Microsoft

High-Performance Windows Store Apps

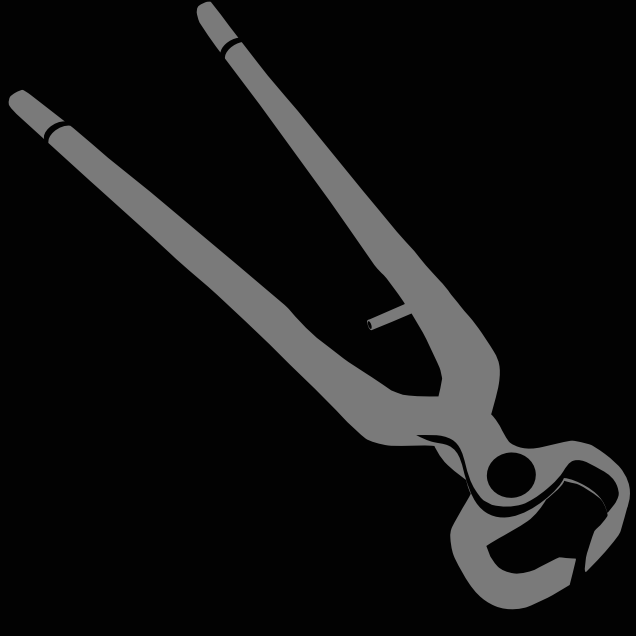
Professional

Brian Rasmussen

Foreword by Eric Lippert,
C# Analysis Architect, Coverity

Microsoft

High-Performance Windows Store Apps



Professional

Brian Rasmussen

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2014935300
ISBN: 978-0-7356-8263-4

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at *http://www.microsoft.com/learning/booksurvey*.

Acquisitions Editor: Devon Musgrave
Developmental Editor: Devon Musgrave
Project Editor: Devon Musgrave
Editorial Production: Christian Holdener, S4Carlisle Publishing Services
Technical Reviewer: Marc Young
Copyeditor: Roger LeBlanc
Indexer: Jean Skipp
Cover: Twist Creative • Seattle and Joel Panchot

To Kirsten for all your love and support. Thank you for going on this adventure with me. I love you!

To Asbjørn and Janus for showing me a new and fantastic view of the world. I love you!

This page intentionally left blank

Contents at a Glance

This page intentionally left blank

Table of Contents

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Foreword

Which is faster, dividing by four or shifting by two?

Should I make this array of 10 numbers shorts *instead of* ints *to save on space?*

What are your favorite tips and tricks for speeding up programs?

I get a lot of questions like these from real-world developers who think they might have performance problems. Though those developers mean well, these are almost always the wrong questions to ask! Why? Because knowing the answers will not actually lead to improvements that any user will notice. A program that downloads and displays a hundred million bytes of video from a server on the other side of the world is not going to suddenly become smooth and fluid because you saved a couple nanoseconds in one math routine or trimmed 20 bytes from an array. Rather, getting good performance is about setting user-focused goals, asking the right questions, using the best possible diagnostic tools to get objective answers, and having the engineering discipline necessary to build a high-performance product every day of the development process.

Brian Rasmussen knows all this inside out. He and I spent many years working together on the Roslyn project at Microsoft: a complete, from-scratch rewrite of the C# and Visual Basic compilers and IDEs. The user-focused performance requirements of this project were daunting: we had to be able to perform an accurate analysis of potentially millions of lines of code in the time between keystrokes, so that the right IntelliSense information could be displayed in the IDE as you type. We were limited to a 32-bit address space shared with the rest of Visual Studio. There is no way we could have achieved our lofty time and memory performance goals without using many of the solid engineering techniques described in this book. With these techniques, you too can succeed in amazing your users with beautiful, fast, fluid, efficient applications.

Good luck, and go fast!

Eric Lippert
Seattle, Washington
March 2014

This page intentionally left blank

Introduction

Performance is both very simple and extremely complex. You don't need to be a ninja programmer or employ cutting-edge tools to spot a performance problem. Detecting performance problems is simple; anyone can identify performance problems just by interacting with an app or a system. If the user feels that your app is too slow, then it is—at least to that particular user. If many users share the experience, you have a legitimate performance problem on your hands. And if the users are not pleased with the performance of your app, they might start looking for a better alternative.

Identifying the reasons for performance problems and subsequently addressing those issues can be mind-numbingly hard. Understanding why performance problems happen can be complex because so many factors are in play. Why is page navigation slow? Why is the movement of the pictures jerky when I pan across the screen? Why does login have to take 10 seconds? Why does this app drain the battery so quickly? To identify the reasons for performance issues, you have to possess detailed knowledge of both the app and the runtime environment and use specialized tools to get the data you need to understand why the app behaves as it does. You need to peek inside a complex system, understand how the gears turn, and come up with a way to make it run faster or more efficiently.

Many developers are obsessed with performance. Go to any developer forum and you'll find numerous questions and theories about the performance of this and that. In many cases, these are observations about specific language constructs or idioms that will not be the source of real issues in the majority of cases, but as developers we pride ourselves on knowing these little bits of performance trivia. So, maybe `for` loops are generally faster than `foreach` loops, but how often will your choice of loop determine the difference between success and failure for your app? Rarely, if ever.

At the end of the day, a successful app is an app that the users enjoy using. Users don't know or care which loop construct you use in your code. Users care about the features your app offers. The more important a feature is to the user, the more you need to pay attention to how the feature appears to the user. This includes whether it is fast enough. Features the users employ all the time should look good, feel good, and be fast. While there's no set goal for exactly what *fast* means, there's research that can help you make reasonable assumptions, which is a good starting point, but ultimately the requirements need to satisfy the users' expectations.

The purpose of this book is to show you how performance is an integral feature of the apps you build. Great performance doesn't happen by accident. It is something you have to design and build. You need to think about performance just like you would think about any other feature; you need to set goals and continually verify that your app meets these goals. I will show you the tools and the techniques you can use to do this for Windows Store apps. After reading this book, you should have a good understanding of what it takes to build high-performance Windows Store apps.

Who should read this book

This book exists to help developers, testers, designers, and project managers who want to build Windows Store apps with great performance. If you want to learn about what affects the performance of your apps; what you can do to build apps that are fast, fluid, and efficient; and how you can investigate performance issues when they arise, this book is for you.

In many cases, performance engineering is something that's left for a few expert developers to fix at the end of the project cycle. One of the goals of this book is to illustrate why that approach is rarely the best option. Addressing performance problems at the end often leads to risky and expensive last-minute changes.

This book offers an alternative approach that acknowledges the challenges of getting performance right. A key aspect of this is to recognize that performance is affected by the visual design, the architecture, and the implementation of the app. As such, the entire team should be conscious about performance goals and how to achieve them.

The other problem with the typical approach is that performance is considered a specialist issue that only the few experts on the team should be concerned about. Like security, performance is something that must be built into the app from the beginning. You cannot make an app and add great performance and security at the end. Again, performance is something everyone on the team should be concerned about, and it takes an explicit engineering effort to get performance right.

Assumptions

This book assumes that you have at least a minimal understanding of how to build Windows Store apps using C# and XAML. Perhaps you have already built, designed, or tested a couple of apps and want to improve your understanding of how performance is affected by the different parts of the app and the underlying runtime system.

The book focuses entirely on C#, but developers using Visual Basic should be able to map the concepts and techniques discussed in the book to their work as well.

If you need a textbook covering how to build Windows Store apps with C#, I recommend *Windows Runtime via C#* (Microsoft Press, 2013) by Jeffrey Richter and Maarten van de Bospoort, and *Windows Store App Development: C# and XAML* (Manning, 2013) by Pete Brown.

This book might not be for you if...

This book might not be for if you are already very familiar with good engineering practices around performance and know and use Windows Performance Toolkit regularly to improve your apps, in which case you'll find few or no new insights in this book. The book is specifically aimed at developers, testers, designers, and project managers, who want to begin the journey to becoming performance experts. Chapter 7 includes a list of additional performance resources, so if you're looking for advanced-level texts, you might be able to pick up some suggestions from there.

Organization of this book

This book is divided into seven chapters as follows:

Chapter 1, "Setting the stage," discusses how performance is typically handled in many projects and how this approach often leads to problems that can be difficult and costly to address. Following that, the chapter gives a brief overview of the approach offered in this book.

Chapter 2, "Platform overview," walks you through the different parts of the platform and how they each affect the performance of the apps you're building. Understanding the platform is necessary to understand why some designs, architectures, and implementations can be problematic for the apps you're trying to build.

Chapter 3, "Designing for performance," shows you how you can design and implement specific features of your app to be fast, fluid, and efficient. Regardless of the kind of app you're building, a number of features or user experiences will always be more important to the users than others. Making sure these are designed and implemented with performance in mind is the key to building apps that are a pleasure to use. Each experience has a set of challenges the implementation must address to succeed. This chapter gives you several examples of challenges and approaches to addressing those.

Chapter 4, "Instrumentation," covers how you can enable your app to tell you what is going on while it is running. In short, instrumentation allows you to measure how time and resources are spent in your app. The chapter introduces Event Tracing for Windows (ETW), which is the premier instrumentation technology in Windows and all the relevant subsystems today. I show you how you can use this technology to measure the performance of specific parts of your app.

Chapter 5, "Performance testing," shows you how you can verify that the performance of your app meets the goals you defined. Between performance goals and instrumentation, you have the basic building blocks for setting up performance tests that will continually let you assert the performance of your app and highlight issues as they appear.

Chapter 6, "Investigating performance issues," discusses the tools and techniques you need to identify and investigate performance issues. The chapter includes several examples of investigating common issues and presents solutions to each of these.

Chapter 7, "Wrap-up," summarizes the advice given in the book and provides a list of further resources for you to study on your journey to becoming a performance expert.

System requirements

You will need the software below to follow the examples in the book:

- Windows 8 or preferably Windows 8.1. (Remember, upgrading to Windows 8.1 is free if you already have Windows 8.) The book specifically targets Windows 8.1, but most of the content applies to Windows 8 as well.
- Microsoft .NET Framework 4.5 or later.
- Microsoft Visual Studio 2013.
- Windows Assessment and Deployment Kit (Windows ADK) for Windows 8.1. Make sure to get the latest versions as described in the following section.
- A PC with an HD display or better is recommended.

Windows Performance Toolkit versions

This book is written using an internal version of Windows Performance Toolkit. As of this writing, the version used is scheduled for release before the book becomes available, so the text should match the software at that point. However, you should make sure you have the latest version of the software used in this book to avoid any confusion.

An easy way to verify if you have the latest version of Windows Performance Toolkit is to look at the name of the recording profile provided for XAML apps. The version that shipped with the Windows 8.1 ADK referred to this profile as "XAML Application Analysis," while the updated version uses the title "XAML App Responsiveness Analysis," as does this book. You want to make sure that the profile listed in the Scenario Analysis section of Windows Performance Recorder (WPR) says "XAML App Responsiveness Analysis." Please refer to Chapters 1, 4, and 6 for additional information on WPR and recording profiles.

Code samples

The code samples used in this book are available for download from:

http://aka.ms/highperf

Acknowledgments

Writing a book is a lot of work, and I couldn't have done it alone. I have received feedback and input from numerous skilled people who have all helped make this a much better book.

I want to thank Brian Braeckel and Kiran Kumar from the XAML performance team for enlightening me about the details of the XAML engine and for answering countless questions about how everything works under the hood.

I want to thank my colleagues Jürgen Schwertl, Will Sergeant, and Kraig Brockschmidt, who all provided a lot of useful input and great ideas for the book. I also want to thank Cenk Ergan, Jason Hendrickson, and Cameron McColl for taking the time to discuss various performance issues covered in the book. Along those lines, I want to thank my friend (and fellow Dane) Mads Torgersen because he provided great feedback

for the book, but more importantly, Mads has been my guide and mentor to life at Microsoft; for that, I am eternally grateful.

A special thank-you goes out to Vance Morrison and Cosmin Radu of the Common Language Runtime team at Microsoft for building PerfView and the entire set of tools for accessing Event Tracing for Windows from managed code. Vance Morrison also provided feedback on the text and answered all my questions about ETW and performance investigations.

On the Windows Performance Analyzer team, I want to thank Robin Giese for helping me understand how WPA handles ETW and for improving the support for using `EventSource` with WPA. Robin also has one of the coolest offices I have seen at Microsoft.

Additionally, I want to thank my editor at Microsoft Press, Devon Musgrave, for giving me the courage to write a book and for skillfully and patiently guiding me through the entire process. In the same spirit I owe a big thanks to my managers, Kyle Marsh and Keith Rowe, for supporting the idea and making the book possible.

