O'REILLY®

Releasing HTML5 Games for Windows 8

FROM THE WEB TO WINDOWS 8 WITH EASE

Jesse Freeman

www.allitebooks.com

O'REILLY®

Releasing HTML5 Games for Windows 8

Windows 8 presents an incredible opportunity for distributing and monetizing HTML5 games, and this guide shows how you can profit from it. You'll learn everything you need to know about porting your original web-based JavaScript game to the new "touch-first" version of Windows, as well as several options for selling your game in the Windows Store.

Windows 8 is a big leap forward for developers because it treats HTML5 as a first-class citizen, alongside C# and C++. Interactive development expert Jesse Freeman explains how Windows 8 works, gets you started with Visual Studio Express (it's free!), and uses a case study to show you how to port an HTML5 game with ease.

- Learn which games and JavaScript libraries work best on Windows 8
- Adjust artwork for different screen resolutions and Windows 8 features
- Accommodate mouse, keyboard, touch, and other game controls
- Optimize your game to run well on any Windows 8 device
- Understand the steps for publishing your game to the Windows Store
- Explore fixed price, trial mode, ad support, and in-app purchase options
- Use a web-first workflow to ensure your game runs on many other platforms

"This book is succinct and to the point. It aids in getting your JavaScript/HTML game project running in a Windows 8 environment fast."

—Dave Voyles
Product Development Engineer, Comcast

Jesse Freeman, for more than 13 years, has been on the cutting edge of interactive development with a focus on the Web and mobile platforms. As an expert in his field, Jesse has worked for Microsoft, MLB, HBO, the New York Jets, VW, Tommy Hilfiger, Heavy and other organizations.

Twitter: @oreillymedia
facebook.com/oreilly

US $19.99 CAN $20.99
ISBN: 978-1-449-36050-4

www.allitebooks.com

Releasing HTML5 Games for Windows 8

Jesse Freeman

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

www.allitebooks.com

Releasing HTML5 Games for Windows 8

by Jesse Freeman

Copyright © 2014 Jesse Freeman. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Rachel Roumeliotis
Production Editor: Christopher Hearse
Proofreader: Christopher Hearse

Cover Designer: Randy Comer
Interior Designer: David Futato
Illustrator: Rebeca Demarest

November 2013: First Edition

Revision History for the First Edition:

2013-10-31: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449360504* for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Releasing HTML5 Games for Windows 8*, the image of an anglerfish, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-36050-4

[LSI]

www.allitebooks.com

I would like to dedicate this book to Phyllis Straus who was my advisor and close friend at FSU. The news of her passing broke my heart and she will truly be missed by me and all the lives she had touched, influenced, and directed to reach their full potential.

www.allitebooks.com

www.allitebooks.com

Table of Contents

www.allitebooks.com

www.allitebooks.com

www.allitebooks.com

www.allitebooks.com

Preface

Designing, building and publishing games is not an easy task. One of the most challenging parts about publishing a game is how to distribute and monetize it. This is even more daunting when it comes to HTML5 games since cross browser compatibility and viable distribution channels are still maturing. Windows 8 offers an incredible opportunity for independent game developers looking to distribute and monetize their HTML5 based game. This book will cover everything you need to know about porting over your web based JavaScript game to Window 8, how to integrate support for WinJS (the JavaScript communication layer to the native OS), and how to publish and sell your game on the Windows 8 Store. This book is a must read for anyone looking to seriously develop HTML5 games!

Audience

While this book was designed for web and game developers who already have a HTML5 game ready to port over to Windows 8, I did my best to cover useful information for anyone simply interested in publishing an HTML5 game to the Window Store. While this book may not directly help you if this is your first time making an HTML5 it will give you a detailed overview of everything that goes into running HTML5 games on Windows 8.

Assumptions This Book Makes

This book assumes that you have a working knowledge of HTML, JavaScript and have made a HTML5 game before. I also assume you already have at least one game ready to port over. While there is a chapter that covers porting an existing open source HTML5 game over you will need to have some working knowledge of the underlying language and technology needed to successfully publish any HTML5 game.

Conventions Used in This Book

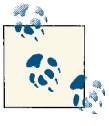
The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

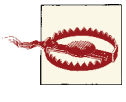
`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.



This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Safari® Books Online

Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technol-

ogy, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/HTML5-Games-Windows-8*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

First and foremost, I would like to thank my wife and sons for all their support while I was making this book. I'd also like to thank my parents and family for all their help and support over the years. I also have a lot of respect for all the thought leaders in the development community who continue to inspire me such as Christer Kaitila, Keith Peters, Chuck Freedman, Sean McCracken, Joel Hooks, Brendan Lee, Scott Penberthy, Seb Lee-Delisle, Rich Shupe, Jobe Makar, and especially Richard Davey who keeps pushing me to make better games.

Thank you as well to Rachel Roumeliotis from O'Reilly Media, Inc. for providing me with this opportunity and to David Isbitski and everyone else at Microsoft who helped make this possible. I also couldn't have done this book without the help from my amazing tech editor Dave Voyles.

Finally I wanted to give a special thanks to Iñaki Diaz for helping me create the pixel art in my games.

CHAPTER 1

Getting Started with Windows 8

In this chapter, I introduce you to Windows 8, talking about what you need in order to get up and running. I'll cover everything from why I chose Windows 8 and how to install it to an introduction to the Windows 8 operating system. Some of the information here may be helpful even if you are an experienced developer, especially as I cover how to use Windows 8 and some of its new paradigms.

Why Windows 8

Windows 8 is the latest version of the Windows operating system from Microsoft. It marks a radical departure from the past and pushes the OS into the post-PC era of tablets and ultra-portable laptops, while still supporting older desktops and laptops. If you think about it, Windows 8 is a huge move forward in terms of approachability for developers due to the three main ways to write software for it: C++, C#, and HTML5. This book focuses on the latter and, more importantly, on helping you port an existing game over to the platform.

What got me excited about Windows 8 is that HTML5 is a first-class citizen. You can literally take a game that runs on the Web in a browser, get it up and running on Windows 8 quickly with very little modification, and start selling it in the Windows Store. Right now it is hard to monetize HTML5 games online, but Windows 8 provides a unique opportunity, which I discuss throughout the book.

I designed this book to be framework agnostic. While I prefer to use the Impact framework for my own games, I have seen games built with all kinds of JavaScript libraries, from homegrown frameworks to DOM-based implementations. It's challenging to try and address all the different development approaches, so I focus on the things you need to consider when porting over your game, along with some code examples to get your brain thinking in terms of how HTML5 development is done on Windows 8. The rest of this chapter will discuss getting set up and ready to begin the porting process.

If you are planning on building an HTML5 Windows 8 game from scratch, I suggest reading through the entire book. This will give you a great overview of how Windows 8 development works. In the final chapter, I discuss simultaneous development workflows for the Web and Windows 8.

What You Need

There are three things you will need in order to start doing development on Windows 8: a copy of Windows 8 plus a PC or Mac that can run it, a copy of Visual Studio Express, and a developer account.

There are a few optional things you may want to consider having as well: test hardware (which I will talk about in more detail later), a touchscreen monitor or touch-enabled device, and a copy of Visual Studio Pro or an MSDN subscription.

The first three items are self-explanatory. In order to do development for Windows 8 you will need to be on Windows 8 and have the right tools. Luckily, when it comes to doing Windows 8 development, you can also test on the same computer you use when coding and/or take advantage of the built-in simulator to test out various configuration and resolution types, if you don't have the test hardware in person.

Also make sure to check out the passage at the end of the chapter about BizSpark and DreamSpark which may be great ways to get free copies of Windows 8 as well as Visual Studio.

Windows 8 Hardware

Chances are good that, if you have already created an HTML5 game, you have a computer that can be used for development. In that case, all you need to do is get a copy of Windows 8. If you are on a Mac, I have some instructions on how to set that up via Boot Camp in the next section. For now I want to focus on the types of Windows 8 hardware and what it means for you as far as testing and developing.

Devices

Windows 8 has an expansive list of supported hardware, both new and old, which you should be familiar with. It's important to remember that you still need to support legacy systems, as well as the new form factors and input devices released along with the new OS. Here is a quick rundown of the typical types of devices your game could run on:

- **Desktop** – This is a standard PC. It is usually a tower and a freestanding monitor, or an "all-in-one" computer. Typically these have powerful video cards, fast CPUs, lots of RAM, and a keyboard as the default input device.

- **Laptop (Non-touchscreen)** – This is your standard laptop. Any laptop that ran Windows 7 should run Windows 8, but there are a few things to keep in mind with these older devices. Laptops usually have slower GPUs/CPUs, less RAM, and have a track pad plus a keyboard as the default input devices.
- **Laptop (Touchscreen)** – Most if not all new Windows 8 laptops are coming out with touchscreens. Windows 8 is a "touch-first" OS, so these new ultra-thin, touch-enabled laptops will work great with Windows 8 out of the box. Plus, the extra touchscreen allows you to do basic touch testing for your game. If you are in the market for new hardware for doing Windows 8 development, I highly suggest making sure your next laptop has a built-in touchscreen.
- **Convertible** – This is a new category of ultra-portables that are tablets that can dock with a keyboard to become a laptop. These offer all the same conveniences of a tablet and laptop at the same time. They have touchscreens as their primary input and usually have very low-powered CPUs with built-in GPUs. These also come in two flavors: Windows 8 and Windows RT, which I will talk about in the next section.
- **Tablet** – Windows 8 was designed to run amazingly on a tablet. These are similar to the convertibles but may not have any way to dock with a keyboard to become a "clamshell" form factor. The flagship of these types of devices would be Microsoft's Surface. Tablets offer lots of options for input from touch to pen, even accelerometer, and can also be connected to a traditional mouse, keyboard, or game controller as well.

This quick overview of the Windows 8 device landscape can help inform you on some of the complexities that may arise when trying to get a game to run on all these different devices with multiple types of input. Over the course of the book, I'll dig into some of these special use cases and work through solutions to help make developing HTML5 games for Windows 8 as straightforward as possible.

Windows 8 Versus Windows RT

After taking a look at the different types of devices ready to run Windows 8, let's go over the difference between Windows 8 and Windows RT. Windows RT is a scaled down version of Windows 8 that was designed to run on ARM devices. On the surface, Windows RT looks identical to Windows 8 and even has a desktop mode, but Windows RT does not allow "classic" Windows apps to run (with the exception of a few Microsoft apps bundled with the device). The user can only run Modern apps, which are distributed through the Windows Store. As an HTML5 developer, this has little to no impact on your game, with the exception of slower performance and needing one as a testing device.

The only reason I bring this up is that ARM devices generally have lower power and performance when compared to Intel-based devices, so if you intend to target Windows

RT (which happens by default when you compile your app), you will need to take that performance hit into consideration.

I highly suggest picking up at least one Windows RT device for testing. They usually start under $500, such as Microsoft's Surface, and other OEMs also offer tablets as well as convertibles. It's also important to note that you will not be able to run Visual Studio on Windows RT, making this device solely for testing or personal use, so don't expect to be doing much development on it.

Introduction to Windows 8

I'm not going to spend a lot of time talking about Windows 8, but I do want to highlight a few key features that you should take note of when thinking about your own app and things you can take advantage of in your own game. Let's get started.

Live Tiles

Live Tiles (Figure 1-1) are possibly the most prominent feature of Windows 8. You are presented with these large icons on the start screen as soon as you turn on the computer, and they are the primary way to launch apps.

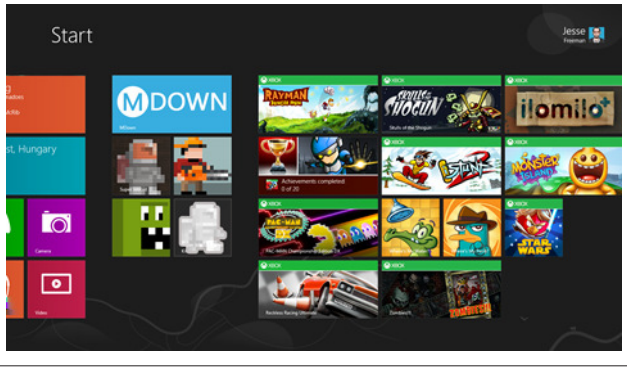


Figure 1-1. Here are active Live Tile on the Windows 8 start screen.

However, Live Tiles are more than glorified icons; they can contain a snapshot of information about the app and entice the user to click on it via animation, displaying status, or highlighting achievements unlocked. I will cover how to create a basic Live Tile in this book and point you to additional resources on how to extend its functionality after you have successfully ported over your game.

Charms

Charms are a new and important way to surface contextual options for Modern apps. Sliding your finger in from the right-hand side of the screen edge or moving your mouse over to the far right of the screen brings up the Charms bar (Figure 1-2).

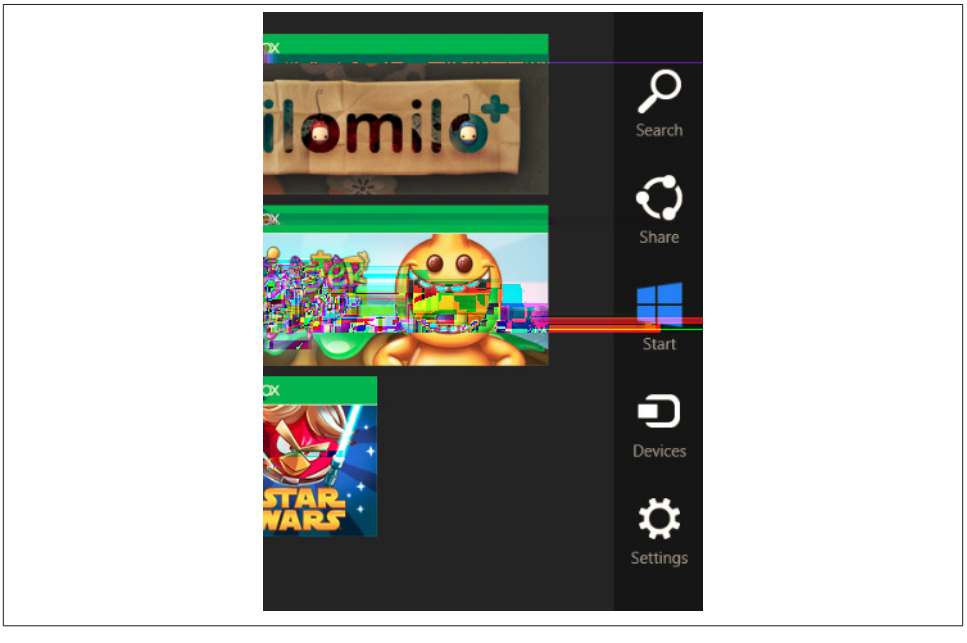


Figure 1-2. You can access Charms by swiping in from the right-hand side of the screen.

Charms are important if your app has search or sharing, but for games, we will focus primarily on the settings option.

Settings

Settings are important for a number of reasons. In Windows 8 games, we may need a place for our player to customize controls, tweak performance options, and display a help or privacy section.

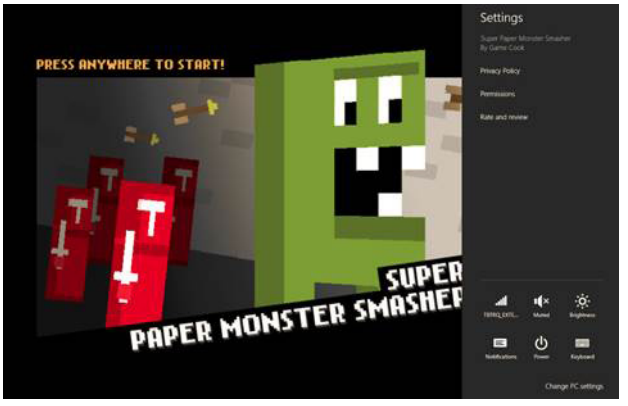


Figure 1-3. Access the settings of an app or game via the Settings Charm.

As you will learn later on in this book, creating setting "fly-outs" (Figure 1-3) is incredibly easy and can be done entirely with JavaScript and HTML.

Windows Store

Perhaps the largest component of this entire process is getting your game into the Windows Store (Figure 1-4).

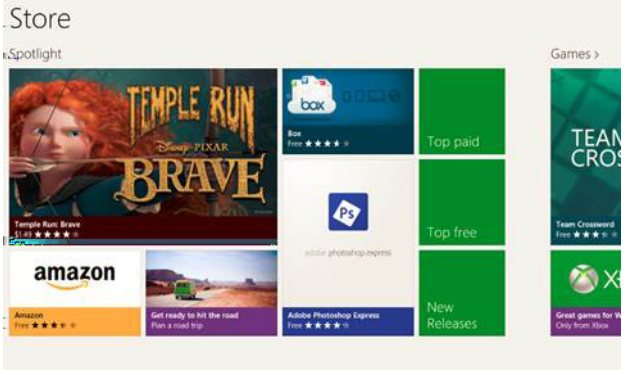


Figure 1-4. The Windows Store is where users are able to purchase and download new Windows 8 apps.

This is the only delivery mechanism for Windows 8 apps, and this book will walk you through how to get your app ready for submission, as well as ways to ensure it is approved quickly.

www.allitebooks.com

In Windows 8.1 the store (Figure 1-5) was dramatically redesigned with a more streamline presentation and surfaces important categories such as featured apps, popular now and new releases.

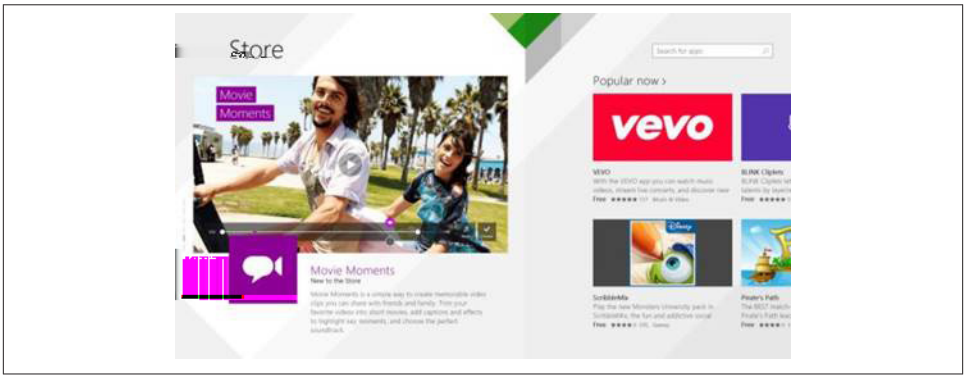


Figure 1-5. The redesigned Windows Store has a larger feature area and to its right is a list of the most popular apps in the store..

I'll talk a little more about Windows 8.1 and how to adapt your game's publishing settings to take advantage of the new design in later chapters.

Shortcuts

Since this may be your first time working on Windows 8, I'll introduce a few shortcuts to make your life easier, especially if you are not using a touchscreen:

- Windows Key – This will switch between the start screen and the currently open app.
- App Search – Simply start typing on the start screen in order to filter out and quickly find apps on your computer. This is a huge time saver.
- Windows Key + C – This will open up the Charms bar, including the clock.
- Windows Key + X – This will display system shortcuts.
- Windows Key + Z – This will pull up the app bar, which most Windows 8 apps use for contextual navigation. You can also pull this up by right clicking when in Modern apps.
- Windows + D – This will quickly throw you into the desktop or "classic" Windows environment.
- Alt + Tab – This will cycle through "classic" Windows apps and Modern UI apps. Holding down Shift while doing this will cycle backwards.

- Windows + Tab – This will cycle through Modern UI apps that are active.

Windows 8.1

Windows 8.1 is an update to the original version of Windows 8 that was released in October of 2012. It's more than just a collection of bug fixes, there are hundreds of new features as well as additional APIs and modifications to the overall OS. When it comes to porting HTML5 games over to Windows 8 there really isn't much of a difference between version 8 and 8.1. Perhaps the most important thing to call out is the new support for WebGL in both IE 11 (which is the default browser in 8.1) and in native apps you publish to the store. That means if you are using WebGL and it runs in IE 11 you will be able to publish it to the store following the same steps outlined in this book.

Windows 8.1 is a free update and because of that it is expected to gain adoption quickly among current Windows 8 users. The update process is incredibly streamlined but keep in mind there may be an existing user base of Windows 8 users. Windows 8 games can run on 8.1 but it doesn't work in reverse. I will be focusing primarily on publishing to Windows 8 in the book and plan on updating more over time as 8.1 gains adoption. When there is something different in 8.1 I will call it out specifically. Keep in mind that this book was originally designed to focus on publishing HTML5 games to the Windows Store on Windows 8.

BizSpark and DreamSpark

The last thing I want to call out are two programs Microsoft has to help developers get free tools, copies of Windows 8 as well as trails for Azure called BizSpark and DreamSpark. BizSpark is designed for small startups and individuals looking to get started developing apps, as well as games, for Microsoft's platform. DreamSpark is designed for students looking to do the same. I highly suggest checking one of these programs out, especially if you are new to developing for Windows 8 and are looking to start your own business making games whether it's on the side or full time. These programs are designed to give you everything you need to be successful and I have suggested to dozens of indie game devs that they sign up while working on porting existing games over to Windows 8.

We'll be covering the topics we introduced in Chapter 1 in more detail over the course of this book. In Chapter 2, I'll go over how to get your game working in Visual Studio and running on Windows 8.

CHAPTER 2

Getting Started with Visual Studio

When it comes to getting an HTML5 game up and running on Windows 8, I use the term porting lightly. Unlike other languages, you do not have to endure the difficult process of truly porting over your code. Windows 8 was designed to make HTML5-based apps run and perform like native ones written in C# or C++. In most cases you simply need to copy your code over to a Visual Studio project, include the files, and hit compile to see your game running. Let's talk a little bit about which games will work and which ones won't on Windows 8.

What Games Work Best

As a rule of thumb, if the game runs in IE10 then you are set. For extra insurance, you can double check your game in the Windows 8 version of IE10 to discover any minor issues that may arise. For the most part, the two browsers run identically. You can also do the same testing on Windows RT to see what performance would be like on ARM devices.

Outside of testing your game in IE10, completely Canvas-based games will generally be the easiest to port. This is true of any HTML5 game you are trying to run on different platforms, as Canvas is self-contained and consistently implemented across each of the major browsers at this point. Also, it's important to note that Canvas is hardware accelerated in IE10, and you will see similar performance boosts that you would have gotten in WebKit-based browsers that support hardware-accelerated Canvas, such as Chrome.

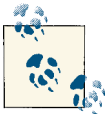
I've also been testing out several different JavaScript game frameworks and have found that all of them run well on Windows 8. ImpactJS, which is one of my favorites, is a great one to use and is fairly easy to get set up out of the box with little to no modifications. You can also use DOM-based games, but expect to have to account for IE10-specific CSS issues. If you already built your game to support IE10 then you shouldn't have a

problem. I've also had great success with "hybrid" games that make use of Canvas for the main game display and HTML for the UI. Just keep in mind you will need to do additional CSS work to make sure your UI scales well on different resolutions.

I'm also happy to tell you that third-party JS libraries also work right out of the box. Here are a few I use in my own games:

- **JQuery** – You can use JQuery or other DOM manipulation libraries. Just be careful of code that writes to the DOM at run time, as there are some sandbox restrictions on dynamic element creation and code execution in HTML5 Windows 8 apps.
- **ImpactJS, EaselJS, etc**. – Almost all Canvas-based JavaScript game/drawing frameworks should work right out of the box.
- **TweenLite and TweenMax JS** – A great library for tween and animations. WinJS also has its own animation classes, but if you want to keep your game cross-platform, you may want to consider a third-party tween library for programmatic animations.

In this chapter, I focus on Canvas-based games. As I mentioned, these games are the easiest to get working on different platforms, and I have some great tips and techniques to get these kinds of games up and running in Windows 8.



