** between the **item.loadImage** call and the **item.isOnScreen** check, and where it’s placed in the promise chain indicates that the return value of this function is a completed handler that itself returns another promise.

Confused? Well, we’ll get to the bottom of it! But we first need a little more background as to what we’re trying to accomplish.

If we just had a ListView with a single item, various loading optimizations wouldn’t be noticeable. But ListViews typically have many items, and the rendering function is called for each one. In the **multistageRenderer** of the previous section, the rendering of each item kicks off an async **item.loadImage** operation to download its thumbnail from an arbitrary URI, and each operation can take an arbitrary amount of time. So for the list as a whole, we have a bunch of simultaneous **loadImage** calls going on, with the rendering of each item waiting on the completion of its particular thumbnail. This is not a problem in and of itself.

An important characteristic that’s not visible in **multistageRenderer**, however, is that the **img** element for the thumbnail is *already* in the DOM, and the **loadImage** function will set that image’s **src** attribute as soon as the download has finished, a fact that’s hidden from view. This in turn triggers an update in the rendering engine as soon as we return from the rest of the promise chain, which is essentially synchronous after this point.

It’s possible, then, to see a bunch of thumbnails coming back within a short amount of time. This will cause excess churn in the rendering engine that results in poor visual performance. To improve this, we want to add those **img** elements to the DOM in batches, thereby combining them together in a single rendering pass.

What accomplishes this is some promise magic a piece of code in a function called **createBatch**, which is structured to be easily copied into other projects. **createBatch** is called just once for the whole app, and its result—another function—is stored in the variable named **thumbnailBatch**:

var thumbnailBatch;  
thumbnailBatch = createBatch();

A call to this **thumbnailBatch** function, as I’ll refer to it from here on, is again inserted into the promise chain of the renderer. This purpose of this insertion, given the nature of the batching code as we’ll see shortly, is to collects the loaded images into batches, releasing them for further processing at suitable intervals. A call to **thumbnailBatch()** again returns a completed handler of some kind, where that handler must return a promise whose fulfillment value is an **img** element that can *then* be added to the DOM. By adding those images to the DOM *after* the batching has taken place, we combine that whole group into the same rendering pass.

This is an important difference between the **batchRenderer** and the previous **multistageRenderer**: in the latter, then thumbnail’s **img** element already exists in the DOM and is passed to **loadImage** as the second parameter. So when **loadImage** sets the image’s **src** attribute, a rendering update is triggered. Within **batchRenderer**, however, that **img** element is separately created within **loadImage** (where **src** is also set), but the **img** is *not* *yet in the DOM*. It’s only added to the DOM after the **thumbnailBatch** step completes, which means it’s done as part of a group within that single layout pass.

So now let’s see how that batching works. Here’s the **createBatch** function in its entirety:

function createBatch(waitPeriod) {

var batchTimeout = WinJS.Promise.as();

var batchedItems = [];

function completeBatch() {

var callbacks = batchedItems;

batchedItems = [];

for (var i = 0; i < callbacks.length; i++) {

callbacks[i]();

}

}

return function () {

batchTimeout.cancel();

batchTimeout = WinJS.Promise.timeout(waitPeriod || 64).then(completeBatch);

var delayedPromise = new WinJS.Promise(function (c) {

batchedItems.push(c);

});

return function (v) {

return delayedPromise.then(function () {

return v;

});

};

};

}

Again, **createBatch** is called just *once* and **thumbnailBatch** is called for *every item in the list*, returning a completed handler that itself returns a promise.

Such a completed handler might just as easily have been inserted directly into the rendering function, but what we’re trying to do here is coordinate activities *across multiple items* rather than just on a per-item basis. This coordination is achieved through the two variables created and initialized at the beginning of **createBatch**: **batchedTimeout**, initialized as an empty promise, and **batchedItems**, initialized an array of functions that’s initially empty. **createBatch** also declares a function, **completeBatch**, that simply empties **batchedItems**, calling each function in the array:

function completeBatch() {

//Copy and clear the array so that the next batch can start to accumulate

//while we're processing the previous one.

var callbacks = batchedItems;

batchedItems = [];

for (var i = 0; i < callbacks.length; i++) {

callbacks[i]();

}

}

Now let’s see what happens within **thumbnailBatch** (the function returned from **createBatch**), which is again called for each item being rendered. First, we *cancel* any existing **batchedTimeout** and immediately recreate it.

batchTimeout.cancel();

batchTimeout = WinJS.Promise.timeout(waitPeriod || 64).then(completeBatch);

The second line shows the future delivery/fulfillment pattern discussed in the All About WinJS Promises post <TODO: link>: it says to call **completeBatch** after a delay of **waitPeriod** milliseconds (with a default of 64ms). This means that so long as **thumbnailBatch** is being called again within **waitPeriod** of a previous call, **batchTimeout** will be reset to another **waitPeriod**. And because **thumbnailBatch** is only called *after* an **item.loadImage** call completes, we’re effectively saying that any **loadImage** operations that complete within **waitPeriod** of the previous one will be included in the same batch. When there’s a gap longer than **waitPeriod** the batch is processed (images are added to the DOM) and the next batch begins.

After handling this timeout business, the **thumbnailBatch** creates a new promise that simply pushes the complete dispatcher function into **batchedItems**:

var delayedPromise = new WinJS.Promise(function (c) {

batchedItems.push(c);

});

Remember from the previous post that a promise is just a code construct, and that’s all we have here. The newly created promise has no async behavior in and of itself: we’re just adding the complete dispatcher function, **c**, to **batchedItems**. But of course, we don’t do anything with **batchedItems** until **batchedTimeout** completes (asynchronously), so there is in fact an async relationship here: when the timeout happens and we clear the batch (inside **completeBatch**), we’ll invoke any completed handlers given elsewhere to **delayedPromise.then**.

This brings us to the last line of code in **createBatch**, which is the function that **thumbnailBatch** returns. This function is exactly the completed handler that gets inserted into the renderer’s whole promise chain:

return function (v) {

return delayedPromise.then(function () {

return v;

});

};

In fact, let’s put this piece of code directly into the promise chain so we can see the resulting relationships:

return item.loadImage(item.data.thumbnail);  
 }).then(function (v) {  
 return delayedPromise.then(function () {  
 return v;  
 });  
 ).then(function (newimg) {

Now we can see that the argument **v** is the result of **item.loadImage**, which is the **img** element it creates for us. If we didn’t want to do batching, we could just say **return WinJS.Promise.as(v)** and the whole chain would still work: **v** would then be passed on synchronously and show up as **newimg** in the next step.

Instead, though, we’re returning a promise from **delayedPromise.then** which won’t be fulfilled—with **v**—until the current **batchedTimeout** is fulfilled. At that time—when again there’s a gap of **waitPeriod** between **loadImage** completions—those **img** elements are then delivered to the next step in the chain where they’re added to the DOM.

And that’s it!