A number of people provided great feedback and did great work during the development of this book, and I want to thank my technical editor Marc Young, copyeditor Roger LeBlanc, proofreader Nicole Schlutt, and indexer Jean Skipp for their awesome contributions.

Last but not least, I want to thank Eric Lippert for writing the Foreword for my book. I had the privilege of working with Eric and a number of other very skilled and name-dropping-worthy people as part of the Roslyn project. We worked on building the next-generation C# and Visual Basic compilers and language services for Visual Studio. I cherish the fact that I could ask Eric Stack Overflow questions in person. I'm very grateful that I got the chance to work on that project, and I am so happy that Eric wrote the Foreword.

Errata, updates, & book support

We've made every effort to ensure the accuracy of this book and its companion content. You can access updates to this book—in the form of a list of submitted errata and their related corrections—at:

http://aka.ms/highperf

If you discover an error that is not already listed, you can submit it to us at the same page.

If you need additional support, email Microsoft Press Book Support at mspinput@microsoft.com.

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to *http://support.microsoft.com*.

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://aka.ms/tellpress

We know you're busy, so we've kept it short with just a few questions. Your answers go directly to the editors at Microsoft Press. (No personal information will be requested.) Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*

This page intentionally left blank

CHAPTER 3

Designing for performance

No matter what kind of app you're building, a number of key usage scenarios are essential for the success of your app. A shopping app must enable users to quickly find and purchase items. A news app must be able to quickly present users with what's going on in the world. These key scenarios are what your app does well. This is where your app provides value to the users, so you need to make sure your design and architecture support them efficiently. All your key scenarios should provide a good user experience. That means they should look good and perform well. By keeping performance in mind for your key scenarios, you will reduce the risk of making the wrong architectural decisions that would be expensive to change later in the game.

For instance, let's assume you're designing a news app. Needless to say, you want the app to be able to retrieve and display the latest news because that would be a key scenario for a news app. One approach is to download the news on startup, but what happens if the back end or the network is not responsive? If your app waits for the news to download, it stalls. How do you think the users will react if they have to wait several seconds every time they want to check the news? How would you like turning on your favorite news channel on TV only to discover a big "Please wait" sign for a couple of seconds? That wouldn't work on TV, and it isn't a good user experience for your app either. Users want to check the news at their convenience; making them wait is a bad idea.

This prompts the question: how can you design a news app that is both immediately responsive and able to show the latest news quickly? As the developer, you can control how fast the app launches and becomes responsive by optimizing the code and design, but unless you hold some secret power over the Internet, you are at its mercy. Users access the cloud through everything from high-speed, optical networks to spotty mobile connections on the outskirts of civilization, so there's no way you can control or even predict the expected performance of their network connections. This leads to a simple conclusion: if your app *has* to wait for the latest news as part of startup, there's just no way you can guarantee a good experience in all cases. To ensure a quick launch, you cannot rely on the news being immediately available.

If you ignore this fact, your app might perform well under ideal circumstances, but most likely you will see bad reviews from users who happen to spend their time in the real world where network performance isn't always as fast and reliable as we would like it to be. You can avoid this problem by designing your experience around the fact that network connections are fickle. The only way you can control the user experience is by relying on the parts you can trust.

Designing your app with performance in mind is the topic of this chapter.

Less is more

The most common performance problem I see is apps that try to do too much at once. Of course, this problem comes in a plethora of guises. Sometimes the app is doing work in advance instead of deferring it until needed, sometimes the app tries to handle more data than it is capable of, and sometimes it is doing the same work over and over again. The examples are plentiful, but the overall problem is the same.

In many cases, the majority of this work is done on the XAML UI thread. As you learned in Chapter 2, "Platform overview," the job of this thread is to keep the app responsive, so you don't want to burden the UI thread with too much work. Doing so results in a slow and unresponsive app.

Regardless of the specifics, the solution to this problem is always the same: do less work. In other words, make sure the app does what is necessary to implement its features, but no more than that. Optimizing the performance of your app is all about getting rid of the nonessential elements or at least moving them out of the way on the critical path.

However simple this might sound, it is probably the best advice I can give you for building fast apps. Of course, the minimum amount of work required for any given key scenario isn't always obvious. Identifying redundant work can be difficult, but keeping the mantra of "less is more" in mind will help you trim your app and achieve great performance. If something isn't needed, don't do it. If it's needed later, defer it. If you have to perform a nontrivial operation repeatedly, consider caching the result and use that instead of doing the work again. You have many options, but they all boil down to getting rid of redundant work.

This might still be a little abstract, so let me give you an example of what I'm talking about. This example is from a sports app I worked on. One of the main features of the app was to display results of recent games. Consequently, the app would retrieve results for all of today's and yesterday's games at startup. Furthermore, it would retrieve the entire set of results at a fixed interval. Unlike stock prices, most game results don't change very often and presumably yesterday's results don't change at all, so the majority of numbers would be unchanged between updates. Yet the app would retrieve these numbers over and over. Obviously, this approach doesn't represent the minimal amount of work, because the same data is retrieved and processed repeatedly.

Unfortunately, that was not the only problem. The app downloaded the data as a couple of XML files, one file per day. Because of the redundancy of retrieving all the results on every request, the XML files were large. There are many good things to be said about XML, but it is rarely the optimal choice from a performance perspective. Unless the data is very verbose, XML markup tends to constitute a significant part of—and in many cases, even the majority of—the content of XML files. In this case, the XML markup dwarfed the actual content of the files. Downloading, reading, and decoding the data that way carries a noticeable overhead. For performance-critical parts of the application, you can usually find better options than XML. Processing these files repeatedly is not optimal either.

Fortunately, solving these problems is not difficult—conceptually, at least. We can easily come up with ways to improve this approach. Instead of retrieving all the results on every request, just use a timestamp to limit the retrieval to the latest results. This reduces the amount of data processed on each request significantly, and it might even eliminate the need to process any data at all on some requests. Similarly, picking more efficient encoding for the data is straightforward. There are numerous, less verbose encodings to choose from.

The solutions to the problems are almost trivial. Unfortunately, to implement these changes the team would have to redesign both the app and the back end to accommodate the different approach for getting the data. That's a significant risk to take on late in the project. This scenario re-iterates the point of Chapter 1, "Setting the stage": design decisions that affect performance are much less expensive to make at the beginning of the project, which is why you should scrutinize all your app's key scenarios and look for redundant work to eliminate as you design the app.

Proof of concept

Unless you're building an app very similar to apps you already built, you will probably need to explore certain areas to get the data you need to make good decisions. A common mistake is to settle on a design or an architecture without verifying that it performs and scales as needed. Your requirements should specify the key scenarios for the app and the expected performance of these given specific volumes of input. You need to know ahead of time what kind of data loads you expect to see and make sure your app can handle these.

Building a prototype or a proof-of-concept app is a great way to try out various approaches and collect information on what works and what doesn't work. You want to make sure the app can handle all the relevant key scenarios with the expected performance before you settle on a design and an architecture.

Some designers build prototypes to test usability issues as well. That's a great idea, but typically these prototypes are not useful for measuring performance. Usability prototypes are often built around limited, hard-coded datasets, which provides enough functionality for the users to interact with the app. However, to properly assert performance, you need to use real data sources, and you need to test with data volumes that match your expected usage scenarios. Real data sources have latency that might affect the user experience. Similarly, most designs and algorithms work well as long as the input is limited. If you want your app to be able to handle specific quantities of input, your prototype must verify that your design and architecture are capable of doing so.

The time you spend up front verifying your assumptions about the design and architecture is a good investment. Any shortcomings you can spot at this stage can be addressed, and the cost of doing so is far less than it will be if you have to fix those problems at a later stage.

Design challenges

In the remainder of this chapter, I go through a number of common app scenarios, look at the design challenges they present, and provide guidelines on how to address these. The scenarios are

- Login
- Live content
- Handling a lot of content
- Handling media

Getting these scenarios to perform well can be a challenge, but if you think about the performance during the design, you can address the biggest issues at this stage. You might still have to tune the implementation later, but getting the architecture right is paramount.

Login

Many social apps, enterprise apps, and so forth need the user to log in before they show relevant content. If you're building an app like that, realize that launching your app consists of the following phases:

1. Launch
2. Authentication
3. Show content

The launch part is really simple. Your app just needs to present itself to the user and offer a login experience. Obviously, the login screen should present the visual identity of the app, but other than that there's really no reason to do any kind of work at this point. The login screen is simple, and consequently the launch experience should be simple and very fast. Yet, I have seen several apps that go through elaborate setup prior to displaying the login screen, which makes the launch experience much longer than it needs to be. There are really no good reasons that this shouldn't be fast.

Once the login screen shows, the app will most likely sit idle for a while—possibly for a long time before the user enters her credentials. If your app needs to perform additional setup steps or fetch resources, this can run in the background as the app is waiting for the user to log in. Launching work at this point means the app can prepare resources while the user enters her credentials.

Authentication itself requires interaction with your app back end, and this interaction might involve a noticeable delay. If your app waits for authentication to complete before proceeding, this latency delays the login experience. While you obviously don't want to present sensitive content to nonauthenticated users, the app can still do a lot of work while waiting for the authentication to complete.

If your app waits until the user is authenticated, you're optimizing for the case where the login is unsuccessful. A better approach is to optimize for the case where login is successful. After all, this is the scenario your users care about and it is probably the most frequent scenario as well. Go ahead and do as much work and render as much of the UI as possible under the assumption that the login will succeed. This reduces or even eliminates the latency of the authentication. If the login doesn't succeed as expected, you can implement a fallback to handle that. It is much less important to get the performance of the failed login scenario right, because fundamentally this scenario isn't very valuable to the users.

Login is an interesting scenario because it includes a lot of time where the app is simply waiting. The key to optimizing the experience is using this delay to your advantage by doing as much as possible while the app is waiting.

Live content

Many apps need to display the most recent information to the users at startup or during page navigation. News apps obviously fall into this category, but so do shopping apps, financial apps, movie and music apps, and many others. Typically, these apps pull content from services in the cloud and, as discussed earlier, that presents a challenge, because networks are inherently unreliable. I go through numerous techniques you can use to improve the user experience when handling live content. The topics are

- Prioritize your content, and make sure it is available.
- Use caching to reduce downloads.
- Use `ContentPrefetcher` to load data in advance.
- Asynchronous I/O.
- Extended splash screen.

Prioritize your content, and make sure it is available

First you should prioritize your content. All your content is not going to be equally important to the users. Your design should reflect this. You need to figure out which are the most important resources and make sure the app handles these before anything else. Basically, you want to identify all the resources needed to populate the first screen of the app or at least part of it.

Your app should retrieve these resources as soon as possible and defer everything else. The goal is to get the data needed and nothing more, present that to the user, and make sure the app is responsive as quickly as possible. All other resources should be handled once the app is responsive.

Once your content is prioritized, you must make sure it is available. This can be tricky if your content isn't local to the app. Any content you have to retrieve at run time might delay progress indefinitely. The only way to guarantee a fast experience is to rely on local resources.

Your app can attempt to download resources at startup, but it shouldn't wait for these, or at least it should limit the wait to a few hundred milliseconds. In case online resources are not available, your app should have a fallback scenario that allows it to launch with slightly stale data. The prefetch feature discussed later in this chapter is a great way to refresh local caches even when the app isn't running. This means that stale data might not be very old.

There are various ways to update local content with retrieved content once this is available. One approach is to use the `FlipView` in a way similar to what the Store app does. (I'm sure you're familiar with the Store app, but for your convenience Figure 3-1 shows what I am talking about.) Your app can load a `FlipView` with local content and start retrieving additional content. Once this content is available, it can be added to the `FlipView` carousel. This provides a smooth transition between the two and guarantees a good user experience even when updated resources cannot be retrieved.