In Windows 8.1, IE 11 is now the default browser. IE 11 has a host of new features with the biggest one being support for WebGL. This means that you can now port your WebGL games over to Windows 8.1 and publish them to the store. One thing to keep in mind is that IE 11's WebGL support is still a work in progress and not every feature found in other browsers may be supported. That is why it is critical to test your games in IE first before attempting to put them in a Visual Studio project in order to work out any issues that may need to be addressed.

I do want to briefly touch on sound. Native HTML5 apps on Windows 8 do support playing multiple audio channels at the same time, which is usually an issue on mobile OSes. That being said since IE 10 only supports Audio Tags, you will have to take advantage of them in your game instead of the newer Audio APIs still being adopted in other browsers. For the most part this shouldn't be too big of an issue but keep in mind that on Windows RT devices there may be some playback issues or latency. I have not noticed it too badly since most of my Web games have very simple sound effects but if your game relies on sound heavily you may need to do some additional testing to make sure there isn't a problem.

Introduction to Visual Studio Express

Visual Studio is an incredibly powerful IDE. In my opinion, it is one of the best out there, and you should be very familiar with it if you have used Eclipse or IntelliJ in the past. It's project based, meaning that you open up a single project to work on at a time and you must specifically tell Visual Studio to include files in a project or it will be ignored. That may take some getting used to if you come from a TextMate, Sublime Text, or VIM background.

Before moving forward it's important to know that there are several versions of Visual Studio. When it comes to building Windows 8 HTML5 apps you can simply use Visual Studio Express which is the free version. It differs slightly from Visual Studio Pro and Ultimate which are used for screenshots in this book. Everything I show in this book can be done in the free version and I will call out specific differences in the two where needed. You can also get a free trail of Visual Studio Pro and Ultimate to follow along with the book's examples. Make sure you check out the BizSpark or DreamSpark programs to see if you can qualify for free licenses of Visual Studio Pro and Ultimate.

When you open Visual Studio, you will be presented with the screen in Figure 2-1.

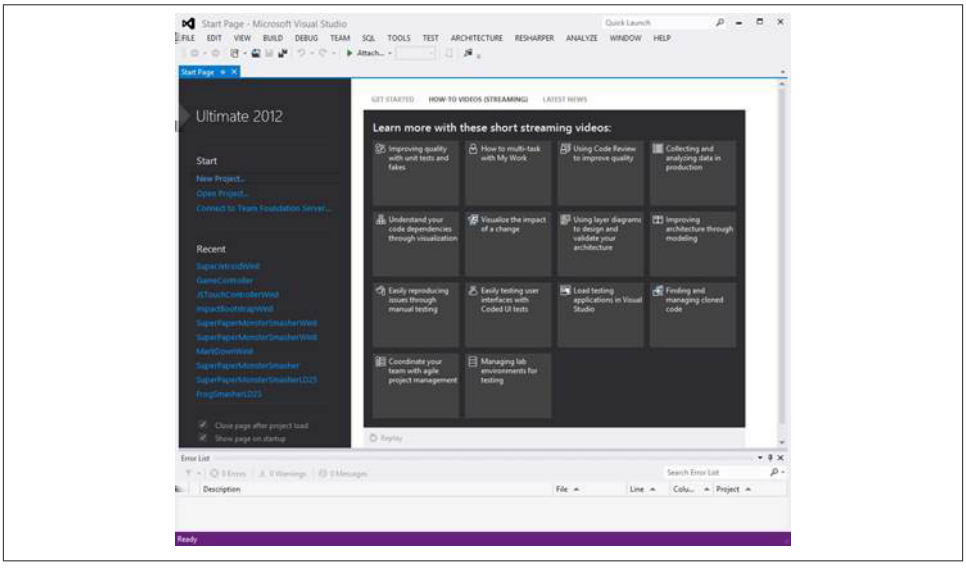


Figure 2-1. Visual Studio's start screen.

From here, you can open a project, create a new one, and even learn more via tutorials, videos, and additional links to resources from the launch screen. Creating a new project is simple thanks to an extensive collection of templates and documentation already included in Visual Studio (Figure 2-2).

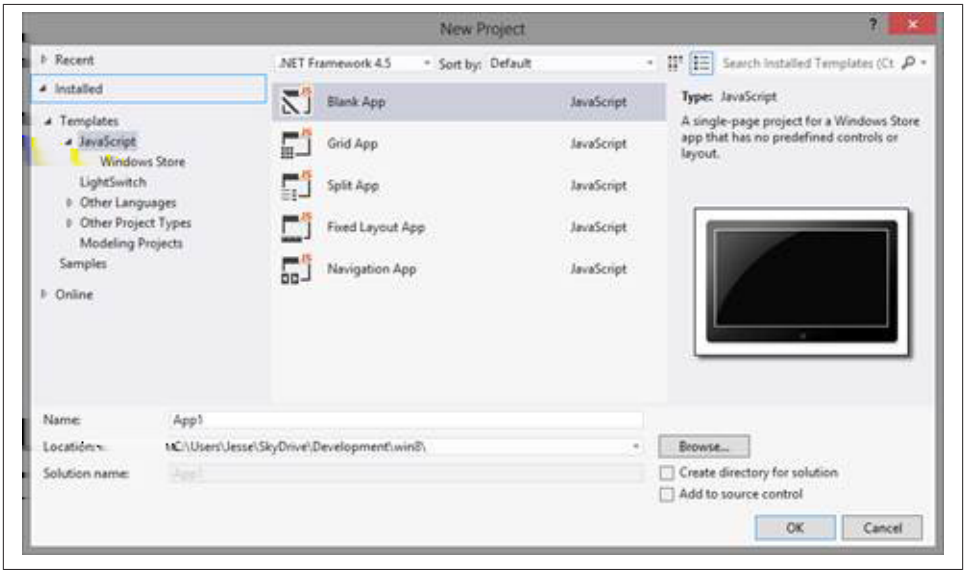


Figure 2-2. Creating a new project from a template in Visual Studio.

In addition to the built-in templates, you can also find more online.

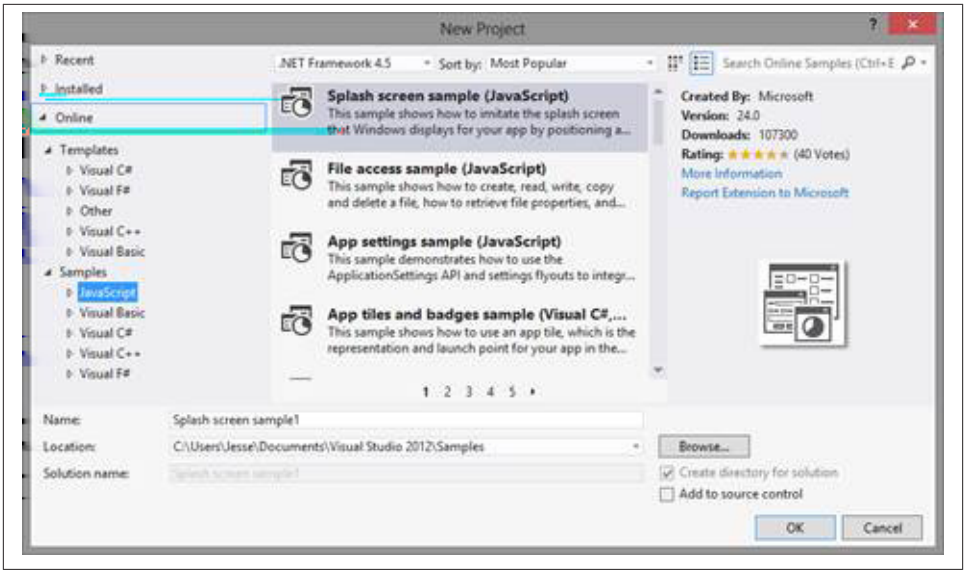


Figure 2-3. Check out the online templates for examples on how to use JavaScript to build Windows 8 apps.

I highly suggest going through the online templates in Samples -> JavaScript (Figure 2-3), as it covers all kinds of examples you may want to play around with to get a better handle on building JavaScript-based Windows 8 apps.

Once you create a project, you will be presented with a nav bar along the top, a code editor in the main window, and Solution Explorer on the right (Figure 2-4).

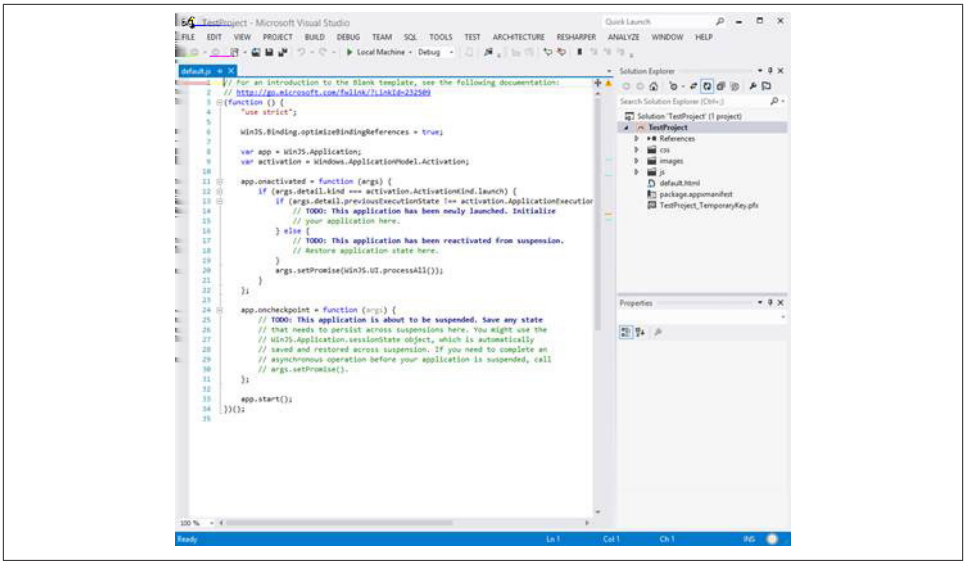


Figure 2-4. This is the layout you will see when starting a new project in Visual Studio.

These are just a few important things you should know while working in Visual Studio. First up is how to run your app, which you can do as soon as you create a new project.

At the top of the toolbar you will see a green arrow button (Figure 2-5).

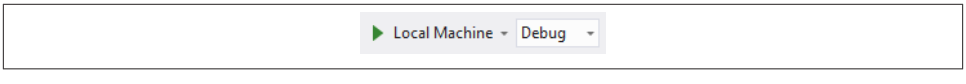


Figure 2-5. This button will launch your app locally for testing in debug mode.

This allows you to run your app locally, remotely, or in the simulator. It also lets you test different types of builds, such as debug and release.

Next is the Solution Explorer (Figure 2-6), which contains all the files in your project.

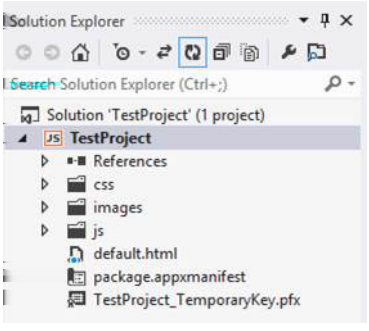


Figure 2-6. This is the Solution Explorer in Visual Studio.

One of the most important things you should keep in mind when working in Visual Studio is including the files you want for your game in the solution. To do this, select the show all files option.

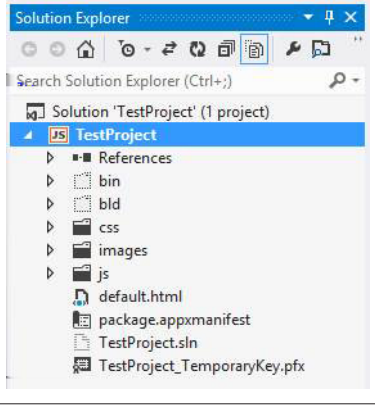


Figure 2-7. Selecting show all files will allow you to add new files into the project solution.

Once you do that, you will see all the files in the project folder (Figure 2-7). Files not included in the solution are outlined in grey dots. To include them, simply right click on a folder or file and select Include In Project from the contextual menu. I'll walk you through this more in the next chapter.

Reset Windows Layout, which you will find under the Window menu at the top right, can be used if you accidently close the Solution Explorer or lose the console window; you can reset the layout and get everything back. I suggest taking some time to poke around and see what there is to do in Visual Studio. You'll also start picking this up as you begin to work more in the IDE.

Understanding Visual Studio Project Structure

The first thing you'll want to do is create a new project in Visual Studio. There are several HTML5-based project templates for you to choose from (Figure 2-8). Unless you want to take advantage of a pre-built Windows Store app's UI, I would suggest choosing the blank project.

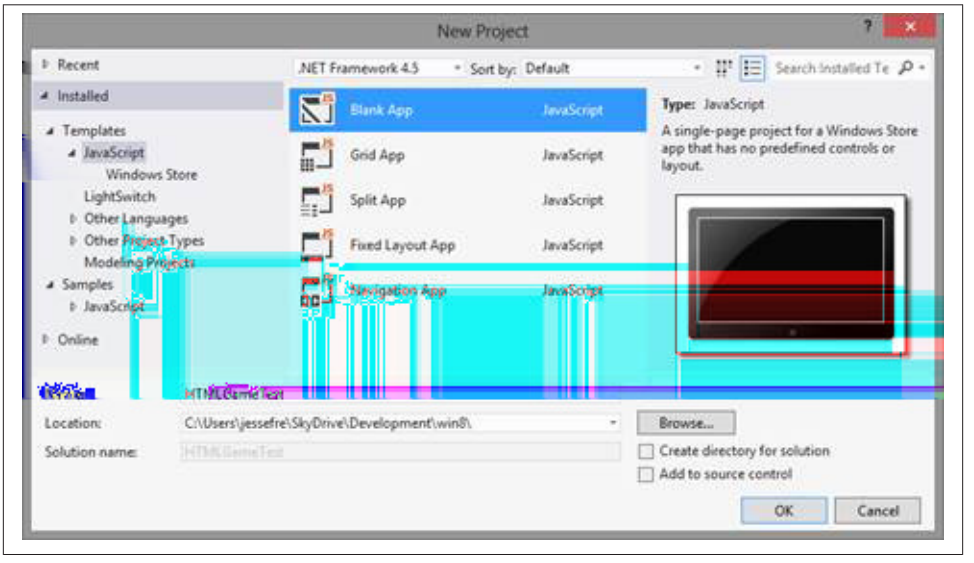


Figure 2-8. You should be familiar with the New Project screen from the first chapter.

It's also important to note that you can uncheck the "Create directory for solution" option. This will put your VS project inside another folder with a supplied name and may be redundant if you don't specifically want that.

Once you create your project, you will see the following folder structure (Figure 2-9):

www.allitebooks.com

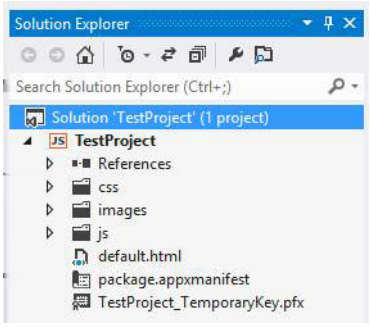


Figure 2-9. Here is the deafult project structure of a Windows 8 HTML5 project.

There are some important directories and files you should know about:

- **default.html** – This is the main page of your project. Think of it as the index.html file you would typically create to host your game.
- **js Directory** – This is where you will find the default.js file, which sets up your project's code when the default.html is run. While this file is optional, you will need to follow a similar setup in your own game if you choose not to use it. I will talk about this more later on in the chapter.
- **images Directory**– This is your default directory for images. It contains your application's loading screen, icons, and store graphics. You can also place your own images in this folder as well.
- **css Directory** – This contains the app's default CSS. This file simply contains a few meta tags to support different resolutions in your app. You can usually ignore or override it.

There is also one more important file we should look at in your project's folder: the `package.appxmanifest` file. If you double click on this, you will be presented with different options to configure your project (Figure 2-10).

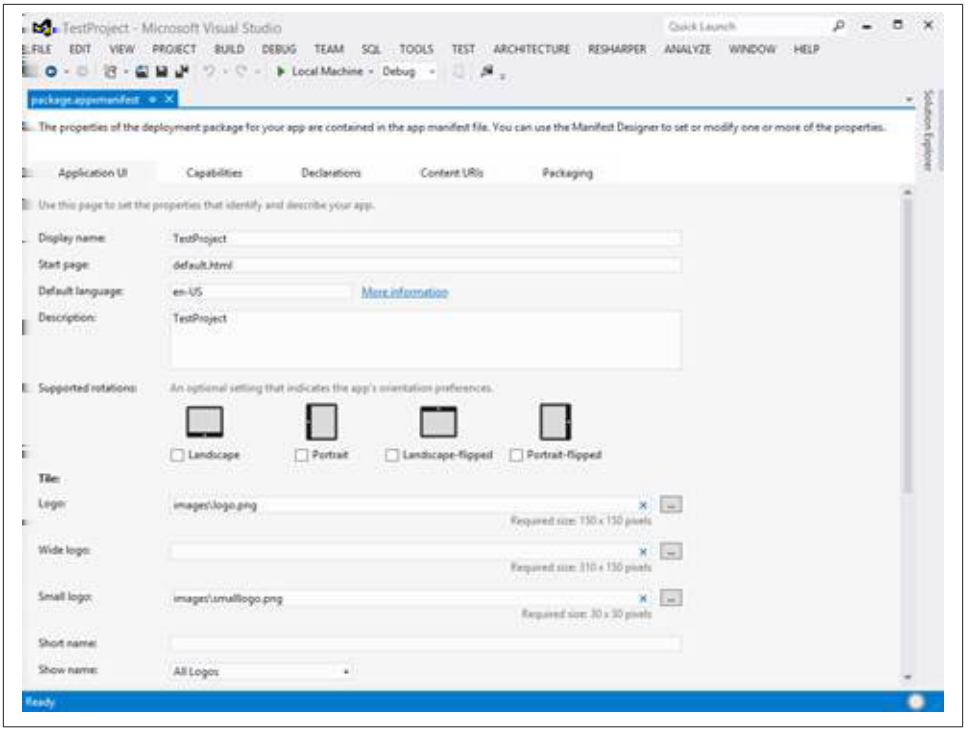


Figure 2-10. You can configure your application's settings and capabilities by clicking on the package.appxmanifest file.

I'll go over some of the more complex options, such as the Capabilities tab, later on in the book.

Finally, I want to cover the anatomy of the default.html file, as you will most likely be replacing some of the code in here or supplementing it with your own. As you can see, the basic code structure is fairly straightforward.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>HTMLGameTest</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- HTMLGameTest references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
```

```
</head>
<body>
    <p>Content goes here</p>
</body>
</html>
```

You'll notice there is hardly any code in this file. We have a reference to the default ui-dark.css file that has all the styling for standard Modern UI apps. Next we have a reference to the base.js and ui.js, which allows us to build out Modern controls and UI. You'll notice these three files are located at //Microsoft.WinJS.1.0/js. These are part of the WinJS library and are only available when you run your app in Windows 8.

WinJS is a built-in Windows 8 library that allows you to communicate with the OS. This helps bridge the game between the JavaScript run time and the native code running under the hood. We'll talk more about WinJS throughout the rest of this book.

The paths to these files will be automatically resolved for you when the app is launched. WinJS is the underlying bridge that allows your app to talk to the OS via JS and also provides code needed to build out Modern UI components. You will want to keep these so that you can leverage WinJS in your own game. Finally, you'll see two references to local files in your project: default.css and default.js.

Now you are ready to start putting your own code into the project.

Moving Your Code Over

You should have everything you need to get started from the first chapter: a copy of Windows 8, Visual Studio, and your own game's source code that has been tested to run in IE 10. Consider this the quick-start guide to fast track running your own HTML5 game on Windows 8.

Since everyone has their own way of building a JS game, I'll walk you through some basic concepts and suggestions on how to get your game running.

First, move all of your game's code into the project. You can do this one of two ways:

First, follow the current project's structure and place all your images in the images folder. Then, move your CSS into the css directory, and anything else you may have. This way you work with the default project structure and only make a few alterations where they are needed.

Second, ignore the current project setup and move your code over as is. For example, Impact games make use of a media folder for all of their assets, including images and sounds, a lib folder for all of the JavaScript code, and have their own default index.html

page. Simply move these files over as is and you'll retain the pre-existing project structure.

Understanding the App Lifecycle

As with any type of software built on a larger framework, there is the notion of an application lifecycle. In our case, we are just referring to the build up, pause/resume, and tear down of our application in the Windows 8 environment. If you open up the js directory and select the default.js file, you will see the base WinJS app launch point.

```
// For an introduction to the Blank template, see the following documentation:
// http://go.microsoft.com/fwlink/?LinkId=232509
(function () {
    "use strict";

    WinJS.Binding.optimizeBindingReferences = true;

    var app = WinJS.Application;
    var activation = Windows.ApplicationModel.Activation;

    app.onactivated = function (args) {
        if (args.detail.kind === activation.ActivationKind.launch) {
            if (args.detail.previousExecutionState !== activation.ApplicationEx-
ecutionState.terminated) {
                // TODO: This application has been newly launched. Initialize
                // your application here.
            } else {
                // TODO: This application has been reactivated from suspension.
                // Restore application state here.
            }
            args.setPromise(WinJS.UI.processAll());
        }
    };

    app.oncheckpoint = function (args) {
        // TODO: This application is about to be suspended. Save any state
        // that needs to persist across suspensions here. You might use the
        // WinJS.Application.sessionState object, which is automatically
        // saved and restored across suspension. If you need to complete an
        // asynchronous operation before your application is suspended, call
        // args.setPromise().
    };

    app.start();
})();
```

Luckily for us, the files are heavily commented. As you can see, there are two places you will need to manage your game being initialized: when the game is launched for the first time and when the app has been reactivated from being suspended.

This is similar to the window.onload or document.load where you would wait until the entire page is loaded then execute the code to run your game. I highly suggest that you modify your game to be initialized here instead of running on its own in the default.html file. This will allow you to better integrate your game into the Windows 8 app lifecycle and give you more control over how your game behaves based on being loaded up or resuming. The best part is that if your game already has some method for initializing it after the page is ready you can probably just comment that out in your index.html page and use this file to call the same method.

It's up to you if you want to place your game's initialization code into the default.js file. Your game will run perfectly fine on its own inside of the default.html file, but you will not be able to handle any resume logic on your own unless you build that into your game's own code. Likewise, you can skip the default.js logic and simply put this in your own game's code. All you need is a reference to WinJS.Application and the activation callback since Windows 8 will handle the rest for you.

Now that we have learned a little more about the app lifecycle, you can give your game a try and see if it will run on Windows 8.

Running Your Game for the First Time

You should have everything you need in place to run your game and see if it works. To test your game, click on the run button (Figure 2-11) at the top of the Visual Studio window.

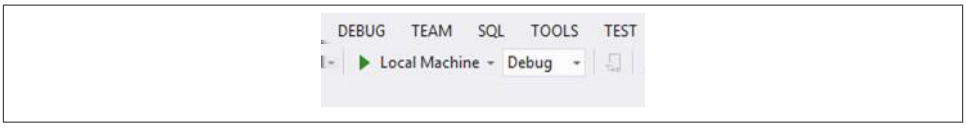


Figure 2-11. Clicking on the green arrow will start the build process for your game's project.

This will run the game locally in debug mode. Once the project is compiled you will be taken to the game, which will most likely be full screen if you are running on a single monitor.

Windows 8 runs great on multiple monitors. One of the best features of having a two-monitor setup is being able to run Modern apps on one screen and the "classic" desktop on the other monitor. What this means is that you can easily build your Windows 8 app side by side with Visual Studio's debug tools visible, which will increase your productivity dramtically. If you are thinking about getting a second monitor or upgrading an exisiting one, I highly susggest getting a touchscreen. Not only will you be able to code Windows 8 games better with a two-screen setup, you can also test out touch controls.

In a single monitor setup, you can still switch back and forth between the game and Visual Studio's debugger. Hopefully your game is running. Depending on your game's controls, you should also be able to play it. In the event that your game is not working, I suggest checking out the chapter on debugging and optimization to help troubleshoot the problem.

Tips and Tricks for Running Your Game on Windows 8

In most cases, setting up your game should be as easy as copying over the files, including them in the project, and hitting compile. If you are running into issues or want to better understand what is going on under the hood to help refactor or clean things up, I'll go over some techniques to help you debug your game in Visual Studio.

Disable Touch Behaviors via CSS

You may notice that pinch to zoom or double tap to zoom may happen in games you move over to Windows 8 from the Web. The quickest way to solve this is with the following CSS:

```
-ms-touch-action: none;
```

You can place this in the HTML, body, or Canvas element CSS to disable touch behaviors.

File Paths and Loading Locally

File paths are resolved to the root of your project when it is running. So, if you wanted to load an image from the images folder, you can simply do:

```
<img src="/images/name-of-image.png"/>
```

Likewise, you can do the same when loading images or assets dynamically via JavaScript. Hopefully you won't have to make a single change to a file path or resource.

In the end, you should have all the tools you need to quickly figure out what may be going wrong with your game's code. Over the next few chapters, I'll discuss what you can do to enhance your game on Windows 8 and better integrate it with WinJS.

Avoid Modernizr Libraries

If your game is already running on the Web, you may be making use of some kind of Modernizr or some type of polyfill to help get your game running across each of the different browsers. While this works great for the Web, you will want to remove this library if it gives you problems. I have seen Visual Studio throw errors if the library violates Windows 8's JavaScript sandbox by trying to inject live code into the DOM at runtime.

CHAPTER 3

Screen Resolution and Artwork

Windows 8 Resolutions

If you have done any desktop software development before, you may be familiar with the dizzying amount of resolution possibilities out there. While Windows 8 still has its share of resolution options, the form factors and the push towards tablets simplifies your baseline of supported resolutions.

Out of the box, Windows 8's native resolution is 1366×768. While Windows 8's minimum resolution is 1024×786, almost all new tablets and convertibles will have their minimum resolution set to 1366×768. But this isn't the full story. Let's take a look at the three main resolutions our game will need to support to pass certification for the Windows Store:

- Full Screen – This is the default resolution when the game is running in full-screen mode. This can be anywhere from 1366×768 and up.
- Snapped – This happens when your game is docked on the side of the screen and another application is running next to it. The minimum width of this resolution is 320 pixels wide.
- Filled – This would be the full-screen resolution minus 320 for the snap view. This resolution happens when you have another app docked in snapped view on the side of your game.

You can easily detect resolution changes by adding an event listener to the window like so:

```
window.addEventListener("resize", onViewStateChanged);
```

And, here is an example handler for the resize event.

