# Optimizing ListView Item Rendering

For many Windows Store apps written in JavaScript that work with collections, cooperating well with the [WinJS ListView control](http://msdn.microsoft.com/en-us/library/windows/apps/br211837.aspx) is essential for great app performance. This isn’t surprising: when you’re dealing with management and display of potentially thousands of items, every bit of optimization you make with those items really counts. Most important is how each of those items gets rendered, which is to say, how—and when—each item in the ListView control gets built up in the DOM and becomes a visible part of the app. Indeed, the *when* part of this equation becomes a critical factor when a user rapidly pans around within a list and expects that list to keep pace.

Item rendering in a ListView happens through either a declarative template defined in HTML or through a custom JavaScript *rendering function* that’s called for every item in the list. Although using a declarative template is the simplest, it doesn’t provide much latitude for specific control over the process. A rendering function, on the other hand, lets you customize rendering on a per-item basis and enables a number of optimizations that are demonstrated in the [HTML ListView optimizing performance sample](http://code.msdn.microsoft.com/windowsapps/ListView-performance-39fb71f0). Those optimizations are:

* Allowing for asynchronous delivery of item data and a rendered element, as supported by basic rendering functions.
* Separating the creation of an item’s shape, which is necessary for the ListView’s overall layout, from its internal elements. This is supported through a placeholder renderer.
* Reusing a previously-created item element (and its children) by replacing its data, avoiding most of the element-creation steps, provided for through a recycling placeholder renderer.
* Deferring expensive visual operations like image loading and animations until an item is visible and the ListView isn’t being rapidly panned, which is accomplished through a multistage renderer.
* Batching together those same visual operations to minimize re-rendering of the DOM with a multistage batching renderer.

In this post we’ll go through all these stages and learn how they cooperate with the ListView’s item rendering process. As you can imagine, optimizations around the *when* of item rendering involves lots of asynchronous operations and therefore lots of promises. So in the process we’ll also gain us a deeper understanding of promises themselves, building upon the earlier All About Promises <TODO: link> on this blog.

As a general note that applies to all renderers, it’s always important to keep your core item rendering time (not counting deferred operations) to a minimum. Because the ListView’s final performance greatly depends on how well its updates are aligned to screen refresh intervals, a few extra milliseconds spent in an item renderer can bump the ListView’s overall rendering time over the next refresh interval, resulting in dropped frames and visual choppiness. In other words, item renderers are one place where it really does count to optimizes your JavaScript code.

## Basic Renderers

Let’s start with a quick review of what an item rendering function—which I’ll simply call a *renderer*—looks like. A renderer is a function that you assign to the ListView’s [itemTemplate](http://msdn.microsoft.com/en-us/library/windows/apps/hh700705.aspx) property instead of a template name, and that function will be called, as necessary, for items that the ListView wants to include in the DOM. (Basic documentation for renderers, by the way, is found on the [itemTemplate](http://msdn.microsoft.com/en-us/library/windows/apps/hh700705.aspx) page, but it’s the sample that really shows the optimizations.)

You might expect that an item rendering function would just be given an item from the ListView’s data source. It would then create the HTML elements needed for that particular item and return the root element that the ListView can add to the DOM. 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 that we’ll see, 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, again, you can expect that promises are involved! For one, the ListView doesn’t just give the renderer the item data directly, it supplies a promise for that data. And instead of the function directly returning the item’s root element, it returns a promise for that element. This allows the ListView to join many item rendering promises together and wait (asynchronously) until a whole page worth of items has been rendered. It does this, in fact, to intelligently manage how it builds up different pages, first building the page of visible items, 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, thereby avoiding unnecessary element creation.

We can see how these promises are used 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 first attaches a completed handler to the **itemPromise**. The handler is invoked when the item data is available and effectively creates the elements in response. But notice again that we’re not actually returning the element directly—we’re returning a promise that’s fulfilled with that element. That is, the return value from **itemPromise.then()** is a promise that’s fulfilled with *element* if and when the ListView needs it.

Returning a promise allows us to do other asynchronous work if necessary. In this case the renderer 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;  
 });  
}

Note that his is a case where we do *not* use **done** at the end of the chain because we’re returning the promise from the last **then** call. The ListView takes responsibility for handling any errors that might arise.

## Placeholder Renderers

The next stage of ListView optimization uses a *placeholder renderer* that separates building up the element into two stages. This allows the ListView to ask for just those parts of an element that are necessary to define the list’s overall layout, without having to build up all the elements inside each item. As a result, the ListView can complete its layout pass quickly and continue to be highly responsive to further input. It can then ask for the remainder of the element at a later time.

A placeholder renderer returns an object with two properties instead of just a promise:

* *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.
* *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 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). So the equivalent placeholder renderer (in the sample) for the previous **simpleRenderer** is as follows:

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 could be moved inside **renderComplete** because the *itemTempl* class in the sample’s css/scenario1.css file 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).