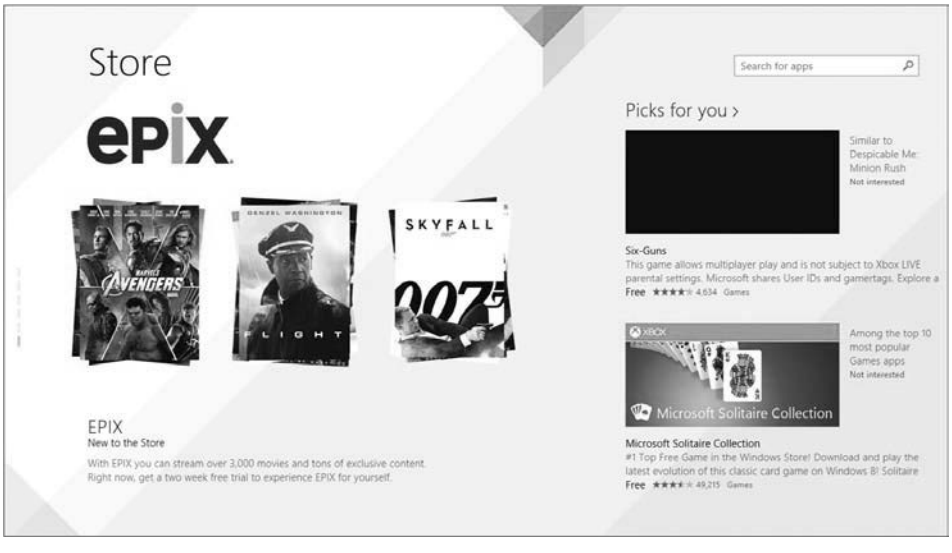


FIGURE 3-1 The Windows Store app uses the FlipView to display both cached and new content.

Use caching to reduce downloads

Caching is a standard feature of all HTTP-based protocols. If resources are cached, they are retrieved locally instead of through the network. Using caching can significantly reduce the need for retrieving data over the network, so the second thing you should do is make sure all your data is cached appropriately. Some resources might be valid only for seconds, but most resources will probably be relevant for minutes, hours, or even days. After all, even high-profile news sites don't change their top story every minute.

Caching is controlled entirely by the server. As long as the server outputs the proper caching information, the Windows Internet (WinINet) stack caches resources as needed and you don't have to

do anything in the app itself to take advantage of this. The `HttpClient` class in `Windows.Web.Http` uses WinINet (but the `HttpClient` defined in `System.Net.Http` does *not*, so make sure you use the right one). If data is cached, it is retrieved locally instead of through the network. Once the caching expires, the resource will be retrieved and cached again on the next request. All of this is completely transparent to the app, so you don't need to do anything in the app itself, but you need to make sure caching is handled correctly on the server.

In some cases, you might not control the caching on the server, such as when you're integrating data from back ends you don't control. In that case, you can introduce a façade server that retrieves the same data as your app would and adds caching information as necessary. Admittedly, this makes the back end part of the app more complex, but doing this improves the performance of your app for all users. If you're doing this, you might also be able to rearrange the data to better suit your needs. Perhaps the original feed is more verbose than what your app needs. Trimming the data feed to your needs is a great way to reduce the amount of data downloaded and the time needed to handle it.

For non-HTTP-based protocols, you typically have to implement your own caching scheme.

Use ContentPrefetcher to load data in advance

Caching requires resources to be retrieved once, so you might be thinking how you can take advantage of caching on something like news stories and accompanying pictures. If the user has to retrieve the data before it can be cached, it is not going to benefit scenarios like that.

The `ContentPrefetcher` class introduced in Windows 8.1 aims at solving that specific problem. The idea is that you can specify—either directly or indirectly—resources that should be retrieved and cached even if your app is not running. Windows will then automatically retrieve those resources periodically and cache them on behalf of your app. This increases the chance that resources can be retrieved locally from the cache. Obviously, all the resources must support caching to take advantage of this. If they don't, prefetching them will not make any difference.

It goes without saying that Windows cannot simply poll the resources every other second because that would drain the power of the device quickly and possibly exhaust the user's data plan. Instead, Windows uses heuristics to determine what resources are downloaded and how. These heuristics take into account network and power conditions, app usage history, and the results of prior prefetch attempts to provide maximum user benefit. The bottom line is that Windows does this in an effective manner, but there is no guarantee that any particular resource will have been downloaded before a given app launches.

In a way, using `ContentPrefetcher` is a bit like owning a lottery ticket. You might not win anything, but the cost of the ticket is low. However, unlike a real lottery ticket, the odds of winning are pretty good in this case. Although the rewards will not yield you a new Ferrari, they will improve the performance of your app. If you ask me, that's well worth the small cost.

Using ContentPrefetcher directly

Using `ContentPrefetcher` is straightforward. If the URIs (locations) of the resources are known to the app, the app simply configures a list of resources with the system and Windows attempts to retrieve them. This is the direct way of using `ContentPrefetcher`, and it works well if the URIs don't change but the content returned does. Listing 3-1 shows how to use `ContentPrefetcher` when the list of resources is known to the app.

LISTING 3-1 Using ContentPrefetcher when the URIs are known to the app.

```
var resources = new[] {
    "http://windowsteamblog.com/windows/b/developers/atom.aspx",
    "http://windowsteamblog.com/windows/b/windowsexperience/atom.aspx",
    "http://windowsteamblog.com/windows/b/extremewindows/atom.aspx",
    "http://windowsteamblog.com/windows/b/business/atom.aspx",
    "http://windowsteamblog.com/windows/b/bloggingwindows/atom.aspx",
    "http://windowsteamblog.com/windows/b/windowssecurity/atom.aspx",
    "http://windowsteamblog.com/windows/b/springboard/atom.aspx",
    "http://windowsteamblog.com/windows/b/windowshomeserver/atom.aspx",
    "http://windowsteamblog.com/windows_live/b/windowslive/rss.aspx",
    "http://windowsteamblog.com/windows_live/b/developer/atom.aspx",
    "http://windowsteamblog.com/ie/b/ie/atom.aspx",
    "http://windowsteamblog.com/windows_phone/b/wpdev/atom.aspx",
    "http://windowsteamblog.com/windows_phone/b/wmdev/atom.aspx",
    "http://windowsteamblog.com/windows_phone/b/windowsphone/atom.aspx"
};

ContentPrefetcher.ContentUris.Clear();
foreach (var res in resources)
{
    ContentPrefetcher.ContentUris.Add(new Uri(res));
}
```

Listing 3-1 adds a number of URIs to the list of `ContentUris` on `ContentPrefetcher`. This populates a global list, so unless you clear the list first, you'll simply add to the existing list. Furthermore, `ContentPrefetcher` limits the number of `ContentUris` to 40 per app. If you add more than 40 URIs, `ContentPrefetcher` throws `Exception`. (For some reason, it doesn't throw a more specific exception.) The bottom line is you should either check the content of `ContentUris` or clear the content before adding to the list. Moreover, you must limit the number of resources to 40 or fewer per app.

The preceding code instructs Windows to retrieve the resources according to the heuristics discussed earlier. To test that prefetching works as expected, you can force the system to fetch the configured resources through the `IContentPrefetcherTaskTrigger::TriggerContentPrefetcherTask` method. Unfortunately, there's no managed wrapper for this call yet, so you have to do this from a C++ app. See the "Triggering prefetching" sidebar for an example of a small C++ console app that triggers fetching for a specific app.

Triggering prefetching

Once you set up `ContentPrefetcher` to retrieve resources for your app, Windows will do so according to its heuristics. To test that the resources are fetched as expected, you can trigger fetching by calling `IContentPrefetcherTaskTrigger::TriggerContentPrefetcherTask`. Unfortunately, there's currently no other way to do that. To make matters a bit more complicated, this method is not exposed to managed apps, so you have to call it from native code.

Listing 3-2 contains the code required to trigger prefetching for a specified app. To compile this project, you need to do the following:

1. Create a C++ Win32 console application named `TriggerPrefetch`.
2. Enter the code from Listing 3-2 into the `TriggerPrefetch.cpp` file.
3. Go to project properties, and locate the C/C++ section. Enter or enable the following:
   a. Under General, add the paths `C:\Program Files (x86)\Microsoft SDKs\Windows\v8.1\ExtensionSDKs\Microsoft.VCLibs\12.0\References\CommonConfiguration\neutral` and `C:\Program Files (x86)\Windows Kits\8.1\References\CommonConfiguration\Neutral`.
   b. Under General > Consume Windows Runtime Extensions, select the value Yes (/ZW).
   c. Under Code Generation > Enable Minimal Build, select the value No (/Gm–).
4. Build the project.
5. Use the `Get-AppxPackage` command in Windows PowerShell to get the full package name for your app.
6. Run the newly built `TriggerPrefetch` utility with the full package name as input.
7. Check the return value by echoing `%errorlevel%`. A value of zero means no errors.

This makes Windows fetch the configured resources immediately. You can verify that the fetch occurs by monitoring network traffic with a network monitor such as Fiddler or by capturing events from the `Microsoft-Windows-BackgroundTransfer-ContentPrefetcher` provider. Admittedly, this is not the most elegant way to test this, so I'm hoping the experience improves with a future release.

LISTING 3-2 C++ code for the TriggerPrefetch utility.

```
#include "stdafx.h"
#include <IContentPrefetcherTaskTrigger.h>
#include <roapi.h>
#include <winstring.h>
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    WCHAR* activableClass = L"Windows.Networking.BackgroundTransfer.ContentPrefetcher";
    int iLen = wcslen(activableClass);
    HSTRING hs_activableClass;
    int rc = 0;

    if (argc > 1) {
        if (SUCCEEDED(WindowsCreateString(activableClass, iLen, &hs_activableClass))) {
            if (SUCCEEDED(RoInitialize(RO_INIT_MULTITHREADED)))
            {
                IContentPrefetcherTaskTrigger *trigger = nullptr;
                if (SUCCEEDED(Windows::Foundation::GetActivationFactory(hs_activableClass,
                                                                        &trigger))) {
                    // supply PackageFullName at prompt, retrieve the name by running
                    // Get-AppxPackage cmdlet
                    if (FAILED(trigger->TriggerContentPrefetcherTask(argv[1]))) {
                        // log error and set bad return value
                        rc = -1;
                    }
                    trigger->Release();
                }
            }
            RoUninitialize();
        }

        if (FAILED(WindowsDeleteString(hs_activableClass))) {
            // log error and set bad return value
            rc = -2;
        }
    }
    else {
        printf("Syntax: TriggerPrefetch <PackageFullName>\n");
    }

    return rc;
}
```

Using ContentPrefetcher indirectly

For some apps, the direct approach is not very attractive. For example, a news app might retrieve stories and corresponding images whose URIs change all the time. Today's top story has a different URI than yesterday's top story, so the app has no way to enumerate the resources it wants to prefetch. To handle this situation, `ContentPrefetcher` offers an indirect way to specify the resources to be fetched. When you use the indirect approach, the `ContentPrefetcher` queries a single resource for a list of resources to retrieve. That is, the list of resources is always available at the same URI, but the content returned through this list differs over time. This allows news apps and the like to constantly refresh the cache based on the new resources on the server side.

The list of resources is just an XML document. The document must conform to the schema in Listing 3-3. Listing 3-4 shows the same resources you used in the example in Listing 3-1 as a

proper XML file for ContentPrefetcher. With the XML file in place, you just need to set the IndirectContentUri property as shown here:

```
ContentPrefetcher.IndirectContentUri = new Uri("http://localhost:46449/resources.xml");
```

This configures ContentPrefetcher to retrieve the list of resources—in this case, called *resources.xml*—from the specified URI. For the purpose of this example, I am just using a local server, but obviously you should point to a real URI to retrieve the list.

LISTING 3-3 Schema for IndirectContentUri.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="PrefetchUris">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="1" name="uri" type="xs:anyURI" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

LISTING 3-4 Sample XML file for IndirectContentUri.

```
<?xml version="1.0" encoding="utf-8" ?>
<prefetchUris>
  <uri>"http://windowsteamblog.com/windows/b/developers/atom.aspx"</uri>
  <uri>"http://windowsteamblog.com/windows/b/windowsexperience/atom.aspx"</uri>
  <uri>"http://windowsteamblog.com/windows/b/extremewindows/atom.aspx"</uri>
  <uri>"http://windowsteamblog.com/windows/b/business/atom.aspx"</uri>
  <uri>"http://windowsteamblog.com/windows/b/bloggingwindows/atom.aspx"</uri>
  <uri>"http://windowsteamblog.com/windows/b/windowssecurity/atom.aspx"</uri>
  <uri>"http://windowsteamblog.com/windows/b/springboard/atom.aspx"</uri>
  <uri>"http://windowsteamblog.com/windows/b/windowshomeserver/atom.aspx"</uri>
  <uri>"http://windowsteamblog.com/windows_live/b/windowslive/rss.aspx"</uri>
  <uri>"http://windowsteamblog.com/windows_live/b/developer/atom.aspx"</uri>
  <uri>"http://windowsteamblog.com/ie/b/ie/atom.aspx"</uri>
  <uri>"http://windowsteamblog.com/windows_phone/b/wpdev/atom.aspx"</uri>
  <uri>"http://windowsteamblog.com/windows_phone/b/wmdev/atom.aspx"</uri>
  <uri>"http://windowsteamblog.com/windows_phone/b/windowsphone/atom.aspx"</uri>
</prefetchUris>
```

Tips for using ContentPrefetcher

ContentPrefetcher has a LastSuccessfulPrefetchTime property that returns a nullable DateTimeOffset indicating when the last prefetch (if any) executed. You can use this to display a message about when the content was retrieved.