```
function onViewStateChanged(event) {
    var viewStates = Windows.UI.ViewManagement.ApplicationViewState;
var newViewState = Windows.UI.ViewManagement.ApplicationView.value;
        if (newViewState === viewStates.snapped) {
            // is snapped
        }else if (newViewState === viewStates.filled) {
            // is filled
        } else if (newViewState === viewStates.fullScreenLandscape) {
            // is full screen
        } else if (newViewState === viewStates.fullScreenPortrait) {
            //is portrait
        }
    }
}
```

Notice how we can also detect screen rotation? You can disable that for your game, but in order to be approved by the store, you need to handle the other three view states properly. As of now, the majority of Windows 8 devices are probably running in landscape mode. While portrait mode is the default orientation on phones, you'll find that Windows 8 is designed for landscape due to its desktop heritage. That's not to say that you should totally ignore portrait mode, but if you are working on minimizing scope creep, especially for your first Windows 8 game, I would suggest sticking to landscape. Likewise, if your game was designed to run in portrait mode since it was created for a mobile device first, you may have to consider centering it on a background image while in landscape mode, which is a popular technique many games are already utilizing in the Windows Store when running on larger resolutions the game wasn't ideally intended to support.

Scaling Games for Full Screen

By design, all Windows 8 apps run at full screen at any resolution. What that means is that, given the huge user base of Windows 7 and the potential of upgrading desktops attached to large monitors, your game will have to support resolutions higher than 1366×768. Let's be practical. While HTML5 games run great on Windows 8 at normal resolutions, the bigger the display, the slower your game will run. This is directly attributed by the increase of pixels your game will be trying to update to the display; the larger the resolution the more pixels being rendered. Fortunately, we can take advantage of a few techniques to stretch out our canvas in a way that will help balance performance and maintain the ascetics of your game.

There are two ways to stretch the canvas:

- Scale to Fit – This stretches out the canvas to fill the screen, and while most people may not notice, your game could look wide or squished on odd resolutions.

www.allitebooks.com

- Maintain Aspect Ratio – While this maintains the quality of your artwork, you may end up with black bars on the sides of your game.

This is a hard decision to make. Luckily, it's very easy for us to test both out. We'll start with scale to fit since you can simply do this in the CSS.

```
<style type="text/css">
        html, body {
            margin: 0px;
            padding: 0px;
        }

        #canvas {
            width: 100%;
            height: 100%;
        }
</style>
```

This will work great on Windows 8, but be careful if you try this on the Web. Since most monitors are designed to be at fixed aspect ratios, like 4:3 or 16:9, your game will always look relatively fine. In the browser, however, the user can set the window to any arbitrary size, and this could look horrible depending on what they set it to. That is why it is sometimes better to try and maintain the aspect ratio, which is something we can do via JavaScript.

The basic concept of maintaining a game's aspect ratio is to figure out its ideal size and divide the width by the height. From there we can attempt to fit the game into the current resolution by using its height as a point of reference. This means that your game will constrain its width to always make sure that the height of the canvas is set to 100 percent. Most likely this will yield black bars on the right and left sides of the game canvas unless the display matches your game's aspect ratio. On the flip side, if your width is smaller than the max height, we will need to add black bars to the top and bottom of the game. This scenario will most likely happen if you design your game to run at a native resolution of 1366×768 and the person snaps another app next to your game, which will lower the resolution to 1024×768.

If you are not familiar with all of this, it may be a hard concept to wrap your head around. Let's look at the code and add this to your own game to see how it affects the visuals at different resolutions. We'll start with the basic CSS.

```
<style type="text/css">

    html, body {
        background-color: #000;
        color: #333;
        font-family: helvetica, arial, sans-serif;
        padding: 0;
        font-size: 12pt;
        margin: 0px;
```

```
        padding: 0px;
    }

    #canvas {
        width: 100%;
        height: 100%;
        -ms-touch-action: none; /* Disable touch behaviors, like pan and zoom */
        position: absolute;
        left: 50%;
        top: 50%;
    }    </style>
```

Next, we will have to add a little bit of JS to control resizing the canvas.

```
<script type="text/javascript">
    window.addEventListener('resize', resizeGame, false);

    function resizeGame() {

        var canvas = ig.system.canvas;

        var canvasRatio = canvas.width / canvas.height;
        var width = window.innerWidth;
        var height = window.innerHeight;
        var windowRatio = width / height;

        if (windowRatio > canvasRatio) {
            width = height * canvasRatio;
            canvas.style.height = height + 'px';
            canvas.style.width = width + 'px';
        } else {
            height = width / canvasRatio;
            canvas.style.width = width + 'px';
            canvas.style.height = height + 'px';
        }

        canvas.style.top = (window.innerHeight - height) / 2 + "px";
        canvas.style.left = (window.innerWidth - width) / 2 + "px";    }
 </script>
```

This also assumes that you have the following setup in your index.html or default.html with the canvas inside a div with an ID called game.

```
<body>
    <canvas id="canvas"></canvas>
</body>
```

What's great about this code is that it will work well on the Web as well as on the desktop, so you can consolidate your game's scaling to a single function. Also, you can call window.resizeGame() whenever you need to manually correct the scaling, such as when the game is finished initializing.

Hopefully, after running the two different examples on your game at different resolutions, you can pick the best one for your use case. No matter which one you chose, the benefits you will get at higher resolutions will be monumental. Rescaling the game on larger monitors with higher resolutions will still yield fast, fluid, responsive playback.

Understanding Snap View

Your game will not be approved for the Window Store unless you fully support snap view. This can be something as simple as throwing up a graphic telling the player they are in snap view and can't continue playing, or doing something even more elaborate. Just keep in mind that making use of the snap view will not only encourage players to keep the game open while they check their email or do other things, but it will also get more attention by the curators of the Windows Store who look to feature developers who are making the best use of Windows 8's new features and capabilities.

There are two basic ways we can handle snap view: using CSS meta tags or managing it via code with JavaScript. Both options allow you to mix and match HTML and Canvas elements, so it's really about finding the right solution for the way your game is already created. Here are two examples to review in order to give you a better idea of how you want to implement it yourself.

In my index.html file, I have my canvas and my snap view image wrapped in the same game div I talked about previously in the scaling part of this chapter.

```
<body>
    <div id="game">
        <canvas id="canvas"></canvas>
        <img id="snap-view" src="media/snap-view.png"/>
    </game>
 </body>
```

Make sure your snap view image is no larger than 320×768 (Figure 3-1).

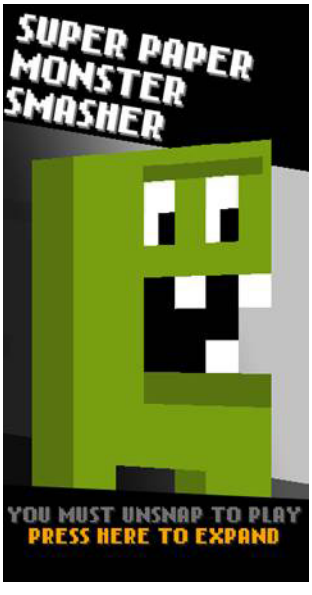


Figure 3-1. The snap view image I am using for my game Super Paper Monster Smasher.

The basic concept is that, while the canvas is automatically being stretched out to take up the full screen, the snap-view image isn't visible. To make sure, we'll alter its CSS properties to hide it. When I detect snap view, I simply hide the canvas and show the snap-view image.

First, let's take a look at the CSS-only approach.

```
@media screen and (-ms-view-state: fullscreen-landscape) {

    #canvas{
        display: block;
    }

    #snap-view{
        display: none;
    }

}

@media screen and (-ms-view-state: filled) {
    #canvas{
        display: block;
    }

    #snap-view{
        display: none;
```

```
    }
}

@media screen and (-ms-view-state: snapped) {
    #canvas{
        display: none;
    }

    #snap-view{
        display: block;
    }
}
```

As you can see, I have set up a CSS meta selector for detecting the snap view. You can also use a min-width meta selector if you want to use this code on the Web and in Windows 8 to help maintain constancy between your code base.

Now, let's take a look at how to maintain this via JavaScript. Here is what the code for that looks like in my game.

```
if (newViewState === viewStates.snapped) {
    canvas.style.display = "none";
    snapView.style.display = "block";
} else if (newViewState === viewStates.filled) {
    canvas.style.display = "block";
    snapView.style.display = "none";
} else if (newViewState === viewStates.fullScreenLandscape) {
    canvas.style.display = "block";
    snapView.style.display = "none";
}
```

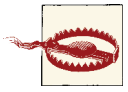
As you can see, I simply tied this conditional into the earlier example where I detect the window's resize event.

You may have also noticed from my snap view picture that I added some text telling the player to click on the image in order to expand the game and exit snap view. This is very easy to do. Simply add a click event to the image like this:

```
var snapView = document.getElementById("snap-view");
snapView.addEventListener('click', function(e) {
    var boolean = Windows.UI.ViewManagement.ApplicationView.tryUnsnap();
})
```

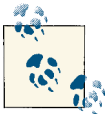
As you can see, I have a simple image that fits perfectly in the 320-pixels-wide area provided by the snap view and can also center it vertically in case the game is running at a higher resolution. Remember that the height of the snap view is determined by the maximum resolution of the attached monitor, so it can be larger than 786 pixels high.

The final thing we will want to make sure of is that whenever we enter snap view we pause our game. Hopefully your game has a way to pause itself. We can simply listen to the same resize event and determine if the game needs to be paused.



It is critical that you also pause the game and any sounds or music playing to offer up a clean experience when transitioning to snap view. If you are using a CSS-based solution, make sure you don't forget to shut everything down in your game via JavaScript. The last thing you want is for your game to be running in the background when the player can't interact with it under the snap view image.

To play it safe, you should pause the game any time you detect a resize event. The simple act of moving from snap to full or full to filled will probably be a little jarring for your players, and they wouldn't have time to react on their own. Also, I wouldn't un-pause your game if you leave snap view. You have to take into consideration that the player may have been doing something else when the app was unsnapped, such as closing another app, and may not be ready to jump back into the game. Whenever you change resolutions from full screen to filled and down to snap view, you should pause the game and let the player manually resume when they are ready.



In Windows 8.1, the user now has more control over how two apps run side by side. In addition to snap view, the user can split the screen between two apps so that each one takes up 50% of the usable screen resolution. That means that your game should be able to scale to accommodate this dynamic resolution changing. If you included the aspect ratio scaling code example from the previous section your game should handle this fine but it's still important to keep in mind in case your game becomes unplayable at half of the screen's resolution. This only applies to devices with a resolution higher than 1024 ×768.

As you can see, supporting snap view is a simple process and something that you don't want to overlook when submitting your game to the Windows Store.

Upscaling Artwork

Another thing to consider is how to upscale your artwork. If you are coming from an existing game, you may have built the game for a lower resolution. Or maybe you are thinking of making a new game from scratch and looking to get an authentic-looking 8- or 16-bit feel. There are some great techniques we can use in order to upscale the artwork in a way that will not only help support higher resolutions but also add a unique look and feel to your game.

Upscaling allows you to show a smaller graphic at a much larger resolution. Since I work with pixel art, the effect tends to accentuate the pixels more, giving it the retro look most people are familiar with. There is no exact science behind how much to scale up, but I tend to do it in multiples of two. So, if I have a 16×16 pixel sprite, I can display it at 24×24 pixels by increasing the scale factor by two or even up to four times its original size.

Figure 3-2, Figure 3-3, and Figure 3-4 are an example of how this works.

Figure 3-2. 1x (actual size 63×42 pixels).

Figure 3-3. 2x (new size 126×84 pixels).

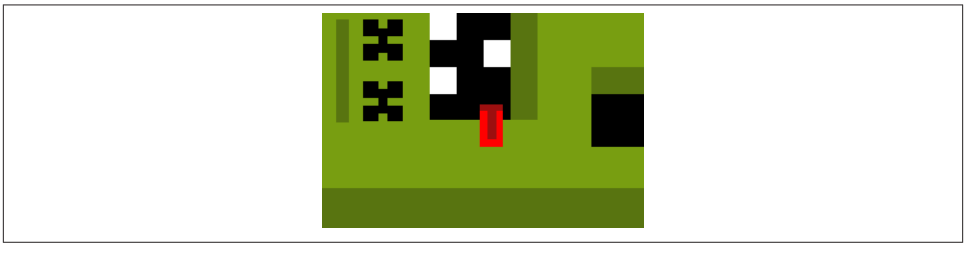


Figure 3-4. 4x (new size 251×167 pixels).

As you can see, the higher the scale factor the more pixelated the artwork looks. This is incredibly effective when you have smaller sized sprites and map tiles.

When resizing artwork in an image editor such as PhotoShop, make sure you use nearest pixel scaling to preserve the crispness of the pixel art.

The one thing you want to make sure of when it comes to upscaling your artwork is that you pre-render the artwork at the higher size ahead of time. Many popular game frameworks allow you to resample the artwork on the fly to an off-screen canvas, but this will double the memory usage of your app and slow down the load time of your game.

If you don't anticipate changing the scale size of your game based on different resolutions, then it may make more sense to output everything at the correct scale factor ahead of time. This is especially important on slower devices. When it comes to getting the best performance out of Windows 8 games, you have to build for the lowest common denominator. If you are forced to resample your artwork on the fly, I suggest caching the generated art to the player's computer and only regenerating it on major resolution changes, or when the app loads up for the first time.

Designing for Multiple Resolutions

While this section may not help you if you already have a completed game, I still want to spend a little time talking about some great ways to better handle multiple resolutions with forward-thinking design. There are a lot of simple tricks you can implement to help accommodate multiple resolutions. Most of the problems usually come up with the game's UI. The first thing I do is open up my design tool and map out a few resolutions to see how they will look in my game. This is a similar technique to what I used to do in video editing to find a "safe zone" where the TV's edges would clip text or video. Here is an example from my game Super Paper Monster Smasher.

Figure 3-5. Here is my design from Photoshop showing off what is visible at each resolution.

Here you can see I have mapped out four different resolutions: full screen, filled, snap, and Web. In Figure 3-5 you will notice that my game's native resolution is 800×480, since this started out as a Web game, and from there I can expand out the camera that renders the in-game graphics to offer more visible area at different sizes.

Having a dynamic camera is a great way to handle resolution changes. Simply expand the view of the game and you can easily fill in the screen and offer more visible real estate for the player. The downside is that you have to have a UI that can also be flexible enough to handle the resolution changes. Also, you run the risk of increasing performance issues since you are rendering more of the game, which is why I made sure to design my maximum resolution just under the 1366×768 pixel resolution of Windows 8 so I can stretch the canvas to fill in the difference.

When it comes to UI for your game, try to keep things as simple as possible because you will use up precious resources trying to render more to the screen, and you want to maximize your game's screen real estate on tablets. Remember that your game will most likely be held in landscape mode on the left- and right-hand sides of the screen, so avoid putting important UI at the bottom of the screen, which could be blocked by a player's hands.

Another important design technique for UI on multiple resolutions is to "dock" them to corners of the screen (Figure 3-6). Here are two examples of how the UI realigns itself based on the screen size.

Figure 3-6. Here is the UI and camera clipping at 800×480 pixels for the Web version of the game.

As you can see in Figure 3-7, the UI elements on the left-hand side always move to the left corner of the screen and the right-hand side elements move to the right. The pause button, which is a critical UI element, is centered inside of the empty space between each side's UI. Another technique you can use on Windows 8 is to completely hide the

pause button and non-critical UI and expose them through the commands bar, which you access by swiping up from the bottom of the screen or right clicking. I tend to avoid doing this because, while modern UI standards dictate all non-essential UI should be hidden from view, games get a lot more leeway during approval. In a game, you want instant access to pausing the game, so make it easy for the player to hit it. If you rely on a gesture, you could make the player do something in the game that would potentially cause a problem.

Figure 3-7. The UI and camera clipping at 1024×768 pixels for the Windows 8 version of the game.

Live Tiles

Live Tiles are probably one of Windows 8's most prominent features. Live Tiles are the icons on the new start screen, but these are more than just app icons. Like the name implies, these tiles are animated and can contain a wealth of high-level information about the app. When it comes to games, we can do some interesting things, such as display player stats and high scores, and even entice the player to come back to the game if they haven't played in awhile. For the purposes of getting your first Windows 8 game to be published, I am just going to focus on the basics of setting up Live Tiles.

Luckily, the easiest thing you can do in order to set up a Live Tile is simply replace the default tile image in your project's images folder. This was automatically created for you by Visual Studio. If you open up your .appxmanifest (Figure 3-8), you will see all the default Live Tile options.

www.allitebooks.com

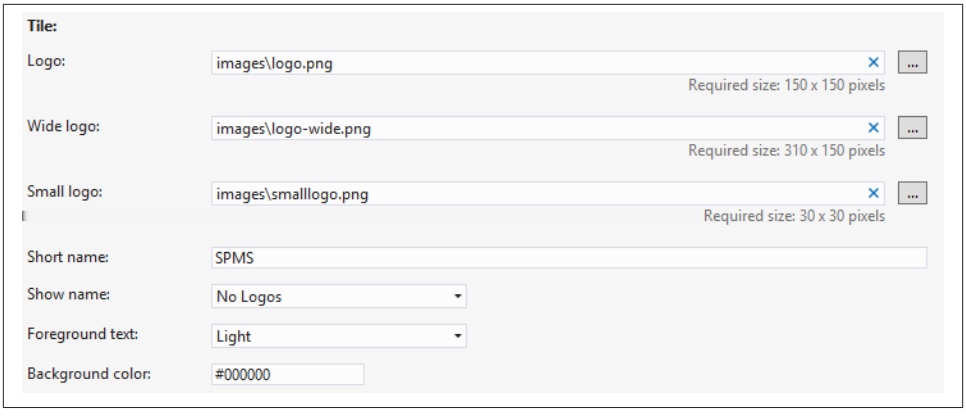


Figure 3-8. These are the options in Visual Studio to add your own images for the default Live Tile graphic.

Figures 3-9, 3-10, and 3-11 show the images I am using in my own game along with their sizes.

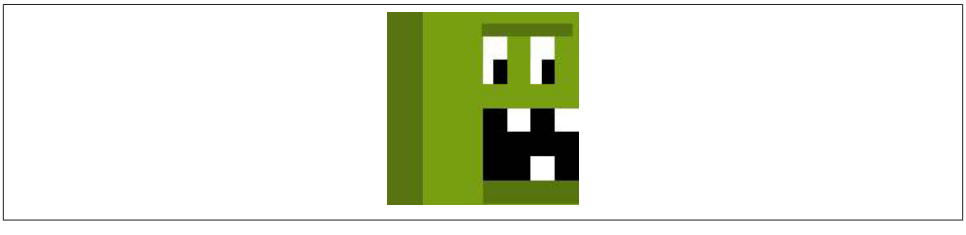


Figure 3-9. The default logo is 150×150 pixels.

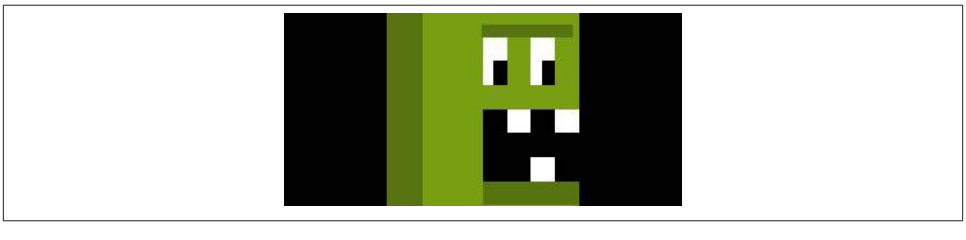


Figure 3-10. The wide logo is 310×150 pixels.

Figure 3-11. The small logo is 30×30 pixels.

In Windows 8, the wide logo is optional. In the Windows Store on 8.1 the app listings actually uses the wide logo which was not done in the earlier version of the store. This means that you should give additional thought into the design of that logo to help it stand out better in the store listing. If you do not supply a wide logo in your game it will automatically be centered on a wide side tile and given a grey background.

Once you have your Live Tile images set up, you can also control if the game's name is displayed automatically for you by Windows 8. Below where you set the Live Tile images, you will see an option to supply a short name and a dropdown with options on how to show the title (Figure 3-12), if you want to.

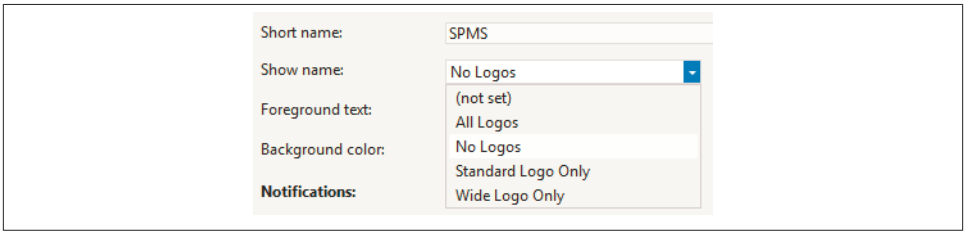


Figure 3-12. Options for adding text to your Live Tile.

As you can see, you can show the text on all logo types, not at all, on the default logo, or on the wide tile. Likewise, if you don't want Windows 8 to automatically display your game's name, I suggest simply setting it to "No Logos" and putting the game's name in the artwork itself.

Splash Screen

You may have noticed an additional graphic in the images folder called splashscreen.png. You can set this in the .appxmanifest (Figure 3-13), just like we did with our game's Live Tile graphics.

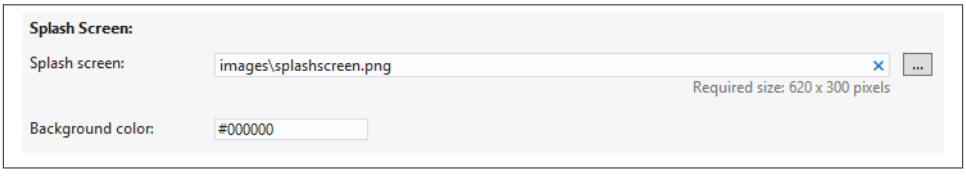


Figure 3-13. Option to customize the splash screen in Visual Studio.

The splash screen is automatically displayed as the game loads up. Its deafult resolution is 620×300 pixels. In Figure 3-14 you will see a sample of what my own game's splash screen looks like.

Figure 3-14. This is the splash screen graphic I use in Super Paper Monster Smasher.

There is nothing fancy about this. The image is centered on the screen and you can set the background color to help match up to your own splash image. I highly suggest making the background color of your splash screen solid so it looks seamless on the default background color. Remember that this image will be centered when the game is loading up so, if you are on a higher resolution monitor, you are going to see a lot of background color surrounding your splash screen image. There is no way to stretch this out to fill the screen.

You can also line up your own splash image to show up after the game is loaded to make this a seamless transition. Simply center the same image on the page and then hide it when you are ready to display the game.

Tips and Tricks for Working with Artwork on Windows 8

Working with images on Windows 8 is similar to how you would on the Web. The best advantage, however, is that you don't have to worry about loading files from a server. Instead, you will need to focus on optimizing your files so they don't take up a lot of memory and also limit your draw calls so you can increase performance on larger monitors. Here are some tips to do just that.

Use Sprite Sheets or Texture Atlases

Try to combine images as much as possible into a single graphic. Not only will this cut down on the number of images you need to manage in memory, it also helps speed up drawing to the Canvas by removing the overhead of having to manage multiple images.

There are generally two different ways to combine images: sprite sheets and texture atlases. Sprite sheets are usually used for images that are the same size and can easily be divisible by their width. A player may be 16×16 pixels wide, so you can line up all the player animations horizontally, vertically, or in a grid. By using the fixed width of the sprite, you can find each frame in the image.

Likewise, if you have lots of graphics that are not perfectly square, or you want to combine all of your game's artwork into larger files, you can use a texture atlas. This is common when working in 3D and allows you to package up all of your textures into a single image and use an atlas file, which defines the coordinates of each image and where to find it on the texture. These atlas files can be in .xml, .json, or .txt. There are lots of free and paid texture packagers out there that can help you optimize your sprites.

Render for Native Resolution

Some JavaScript frameworks, such as ImpactJS, can automatically upscale your graphics when the game starts. While this works well on some devices, I have noticed considerable slowdown on lower-powered Windows 8 devices trying to do the resizing. Instead, you should have all of your artwork pre-rendered at the correct scale factor.

You can also supply multiple scale images to choose from at runtime or, if you must have your game generate images at runtime, cache them and save them to the hard drive to use later. I've seen many games use this so that you only have one long load time the first time you run the game and then it resizes everything so consecutive loads are faster.

Handling Edge Cases

Be prepared for edge cases and situations where you can't totally anticipate the resolution your game is going to run on. Because of this, it is important to test your game at multiple resolutions and DPIs. You can easily do this with the Windows 8 Device Simulator in Visual Studio.

Figure 3-15. Testing out different resolutions in the simulator.

As you can see in Figure 3-15, simply select the icon that looks like a monitor on the right-hand side and select a different resolution to test your game out and see how it responds.

Set a Maximum Resolution

No matter what you end up doing to support multiple resolutions, just keep in mind the maximum resolution your game should render then scale the game up after that. For my Super Paper Monster Smasher game, my max resolution is 1076×600 pixels, which scales up perfectly to 1366×768 pixels. Anything higher than this and I simply scale the canvas up. This way the game is always rendering at a resolution I am comfortable knowing will perform well. There is very little performance impact for scaling the canvas via CSS. If you were to create your game to run at 2560×1440, which is the native resolution on my 27″ monitor, the game would run incredibly slow.

CHAPTER 4

Handling Game Controls On Windows 8

When it comes to game controls for Windows 8, things can get a little overwhelming if you don't have a plan in place when porting your game. If you remember back to Chapter 1 where we reviewed the different Windows 8 device form factors, you may recall the diversity of Windows 8-capable hardware. With that in mind, the following input options could be available at any time to control your game:

- Mouse
- Keyboard
- Touch
- Pen
- Game Controller
- Accelerometer

Luckily, if you are coming over with an existing HTML5 game, chances are good that you are already using keyboard or mouse controls. In special cases, if your game was designed to run on mobile multi-touch, you may be ready for multi-touch on Windows 8 with a few minor modifications to your code.

The good news is that, if your game only uses the mouse, as in a single-input method, chances are good that you may not have to change a thing. We'll get into this more when I discuss touch-first design. If you are using a keyboard, you may need to offer up additional controls on touchscreen devices. Usually, this is done with a virtual D-pad and buttons or a virtual controller stick.

Either way, it helps to take some time out to really think through the best controls for your game. Bad controls on a game could easily ruin an otherwise amazing creation. Since you are porting the game over to Windows 8, you don't have to make concessions and have a great opportunity to supply the bare minimum touch controls to satisfy tablet

users and encourage them to use something better, like a game controller. We'll explore this and more in the next few sections.

What Is a Touch-First Experience?

In Windows 8 design terminology, you will hear the term "touch-first experience" a lot. Windows 8 was built from the ground up to support touch input as the primary means of input, followed by a mouse and keyboard as secondary input. This has a lot to do with the shift from traditional desktops and laptops to tablet computers. The good news is that a mouse and a finger work similarly, so if you think touch first, a mouse will always work. Of course there are special cases where this may not always be true, especially in games.

The real thing to keep in mind when building any HTML5 game is how it will be controlled. I design most of my games to work with a single touch or mouse click control when possible. This way my games will always work in a browser and also on mobile devices. Because of this, I am able to get my games running with no modification on Windows 8. Unfortunately, this may not always be the case.

You can learn more about touch-first experience for Windows 8 here.

Working with Traditional Input

When I refer to "traditional input" I am talking about a mouse and keyboard. That's because the majority of computers out today have a mouse and keyboard. Touch-enabled devices, such as phones and tablets, are relatively new, especially when it comes to HTML5 games. If your game supports a mouse and keyboard, you will not have to change a thing in your code. Simply listen to keyboard and mouse events like you normally would in a browser. Just remember that if you rely too heavily on the keyboard you are going to want to offer up some additional touch-centric controls for tablet users in order to pass Windows Store certification.

Working with Touch

In Windows 8 apps, we have very granular control over detecting the type of touch input our app is receiving. When it comes to WebKit games, there is a clear distinction between touch and mouse clicks. Usually in a WebKit app, you would do something like this:

```
var touchEnabled = 'createTouch' in document
```

From there, you could split up the logic in your app to handle touch differently than mouse controls.

```
if (touchEnabled) {
  //Touch is enabled
  canvas.addEventListener( 'touchmove', onTouchMove, false );
} else {
  // Use mouse events instead
  canvas.addEventListener( 'mousemove', onMouseMove, false );
}
```

Instead, on Windows 8, you can simply handle all mouse and touch events the same since you have access to an input abstraction class called the MSPointer. On Window 8, we refer to this as "handling touch-first input without compromising mouse," and there is a great article on it here. It's important to note that you can also check if the device is touch enabled, just like you do on WebKit, like so:

```
var touchEnabled = window.navigator.msPointerEnabled
```

To help illustrate how touch and mouse input is handled on Windows 8, take a look at Figure 4-1.

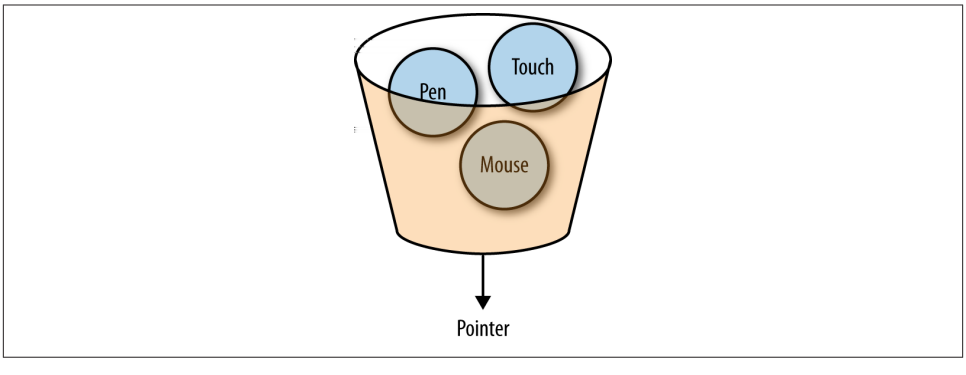


Figure 4-1. Here you can see a graphical representation of how the MSPointer works by encapsulating mouse, touch, and pen input into a single point object.

As you can see, references to a pointer object abstracts the type of input and gives you the necessary x,y values along with other properties you would need to have access to in your game. Let's look at how touch events work in Windows 8 compared to WebKit.

When it comes to WebKit touch events, you would usually set up the following three events:

```
canvas.addEventListener( 'touchstart', onTouchStart, false );
canvas.addEventListener( 'touchmove', onTouchMove, false );
canvas.addEventListener( 'touchend', onTouchEnd, false );
```

This would cover your three main input states. It's similar in Windows 8. The only difference is that you need to reference the MSPointer event instead.

```
canvas.addEventListener('MSPointerDown', onTouchStart, false);
canvas.addEventListener("MSPointerMove", onTouchMove, false);
canvas.addEventListener('MSPointerUp', onTouchEnd, false);
```

As you can see, the two are basically the same. If you wanted, you could easily detect the platform and swap out the correct event ID if you are building a cross-platform HTML5 game. One of the advantages of supporting this in your Windows 8 app is that IE 10 also supports the same touch model, so it will work flawlessly on the Web as well.

From here, there are a few additional differences in the event that you would get back on Windows 8 versus WebKit. In WebKit, you could get the touch point's ID via the .identifier property. In Windows 8, you would use .pointerID. From there, you can continue to access x,y and clientX,clientY just like you would do in WebKit. After that, your code should work basically the same. Also, since input can come from multiple sources in a Windows 8 game, you can test the pointerType of the event object you receive with the following constants:

```
MSPOINTER_TYPE_MOUSE  4
MSPOINTER_TYPE_PEN    3
MSPOINTER_TYPE_TOUCH  2
```

Simply get the value of pointerType from the event and see if it matches up to one of the above values. This is critical if you want your game to show virtual controls when it detects a touch event instead of the mouse. As you start to anticipate your game being played on devices with a mouse and keyboard, you may want to enable or disable the touch controls based on the input type. I have seen this in several apps that will show the onscreen keyboard when a touch is detected versus a mouse click.

I highly suggest checking out a library called Hand.js which is a polyfill for supporting pointer events on every browser. You can learn more about the project here. What is great about this library is that you simply code to the Pointer events and it works on everything from desktop to mobile devices, including iOS and Android without needing to worry about what the capabilities of the device are. You get pointer events with IDs if there is multi-touch and just like MSPointer outlined above, you can detect if it was a mouse, pen or touch.

www.allitebooks.com

Working with Controllers and Game Pads

One of the key advantages of building a game for Windows 8 is the ability to access physical hardware via the USB ports. Of course the first thing that may come to mind is how do I get my game to work with an Xbox controller or other compatible USB gamepad? While WinJS doesn't directly expose access to the Windows 8 controller API, we will go over how to access this via the C++ bridge that allows JavaScript to communicate with the underlying OS.

In order to get started, we are going to need to download this library, which abstracts the Windows 8 C++ controller API and exposes it to JavaScript.

For more information on how the underlying Windows 8 controller API works, make sure to read this.

To get started, copy over the cpp folder to the project and click on Solution and Add Existing Project.

As you can see in Figure 4-2, we have added the GameController project to our Visual Studio solution. Now we need to sort out any dependencies between the two projects. You can do this by opening up the Project Dependencies window.

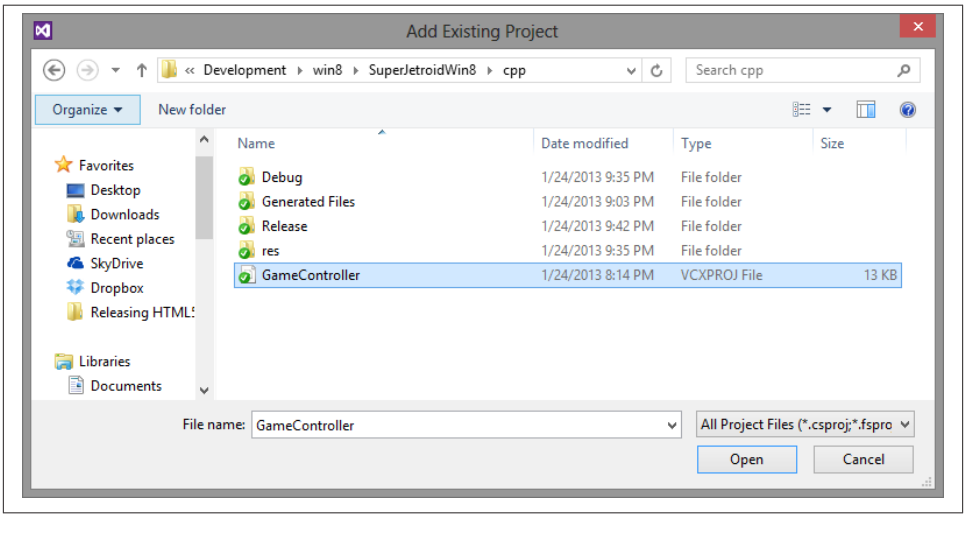


Figure 4-2. Navigate to where you put the GameController project to add it into your game's project solution.

We will need to tell our main game project that it depends on the GameController project. Simply check the box (Figure 4-3). Now we need to reassign the default project for the solution. You can do that by selecting Startup Project.

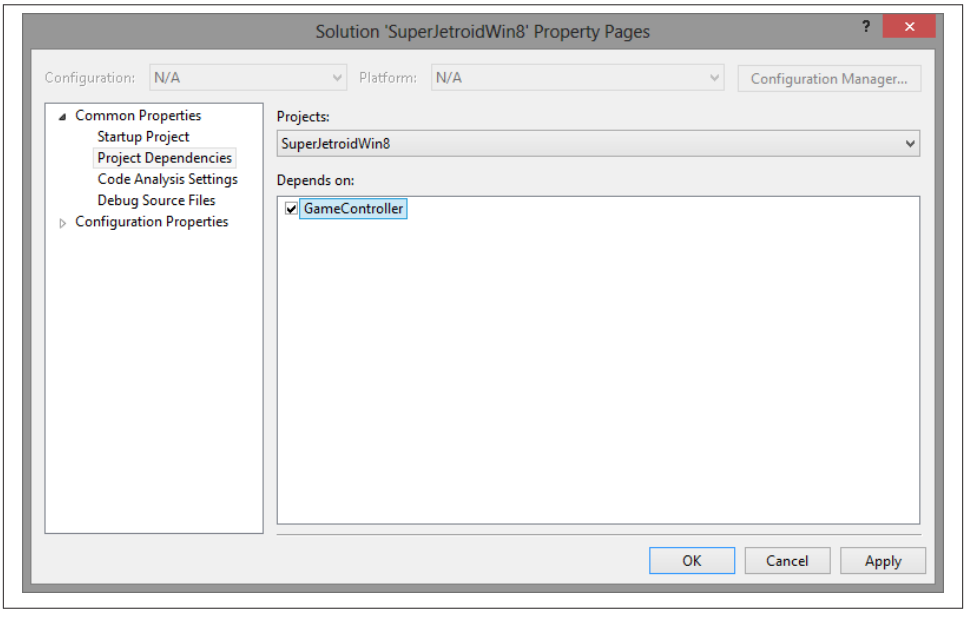


Figure 4-3. Add the GameController project as a dependency.

Here you can see I have selected my default project SuperJetroidWin8 (Figure 4-4), which contains my main game. Once you exit this screen, Visual Studio is probably going to throw a warning (see Figure 4-5).

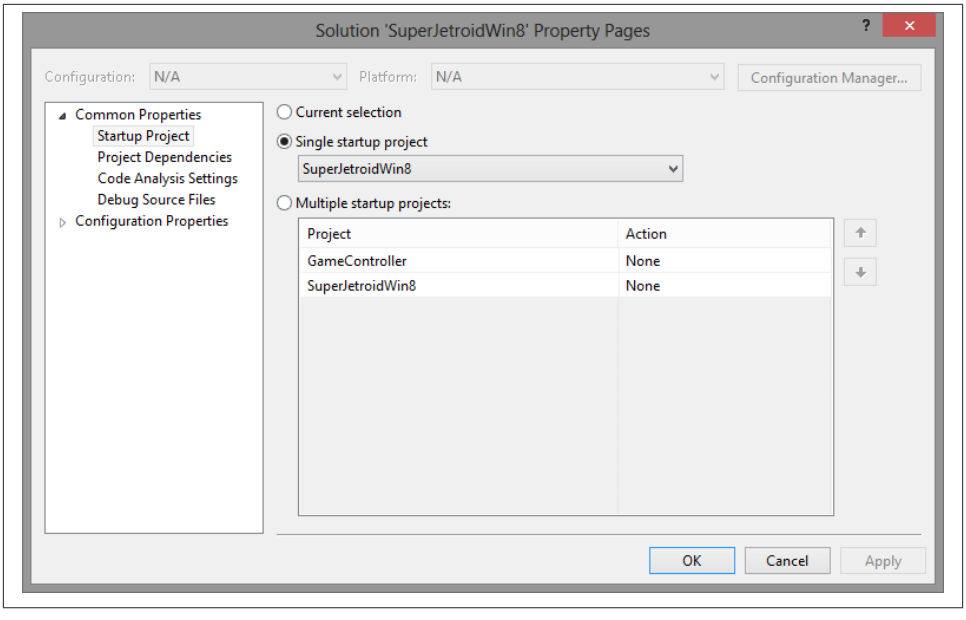


Figure 4-4. Set your game to be the startup project.

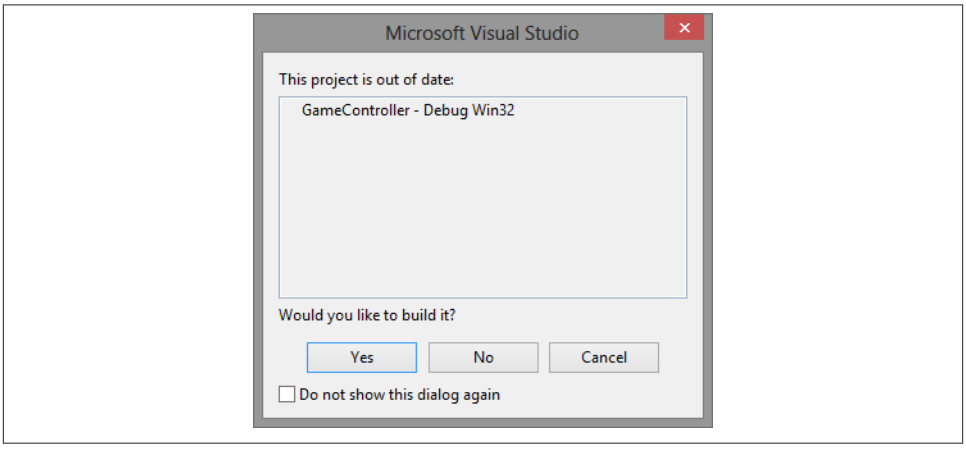


Figure 4-5. Visual Studio will need to rebuild the GameController project once it is set up.

There is nothing to worry about. This happens because the original GameController project just needs to be recompiled so that our main project can reference it properly.

Once the project has been recompiled, you will be ready to add it as a reference to your game's project (Figure 4-6). Simply use the Reference Manager in Visual Studio and you will be able to set this up easily.

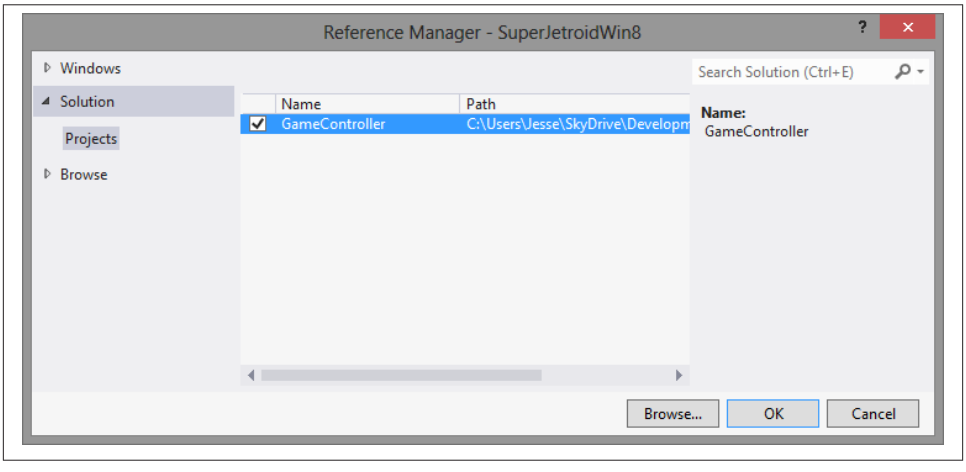


Figure 4-6. Here you can see the GameController project is now referenced in the main game's project.

Now that everything is connected up in Visual Studio, you can start adding support for the controller in your own game. To help you out, I suggest checking out the Controller.h file, which has all the constants you will need to reference in your own code that offers access to various values from the controller. Now let's take a look at how each of these map to a physical Xbox controller (Figure 4-7).

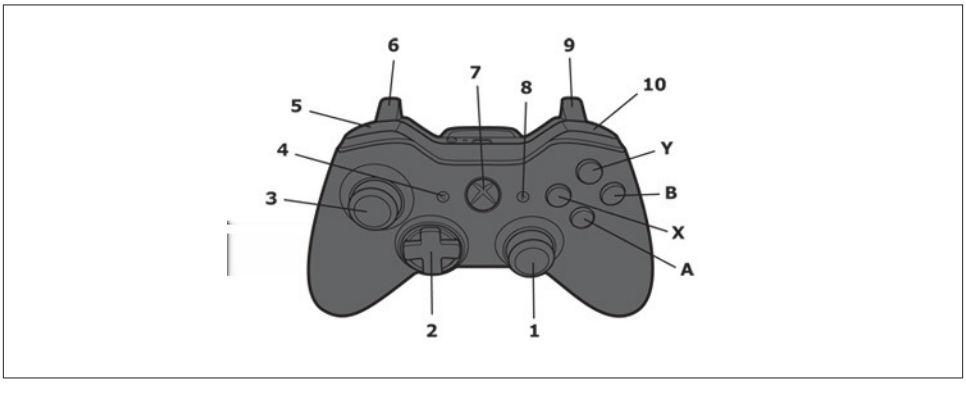


Figure 4-7. Use the following table to match up the buttons on an Xbox controller to their JavaScript equivalent.

Table 4-1. Xbox Controller Key Cheat Sheet

Key	Description	Reference
Y	Y Button (Yellow Button)	y
B	B Button (Red Button)	b
X	X Button (Blue Button)	x
A	A Button (Green Button)	a
1	Right Stick	RightThumbX, RightThumbY
2	Directional Pad (D-pad)	dpad_up, dpad_right, dpad_down, dpad_left
3	Left Stick	LeftThumbX, LeftThumbY
4	Back Button	back
5	Left Bumper	left_shoulder
6	Left Trigger	LeftTrigger
7	Guide Button	N/A
8	Start Button	start
9	Right Trigger	RightTrigger
10	Right Bumper	Right_shoulder

Getting this up and running in your own game is now going to be relatively easy. First, you are going to need to set up a reference to the controller. You can do this in your default.js class with the following line of code in the app.activate function right before you initialize your game.