## Recycling Placeholder Renderers

The next optimization, the *recycling placeholder* renderer, doesn’t add anything new where promises are concerned. It rather adds awareness of a second parameter to the renderer called *recycled*, which is the root element of an item that was previously rendered but is no longer visible. That is, the recycled element already has is child elements in place, so you can simply replace the data and perhaps tweak a few of those elements. This avoids most of the otherwise costly element-creation calls needed for a wholly new item, clearly saves saving quite a bit of time in the rendering process.

The ListView may provide a recycled element when its [**loadingBehavior**](http://msdn.microsoft.com/en-us/library/windows/apps/br211835.aspx) is set to "randomaccess". If *recycled* is given, you can just clean data out the element (and its children), return it as the placeholder, and then fill in data and create any additional children (if necessary) within *renderComplete*. If a recycled element is 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 more generally, note that you can provide a function to the ListView’s [**resetItem**](http://msdn.microsoft.com/en-us/library/windows/apps/br211846.aspx) property. This function would contain similar code to that shown above. The same is true for the [**resetGroupHeader**](http://msdn.microsoft.com/en-us/library/windows/apps/hh700726.aspx) 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*. This 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. This recognizes that users often pan around within a ListView quite rapidly, so it makes sense to asynchronously defer the more expensive operations until the ListView has come to a stable position.

The ListView provides the necessary hooks as members on the **item** result from **itemPromise**: a property called **ready** (a promise) and two methods, **loadImage** and **isOnScreen**, both of which return (more!) promises. That is:

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**. Note that **loadImage** will create an *img* element if one isn’t provided.
* **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 a longer 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 structure, such as text. It then returns the promise in **item.ready**. When that promise is fulfilled—or, more accurately, if that promise is fulfilled—we use the item’s async **loadImage** method to kick off an image download, returning the **item.isOnScreen** promise from that completed handler. This means that the *onscreen* visibility flag gets passed to the final completed handler in the chain. When and if that **isOnScreen** promise is fulfilled—meaning the item is actually visible—we can perform relevant operations like animations.

I emphasize the if part because it’s again 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 whenever these items are panned 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 again 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 mysterious call to a function called **thumbnailBatch** between the **item.loadImage** call and the **item.isOnScreen** check. The placement of **thumbnailBatch** in the chain indicates that its return value must be 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 might have a bunch of simultaneous **loadImage** calls going on, with the rendering of each item waiting on the completion of its particular thumbnail. So far so good.

An important characteristic that’s not at all 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. 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 that point.

It’s possible, then, that a bunch of thumbnails could come back to the UI thread within a short amount of time. This will cause excess churn in the rendering engine and poor visual performance. To avoid this churn, we want to fully create these *img* elements *before* they’re in the DOM, and then add them in batches such that they’re all handled in a single rendering pass.

The sample does this through a piece of promise-magic code a function called **createBatch**. **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 group together a set of loaded *img* elements, releasing them for further processing at suitable intervals. Again, just looking at the promise chain in the renderer, a call to **thumbnailBatch()** must return a completed handler function that returns a promise, and the fulfillment value of that promise (looking at the next step in the chain) must be 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 **multistageRenderer** of the previous section: in the latter, the 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, making it 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 its **thumbnailBatch** result is called for *each* rendered itemin the list. The completed handler that **thumbnailBatch** generates is then called whenever a **loadImage** operation completes.

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 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 the *batchedItems* array:

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

batchedItems.push(c);

});

Remember from All About Promises <TODO: link> 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 the dispatcher 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 lines of code in **createBatch**, which is the function that **thumbnailBatch** actually 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 created 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!

## In Closing

The five different rendering functions demonstrated in the [HTML ListView optimizing performance sample](http://code.msdn.microsoft.com/windowsapps/ListView-performance-39fb71f0) all have one thing in common: they show how the asynchronous relationship between the ListView and the renderer—expressed through promises—allows the renderer tremendous flexibility in how and when it produces elements for items in the list. In writing your own apps, the strategy you use for ListView optimization will greatly depend on the size of your data source, the complexity of the items themselves, and the amount of data that you’re obtaining asynchronously for those items (such as downloading remote images). Clearly, you’ll want to keep item renderers as simple as you can to still meet your performance goals. But in any case, you now have all the tools you need to help the ListView—and your app—perform its best.

Kraig Brockschmidt  
Program Manager, Windows Ecosystem Team  
Author, [*Programming Windows 8 Apps in HTML, CSS, and JavaScript*](http://blogs.msdn.com/b/microsoft_press/archive/2012/10/29/free-ebook-programming-windows-8-apps-with-html-css-and-javascript.aspx)