Common to both the direct and indirect approach is that they work only with the HttpClient class defined in the Windows.Web.Http namespace. ContentPrefetcher does *not* work with the HttpClient class from the System.Net.Http namespace, so you have to make sure you use the right HttpClient for this to work.

I hope this goes without saying, but setting up `ContentPrefetcher` should not be part of the critical path for the launch experience. `ContentPrefetcher` doesn't alter the current launch, but it might improve subsequent launches by making resources available locally. Consequently, this is something you should set up once the app is up and running.

Asynchronous I/O

Even with the help of `ContentPrefetcher`, many apps still need to retrieve resources from the cloud at run time. Retrieving data from the network easily takes hundreds of milliseconds, and it might even take seconds to complete. In the past, it was easy to block the UI thread doing I/O like that. Blocking the UI thread for more than 50 milliseconds (ms) or so can make your app appear unresponsive. Fortunately, WinRT and C# make it easy to write asynchronous code, which prevents you from blocking the UI thread because of I/O.

Despite the asynchronous design of the WinRT API and the excellent support for writing asynchronous code in C#, you still need to keep in mind a couple of pitfalls. Although asynchronous I/O prevents the UI thread from blocking and thus keeps your app responsive, it doesn't change how long a given operation takes. If it takes 400 ms to download some resource, making this operation asynchronous allows your app to do other work while waiting for the operation to complete, but it doesn't change the fact that the app has to wait 400 ms. Asynchronous code allows your app to do something while it is waiting, but it doesn't change the duration of the wait.

Consider Listing 3-5. It retrieves an RSS feed asynchronously and does some work as represented by the `DoMoreWork` method once that operation has completed. Using `await` accomplishes two things here. It makes `SomeMethod` return immediately so that the caller is not blocked, and it captures the state and the remaining code of the method so that it can run once `RetrieveFeedAsync` completes. Now assume that retrieving this RSS feed takes 400 ms. That means the total running time for `SomeMethod` is 400 ms or more. The calling thread can do other work in the meantime, but `DoMoreWork` doesn't run until at least 400 ms have passed.

LISTING 3-5 Simple asynchronous method.

```
public async void SomeMethod()
{
    var feedUri = new Uri("http://windowsteamblog.com/windows/b/developers/atom.aspx");
    var client = new Windows.Web.Syndication.SyndicationClient();
    var feed = await client.RetrieveFeedAsync(feedUri);

    DoMoreWork();
}
```

If `DoMoreWork` needs the result of the asynchronous operation, this approach makes perfect sense, but it doesn't change the fact that the invocation of the method is delayed by the duration of the asynchronous operation in `RetrieveFeedAsync`.

This is something to keep in mind when you design the launch and navigation experiences for your app. For example, if you `await` an asynchronous operation before you create the main frame in the `OnLaunched` method, the launch experience is delayed by the duration of said operation because the app cannot proceed before the asynchronous operation completes.

Similarly, if you wait before an asynchronous data source is fully populated before you set up binding, the app will not show any data before all the data is available. This might be desirable for a reliable, fast data source, but if the data source could introduce arbitrary delays, this likely turns into a bad user experience. On the other hand, if you set up binding and then populate the data source asynchronously, the app displays data as it is added to the data source. If any of the elements take a long time to retrieve, they will not show up before they are ready, but at least the app remains responsive and shows the data that is available while the remaining data is retrieved.

Cancelling asynchronous operations

When you write asynchronous code, the `await` keyword acts like a rendezvous point in the code. Whenever the asynchronous method completes, your code continues to execute just as if the call had been synchronous.

That's great, until you realize that "whenever" is unbounded. Your code could end up waiting forever to run. If an asynchronous calls stalls forever, the remaining code will never run.

If that's a concern, you want to be able to detect situations like that and cancel the asynchronous operation. For instance, if your app requests live content at startup, you should implement a fallback mechanism that cancels the outstanding requests and provides content in an alternate fashion if the request takes too long.

Using CancellationToken with Task

If you are familiar with the Task Parallel Library (TPL), you probably know about `CancellationToken`. A `CancellationToken` is useful when you want to cancel a running task. You simply pass in the token and your task can then check whether cancellation has been requested. If you want to cancel a CPU-bound `Task`, you should use a `CancellationToken`, just like you would when using TPL for desktop or server applications. Listing 3-6 shows a simple example of doing this.

LISTING 3-6 Using CancellationToken to cancel a long-running computation.

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;

// The Blank Page item template is documented at http://go.microsoft.com/fwlink/?LinkId=234238
```

```
namespace CancelTask
{
    /// <summary>
    /// An empty page that can be used on its own or navigated to within a Frame.
    /// </summary>
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
        }

        // input for Fibonacci calculation
        private int Counter = 35;

        private CancellationTokenSource Cts;

        private async void Calculate_Click(object sender, RoutedEventArgs e)
        {
            Cts = new CancellationTokenSource();
            try
            {
                Output.Text = (await LongRunningComputationAsync(Cts.Token)).ToString();
            }
            catch (OperationCanceledException)
            {
                Output.Text = "Cancelled";
            }
        }

        private void Cancel_Click(object sender, RoutedEventArgs e)
        {
            Cts.Cancel();
        }

        private Task<int> LongRunningComputationAsync(CancellationToken token)
        {
            return Task.Run(() => RecursiveFibonacci(Counter++, token), token);
        }

        // Very ineffective implementation of Fibonacci
        // For illustration purposes only - Don't use!
        private int RecursiveFibonacci(int n, CancellationToken token)
        {
            if (n <= 1)
            {
                return 1;
            }

            token.ThrowIfCancellationRequested();

            return RecursiveFibonacci(n - 2, token) + RecursiveFibonacci(n - 1, token);
        }
    }
}
```

Listing 3-6 sets up two event handlers. The first event handler triggers a long-running, asynchronous calculation. In this case, the calculation is an extremely inefficient implementation of Fibonacci. There are much better ways to implement this function. The only nice feature of this implementation is that it is simple to read. The performance of this implementation is horrible, so please don't use this code.

The interesting thing about this implementation is that it takes a `CancellationToken` as input and checks whether cancellation has been requested. If cancellation is requested, it throws an `OperationCanceledException`, which must be caught by the caller. The exception is automatically marshalled from the worker thread running the asynchronous operation to the calling thread that initiated the operation.

The second event handler simply signals that cancellation was requested. This is done by calling the `Cancel` method on the `CancellationTokenSource`. This changes the state of the `CancellationToken` and aborts the asynchronous operation with an `OperationCanceledException`.

Notice how the `CancellationToken` in `LongRunningComputationAsync` is passed to both `RecursiveFibonacci` and to `Task.Run`. The reason for this is that both methods need to be able to detect whether cancellation was requested. If cancellation was requested before `Task.Run` executes, no task is scheduled. If cancellation was requested while the calculation was in progress, the call to `ThrowIfCancellationRequested` will cancel it.

Using CancellationToken with WinRT APIs

`Task` and `CancellationToken` are .NET Framework concepts and thus unknown to the WinRT APIs. That is, none of the asynchronous WinRT methods take a `CancellationToken` as input, so by default the approach in Listing 3-6 doesn't work with WinRT methods. Luckily, the `WindowsRuntimeSystemExtensions` class provides the `AsTask` extension method that turns the object returned by a WinRT asynchronous method into a `Task`.

The `WindowsRuntimeSystemExtensions` class provides many overloads of the `AsTask` extension method, and several of those take a `CancellationToken`. In other words, the `AsTask` extension method effectively bridges the gap between `Task`-based asynchronous methods and WinRT-based asynchronous methods. All you need to do to use a `CancellationToken` with WinRT-based methods is call the `AsTask` extension method on the objected returned by the asynchronous method.

Listing 3-7 shows how you can use a `CancellationToken` to cancel an asynchronous WinRT method.

LISTING 3-7 Using CancellationToken with WinRT-based asynchronous methods.

```
using System;
using System.Threading;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
```

```
// The Blank Page item template is documented at http://go.microsoft.com/fwlink/?LinkId=234238

namespace CancelAsyncWinRT
{
    /// <summary>
    /// An empty page that can be used on its own or navigated to within a Frame.
    /// </summary>
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
        }

        private CancellationTokenSource Cts;

        private async void Download_Click(object sender, RoutedEventArgs e)
        {
            Cts = new CancellationTokenSource();

            var feedUri = new Uri("http://windowsteamblog.com/windows/b/developers/atom.aspx");
            var client = new Windows.Web.Syndication.SyndicationClient();

            // Disable cache, so method runs for a while
            client.BypassCacheOnRetrieve = true;
            var feed = client.RetrieveFeedAsync(feedUri).AsTask(Cts.Token);

            try
            {
                Output.Text = (await feed).Title.Text;
            }
            catch (OperationCanceledException)
            {
                Output.Text = "Cancelled";
            }