```
// Although the API supports up to 4 controllers per machine,
// this sample only works with a single controller.
controller = new GameController.Controller(0);
```

Now in your game, you can simply query the controller for its state. Here is an example of how I do this in one of my own games. I use this in the update loop of my game to make sure I capture the controller events before anything else is updated in the game.

```
// Joystick controls
if (typeof(controller) != "undefined") {
    var state = controller.getState();

    if (state.connected) {

        if (state.dpad_right)
            this.player.rightDown();
        else {
            this.player.rightReleased();
        }

        if (state.dpad_left)
            this.player.leftDown();
        else {
```

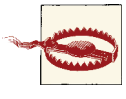
```
                this.player.leftReleased();
            }

            if (state.dpad_up || state.a)
                this.player.jumpDown();


            if (state.start)
                this.togglePause();

        }
    } else {
        //console.log("no controller");
    }
```

Once you have this basic logic set up in your own game, you shouldn't have any issues accessing the other buttons on the game controller, such as the analog sticks, triggers, and even the back and start buttons.



It's important that you make your entire game controllable via a gamepad if you chose to support one in your game. That means that everything from starting the game to navigating menus should all work with the controller. The last thing you want players to do is put down the controller in a pause screen or to start the game. This means adding a little extra logic to your game's UI, but one of the tricks I use is to also map these controls to the keyboard so you can support both and the user doesn't have to touch the mouse if there is no need to.

Knowing When to Use What

As you can see, all of this can really be overwhelming. I have put together scenarios to help you pick when to support touch or physical (see Table 4-2).

Table 4-2. Windows 8 Game Controls Cheat Sheet

Input Type	Support Physical Controls	Support Touch Controls	Support Controller	Notes
Keyboard	Yes	No	No	Only support keyboard and hide virtual controls
Mouse	Yes	Possibly	NA	Mouse can activate touch controls if you have a virtual joystick but may not want to show if using virtual D-pad
Touch	No	Yes	No	Ignore physical controls when working with touch events
Controller	No	No	Yes	Ignore all other input types when using a controller

Table 4-2 should help you break out your controller logic to be more contextual to what a user would expect based on the type of input they are using. Of course, you can simply keep touch controls on the entire time and always check for a physical controller, but a big part of publishing any game is the level of polish you add to the final product. It would be silly to see touch controls on a desktop with no touch screen at all.

Tips and Tricks for Windows 8 Game Input

We have covered a lot of material in this chapter, but I wanted to share some tips and tricks that have helped me when making my own games.

Contextual Controls

If you are supporting touch controls and the device has a keyboard, you should hide the touch controls when the keyboard is showing and show them when you detect a touch event. The Windows 8 virtual keyboard is a great example of this in action. If you use a standard keyboard, you never see the virtual one until you touch the screen with your finger. Reference the chart in the previous section to solve this problem.

Avoid Configuration Screens

It's easy to throw all of your input options into a settings window and let the user pick their own, but a well-designed Windows 8 game should automatically detect the correct input the player is using. This doesn't mean that you shouldn't have a way to let the player customize or disable controls that are not used, but keep in mind the user just wants to play a game. The less work they have to do to set it up the better.

Instruction Screen

The final and most important tip should be self-explanatory, but make sure you tell your user about all of the control options, or at least what the keys are.

As you can see in Figure 4-8, I have a simple illustration that shows how to use the controls for my game Super Paper Monster Smasher. This plays in the "attract loop" of the game, which is a series of screens that animate if you stay on the splash screen for a set amount of time and don't start a game. This was a very popular technique for arcade games, and one I make sure to include in each of my own games.

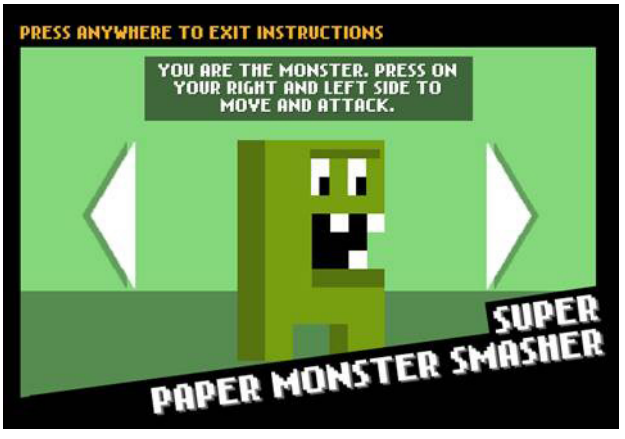


Figure 4-8. The instructions screen for Super Paper Monster Smasher.

CHAPTER 5
Debugging and Optimization

One of the most rewarding parts about building HTML5 games for Windows 8 is the fact that, if your game is already running in the browser on the desktop, chances are good that it will run perfectly as a Windows Store app. Of course, in the real world, nothing works exactly like you expect it to. This chapter will go over some of the best ways to optimize your game so that it runs well on your computer and all the other Windows 8 devices out there.

Using the Console

If you are a fan of console.log(), you'll be happy to know that all of your log traces will still show up in Visual Studio's console window. This means the quickest way to debug your app is probably to throw a console.log() statement in there and see what's happening. There are a few things that Visual Studio doesn't support in the console output, which you may be used to in a browser log window:

- Visual Studio will not output the structure of an object. This means you simply get a string equivalent of the object. For exploring objects, you will need to use the debugger and a breakpoint, which we will talk about next.
- The console also doesn't automatically add spaces to multiple items separated by a comma. So, if you do `console.log("show", "me", "this")` in the console, you will see "showmethis." In WebKit browsers, the console would give you "show me this." It's not a deal breaker but something to keep in mind if you spend a lot of time trying to output multiple items in a single log statement.

Finally, the most important feature of the console is the ability to execute code or modify values while your app is running. Simply type into the console window and you will be able to test code by hand, just like you would do in a browser's console window.

Debugging and Breakpoints

For heavy-duty debugging, I would use breakpoints and Visual Studio's debugger. Hopefully you are familiar with how breakpoints work. For a quick refresher, you simply make a line of code where it should pause when executing. This allows you to explore the state of the application and view any objects in memory. To add a breakpoint, left click in the grey area on the left-hand side of the line numbers running down the editor (Figure 5-1).

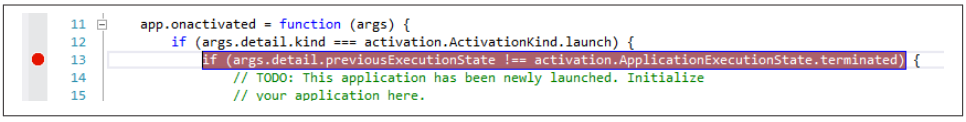


Figure 5-1. Here is what a breakpoint looks like when the code is executed and stopped for you to debug it.

Once you do this a red dot will show up. When you run the app, the debugger will pause on that line and throw you back into Visual Studio. At this point you can start exploring the objects in your game. Simply roll over an element, such as the argument being passed into a function, and you will be able to see everything within that scope (Figure 5-2).

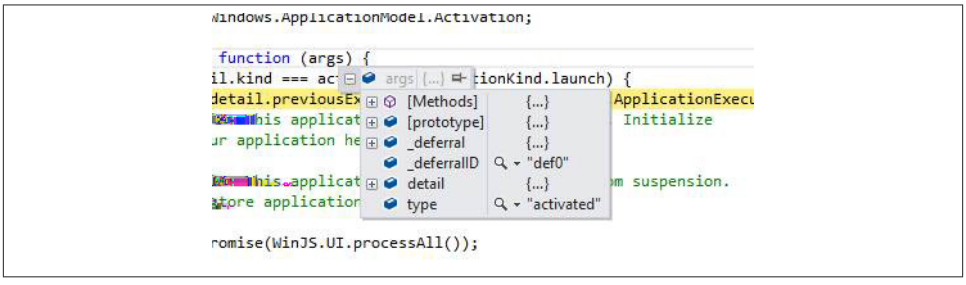


Figure 5-2. Once the debugger has stopped at a breakpoint, you can explore any object in the code to see its properties and methods.

This is incredibly helpful since you can see the properties of an object and also all of the methods associated with it.

If you are doing lots of object inspection and need to keep track of each one, try adding them to a watch list. Right click on an object in the debugger when at a breakpoint and you will get a contextual menu with additonal debugger features to track that instance while your game is running in debug mode (Figure 5-3).

www.allitebooks.com

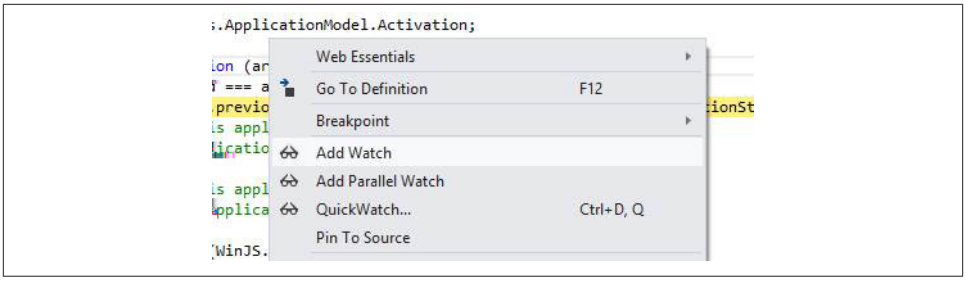


Figure 5-3. You can also add objects to a watch list for easy reference as you explore other parts of the application.

Once you watch an object it will be added to the watch panel (Figure 5-4).

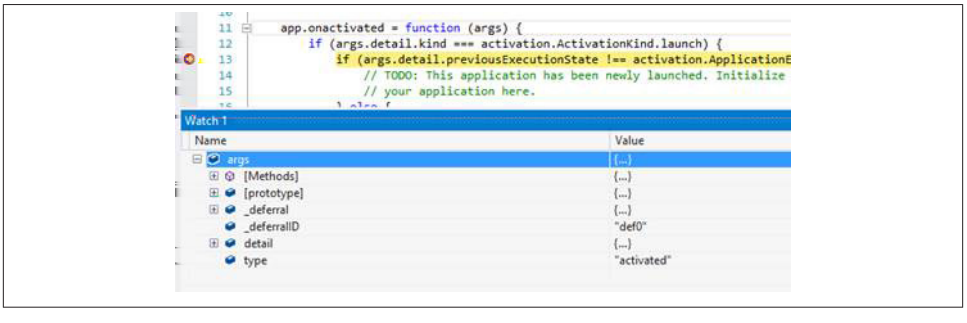


Figure 5-4. Here is the object in the watch panel.

Now you can continue moving throuh multiple breakpoints and still keep an eye on how an instance of an object or varible is behaving. To move forward through the debug process, use the buttons at the top of the Visual Studio toolbar (Figure 5-5).

Figure 5-5. Once your game is running, the compile toolbar changes to the debug panel with new options.

From left to right: pause, stop, restart, refresh, next statement, step into, step over, and step out. As you can see, the debugger is an incredibly powerful tool and much better than simply using the console to log text.

DOM Explorer

The DOM Explorer is one of the most powerful tools you have when it comes to inspecting HTML markup structure and CSS styling for Windows 8 apps. It is similar to the F12 tools you would find in IE. To activate it, simply go to Debug -> Windows -> DOM Explorer (Figure 5-6).

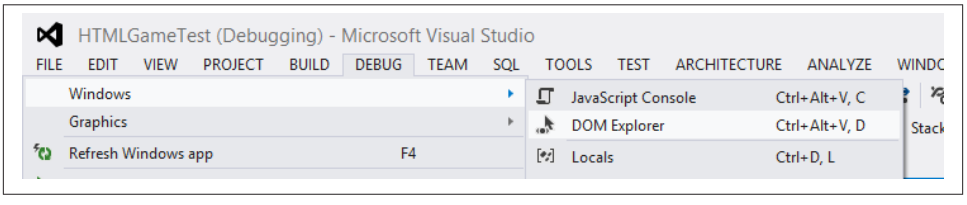


Figure 5-6. You can easily pull up the DOM Explorer via the debug menu.

Once it opens, you will be presented with the currently running HTML page and its elements (Figure 5-7). You can explore them and view styles just like you would in IE.

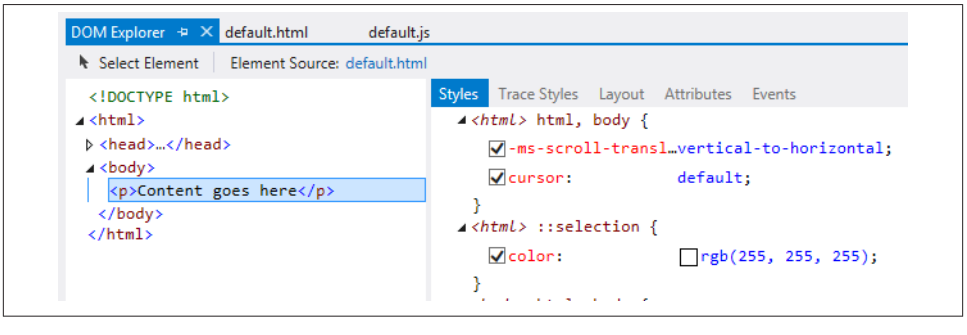


Figure 5-7. The DOM Explorer is incredibly powerful, especially if you are doing any HTML-based layout in your game.

Remote Debugging

One of the most important ways to test out the performance of your game is to run it on other devices. Luckily you can do this easily by setting up a Windows 8 device for remote debugging. To get started you are going to need to download the Remote Tools for Visual Studio, which you can find here.

You will see under the Remote Tools for Visual Studio tab three different builds: one for x86, one for x64, and one for ARM. Once you have this app installed on your remote test device, load it up and go back into Visual Studio. From there you can select Remote Machine from the build drop down (Figure 5-8).

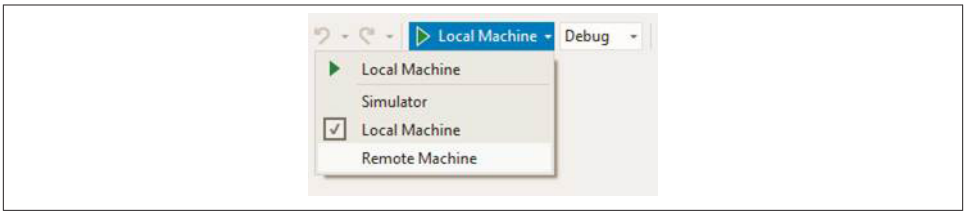


Figure 5-8. The different deployment options in Visual Studio.

Visual Studio will now ask you to configure the remote machine (Figure 5-9).

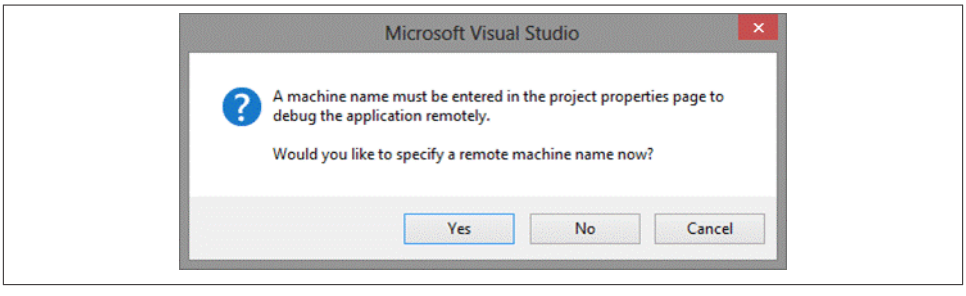


Figure 5-9. Setting up a remote machine for debugging.

This will bring open a configuration panel (Figure 5-10) with options for what happens when you do a remote deploy. You can configure if the app will automatically launch on the test device, give the device a name, and some other non-critical options.

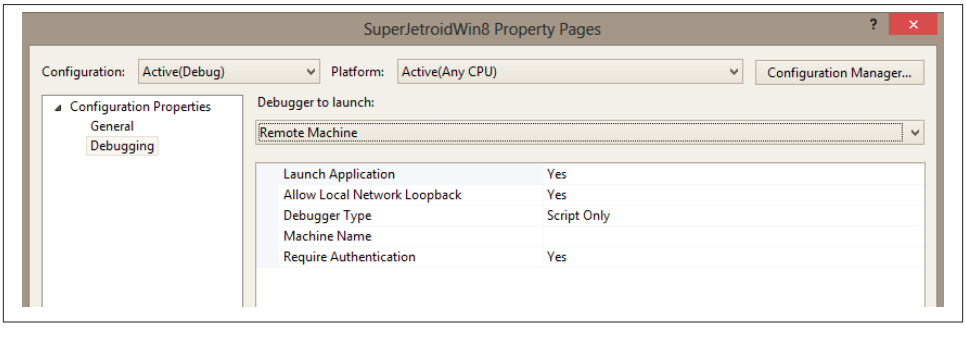


Figure 5-10. Configuring the remote deployment options in Visual Studio.

To set up the Machine Name, you will need to click on the empty name field and then you will be able to scan the local network for any Windows 8 devices running the remote tools (Figure 5-11).

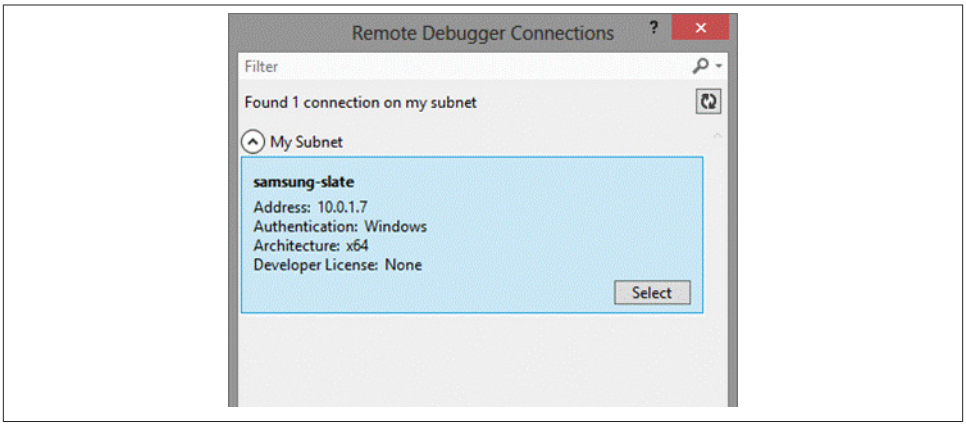


Figure 5-11. Visual Studio has found the remote machine.

Once you have found your device, simply select it and its name will automatically be filled into the empty field you saw on the previous window (Figure 5-12).

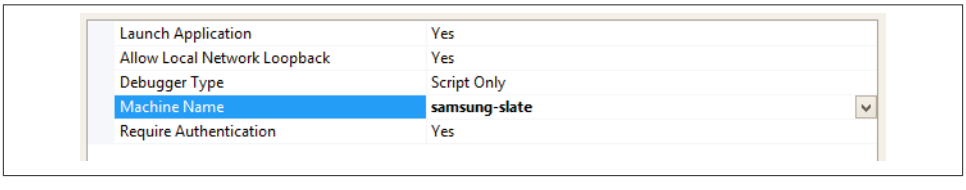


Figure 5-12. Once selected, the remote machine's name will show up.

From here you will need to set up a developer license, which is automated for you via the dialog box that pops up (Figure 5-13).

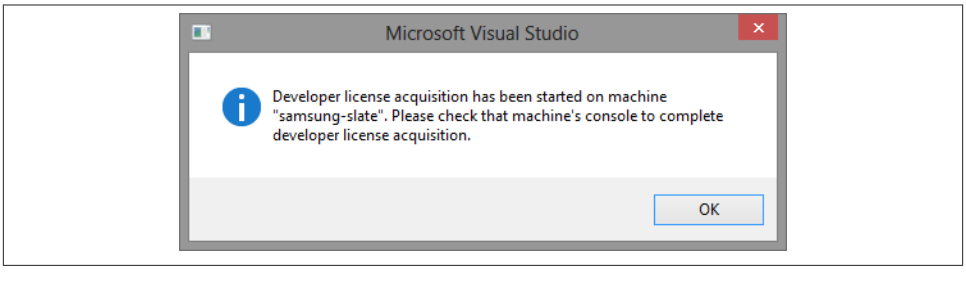


Figure 5-13. You will need to set up a developer license once everything is connected up.

Now you should be ready to deploy to the remote device. Simply hit compile (Figure 5-14) like you normally would with a local build and the game will show up on the remote device and automatically load up the game for you to start testing.

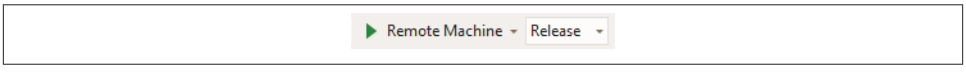


Figure 5-14. Now you can deploy your game to the remote machine inside of Visual Studio.

The best part about remote deploying is that you continue to use all of the same debugging tools I outlined previously in the chapter over the network connection. If you are working on a single monitor setup, this will allow you to continue testing the game out on one device while still being able to debug it on your main development computer.

Optimizing Graphics

I have already talked about ways to get the best-looking visuals out of you games, but I wanted to dig in a little deeper to a way of optimizing your artwork. The first thing you should do is optimize the artwork itself. Image editors, such as Photoshop and Fireworks, offer ways of compressing artwork for the Web, and I highly suggest looking into these solutions. Most of the time, the difference between using 8-bit PNGs versus 24-bit PNGs can make all the difference between how long a game loads up on the Web and on Windows 8.

Another solution, which I talked about at the end of Chapter 3 was using something called texture atlases. Texture atlases are similar to sprite sheets in that you group a lot of artwork into a single image. Texture atlases, however, rely on another file, the atlas, which defines the x,y (width and height) of the actual sprite in the texture image. What is great about using texture atlases is that you can include images that are not perfectly square in the same sheet. Sprites generally are divisible by their width and height; images in texture atlases can be any size. Let's look at an example (Figure 5-15).

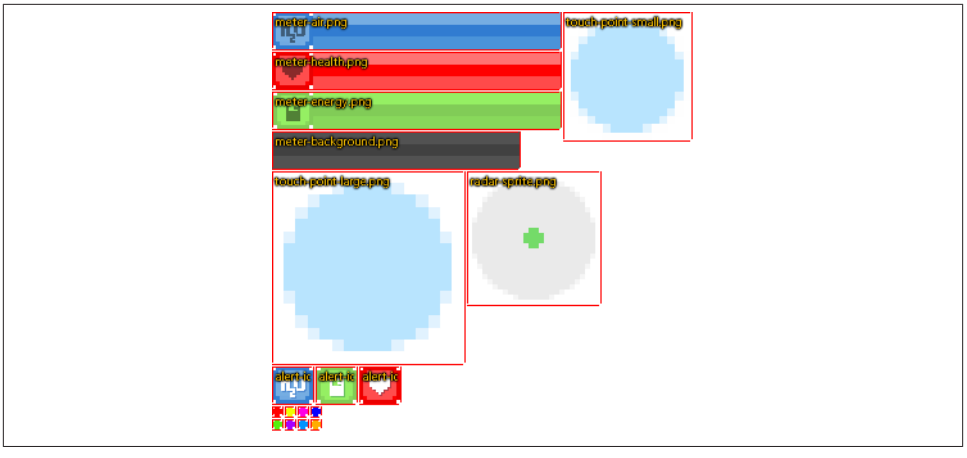


Figure 5-15. An example of a texture atlas for Super Jetroid.

And here is a sample of the JSON data I use to pull out each sprite.