        }

        private void Cancel_Click(object sender, RoutedEventArgs e)
        {
            Cts.Cancel();
        }
    }
}
```

Listing 3-7 is similar to the code in Listing 3-6. However, this time the app calls `RetrieveFeedAsync` on `SyndicationClient`. That's a native, asynchronous WinRT method. By default, this doesn't support the use of a `CancellationToken`. Notice how `AsTask` turns the object returned by `RetrieveFeedAsync` into a `Task<SyndicationFeed>`. The `AsTask` overload takes a `CancellationToken` as input, which enables you to cancel the WinRT API call just like you would do if this was a regular .NET task. Furthermore, you can `await` the `Task` just like any other awaitable type, and once the asynchronous operation completes, you can fetch the `Title` from the downloaded feed.

Extended splash screen

The MSDN guidelines state that your app can use an extended splash screen if it needs more time during startup. The reasons for using an extended splash screen are twofold. First of all, you want to prevent Windows from terminating the app because of excessive startup time. Second, you might want to let users know that the app is still launching. The extended splash screen typically replaces the system-controlled splash screen with an identical or similar screen that is displayed while the app loads.

From the point of view of Windows, the app is actually running at this point. Using an extended splash screen prevents Windows from shutting down your app because it spends too much time getting started, but it doesn't change the fact that the user has to wait a long time for the app to become responsive. Waiting several seconds for an app to launch is not a good user experience, extended splash screen or not.

Many apps simply display their logo accompanied by a progress indicator as their extended splash screen. There might be a few users out there who get all excited by seeing spinning dots for seconds, but for the rest of us that's not a great experience. I'm not a big fan of extended splash screens, but if your app must use an extended splash screen at least make sure it adds some value. You can beef up the progress indicator by telling the user what the app is doing. I don't feel that it adds a lot of value, but it does break the monotony, which makes it a tad better than the progress indicator on its own. Instead, try to come up with some interesting information you can show the user. This could be in the form of helpful tips for using the app, interesting stats about the usage, or something else that might make the user interested in exploring new areas of the app.

Extended splash screens make sense for advanced games or apps that have to process large amounts of data on startup. Most regular apps should not need an extended splash screen, and using one should definitely be the last resort. Make sure your app's start experience is as fast as it can get using the tips covered in this chapter. Before you even consider using an extended splash screen, you should measure the startup performance of your app and optimize the critical path as described in Chapter 6, "Investigating performance issues." It is likely that you can optimize the launch experience to the point where you don't need an extended splash screen. All things considered, a faster startup makes for a much better user experience than simply using an extended splash screen to mask performance issues.

Handling a lot of content

Some apps handle large datasets. Many shopping apps make thousands of items available to the user, photo browsers let the users browse large image libraries, and social apps handle countless messages, replies, comments, and so forth. If you're building an app that needs to handle a lot of content, there are several things to keep in mind. I go through the following topics:

- Prioritize your content.
- Partition content to reduce workload.

- Size or decode images.
- Cache information to improve transitions.

Prioritize your content

I know this repeats one of the points of the previous section, but prioritizing is a very important part of performance work in general, so it bears repeating. Every time you make some scenario run faster, you do so at the expense of something else. There's no way every part of your app can be as fast as possible, so you need to prioritize.

If your app needs to show a lot of content, some content will be more important than other content. If you do not figure out what the important parts are and focus on those, you force the user to make this prioritization and that is not a great user experience. This is the reason news apps have top stories—they help the readers focus their attention on the important parts. Furthermore, it is virtually impossible to avoid suggesting some kind of prioritization even when you try not to. A news app that simply lists all the available stories without any embellishment still suggests priority by the order in which the stories are listed.

The key here is that priority is a useful tool that shapes the user experience. You can use this tool to your advantage. Users are looking for order, so don't give them chaos. If your app has a lot of content, the best way to improve the user experience and the performance is to acknowledge that some of the content is going to be more important to the user and make sure this is readily available.

In practical terms, this means everything visible on the first page of the app is more important than all the off-screen items. Furthermore, you probably even want to prioritize the content of the first page as well. The new `Hub` control introduced in Windows 8.1 is an excellent way to do that, but there are other useful ways to achieve the same effect.

Many things affect our perception of what is important and what isn't. Order, size, and color of elements all affect how we interpret importance. The `Hub` control uses all these elements to create a clear ordering of things, as you can see in Figure 3-2. On the left, a large picture attracts the user's attention. Because many users read from left to right, both the size and the placement of this item underlines its importance. The next item in the second column is smaller, suggesting it is less important. However, it has a picture to draw attention to it, which makes it more important than the item in the third column, and so forth.

FIGURE 3-2 Mockup of a Hawaii app based on the Hub app template.

The Hub control establishes an obvious order of the elements on the page. To support this, your app must make sure the necessary resources are retrieved in the same order.

Ideally, all the important resources should be loaded locally. The app can attempt to retrieve the resources online, but the only way to guarantee good performance is to have a mechanism to fall back to local content in case network latency is too high. Your app can launch asynchronous tasks to retrieve the elements, but if these run for more than a few hundred milliseconds, the app should use cached resources instead, even if these are stale. The content can be refreshed once the resources are available. If you optimize for responsiveness, your users will be able to use the app but might not have the latest data. On the other hand, if you optimize for absolutely up-to-date content, users will get neither responsiveness nor fresh content when the network isn't cooperating.

A common problem in this area is that the server-side APIs might not be designed to support the prioritization of the content your app needs. I have seen several apps that get all their resources in bulk. This means the important resources are bundled with the less important resources, and the result is that the app cannot do anything before everything is ready. In other words, content cannot be prioritized and the overall workload increases. That hurts performance. You need to make sure your back end supports specific queries so that your app can retrieve just the resources it needs at any given time. When your app uses a back end, performance is affected by both the app itself, the back end, and the protocol used to communicate between the two. All three must be designed with performance for the specific scenarios in mind and the maxim "less is more" applies to all of them. Make sure your app is simple; make sure you don't send, receive, and process more data than necessary; and design your back end to support the queries your app needs to make.

Partition content to reduce workload

Very large datasets present a number of challenges. If your app needs to handle large datasets, you should spend some time identifying the maximum reasonable size. Although XAML offers several ways to help you partition the content on the screen, you still need to figure out what a manageable amount of content looks like.

Even though random access virtualization allows the user to page through huge collections of data, you need to ask yourself if that's the best way to present the content. Nobody wants to flip through dozens of pages to find an item. After all, there's a reason department stores have different departments and your newspaper has different sections. Your app should make it easy for the user to zoom in on the content she desires.

Virtualization

Once you identify the right partitioning of your app's data, you might still need to present a lot of items on the screen. As discussed in Chapter 2, XAML offers two ways to handle lists or grids with a lot of items. To avoid repeating myself here, I will just talk about grids, but keep in mind that these ideas apply to lists as well.

As you know, grids can be either virtualized or not virtualized. When a grid is virtualized, the XAML engine renders content on demand. When a grid is not virtualized, everything is rendered up front. Grids that are not virtualized can have great performance once everything is set up, but the time needed to do so can easily hurt the performance significantly. Nonvirtualized grids don't scale well because of this. If your app has to build a large nonvirtualized grid as part of startup or page navigation, performance suffers. Consequently, the recommendation is to use nonvirtualized grids only for small and simple grids.

Virtualized grids are much better at handling large collections because most of the work can be deferred until needed and the underlying data structures can be recycled. Virtualization reduces the work and the storage needed to handle and display the collection, which improves the performance. Virtualized grids scale much better with the number of elements, and they should be your preferred choice for anything but the simplest grids.

However, virtualized grids pay for great startup performance by doing more work during the navigation of the data, so you need to pay attention to how much work is required to handle each element in the grid. XAML offers a couple of ways to customize how this work is handled.

As you saw in Chapter 2, virtualization requires that the `ItemsPanel` of the `GridView` support virtualization. In Windows 8, the default `ItemsPanel` was `WrapGrid`, which supports virtualization. If you change the `ItemsPanel` to `VariableSizedWrapGrid`, you disable virtualization for the `GridView` because this panel doesn't support virtualization.

If your data source was grouped, you would use a `VirtualizingStackPanel` as your `ItemsPanel` for the `GridView` (or nothing at all because `VirtualizingStackPanel` was the default). The `VirtualizingStackPanel` virtualizes the content based on entire groups, which means that all groups that are completely or partially on the screen are fully rendered. Groups that are completely off the screen are not rendered. If panning brings these groups on the screen, they get rendered as expected.

Group-based virtualization works well if you have many smaller groups. Most of the groups will be off the screen at any given time, so the XAML engine doesn't spend time rendering those. However, if you have few, large groups, the majority of any given group will be off the screen most of the time. In that case, XAML still spends a lot of time rendering items in the group that are not visible.

To work around this, Windows 8.1 introduced a new `ItemsPanel` called `ItemsWrapGrid`, which supports virtualization at the item level. This support allows XAML to virtualize within groups, allowing it to render only a part of any group as necessary. Furthermore, in Windows 8.1, `ItemsWrapGrid` replaces `VirtualizingStackPanel` as the default `ItemsPanel` for grouped grid views. This means that if you have a Windows 8 app that uses the default `ItemsPanel`, all you have to do is retarget your app for Windows 8.1 and your app will automatically use the improved `ItemsWrapGrid`. However, if you customized the `ItemsPanel` to explicitly use another panel (or explicitly specified the use of `VirtualizingStackPanel`), you need to update this to use `ItemsWrapGrid` if you want your app to use the improved version.

To illustrate the performance differences between the different kinds of virtualizations, let me walk you through an example. Figure 3-3 shows a simple grid app with a grouped data source. The app displays 900 abstract pictures along with a title and a short description for each picture. The pictures are all local assets, so network response times don't affect the performance in this case. The pictures are grouped into five groups of 180 pictures each.

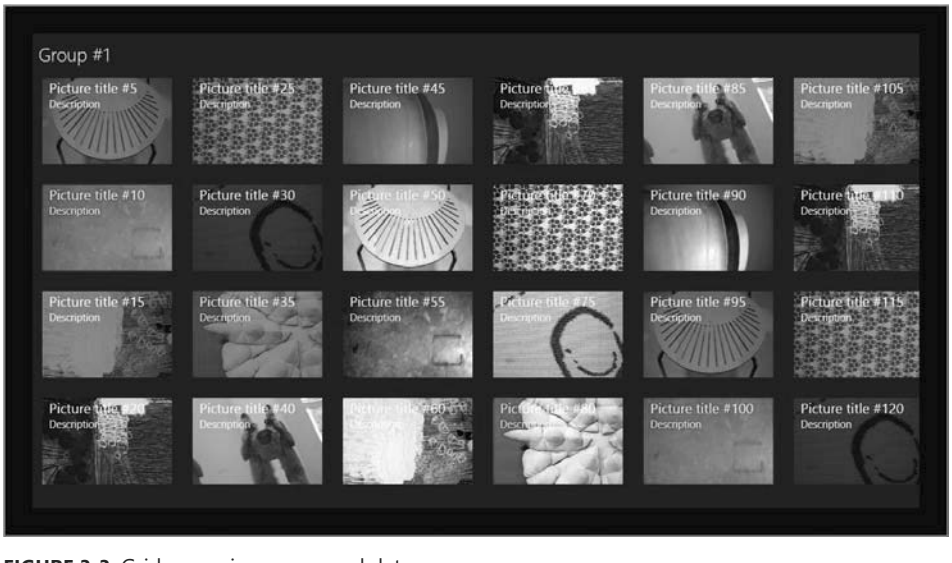


FIGURE 3-3 Grid app using a grouped data source.

The XAML corresponding to the preceding app is shown in Listing 3-8 next.

LISTING 3-8 XAML for grid page.

```
<Page
    x:Class="GridVirtualization.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:GridVirtualization"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Page.Resources>
        <CollectionViewSource x:Key="PictureSource" IsSourceGrouped="True" Source="{Binding}"/>
    </Page.Resources>

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}" Margin="40">
        <GridView ItemsSource="{Binding Source={Binding PictureSource}}" SelectionMode="None" >
            <GridView.ItemTemplate>
                <DataTemplate>
                    <Grid Width="188" Height="125" Margin="10">
                        <Grid.RowDefinitions>
                            <RowDefinition Height="auto" />
                            <RowDefinition Height="*" />
                        </Grid.RowDefinitions>

                        <Image Grid.Column="0" Source="{Binding Location}" />
                        <StackPanel Grid.Column="1" Margin="10,0,10,0">
                            <TextBlock Text="{Binding Title}" FontSize="20" />
                            <TextBlock Text="{Binding Tagline}" />
                        </StackPanel>
                    </Grid>
                </DataTemplate>
            </GridView.ItemTemplate>

            <GridView.ItemsPanel>
                <ItemsPanelTemplate>
                    <VirtualizingStackPanel Orientation="Horizontal" />
                </ItemsPanelTemplate>
            </GridView.ItemsPanel>

            <GridView.GroupStyle>
                <GroupStyle>

                    <GroupStyle.HeaderTemplate>
                        <DataTemplate>
                            <TextBlock Style="{StaticResource SubheaderTextBlockStyle}"
                                       Text="{Binding Key}" Margin="10"/>
                        </DataTemplate>
                    </GroupStyle.HeaderTemplate>

                    <GroupStyle.Panel>
                        <ItemsPanelTemplate>
                            <VariableSizedWrapGrid/>
                        </ItemsPanelTemplate>
```

```
                    </GroupStyle.Panel>

                </GroupStyle>
            </GridView.GroupStyle>
        </GridView>

    </Grid>
</Page>
```

The XAML shouldn't contain any big surprises. This is similar to how the Grid App template would arrange the markup for a Windows 8 app. The XAML sets up a grouped data source and binds that to a `GridView`. The interesting part is the `GridView.ItemsPanel` section. This section defines the `ItemsPanelTemplate` for the grid elements. In this case, I explicitly specify a horizontal `VirtualizingStackPanel` as highlighted in the code. Because this panel doesn't support virtualization on the item level, performance suffers because of the large groups in the grid. On my Surface 2, displaying the grid takes around 2.5 seconds with another 2.5 seconds before the entire set of visible images are rendered. That's 5 seconds in total—far beyond acceptable.

These numbers obviously depend on the size of the total grid as well as the size of each group. In this case, each group contains 180 elements. With 24 elements shown on the screen on a Surface 2, the bulk of any group is off the screen at any given point. Because `VirtualizingStackPanel` causes virtualization to happen per group, XAML has to do a lot of work to render the off-screen elements for the group or groups being displayed.

If I change the markup to use the new `ItemsWrapGrid` instead, the grid virtualizes on a per-item basis, which improves the performance drastically in this case. I measured less than 1.5 seconds for the grid to appear with all visible items rendered on my Surface 2. At this point, the app is responsive to the user. That's certainly within the desired target values, and an example of how you can make a huge difference just picking the proper `ItemsPanel`. Further optimizations might be possible, but obviously picking the proper panel was important. Of course, you don't actually have to specify the `ItemsPanel` explicitly, because the default is `ItemsWrapGrid` if you target Windows 8.1. However, if you do specify an `ItemsPanel`, make sure you pick the right one for the task.

Of course, if you use a non-virtualizing `ItemsPanel` such as `VariableSizedWrapGrid` instead, performance suffers as XAML has to render the entire grid. In this case, I measured around 8 seconds to render the grid on a Surface 2 when using a non-virtualizing `ItemsPanel`.

Placeholders

In Windows 8, a common problem with large, virtualized grids was what we call *panning to black*. Recall that when a grid or a list is virtualized, XAML has to create and render elements as they are panned into view. Creating and rendering each of these items take time. If the items are too complex, too numerous, or both, XAML might not be able to keep up and render these as quickly as needed, which means that nothing is displayed for these elements until XAML is able to catch up. Because many apps use a black background, the result of this problem is usually a large black area, as illustrated in Figure 3-4, and hence the term *panning to black*.

FIGURE 3-4 Panning to black—a common problem with large, virtualized grids in Windows 8.

The app in Figure 3-4 is based on the abstract pictures example mentioned earlier, but as you can see I made the formatting of each item a little more complex. In the new layout, there's a border around each picture and a drop shadow made from placing a rectangle underneath each picture, and the text is now placed on a rectangle on top of the picture itself. Furthermore, I reduced the size of each picture element to fit more items on the screen. The more items and the more complex they are, the more likely it is that XAML might not be able to keep up during fast panning on slower devices. In this case, I have enough items on the screen to cause rendering problems on low-end devices.

Fortunately, this issue has been addressed in Windows 8.1 as well with the `ShowsScrollingPlaceholders` property on `ListViewBase`. Because both `ListView` and `GridView` specialize `ListViewBase`, this addition applies to both lists and grids.

When `ShowsScrollingPlaceholders` is enabled, XAML shows gray placeholder items in place of any items that it cannot render during panning as illustrated in Figure 3-5. This behavior immediately lets the user know that more items are being rendered. This makes for a much better user experience than simply panning through a sea of black. With the placeholders, the user is no longer in doubt about whether more content is coming.

FIGURE 3-5 When you enable ShowsScrollingPlaceholders, XAML renders gray boxes in place of content that has not been rendered.

`ShowsScrollingPlaceholders` is enabled by default, so you don't need to do anything to take advantage of this feature if you're building for Windows 8.1, but you might actually want to disable it in some cases. Why, you ask. Well, let me tell you about another feature in Windows 8.1.

Customized placeholders

Placeholders let the user know that additional items are on their way, but they are generic and don't provide any specific information on what's coming—after all, they are just gray boxes. If the XAML engine cannot keep up during panning, it is typically because each element being rendered is complex. By default, XAML attempts to render each visible item in full, and failing to do so, it renders the placeholder instead until it has the bandwidth to render the missing items.

But what if XAML could render the simple parts of each element before moving on to the more complex parts? The overall time to render the element wouldn't improve, but instead of waiting for the entire element to render, the user would see some useful information for each element during panning and eventually all of each element on the screen.

The `ContainerContentChanging` event defined on `ListViewBase` allows just that. If you attach an event handler to this event, XAML calls your handler as it renders each element. This behavior allows you to render a simple version of the element initially. This could be something like a title or a brief description. From the handler, you can set up another handler to be called during the next phase of rendering the item. You can chain these handlers as you like and thus partition the work needed to render each element as you like. This is essentially an improved version of the placeholders feature, so if you implement this, you want to disable placeholders to avoid rendering both the generic placeholders and your own improved placeholders.

Let me walk you through an example. The app in Figure 3-6 is the same as you saw earlier for the example on placeholders. In the following, I walk you through how you can change this app to use the `ContainerContentChanging` event to provide custom placeholders instead of generic placeholders. The XAML for the app is listed in Listing 3-9.

FIGURE 3-6 This grid app shows a large number of somewhat complex elements and thus might not be able to provide a smooth panning experience on low-end devices.

LISTING 3-9 XAML for the abstract pictures viewer app.

```
<Page
    x:Class="GridVirtualization.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:GridVirtualization"
    xmlns:d=http://schemas.microsoft.com/expression/blend/2008
    xmlns:mc=http://schemas.openxmlformats.org/markup-compatibility/2006
    mc:Ignorable="d">

    <Page.Resources>
        <CollectionViewSource x:Key="PictureSource" IsSourceGrouped="True" Source="{Binding}"/>
    </Page.Resources>

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}" Margin="40">
        <GridView ItemsSource="{Binding Source={Binding PictureSource}}"
                  SelectionMode="None" ShowsScrollingPlaceholders="True" >
            <GridView.ItemTemplate>
                <DataTemplate>
                    <Border BorderBrush="Black" BorderThickness="1" Background="DarkGray">
                        <Grid Width="97" Height="65" Margin="5">
                            <Grid.RowDefinitions>
                                <RowDefinition Height="auto" />
                                <RowDefinition Height="*" />
                            </Grid.RowDefinitions>
```

```
                        <Rectangle x:Name="dropShadow" Fill="Black" Opacity="0.8"
                                   Width="94" Height="62" Margin="5,5,0,0"/>
                        <Image Grid.Column="0" Source="{Binding Location}"
                               VerticalAlignment="Top" HorizontalAlignment="Left"
                               Width="94" Height="62"/>
                        <Rectangle x:Name="textBackground" Fill="Black" Opacity="0.75"
                                   Height="35" VerticalAlignment="Bottom"
                                   Margin="0,0,3,3"/>
                        <StackPanel Grid.Column="1" Margin="3,0,0,6"
                                    VerticalAlignment="Bottom" >
                            <TextBlock Text="{Binding Title}" FontSize="12" />
                            <TextBlock Text="{Binding Tagline}" FontSize="10" />
                        </StackPanel>
                    </Grid>
                </Border>
            </DataTemplate>
        </GridView.ItemTemplate>

        <GridView.ItemsPanel>
            <ItemsPanelTemplate>
                <ItemsWrapGrid />
            </ItemsPanelTemplate>
        </GridView.ItemsPanel>

        <GridView.GroupStyle>
            <GroupStyle>

                <GroupStyle.HeaderTemplate>
                    <DataTemplate>
                        <TextBlock Style="{StaticResource SubheaderTextBlockStyle}"
                                   Text="{Binding Key}" Margin="10"/>
                    </DataTemplate>
                </GroupStyle.HeaderTemplate>

                <GroupStyle.Panel>
                    <ItemsPanelTemplate>
                        <VariableSizedWrapGrid/>
                    </ItemsPanelTemplate>
                </GroupStyle.Panel>

            </GroupStyle>
        </GridView.GroupStyle>
    </GridView>

</Grid>
</Page>
```

The markup is similar to what you saw in Listing 3-8, but I added some additional presentation elements and changed the size of each picture listing. Notice the use of the `ShowsScrollingPlaceholders` in the `GridView` tag. This is explicitly set to `true`, which isn't strictly necessary because that's the default. In other words, you could delete this attribute and achieve the same effect. However, because part of this exercise is to disable the generic placeholders, I included the attribute so that you can see where it goes.

To change this from using generic placeholders to using `ContainerContentChanging`, I need to do a couple of things. First and foremost, I need to hook up the event handler in the `GridView` tag. Second, I need to name the element for each abstract picture so that I can reference it in the event handler. The last thing I need to do in the markup is turn the generic placeholders off so that XAML doesn't spend time rendering those as well as my custom placeholders. The updated XAML is shown in Listing 3-10.

LISTING 3-10 XAML updated to use ContainerContentChanging instead of ShowsScrollingPlaceholders.

```
<Page
    x:Class="GridVirtualization.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:GridVirtualization"
    xmlns:d=http://schemas.microsoft.com/expression/blend/2008
    xmlns:mc=http://schemas.openxmlformats.org/markup-compatibility/2006
    mc:Ignorable="d">

    <Page.Resources>
        <CollectionViewSource x:Key="PictureSource" IsSourceGrouped="True" Source="{Binding}"/>
    </Page.Resources>

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}" Margin="40">
        <GridView ItemsSource="{Binding Source={Binding PictureSource}}"
                  SelectionMode="None" ShowsScrollingPlaceholders="False"
                  ContainerContentChanging="ContainerContentChanging" >
            <GridView.ItemTemplate>
                <DataTemplate>
                    <Border BorderBrush="Black" BorderThickness="1" Background="DarkGray">
                        <Grid Width="97" Height="65" Margin="5">
                            <Grid.RowDefinitions>
                                <RowDefinition Height="auto" />
                                <RowDefinition Height="*" />
                            </Grid.RowDefinitions>

                            <Rectangle x:Name="dropShadow" Fill="Black" Opacity="0.8"
                                       Width="94" Height="62" Margin="5,5,0,0"/>
                            <Image x:Name="picture" Grid.Column="0" Source="{Binding Location}"
                                   VerticalAlignment="Top" HorizontalAlignment="Left"
                                   Width="94" Height="62"/>
                            <Rectangle x:Name="textBackground" Fill="Black" Opacity="0.75"
                                       Height="35" VerticalAlignment="Bottom"
                                       Margin="0,0,3,3"/>
                            <StackPanel Grid.Column="1" Margin="3,0,0,6"
                                        VerticalAlignment="Bottom" >
                                <TextBlock x:Name="titleText" Text="{Binding Title}"
                                           FontSize="12" />
                                <TextBlock x:Name="descText" Text="{Binding Tagline}"
                                           FontSize="10" />
                            </StackPanel>
                        </Grid>
                    </Border>
                </DataTemplate>
            </GridView.ItemTemplate>
```

```
            <GridView.ItemsPanel>
                <ItemsPanelTemplate>
                    <ItemsWrapGrid />
                </ItemsPanelTemplate>
            </GridView.ItemsPanel>

            <GridView.GroupStyle>
                <GroupStyle>

                    <GroupStyle.HeaderTemplate>
                        <DataTemplate>
                            <TextBlock Style="{StaticResource SubheaderTextBlockStyle}"
                                       Text="{Binding Key}" Margin="10"/>
                        </DataTemplate>
                    </GroupStyle.HeaderTemplate>

                    <GroupStyle.Panel>
                        <ItemsPanelTemplate>
                            <VariableSizedWrapGrid/>
                        </ItemsPanelTemplate>
                    </GroupStyle.Panel>

                </GroupStyle>
            </GridView.GroupStyle>
        </GridView>

    </Grid>
</Page>
```

The updated XAML markup hooks up an event handler for the `ContainerContentChanging` event, so obviously we need to implement that.

Before I go through the implementation though, let me stress that this is called for each element in the grid as part of rendering. This affects the time XAML spends creating display frames—recall from Chapter 2, that XAML breaks down the layout of the user interface in a number of frames. If creating the frames requires a lot of work, XAML will not be able to keep up with user interactions and responsiveness will suffer.

Take a look at the implementation of the initial event handler in Listing 3-11. It doesn't do a lot, which is intentional because this method is called synchronously for each element. Whatever complex work you need to do to render the elements should be done at later stages, as I'll discuss momentarily. Initially, `ContainerContentChanging` informs the XAML engine that the current element has been handled. This is an important optimization that tells XAML that it can skip its usual work for this element.

It then digs out all the references for the UI elements it needs to access and sets the `Opacity` of those to 0. This instructs the XAML engine to skip those elements during rendering because they are not visible. The elements are still part of the visual tree, however. The border element is left unchanged, so it still renders for each element. In other words, the border is the first visible piece in my custom placeholder. Finally, the handler sets up an additional callback for this element, signaling that there's more work to do before rendering is complete. Unlike the first handler, subsequent callbacks are invoked asynchronously when XAML has the bandwidth to render additional details.

LISTING 3-11 Implementation of ContainerContentChanging.

```
private void ContainerContentChanging(ListViewBase sender,
                                      ContainerContentChangingEventArgs args)
{
    // For improved performance, set Handled to true
    // so the app does not set content on this item
    args.Handled = true;

    var templateRoot = (Border)args.ItemContainer.ContentTemplateRoot;
    var textBg = (Rectangle)templateRoot.FindName("textBackground");
    var title = (TextBlock)templateRoot.FindName("titleText");
    var desc = (TextBlock)templateRoot.FindName("descText");
    var dropShadow = (Rectangle)templateRoot.FindName("dropShadow");
    var image = (Image)templateRoot.FindName("picture");

    textBg.Opacity = 0;
    title.Opacity = 0;
    desc.Opacity = 0;
    dropShadow.Opacity = 0;
    image.Opacity = 0;

    args.RegisterUpdateCallback(ShowText);
}
```

You can chain as many callbacks as you like, but if you need to chain more than a few, your elements might be too complex to be displayed efficiently in a grid or a list. This sample sets up a chain of three event handlers. The next handler displays the title and the description for the picture. The final handler displays the image and the drop shadow. The two remaining callbacks are shown in Listing 3-12.

LISTING 3-12 Remaining callbacks for the movies app.

```
private void ShowText(ListViewBase sender, ContainerContentChangingEventArgs args)
{
    var picture = (AbstractPicture)args.Item;
    var templateRoot = (Border)args.ItemContainer.ContentTemplateRoot;

    var textBg = (Rectangle)templateRoot.FindName("textBackground");
    var title = (TextBlock)templateRoot.FindName("titleText");
    var desc = (TextBlock)templateRoot.FindName("descText");

    title.Text = picture.Title;
    desc.Text = picture.Description;

    textBg.Opacity = 1;
    title.Opacity = 1;
    desc.Opacity = 1;

    args.RegisterUpdateCallback(ShowPicture);
}
```

```
private void ShowPicture(ListViewBase sender, ContainerContentChangingEventArgs args)
{
    var picture = (AbstractPicture)args.Item;
    var templateRoot = (Border)args.ItemContainer.ContentTemplateRoot;

    var dropShadow = (Rectangle)templateRoot.FindName("dropShadow");
    var image = (Image)templateRoot.FindName("picture");

    image.Source = picture.Location;

    dropShadow.Opacity = 1;
    image.Opacity = 1;
}
```

They both follow the same general idea. Each callback surfaces additional detail about the item being handled. Notice how the actual data item is passed in the `Item` property of `args`. In `ShowText`, this is used to retrieve the title and description of the picture and expose those. When data is ready, the corresponding UI elements are made visible by setting the `Opacity` property to 1, thus rendering the data during that phase.

`ShowPicture` doesn't set up another callback, so it completes the chain of callbacks. When `ShowPicture` completes, the current item is rendered completely.

With the preceding implementation, each element renders as follows:

1. Border
2. Title and description on top of a rectangle
3. Picture and drop shadow

`ContainerContentChanging` offers a great way to customize rendering for complex elements in grids and lists during panning. However, keep a couple of things in mind when using this approach. Splitting the rendering into multiple phases adds overhead to the work needed for each element, so you might need to try various approaches to get the balance right. The first method in the chain of methods is special. It must make sure to set the `Handled` property on the `ContainerContentChangingEventArgs` instance, and it should execute as quickly as possible because it is invoked synchronously for each element. The work you do in this method adds latency to the act of rendering the grid, so keep it as brief as possible.

Before you use `ContainerContentChanging`, you should simplify the layout for each element as much as possible. The goal is to make panning fast and smooth. The best way to do that is to reduce the amount of work XAML has to do for each element, reduce the number of elements, or both. Once you have done that, you can use `ContainerContentChanging` to partition the remaining work if needed.

Cache information to improve transitions

It doesn't matter if your app does heavy calculations, performs complex queries, or downloads truckloads of data from the cloud—retrieving the necessary data can be time consuming. As discussed, you don't want your app to wait for the data to become available because that leads to a bad user experience. Furthermore, when the data is finally present, you might want to hold on to it to avoid the overhead of calculating or retrieving the data again. Of course, holding on to data is not free either, so you need to come up with a good strategy for what you want your app to cache and how long data should be cached. Remember, a cache without an expiration policy is just a fancy word for a *memory leak*.

Getting caching right can be tricky. If you don't get it right, you might cause your app's memory usage to balloon, with no noticeable gain in performance. Before you implement caching, make sure you collect the data to support your decision. Once you implement your caching, you need to measure and adjust the implementation as necessary.

When it comes to caching, you're generally looking for scenarios where getting the required data takes longer than desired *and* the bulk of the data is reused. If the data isn't reused, caching it makes no sense regardless of how long it takes to retrieve it. If some of the data is reused, you need to consider the hit/miss ratio. If your app has to hold on to large amounts of data to facilitate caching, you might run into problems because of excessive memory usage, as described in Chapter 2. You might need to experiment with different approaches to get the balance right. As I said, getting caching right isn't easy.

Once you identify scenarios that might benefit from caching, consider the following:

- Is caching already available? Anything your app retrieves over HTTP can be cached, and the protocol has its own scheme for controlling caching. To use HTTP caching, you need to be able to control how data is handled on the server. If that's an option, HTTP caching is a good choice because it provides an easy-to-use, well-understood mechanism for caching resources, and best of all, it is completely transparent to the app.
- Can results be cached for multiple users? Some calculations and queries are common across many or all the user sessions. These can be executed and cached on the server instead of letting each client do the work. This can eliminate the work in the majority of cases if many scenarios use the same data. However, it obviously adds additional network latency to the equation. For this to be attractive, the cost of doing the work locally must exceed the latency.
- Does data retrieval follow a pattern? In some scenarios, there's a good chance you can predict subsequent queries or calculations based on the current action. If that's the case, you can use this to prepopulate your app cache. This works for both local caches and when interacting with a back end. For example, if your app uses a local cache, the current action can trigger a background task that proactively calculates the next expected result set and stores that. A `CancellationToken` can be used to cancel the action if the prediction turns out to be incorrect. For back-end-based solutions, the server can return the requested data as well as data likely to be returned for the next query or queries. Doing so reduces the number of network round trips between the app and the back end for each request.

- Is the data valid beyond the current user interaction? If the user closes and restarts the app, will it need the same data again? If the app needs the same data, the cache should be persisted locally and read upon restart. There are various options for persisting data. For data that is read in bulk, just serializing the objects is a good approach. If data is read back selectively, a local database such as SQLite might be a better option. Persisting and releasing the data can also reduce the memory usage of the app. This works well if different parts of the app use distinct sets of data.

Releasing memory on demand

The problem with holding on to too much data is that the memory usage of the app grows to undesirable levels. This affects overall system performance and increases the risk of the app being terminated by the operating system. In a .NET app, you don't manage memory usage explicitly—instead, the Common Language Runtime (CLR) allocates and frees memory based on the lifetime of objects. As long as your app holds references to objects, they are not reclaimed during garbage collection and the associated memory segments are not released.

Weak references allow your app to release memory on demand. When your app holds a strong reference to an object or a graph of objects, these are considered in use and thus not reclaimed during garbage collection. A weak reference allows your app to hold on to an object or a collection of objects while still permitting the objects to be collected during garbage collection. By using weak references, your app can hold on to data and release it automatically if needed. Listing 3-13 shows how to use the `WeakReference` class to hold on to a list of items.

LISTING 3-13 Using a weak reference to hold on to a list.

```
// Instance field
WeakReference<List<SomeType>> weakRef;

…
// Some place in the code
…

var list = GetListOfObjects();

// Create a weak reference to the list
weakRef = new WeakReference<List<SomeType>>(list);

…
// Somewhere else in the code
…

List<SomeType> list = null;
if(!weakRef.TryGetTarget(out list))
{
    // The list was reclaimed during GC, so we have to re-create it
    list = GetListOfObjects();
}
```

The code is straightforward. `weakRef` is an instance field on some class that uses a `List<SomeType>`. Initially, I create a list using a strong local reference. As long as that reference is alive, the list cannot be reclaimed. Next, I assign a `WeakReference` to the field on the current object that points to the same list. When the original strong reference is no longer valid, the weak reference holds on to the list, but it will not prevent the list from being reclaimed if a garbage collection needs to reclaim the associated memory.

Because the list might have been reclaimed, I need to check if the `weakRef` is still referencing it before I can access the list. This is done through the `TryGetTarget` method. If it returns `false`, it means that `list` could not be set and I need to re-create the list. If it returns `true`, the `list` reference is set through the `out` parameter. This makes the original list object accessible and prevents the list from being collected.

You can use weak references in your data model to hold on to data as needed. Weak references are useful for data that can be re-created quickly. For data retrieved from the network, weak references might not be the best solution because garbage collections trigger the need to download the data again. For such data, it is better to persist it locally. Once that's done, you can use weak references to hold on to in-memory copies of the data.

Handling media

Including media in your app is a compelling way to enhance the user experience. XAML makes it easy to include video, audio, and images in your application. Just pick the right UI element, set the source, and voila, XAML and WinRT take care of all the heavy lifting of decoding and presenting the media appropriately.

For example, to include video in your app, you just drop a `MediaElement` on your user interface and set the source to a local or remote video. Listing 3-14 shows the simple XAML you need to write to do this. Using audio and images are equally straightforward.

LISTING 3-14 Including media in your app is simple.

```
<MediaElement x:Name="mediaSimple"
              Source="Videos/video.mp4"
              Width="400" />
```

While including media in your app is deceptively easy, you can do a couple of things to ensure proper and efficient playback. In the following sections, I go through a number of things to keep in mind when handling media content:

- Playing video
- Displaying images
- Playing audio
- Releasing resources

Playing video

WinRT supports a number of popular video formats. Selecting the right format can have a great impact on both performance and power utilization. For videos, the recommended encoding is H.264, and for local playback, the preferred file container is MP4. H.264 decoding is accelerated through most recent graphics hardware. Although hardware acceleration for VC-1 decoding is broadly available, for a large set of graphics hardware on the market, the acceleration is limited in many cases to a partial acceleration level, rather than a full-steam-level hardware offload.

Apart from picking the optimal format, there are a number of things to keep in mind when playing video. You should prefer videos that are scaled to the display resolution. Decoding video takes a lot of memory and GPU cycles, so choose a video format close to the resolution it will be displayed at. There is no point in using the resources to decode high-definition video if it's going to get scaled down to a much smaller size. If the back end offers multiple formats, make sure to pick the proper format.

Defer processing of media content

By default, `MediaElement` retrieves one of the initial frames to display before the video is played. Opening the file, decoding the frames, and picking an image is not a trivial operation. If the user never plays the file, this is a lot of work to do up front. Fortunately, `MediaElement` has a `PosterSource` property that allows you to specify an image that can be used instead. Note, however, that this property is useful only if `Source` is not set in the markup as well. To use `PosterSource`, set it to some image and then dynamically set the `Source` as needed at a later point. The `PosterSource` image is displayed while the work to retrieve an appropriate frame from the video is in progress. Once a frame has been identified, it is cached to reduce the necessary work the next time the video is used. The `PosterSource` is also displayed if the `MediaFailed` event fires.

If your app displays a collection of videos, the use of `PosterSource` can have a significant effect on the user experience because XAML doesn't have to retrieve and display an image from each of the files. Furthermore, if your app handles multiple videos, you should defer setting the source until needed. When you set the source, XAML does work to prepare the video for playback. If you have many videos, that's a significant amount of work, and chances are that most of it will not be needed because the user might not play all the videos. By deferring this work, the app can present the users with the available options much faster.

Controlling playback

In Windows 8, the `MediaElement` tag didn't provide any controls to manage the playback of video or audio by default, so you had to implement those yourself. If you want to customize the controls, this approach is still viable, but for all those cases where you just want to play back media content, it is a little tedious to implement this. Fortunately, the `AreTransportControlsEnabled` property was added to `MediaElement` in Windows 8.1. If you set this to `true`, you get a set of standard media controls.

The controls include everything you need for controlling regular playback of both audio and video content. More importantly, if you enable `AreTransportControlsEnabled`, the `MediaElement`

automatically does the right thing in many cases. For instance, the controls are displayed on demand and automatically removed when no longer used. This optimizes the playback of video because XAML doesn't have to composite the controls on top of the video frames.

Full-screen playback

If video playback is an important part of your application, you should prefer full-screen playback over embedded playback. Full-screen playback enjoys several optimizations that provide for a better user experience as well as better power utilization. In Windows 8, you had to make sure the size of the `MediaElement` matched the display resolution. In Windows 8.1, you can use the `IsFullWindow` property on `MediaElement`. If you set this to `true`, XAML makes sure your video is played at the proper resolution and you don't have to set the dimensions on the `MediaElement`.

To further optimize playback, make sure no elements are rendered on top of the `MediaElement`. If you're implementing your own playback controls, make sure they are removed when no longer needed. Rendering elements on top of the `MediaElement` forces composition to happen, which adds a bit of overhead and prevents some internal optimizations. If you're using `AreTransportControlsEnabled`, you don't have to do anything else because it automatically removes the controls as needed.

A common problem in this regard is progress indicators. Retrieving media content can take some time, so showing a progress indicator can be a good idea to let the user know that the app is working on retrieving content. However, if you do that, you must make sure to disable the progress indicator by setting the `IsActive` property to `false` once the media is ready. If you don't do that, the progress indicator continues to run its animation even if it isn't visible. That hurts performance and reduces battery life significantly because WinRT cannot apply its optimizations in this area. If the media is not full screen, this also causes overdraw.

A good way to detect this problem is to enable `DebugSettings.EnableRedrawRegions` and run the app. This causes the Desktop Window Manager (DWM) to display sections in different colors each time it updates the display. If you have a progress indicator working in the background, you'll see the DWM compositing the dots along with the video.

One thing to be aware of when using `EnableRedrawRegions` is that this works on the DWM level, so you might see other applications trigger DWM work as well. Somewhat surprisingly, the artifacts of this work show up even though your app uses the entire screen. If you see any DWM updates you didn't expect, this might be caused by other applications. To reduce the noise, close or minimize all other applications while you test with this property enabled.

Embedded playback

For some apps, full-screen video is not really an option. If your app edits video, you might want to show controls and data alongside the media content. If the media content is not available in a decent resolution, showing it in full screen leaves a lot to be desired. Whatever the reason, there are scenarios where displaying full-screen video is not the way to go. For those cases, you can embed the `MediaElement` on the page.

If you embed media on the page, WinRT is not able to apply all the optimizations available for full-screen display. However, the same tips still apply. To optimize performance of the playback, you don't want to place any XAML elements on top of your `MediaElement`. If you implement your own media controls, make sure to place these next to the `MediaElement` instead of on top as illustrated in Figure 3-7. If you do place them on top, make sure to remove them after a while, similar to what `AreTransportControlsEnabled` does.

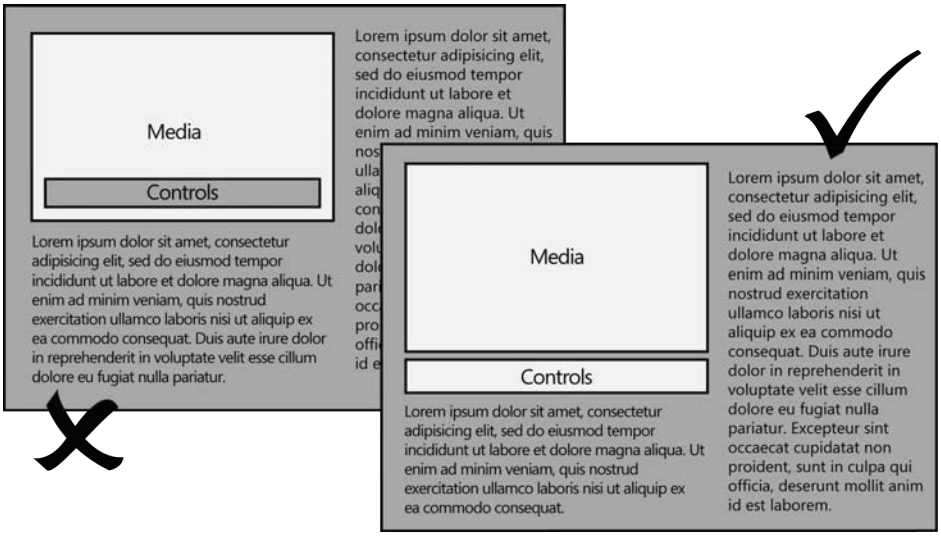


FIGURE 3-7 Place controls next to MediaElement to optimize playback.

Displaying images

Images often play an important part of the presentation in Windows Store apps. As discussed in Chapter 2, handling images is far from trivial, and the amount of work needed to handle an image increases with the size of the image.

The optimal approach is to make sure all your image resources are stored in the size needed. Using correctly sized images affects performance noticeably in many cases. If the images are downloaded dynamically, smaller images load faster. Smaller images are also decoded and transferred to the GPU faster, so there are several reasons why getting the size right is important.

If you cannot specify the native size of the images, the next best thing to do is to make sure your app decodes the images in the proper size. If you're using the `BitmapImage` class, you can use the `DecodePixelWidth` and `DecodePixelHeight` properties to specify the decode size. This affects decoding itself and the amount of data transferred to the GPU, both of which can affect performance significantly.

Listing 3-15 is not very effective because it forces the entire image to be loaded, decoded, and then resized. This increases memory usage significantly and might exhaust GPU bandwidth as well.

LISTING 3-15 The ineffective way to handle a large image—don't do this.

```
<!-- Don't do this -->
<Image Source="ms-appx:///Assets/highresPicture.jpg" Width="300" Height="200"/>
```

Listing 3-16 does the right thing by specifying the height and width for the decoded image. This reduces the size of the decoded image and thus the memory usage, and it decreases the amount of data sent to the GPU. Be sure to either scale your images to the desired size or set the decode size.

LISTING 3-16 The proper way to handle a large image.

```
<Image>
    <Image.Source>
        <BitmapImage UriSource="ms-appx:///Assets/highresPicture.jpg"
                     DecodePixelWidth="300" DecodePixelHeight="200"/>
    </Image.Source>
</Image>
```

Also, to prevent images from being decoded more than once, assign the `Image.Source` property from an `Uri` rather than using a memory stream. The XAML engine can associate multiple `Bitmaps` based on the same `Uri` with a single decoded image. There's no similar optimization for memory streams, so if you use a `Bitmap` based on a memory stream as the source of multiple images, XAML has to decode the image multiple times.

Specifying `DecodePixelWidth` and `DecodePixelHeight` is useful for handling images efficiently. However, if all you really need are thumbnails, the `StorageFile.GetThumbnailAsync` is a better option because it can read the thumbnails cached by the file system. Listing 3-17 shows how to use `GetThumbnailAsync` to retrieve thumbnails of images in a folder.

LISTING 3-17 Using GetThumbnailAsync to retrieve a thumbnail of an image file and assign it as the source of an image in XAML.

```
var picker = new FileOpenPicker();
picker.FileTypeFilter.Add(".jpg");
picker.FileTypeFilter.Add(".jpeg");
picker.FileTypeFilter.Add(".png");
picker.SuggestedStartLocation = PickerLocationId.PicturesLibrary;

var file = await picker.PickSingleFileAsync();

var fileThumbnail = await file.GetThumbnailAsync(ThumbnailMode.SingleItem, 64);

var bmp = new BitmapImage();
bmp.SetSource(fileThumbnail);

image.Source = bmp;
```

Listing 3-17 creates an instance of `FileOpenPicker`, adds some commonly used image formats, and asks the user to pick a file. `GetThumbnailAsync` is then used to create a `StorageItemThumbnail` with a requested size of 64 for the picked file.

Using thumbnails is appropriate for apps that let the user browse any kind of media. `GetThumbnailAsync` can retrieve thumbnails for images, audio files, videos, documents, and even folders containing these types by specifying the proper `ThumbnailMode`. If your app needs only thumbnails, this approach is preferable to using `DecodePixelWidth` and `DecodePixelHeight`.

Playing audio

WinRT also supports numerous audio formats, but for the best results you should prefer Advanced Audio Coding (AAC) and MP3, because they offer the best performance. If your app plays short, low-latency sound effects, use WAV files with uncompressed Pulse-Code Modulation (PCM) data to reduce processing overhead that is typical for compressed audio formats.

When playing sound, it might make sense to turn off the display after a while. However, if your app shows information related to the playback, you can instruct Windows to leave the display on as described in the upcoming "Display deactivation" section. Remember to call `DisplayRequest.RequestRelease` if the playback stops for some reason, so your app doesn't keep the display needlessly on.

Windows also supports hardware offload of audio playback. To enable this, your app must set `MediaElement.AudioCategory` to either `ForegroundOnlyMedia` or `BackgroundCapableMedia`. Hardware audio offload optimizes audio rendering, which can improve both functionality and battery life.

Releasing resources

Media files can be huge, and consequently accessing them increases the memory usage of your app significantly. To use media content efficiently, you must release the resources as soon as possible when you're done using them.

Streams

An obvious example of resources is *streams*. Streams can be used to read and write media content if you want to do more than the basics provided by the media UI elements. For instance, your app could read media content from a stream in order to modify it. If you read or write media content through a stream, make sure to close the stream as soon as the app is done reading from it.

Listing 3-18 shows an example where an app reads a picture from a file, modifies the content, and writes the updated picture to the stream of a bitmap. Notice how the streams are used inside `using` blocks. This, of course, implicitly calls the `Dispose` method at the end of the block and thus releases the resources as necessary.

LISTING 3-18 Reading and writing media content via streams.

```
// Get picture from library
var folder = KnownFolders.PicturesLibrary;
var file = await folder.GetFileAsync("pic.png");

// Read stream from file
var streamRef = RandomAccessStreamReference.CreateFromFile(file);
using (var fileStream = await streamRef.OpenReadAsync())
{
    // Decode format
    var decoder = await BitmapDecoder.CreateAsync(fileStream);
    var frame = await decoder.GetFrameAsync(0);

    // Get pixels as byte array
    var pixelDataProvider = await frame.GetPixelDataAsync();
    var pixels = pixelDataProvider.DetachPixelData();

    // Change picture by manipulating the byte array
    ManipulatePixels(pixels);

    // Create bitmap and write the modified pixels
    var bitmap = new WriteableBitmap((int)frame.PixelWidth, (int)frame.PixelHeight);
    using (var stream = bitmap.PixelBuffer.AsStream())
    {
        stream.Write(pixels, 0, pixels.Length);
    } // this closes the stream

    // Display modified picture
    image.Width = frame.PixelWidth;
    image.Height = frame.PixelHeight;
    image.Source = bitmap;
} // this closes the file stream
```

Display deactivation

If you leave your device idle for a while, Windows dims and eventually turns off the display to preserve power. That's rarely the desired behavior when playing media content such as video. The user should be able to watch videos without having to tap the display every now and then. Yes, this sounds silly, but believe me I have seen media apps that didn't do this.

To prevent Windows from turning off the display during media playback, your app must call the `DisplayRequest.RequestActive` method. This informs Windows it should not turn off the display to preserve power. That ensures the user can enjoy the media playback uninterrupted, which is exactly what you want as long as the user is actually watching the media.

If the user pauses the media or watches the content to the end, or the app encounters a media playback error, the display is no longer required and your app should call `DisplayRequest.RequestRelease` to let Windows know that the display can be turned off again when idle. If your app doesn't do that, the display will drain the device's battery, and possibly leave the user with an unusable device.

If your app uses `AreTransportControlsEnabled` as described earlier and the playback is full screen, or if your app sets the `IsFullWindow` property, XAML automatically calls `DisplayRequest.RequestActive` and `DisplayRequest.RequestRelease` as appropriate.

Summary

Making the key scenarios of your app perform well is paramount for a good user experience and positive ratings on the Windows Store. Unless your app accurately predicts tomorrow's stock prices, users don't want to wait for your app to respond. Of course, if your app can predict the market, I'm pretty sure most users will accept a little waiting and you can ignore most of the advice in this chapter. For the rest of us, here's a summary of what I covered.

Designing for performance means getting rid of everything that isn't essential for the experience. If it isn't needed, don't do it. If it's needed later, do it later. If it takes a long time to do, save the result.

You need to identify the key scenarios and the resources needed to implement those. Once you identify the resources, you need to prioritize them so that your app can retrieve and handle the most important assets first. Your goal should be to optimize for responsiveness. Make sure the user can interact with the important parts of your app with as little delay as possible. Everything else can be handled once the app is responsive.

If your app needs to retrieve data from the network, you need to address the fact that you cannot guarantee a fast and steady rate of data. If your app is not useful without a fast network connection, you're cutting off a lot of users. You need to design your app so that it handles poor network connectivity. Windows offers several tools to help you do that.

The `ContentPrefetcher` class introduced in Windows 8.1 lets your app subscribe to automatic content updates even when it isn't running. This reduces the risk of cached data being stale. Launching with local content is the only way to guarantee a specific performance.

Asynchronous I/O lets your app retrieve data without blocking the UI thread. This keeps your app responsive, but it doesn't change the fact that I/O might take a long time, asynchronous or not. You need to design around this. If your app retrieves data as part of launch or navigation, you need to provide a fallback option in the case of a slow network connection. The `WindowsRuntimeSystemExtensions` class provides a series of extension methods that bridges the gap between the WinRT APIs and the .NET Task Parallel Library. This means you can wait for and cancel asynchronous calls using the familiar mechanisms of the .NET library.

A lot of work went into making Windows 8.1 even better at presenting large datasets to the user efficiently. The grid and list controls were both updated to support faster virtualization. If you're building for 8.1, you get these benefits automatically, but if you're upgrading your Windows 8.0 app, you should make sure your grids use the `ItemsWrapGrid` panel. This panel supports item-level virtualization, which improves performance significantly for large groups in a grid. Furthermore, it allows you to display generic placeholders for items that are still being rendered. To improve the experience beyond that, you can even implement your own placeholders, which allows you to render each element in phases.

Media content is an integral part of many Windows apps. If your app handles images, video, or audio, you can do several things to improve the experience. The most important part is to make sure you're not forcing XAML to do more work than necessary for each item. For images and video, that means scaling resolution as appropriate. For videos, that means deferring as much work as possible and preferring full-screen playback over embedded playback. Because media content is demanding on the system, your app needs to release resources as soon as they are no longer needed.

To help you measure the performance of your code and ensure the techniques discussed in this chapter are working in your favor, you should instrument your code, and doing so is the topic of Chapter 4, "Instrumentation."

This page intentionally left blank

Index

Symbols and Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z