```
{
    "filename": "alert-icon-air-on.png", "rotated": false,"trimmed": true,
    "frame": {"x":0,"y":276,"w":32,"h":29},
    "spriteSourceSize": {"x":0,"y":0,"w":32,"h":29},
    "sourceSize": {"w":32,"h":29}
}
```

The other advantage of texture atlases is not having to load in lots of individual sprite sheets for your game. Since I can put all of my entity and UI artwork in the same texture, I can dramatically cut down on my game's loading time. While most browsers cap out at six connections at a time, being able to put all of your artwork into less images, even if they are larger, will make your game load faster. Also, when you have all of your artwork in a single image, you can perform more advanced compression on the entire set, helping bring down your artwork size even more.

Figure 5-16 is a snapshot of the amount of requests my game was making before moving over to texture atlases. Figure 5-17 shows the optimized build where about 90 percent of the artwork is now being loaded via texture atlases.

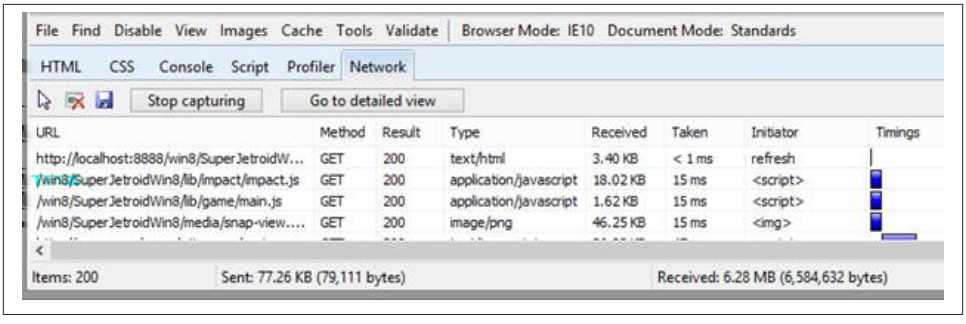


Figure 5-16. Here you can see the total number of connections before moving over to texture atlases.

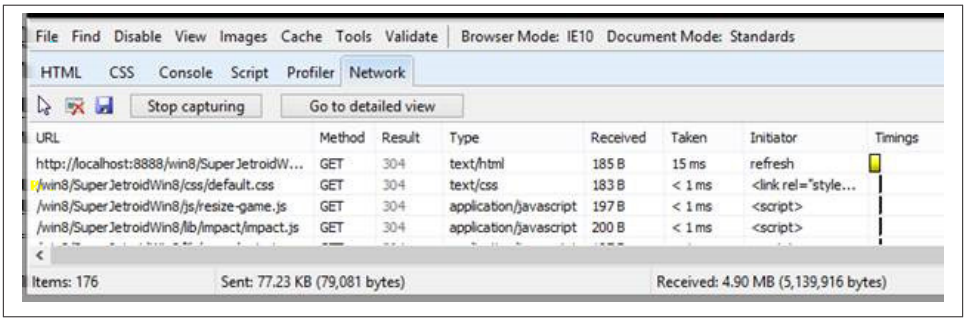


Figure 5-17. The number of connections is now lower after moving over to texture atlases.

As you can see when comparing Figure 5-16's Items count to Figure 5-17, I was able to cut down 12 percent of my requests going from 200 down to 176. While that may not sound like a big savings, check out the size of the game. My game went from 6.8mb down to 4.9mb, which is almost a 23 percent reduction in file size. This is huge if you are trying to run your HTML5 game on mobile browsers and also improves the memory footprint on any platform.

All of these Web-based optimizations directly translate into performance improvements on the Windows 8 side of things as well. The reduction in file size makes it easier for users to download your game and store it on their computer, and it lowers the overall memory footprint as well. This last point is especially critical on lower-powered Windows 8 devices that may only have 32 or 64 gigs of storage and only two gigs of memory.

There are a lot of great texture packagers out there and some are free, such as ShoeBox, while others cost money but are more robust, such as TexturePacker.

Optimizing Code

It's next to impossible to cover all of the ways you can optimize the JavaScript in the scope of this book, but I have a handy list of "code smells," which is a developer term for code that could contain deep problems you should look out for before publishing:

- Always use multiplication instead of division and avoid other complex math equations, such as square roots, unless it is absolutely needed.
- Avoid complex, deeply nested loops. If you have multiple nested loops, try to simplify them and see if you can get away with a single loop. Loops are code blockers, meaning nothing can execute until they complete, so try to limit or avoid them where possible.
- Use RequestAnimationFrame instead of SetInterval for your game loop. I had a dramatic speed increase in my own games by making this switch, and it should help anyone making a Windows 8 game or even a Web version of their game on modern browsers.

For the difficult-to-track-down performance issues, I suggest taking advantage of Visual Studio's Performance Explorer, which you can find under the Analyze -> Window menu. Figure 5-18 shows when I run my own game.

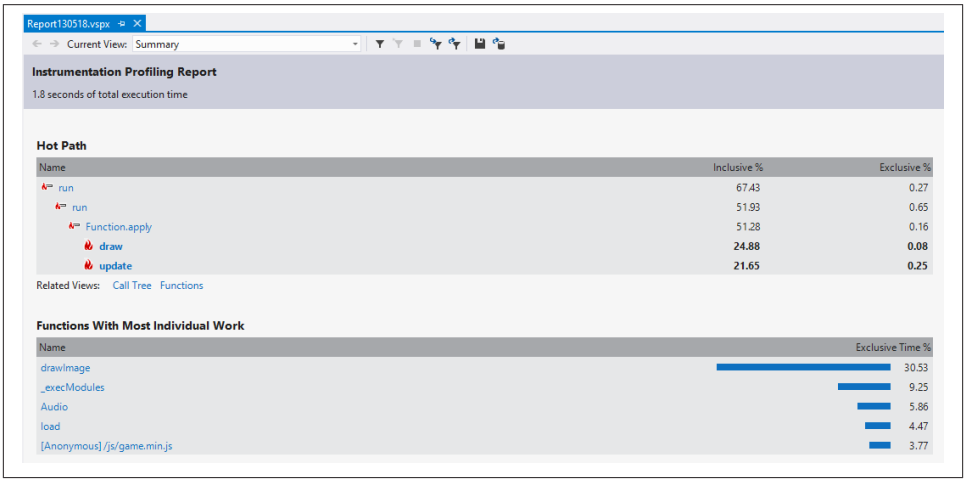


Figure 5-18. Here you can see the Performance Explorer running for my game.

There isn't much out of the ordinary in here but, as you can see, my game loop's two main functions (draw and update) are in the "Hot Path" showing what methods are taking up a lot of resources. There is not much I can do about this outside of trying to lower the number of executions in these two calls, but you may see something else

showing up on here that you may have overlooked when building the game for the browser.

Optimizing for the Lowest Common Denominator

This is a common technique when building any type of game; you always want to build for the lowest end of your game's specs. More times than not, your audience will be using slower computers than what you are developing on. Even if your game runs great on a top-of-the-line gaming PC, you still need to take into account the slower end of the computing spectrum: budget PCs. One of the best ways to test is to have a device of your own. In the beginning of the book I talked about different form factors, but here is a basic breakdown of processors you will want to support:

- ARM – The slowest Windows 8 devices, these are basically mobile devices. These will be on par with other mobile devices, such as phones and tablets. They also have the lowest resolution.
- Low-Power x86 Chips – These are Atom-class processors and are faster than ARM but still have issues with very taxing graphics and calculations. These devices have similar resolutions as ARM-based devices.
- High-Power x86 and x64 Chips – These are top-of-the-line processors you would find in full desktops, laptops, and even some Windows 8 tablets, such as the Surface Pro. These can be paired with integrated graphics chips or full standalone GPUs.

I tend to code for the ARM devices. My goal is that if it doesn't run on ARM then the game is not shippable. Of course you can simply publish an x86- or x64-based game, but you are limiting your reach across the full Windows 8 platform. I highly suggest testing on an ARM device as much as possible while getting your game up and running on Windows 8. If your game works on ARM, you can probably skip the low-powered processors as long as your main develop machine has one of the higher-end processors.

Tips and Tricks for Further Optimization

As you already probably know, optimizing JavaScript based games is part trail and error and part black magic. While I did my best to cover a lot of the common ways to get better performance on Windows 8 here are some addition tips to try out.

Debug Builds Are Slower Than Production Builds

As you end up testing your game on your local computer or on a remote device, you may notice some serious performance slowdowns depending on the hardware running the game. At first you may get nervous and think your game isn't going to run well in production, but I have noticed these slowdowns on slower hardware. If you are doing

testing and need to use the debugger, see if you can work through the slowdowns. If you don't need to debug, I suggest exiting the game by stopping the Visual Studio build, closing the game, and reopening it from the start screen. It will run at full speed and you will hopefully see a dramatic difference in performance.

Avoid Multiple Draw Calls to the Canvas

You should be keeping track of the number of draw calls you make to Canvas. Every draw call has the potential of slowing down your game. Unfortunately, there aren't any debug tools to help optimize rendering performance in your game unless the game framework you are using has it built in. Limit the number of draw calls by caching complex renderings to an off-screen canvas ahead of time or by trying to pre-render graphics wherever possible. A good example of something that may cause performance problems would be manually rendering pixel fonts. If every letter is a draw to Canvas, large text or text in the UI can quickly slow down your game's performance. Speaking of fonts, I would also avoid drawing Canvas fonts as well.

Use Best Practices

There are lots of great websites and resources out there on best practices for optimizing JavaScript and anything IE 10 specific will work well in Windows 8 HTML5 apps. Take some time out to get up to speed with the best practices of developing for IE 10 and you will be able to get some extra performance out of your game. Remember, at the end of the day this is no different than any other JavaScript based web game, it just happens to run natively on Windows 8.

CHAPTER 6

Publishing Your Game to the Windows 8 Store

It is incredibly easy to set up a developer account for the Windows Store. Like every app store out there you will have to register, pay a developer fee, and work through a submission process in order to launch your game. Fortunately, submitting to the Windows Store is a painless process, and it all starts by creating your developer account here.

From here you will need to register as a developer. I suggest getting a Microsoft Outlook account before registering. You may have already done this when setting up Windows 8 on your computer. This will allow you to keep your PCs in sync and keep all your account information in one place. Likewise, you can use this account to log into everything you need on Microsoft's developer accounts and resources.

When you are ready to continue with the registration you will have to pay a small fee. The registration fee for individuals is $49, with a $99 fee for companies. The revenue share is 70 percent to the developer, but when an app achieves $25,000 in revenue—aggregated across all sales in every market—that changes to 80 percent revenue share for the rest of the lifetime of the app. So, unlike other markets, the more you sell on the Windows Store the higher your revenue share goes, which is a great deal!

After paying the fee, you should be all set with a developer account and ready to publish your first game.

Reserving Your Game's Name

It's important to reserve your game's name as soon as you have a developer account. These names are unique in the store, similar to domain names, so if you want to make sure your game's name doesn't get taken you should register it quickly. Also, you can register multiple app names, so don't feel pressured if you haven't finished your other game ideas yet. Simply select the option to create a new App in the developer dashboard and you can begin the process of registering your game's name, see Figure 6-1. After you have reserved your game's name you can start filling out details, such as the description, screenshots, and other relevant information, including price and markets you want to distribute your game in.

As you can imagine, this should all be self-explanatory, but I have included a quick walkthrough with some screenshots so you know what to expect when you are ready to do it. Usually the hardest part of releasing a game is simply filling out the forms and preparing all the assets you need to be approved on the store.

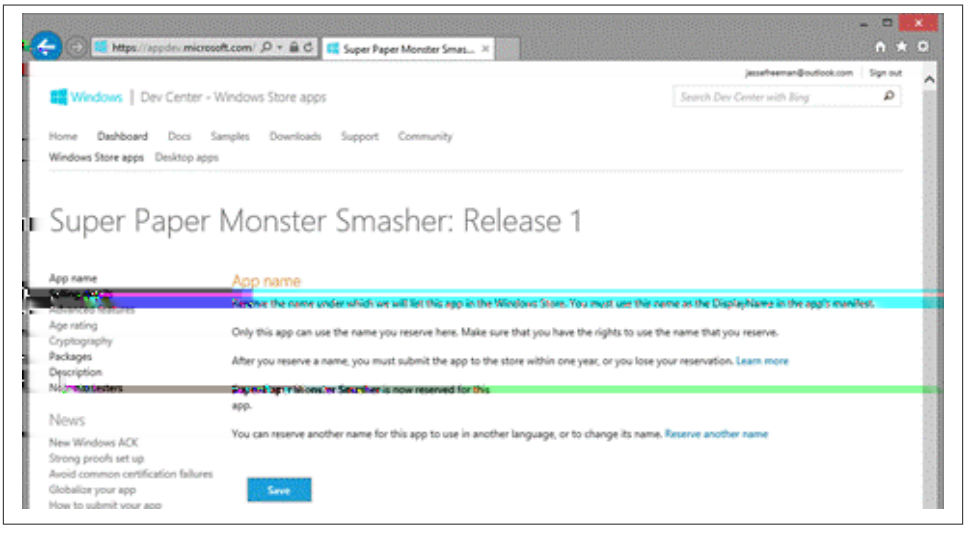


Figure 6-1. Here you can see I have registered my game's name.

Now you will need to add some key data to get the game ready for release. This will include your desired release date, what categories your game falls into, and hardware requirements (Figure 6-2).

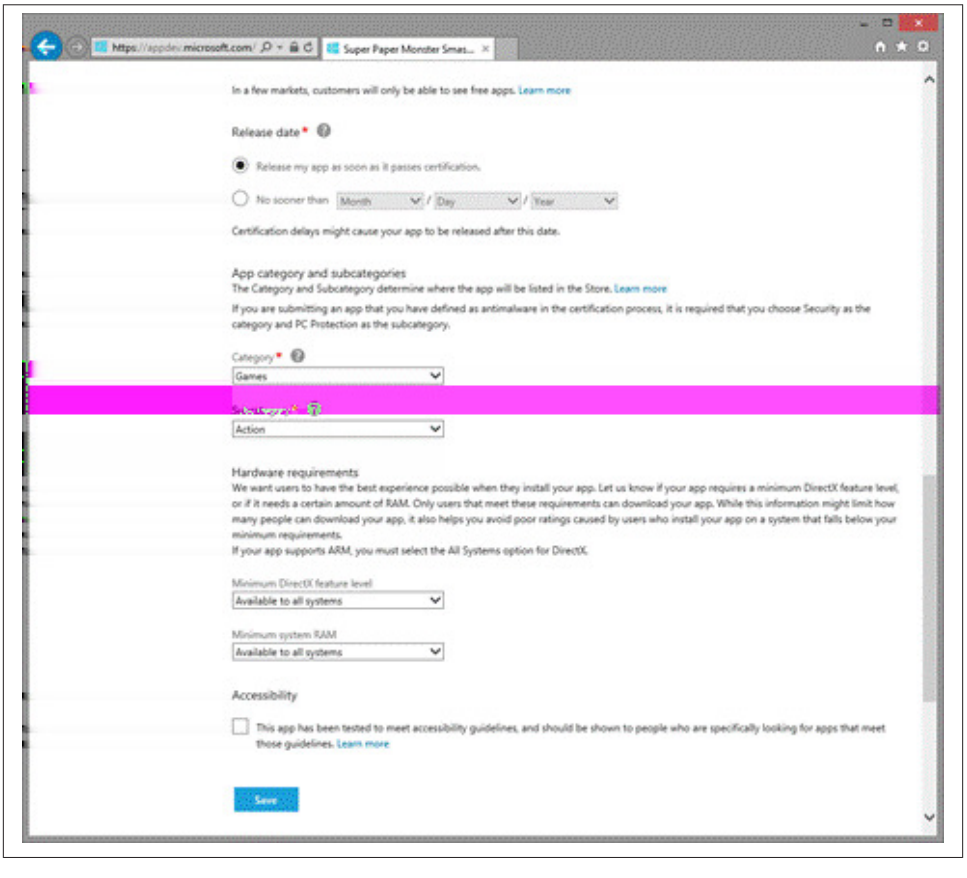


Figure 6-2. Here you can set up release date, category, and other important options.

It's important to also pick the appropriate age range for your game (Figure 6-3). Be conservative or you may not pass the submission process.

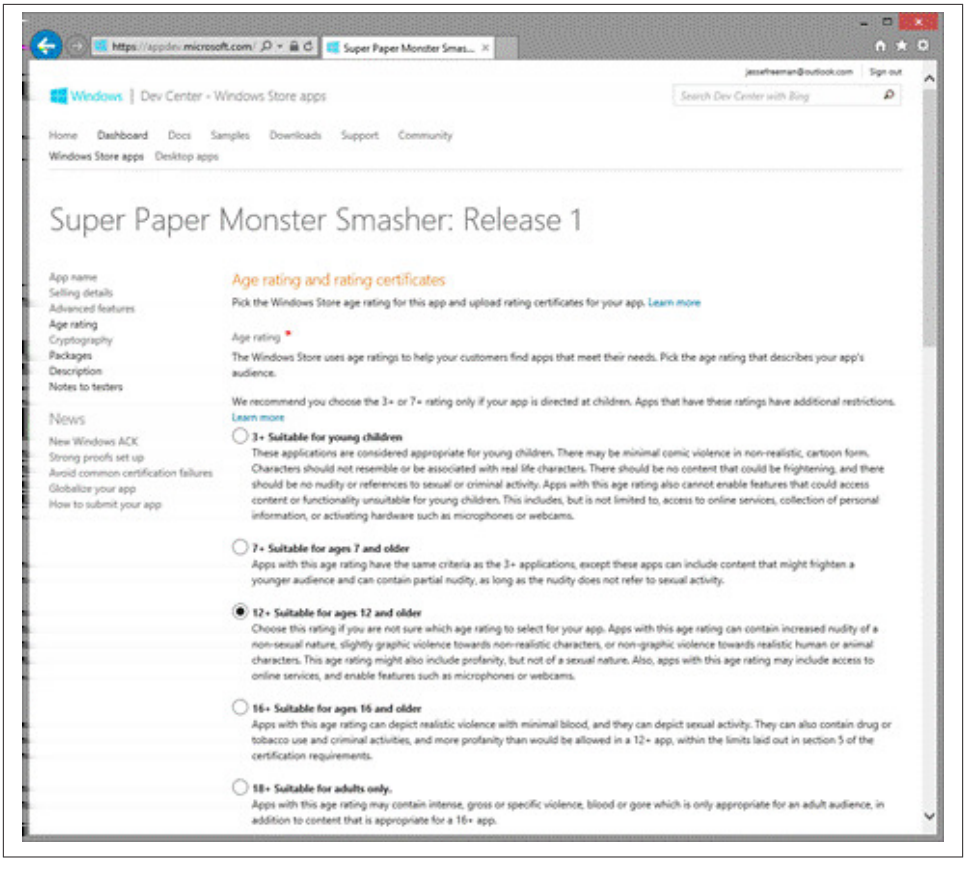


Figure 6-3. It's important to pick the correct age rating and rating certificates (if needed).

There is no need to harp on the details of this process. The Windows Store submission process is well laid out and broken up into clear tasks that should easily walk you through the process of getting your game ready for submission. Plus, you can do it at your own pace. Once you fill out a section you can return to the rest of the submission process at any time, so don't feel pressured to do it all in one sitting. Also make sure to check out Chapter 7 for tips on monetization and how to sell your app on the store once it's ready for submission.

Submitting Your Game with Visual Studio

When you are finally ready to publish your game, you will be happy to know that it can easily be done through Visual Studio. Simply go to the Project menu in Visual Studio and go down to the Store option.

If you are using Visual Studio Express, the option for publishing to the store is in its own menu next to Tools at the top. Other than that, the same process applies.

Select "Upload App Packages" as you can see in Figure 6-4 and Visual Studio will begin the process for validating, testing, and submitting your game. To do this, Visual Studio must create an app package (Figure 6-5).

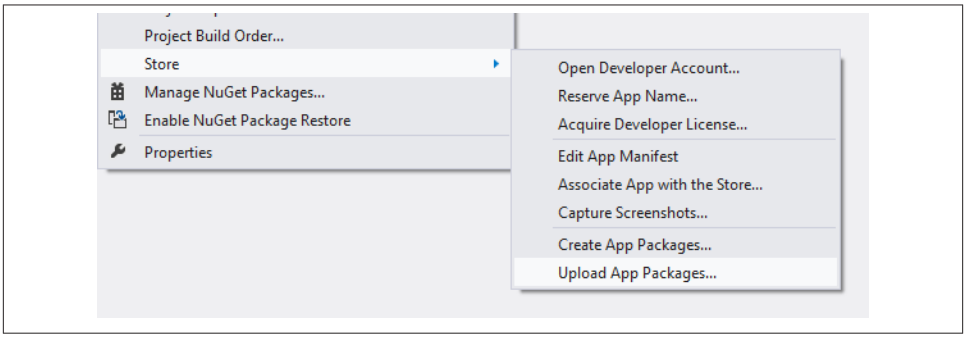


Figure 6-4. In the Project menu you will find options for submitting a game to the store.

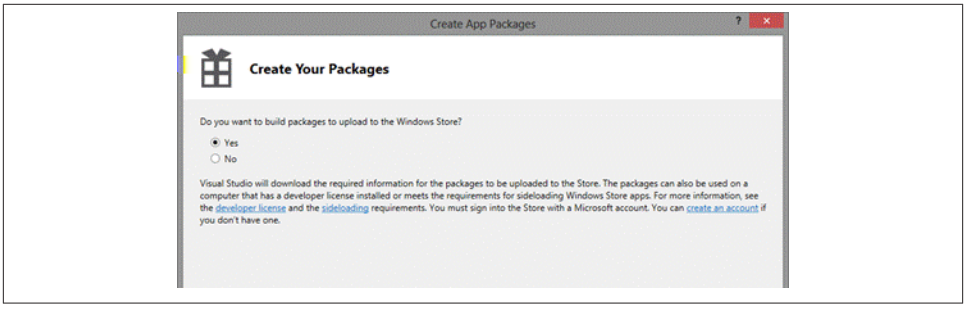


Figure 6-5. The Visual Studio wizard will walk you through creating an app package for your game.

As you continue, you will need to select the reserved name of the project you want to upload to. Figure 6-6 shows a few projects registered that I have yet to complete.

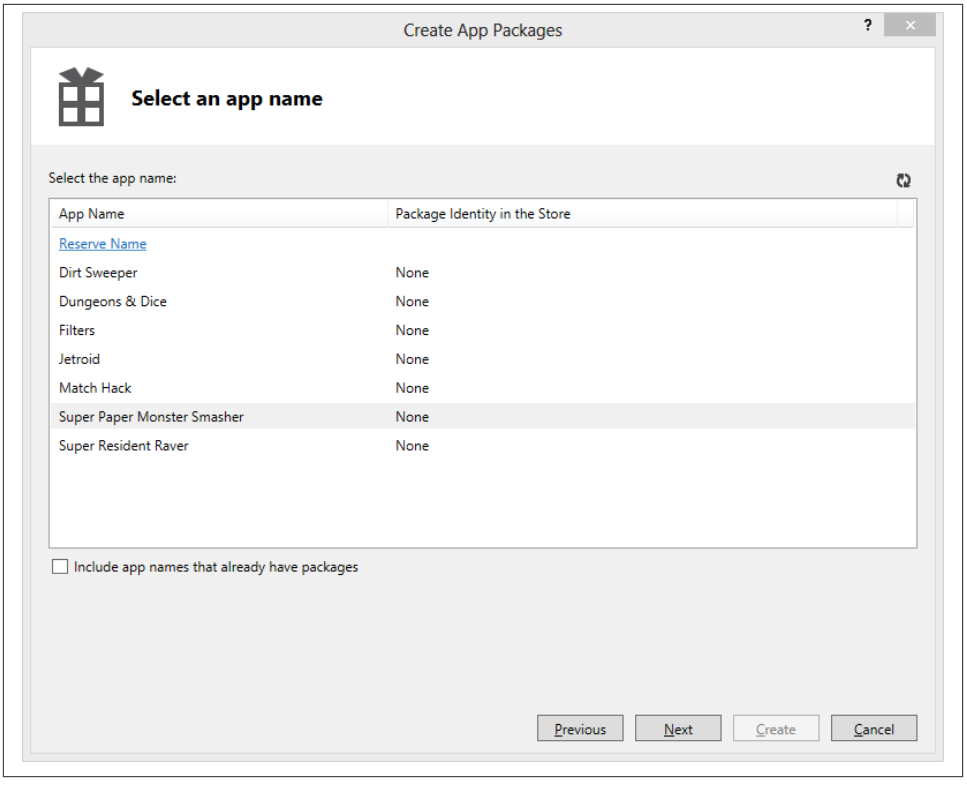


Figure 6-6. Here I am getting ready to submit my Super Paper Monster Smasher game to the store.

It's important to highlight two things here. First, you can also reserve a name from the panel in Figure 6-6 so, if you haven't set one up yet, you still can without exiting the Create App Packages wizard.

I would suggest getting in the habit of reserving an app name as soon as you create a new Windows 8 project in Visual Studio. This way you can rest assured you will get the name you are looking for right up front.

Second, you should leave "Include app names that already have packages" unchecked. This is used for submitting an update to an existing app. We will discuss submitting updates in the next section.

Next, you will need to set up the version number and platforms you want to release your game on (Figure 6-7).

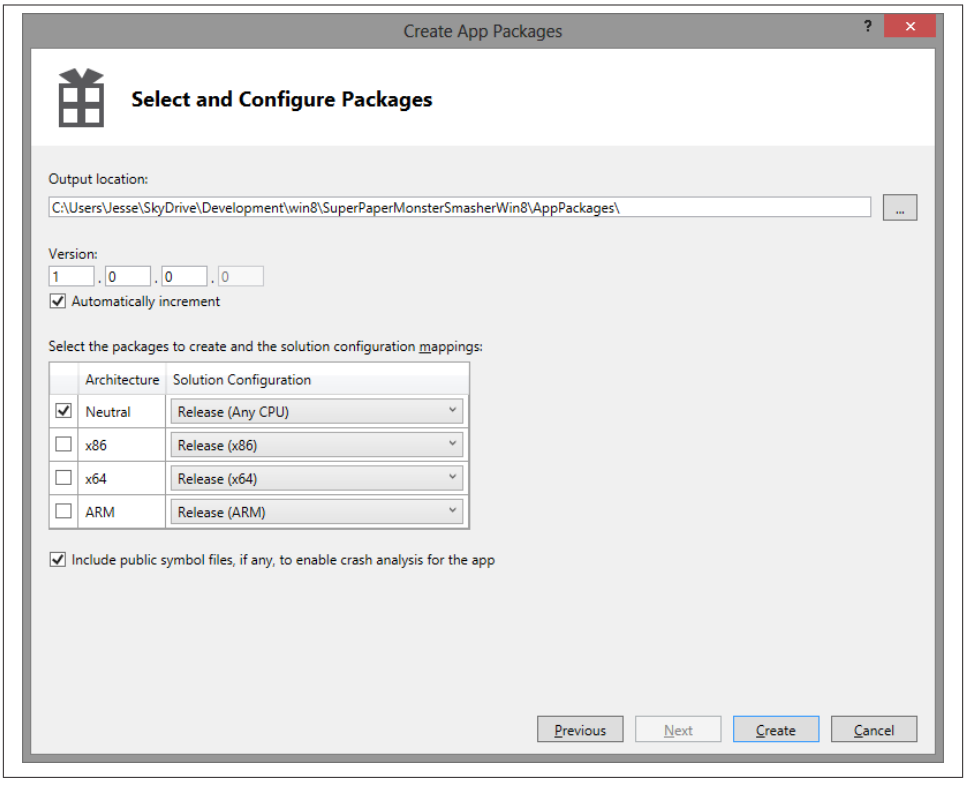
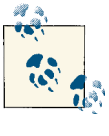


Figure 6-7. Set up the version number and platforms your game will run on.

As you can see, you can choose x86, x64, and ARM as the desired platform architecture, or Neutral for all of them. If you are building an HTML5 game there shouldn't be any issues having it run on all platform architectures, outside of performance differences.



It's important to take note of the output location. This is where your app package will be saved. You will need to know this path when you are finally ready to upload the package in the last step of this process.

I also leave the version number to be automatically incremented for me. This takes care of continual updates if you run into any issues during the approval process so you don't have to manually manage the version number when submitting.

Now you should be ready to validate your app. You will see a screen that prompts you to Launch Windows App Certification Kit (Figure 6-8).

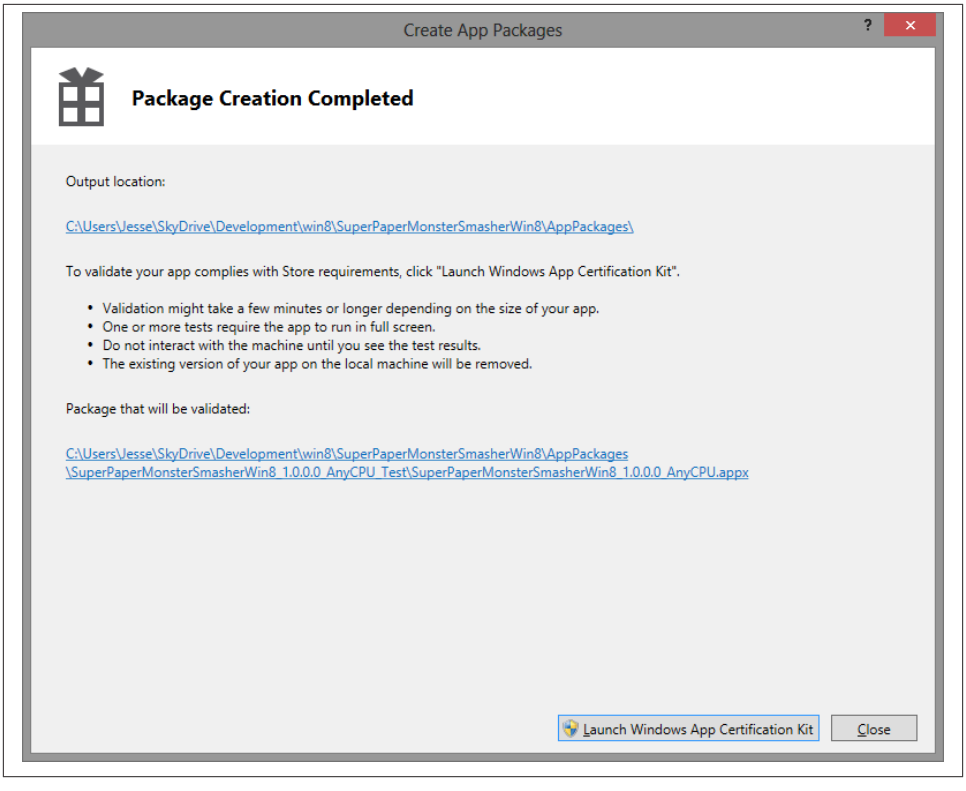


Figure 6-8. Once you have completed everything you will need to select the Launch Windows App Certification Kit option.

Once you launch the certification kit it will begin validating your game. This is done locally, and you will not be able to submit your game until it passes (Figure 6-9).

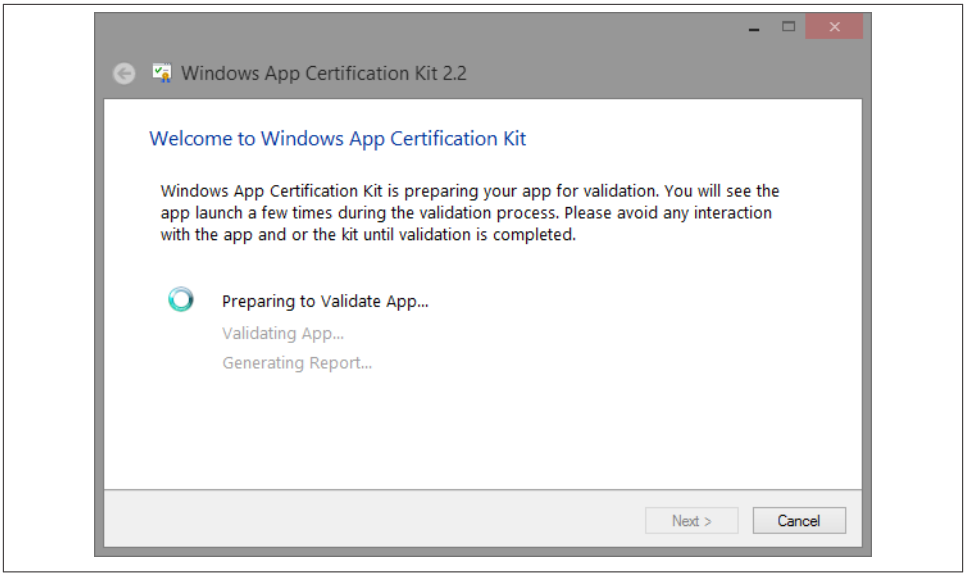


Figure 6-9. Here you can see the app certification kit running.

This should be a smooth process and you shouldn't have any issues. Unfortunately, chances are good that if you built your game outside of Visual Studio and copied it over during your development process your game is going to fail the certification process (Figure 6-10).

If your app fails certification there is nothing to worry about. There are a few common issues that will trigger a failure, and I am going to go over the top reason next. No matter what the game's issue is, there will be a detailed report generated outlining exactly what caused the problem (Figure 6-11). The report is an XML file, which you can easily open and view in Internet Explorer.

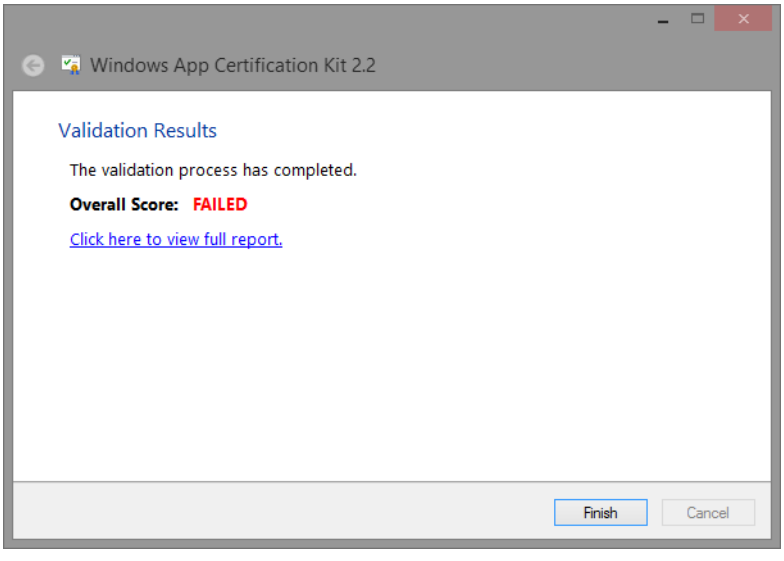


Figure 6-10. If your game fails certification you will be presented with a full report.

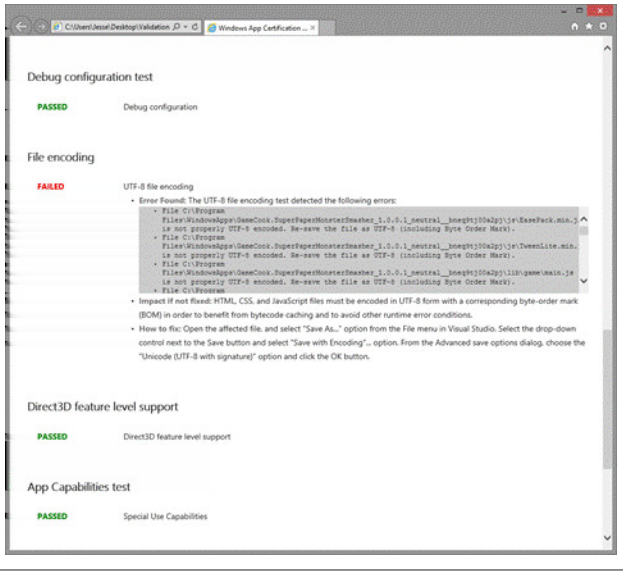


Figure 6-11. You can see issues in the report in any section marked as failed.

You may be wondering what to do now. You may have gotten a similar error:

```
File C:\Program Files\WindowsApps\GameCook.SuperPaperMonsterSmasher_1.0.0.1_neu-
tral__bneq9tj00a2pj\game.min.js is not properly UTF-8 encoded. Re-save the file
as UTF-8 (including Byte Order Mark).
```

While this error may seem strange it's actually quite easy to fix. You have to re-save your problematic JavaScript files in a different encoding. You can quickly do this by opening the file in question and selecting Advanced Save Options from the file menu. You will see the option window in Figure 6-12.

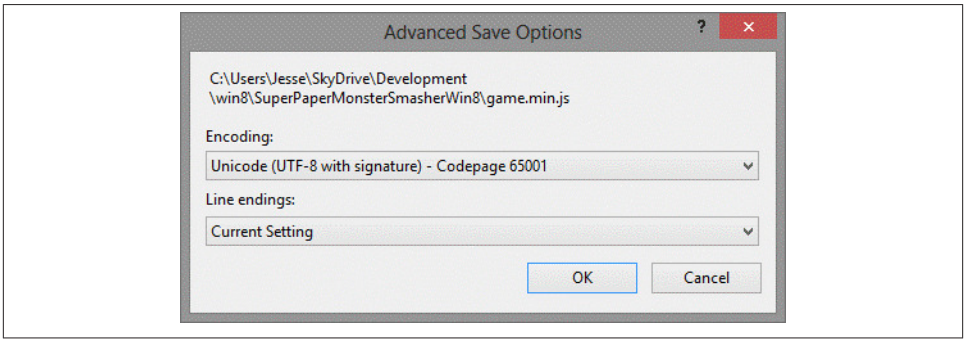
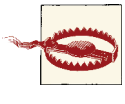


Figure 6-12. This window allows you to save your file in different encodings.

Once you correct the files by saving them with the right encoding you should be all set. This is something I try to do from the very beginning when possible.



It is important to make sure that your JavaScript files are saved as Unicode (UTF-8 with signature) – Codepage 65001. Without this your app will fail certification every time.

Now we are ready to upload our app package to the store once you pass certification (Figure 6-13). You will need to go back to the developer portal and upload the file manually. Select the Packages tab and you will be prompted to choose the final package that was generated by Visual Studio. Remember how I told you to take note of where it was going to be saved? Now is the time to navigate to that location, which is usually in your own project (unless you changed the default output path) and select the package to be uploaded (Figure 6-14).

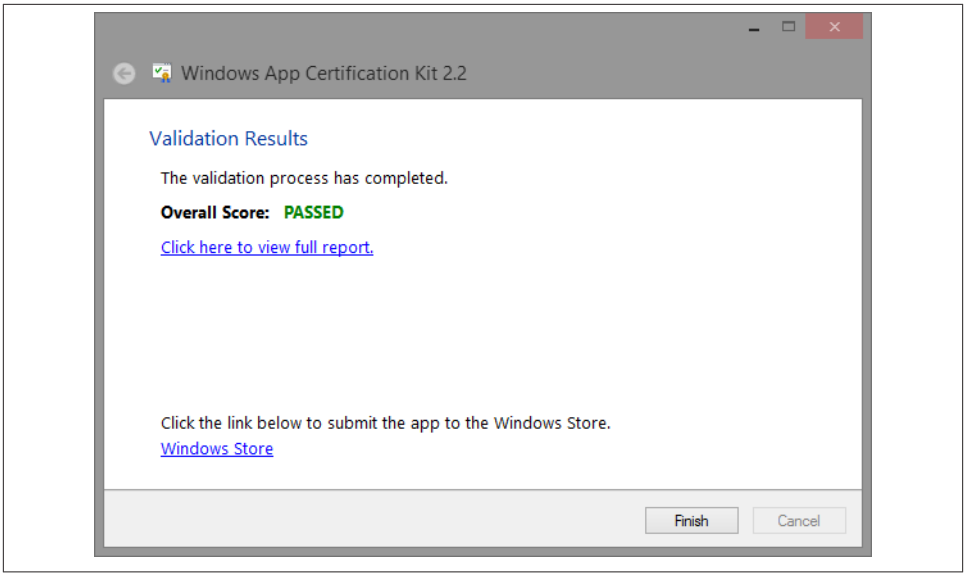


Figure 6-13. The game has passed certification.

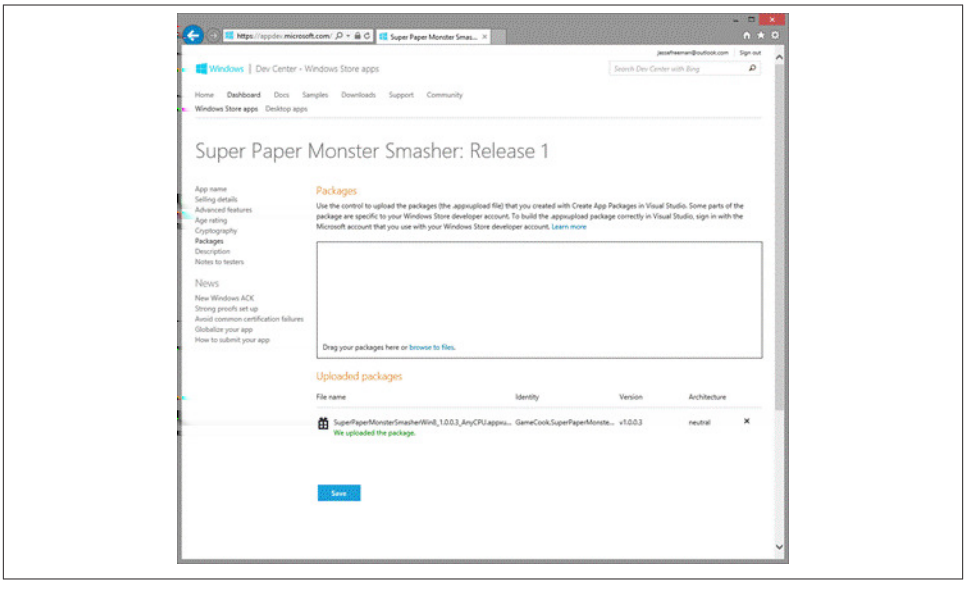


Figure 6-14. Once your package has been uploaded you are ready to move onto the next part of submitting your app.

Creating Collateral for Your Game

Perhaps the most important part of the entire submission process is filling out your app's description. It's mandatory, and it will be the driving force to getting people to buy or try out your game. Before you start this next process, think long and hard about how you are going to market your game.

Here is a breakdown of the key pieces of information you will need to complete this next section:

- Description: There is a 10,000 character limit for your description.
- Keywords: You'll need at least two to four of these to help with searching and discoverability.
- Screenshots: These can be 1366x7×68 or 768×1366 and saved as PNGs. You will also need to provide no more than 200-character captions for each screenshot.
- Promotion Imagery: While not mandatory, this will help you get your app featured if the editors like it. You will need to supply the following size PNGs: 846×468, 558×756, 414×468, and 414×180.

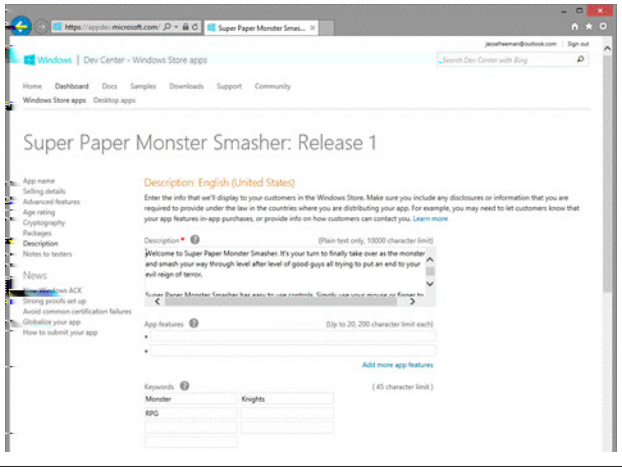


Figure 6-15. Be prepared to fill out a description, app features, and keywords.

I highly suggest planning out your screenshots ahead of time. It's not something you want to rush at the last minute, considering your screenshots will directly impact if the person buys or at least tries out your game (see Figure 6-16).

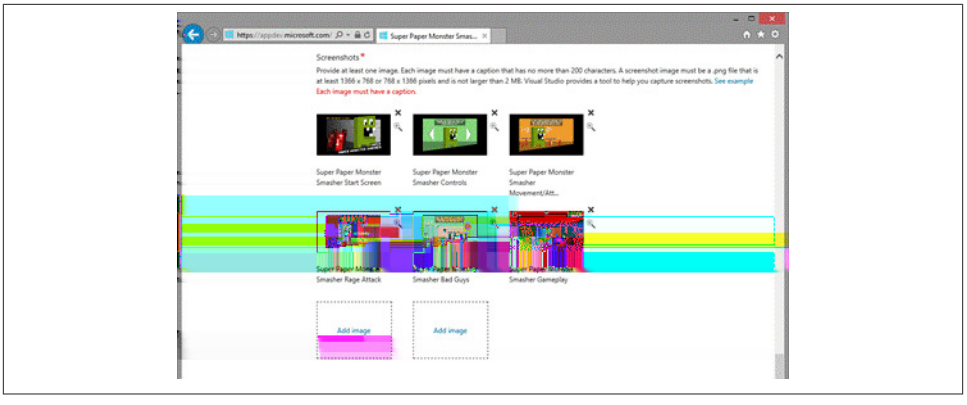
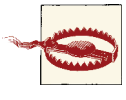


Figure 6-16. Here you can see the screenshots section of the form.

Also, make sure you fill in the website and support contact for your game as well. Not only can you help drive some additional traffic back to your site but buyers may be interested in seeing other games you have made or contact you to suggest features or file bugs.



Don't forget to add a URL to the location of your privacy policy. If your app uses an Internet connection in any way you will have to submit this in order to pass certification. See the Privacy Policy section later in this chapter for more information on the privacy policy.

Now we are ready to finally submit the game. The last thing you need to do is fill out the notes section for the testers (Figure 6-17). On your first attempt to submit you may not have anything to share with the testers, but this space is invaluable if you are having issues re-submitting a game due to failing certification on the testing side of things.

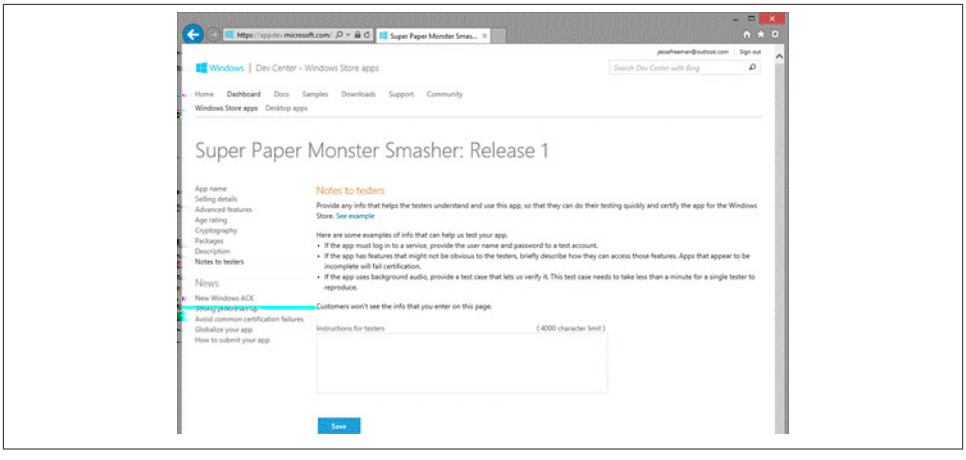


Figure 6-17. Once you are done you can add comments for the testers.

At this point you should have checks on every section of filling out your game and are ready to submit.

During the submission process you can check on the status of your app (Figure 6-18). Each step has an estimated timeframe on when it will complete and be ready for the next step.

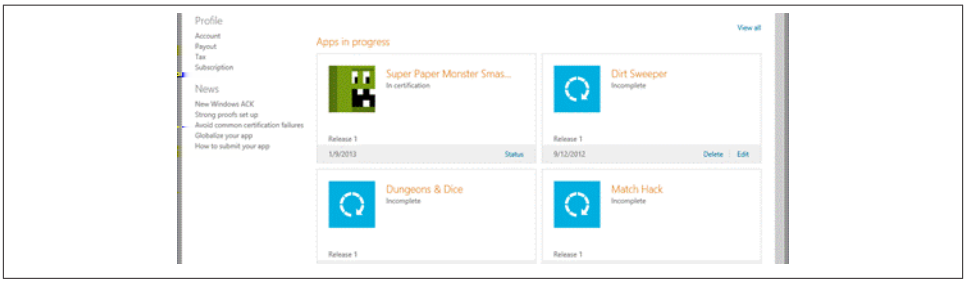


Figure 6-18. You can monitor the progress of your submitted game from the developer dashboard.

At each step of the certification process you will be notified of the estimated time it will take and whether that step has passed or not (Figure 6-19).

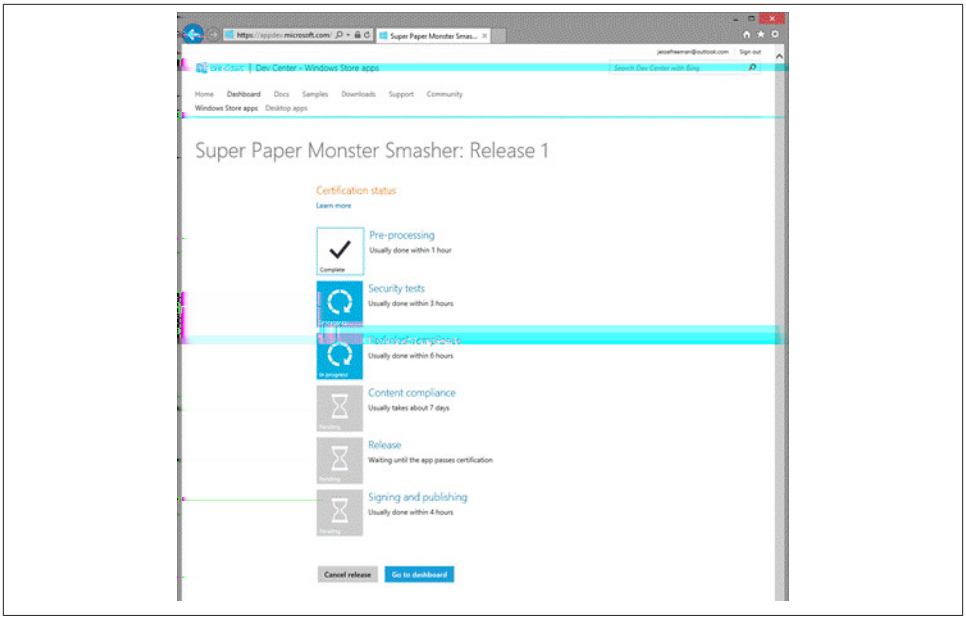


Figure 6-19. Here you can see my game has just been submitted and is going through the certification process.

If for any reason your app fails certification you will be presented with a report detailing the reason why. I have seen apps take anywhere from a few days to be approved to less than 12 hours. Here is a list of common reasons your app may fail:

- Privacy Policy – Make sure you include a privacy policy in your app in the Settings Charm.
- Screenshots – You need to include screenshots that accurately reflect your game. If you modify them in a way that doesn't match up with the real experience of your game it will fail.
- Crashes – If your game crashes during the testing process it will fail and a crash report will be included.
- Feature Incomplete – If you declare your game as a beta or work in progress they will fail your app. Make sure your app is 100 percent complete before submitting.

Outside of this, it should be easy to get your game certified and in the store in a few days.

Submitting Updates and New Releases

Eventually you will want to update or patch your game. New releases are straightforward and will follow similar steps to the submission process. To start with an update you will need to create a new app package.

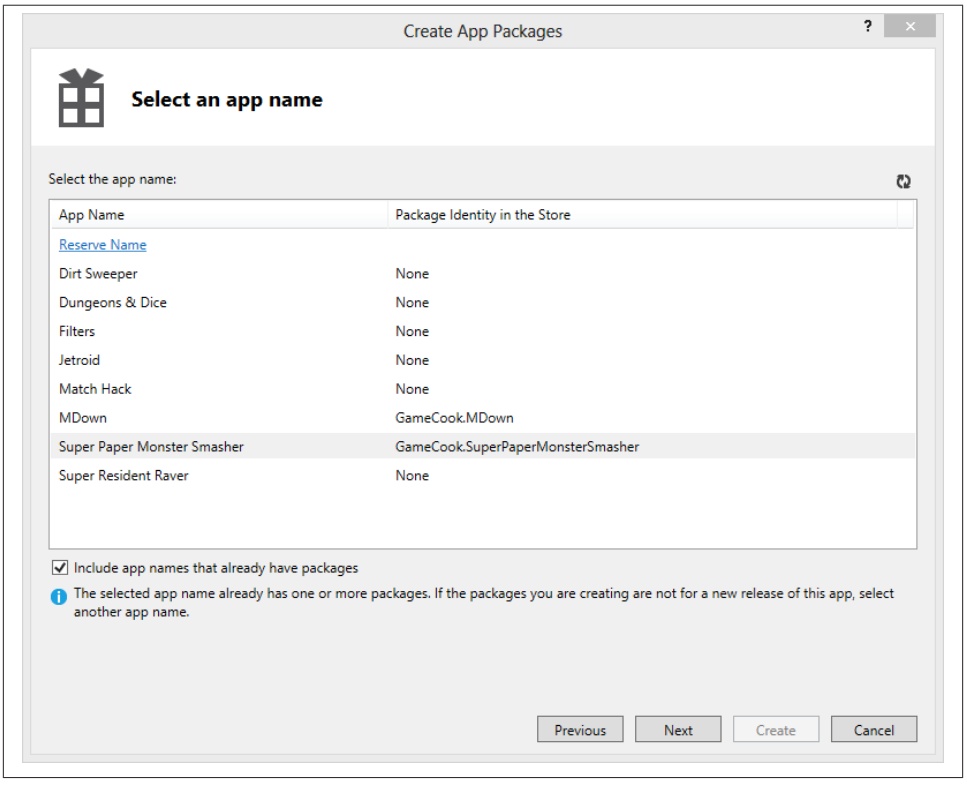


Figure 6-20. Your game will automatically be selected since you already submitted it.

Figure 6-20 shows "Include app names that already have packages" is auto-selected. This will allow you to submit your package to a project that has already been approved in the store.

You will also notice that, during the submission process, your app's version number will continue to increment with each new submission if you had already auto-selected that option (Figure 6-21).

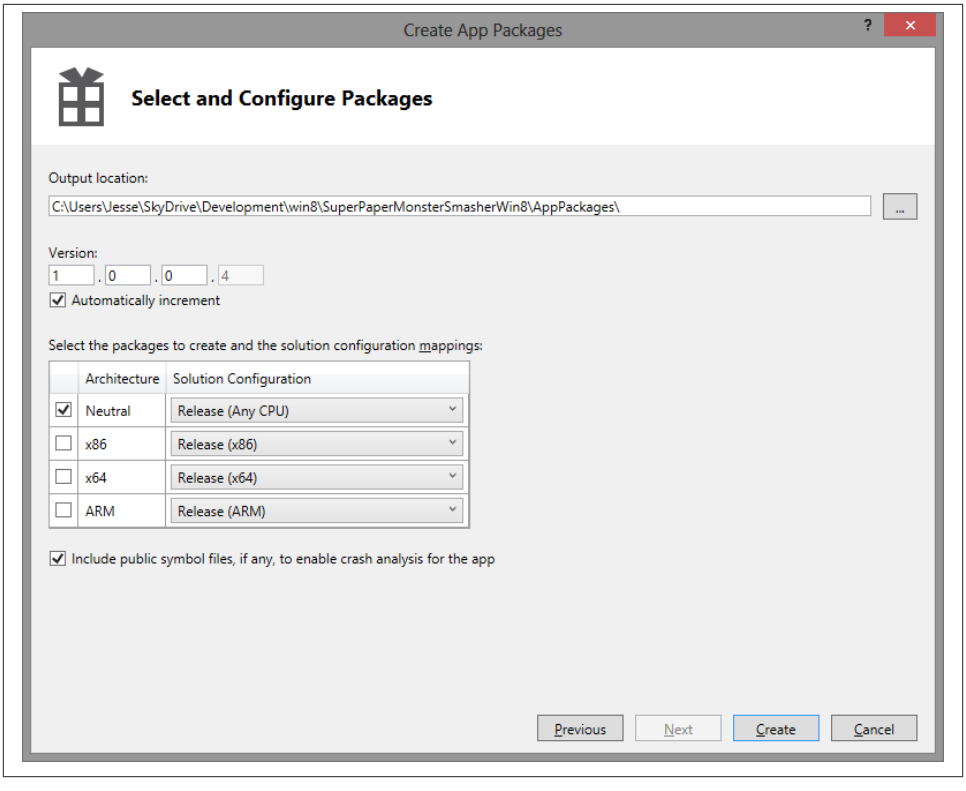


Figure 6-21. Here is the new version of my game's package.

You will also need to take note of where the new build is saved. Just like before, you will need to locally verify your game's package and upload it through the developer dashboard (Figure 6-22).

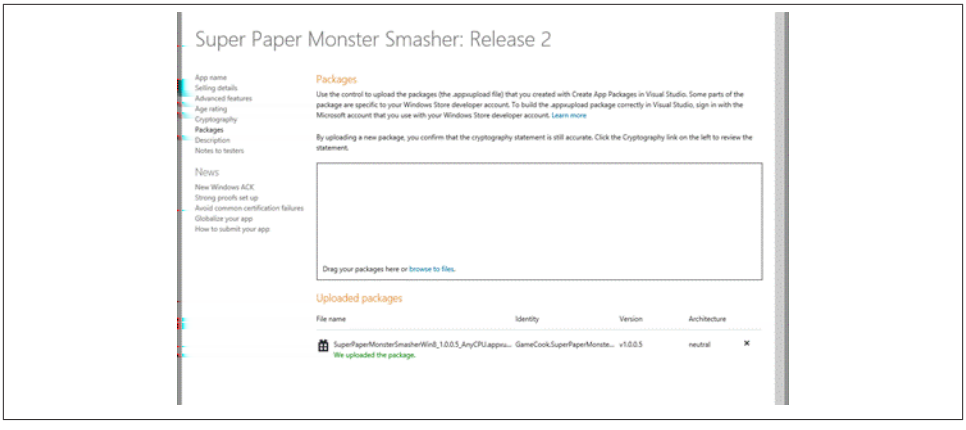


Figure 6-22. Here you can see this is automatically being labeled as my second release.

Next, you have to submit an update description for the store to display (Figure 6-23).

Figure 6-23. Here you can see the mandatory update description field.

While your app is being approved you are able to view its progress and update log in your developer dashboard (Figure 6-24).

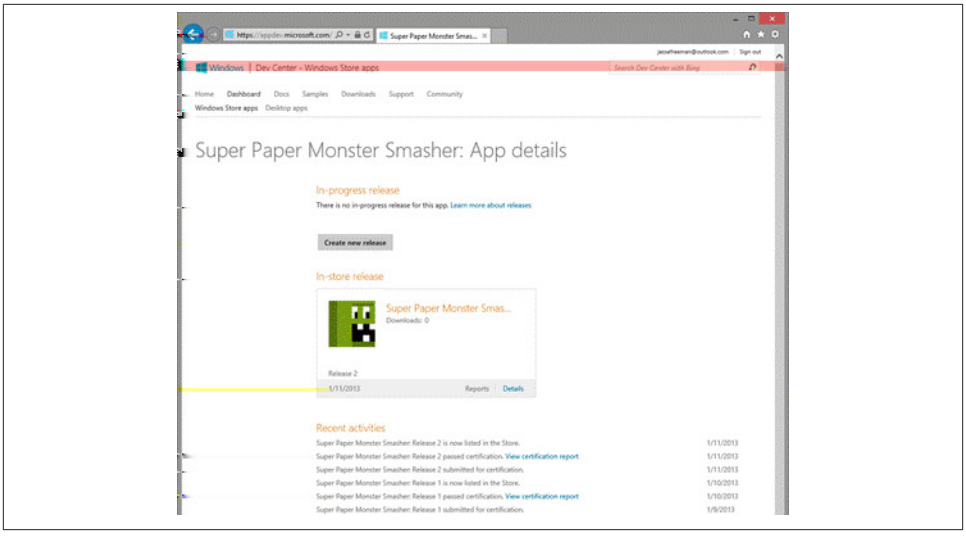


Figure 6-24. Here you can see the game is in the process of being released.

Once the update has been certified, which will take less time than the original submission process, your game will be released to the Windows Store.

Reviewing Your Game's Stats and Ratings

The developer dashboard is not just a place to submit your apps, it's also home to important stats on your game. You can access all of this by selecting Reports under your game's title in the dashboard. Here you will find the following reports:

- Crashes and App Quality
- Download Stats
- User Demographics
- Usage Stats
- Ratings and Reviews

Let's take a look at the most important report, which is quality.

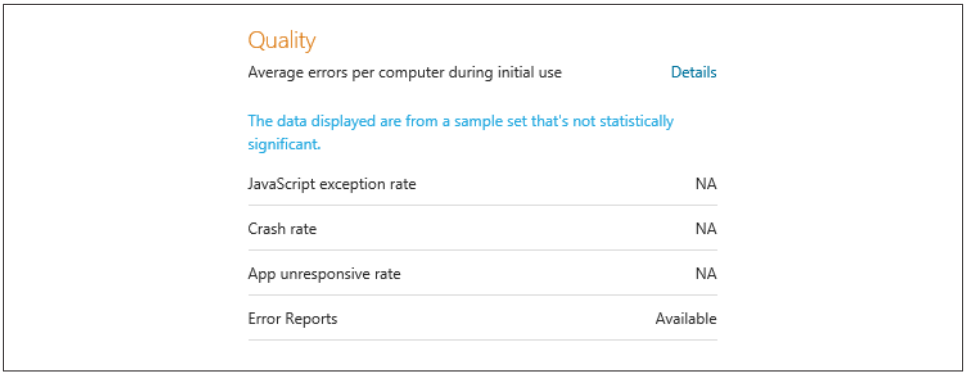


Figure 6-25. Here you can see a summary of your game's crashes.

Hopefully this section (Figure 6-25) is not showing a lot of activity but, if you do see reports of crashing in user reviews, it's important to be able to access the crash data. It might take a large amount of crashes before they start showing up on the summary section of this report. You can drill into it and see if any minor crashes are showing up. In the detail view, you can see all crashes your players may have experienced (Figure 6-26).

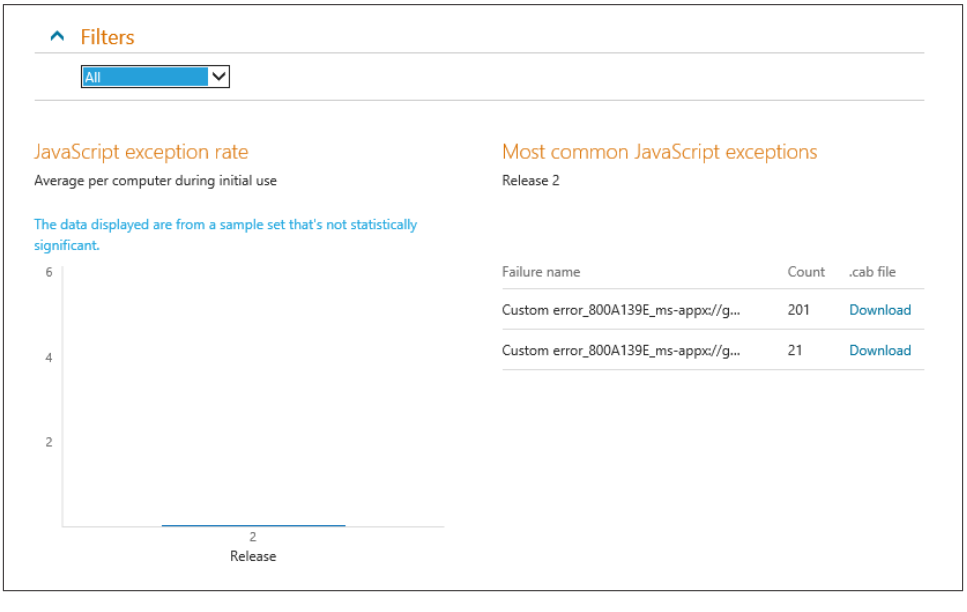


Figure 6-26. Here you can see more granular details on the crashes in your game.

At first I wasn't seeing any crashes in the summary section, but I received a few reviews saying the game did crash. After going into the details I was able to track the issue down

to a sound file that wasn't properly loading. Here is what my own crash report looked like:

```
ErrorDescription=Failed to load resource: media/sounds/hurt2.*
ErrorType=7
ErrorTypeText=Custom error
ErrorNumber=800A139E
SourceFile=ms-appx://gamecook.superpapermonstersmasher/game.min.js
Line=248
Character=20
DocumentURL=/default.html
OSProduct=Windows 8 Pro
OSVersion=6.2.9200.0
OSServicePack=0
UserAgentString=Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; Trident/
6.0; MSAppHost/1.0)
StackTraceAvailability=2
StackTrace=ms-appx://gamecook.superpapermonstersmasher/game.min.js:
248:20           _loadCallback(string, boolean, object)
                        ms-appx://gamecook.superpapermonstersmasher/game.min.js:
178:266           Anonymous function(object)
StackTraceHash=93186aa1da5762c6cceb8cfff38f9d13
PackageFullName=GameCook.SuperPaperMonsterSmasher_1.0.0.5_neutral__bneq9tj00a2pj
AppUserModelID=GameCook.SuperPaperMonsterSmasher_bneq9tj00a2pj!App
IsTerminal=True
DependentPackageList=Microsoft.WinJS.1.0_1.0.9200.20512_neutral__8wekyb3d8bbwe
```

As you can see, these kinds of reports can be very helpful for figuring out the source of a problem in your game.

The other reports are not as grim and are a great insight into the userbase of your game. Here (Figure 6-27) you can see the download summary for my game compared to other apps in similar categories:

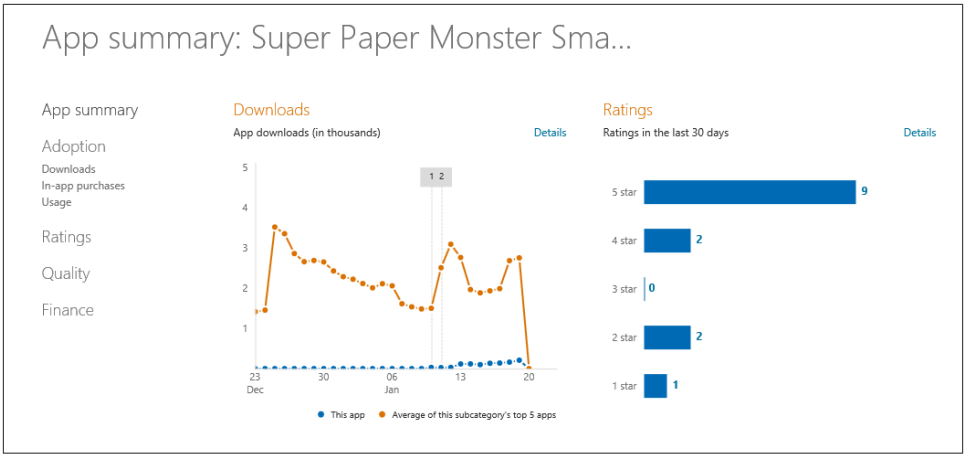


Figure 6-27. Your downloads compaired to other games in the same category.

I can also dig into this user data and look at some interesting user stats, such as age, location, and gender of my players (Figure 6-28).

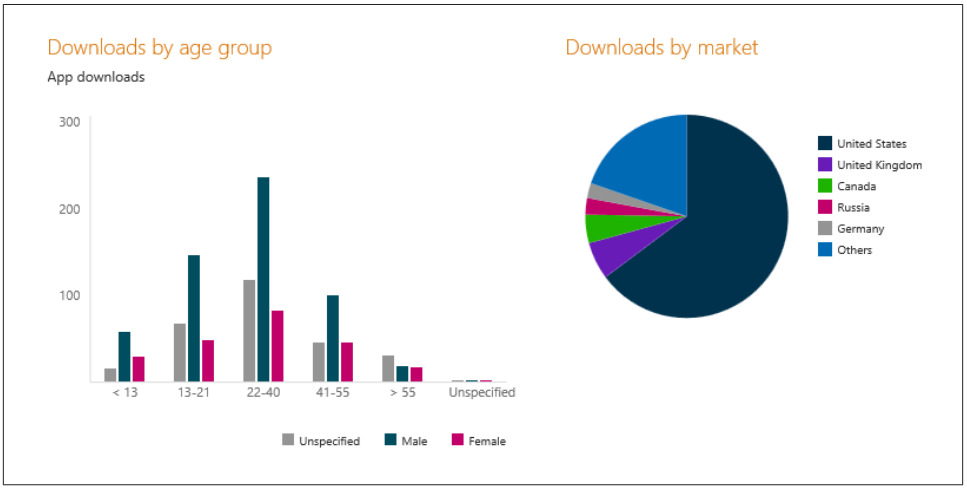


Figure 6-28. Here is a visualiztion of the age and location breakdown of my userbase.

Since this information is optional to peole who register in Windows 8, you will see a group of users marked as unspecified (Figure 6-29). You can also dig into how long people are playing your game. This is also compared to the average of other apps in the same category.

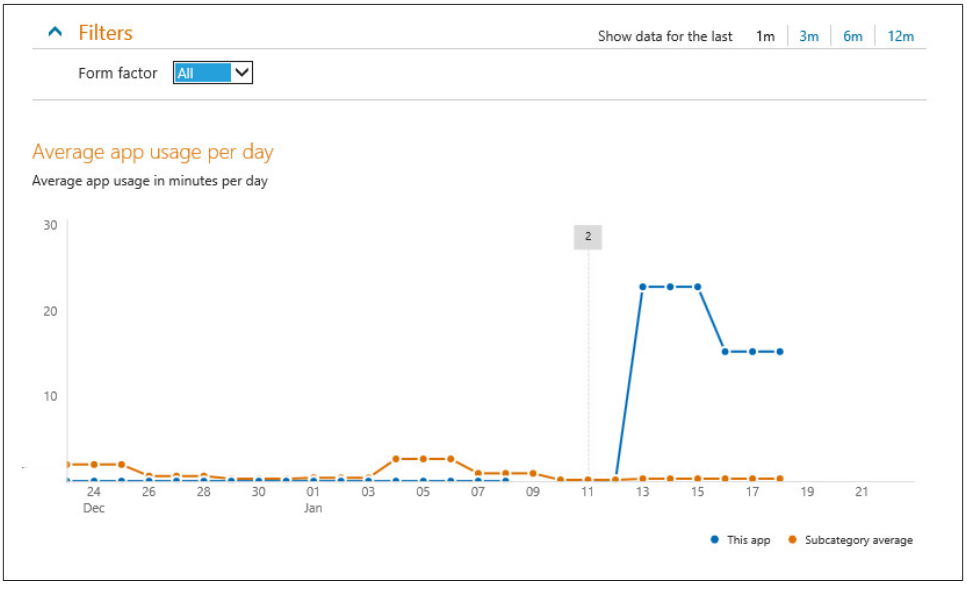


Figure 6-29. Here you can see breakdowns for 1-, 3-, 6-, and 12-minute usage.

You can also get reports on each type of form factor, which may be critical if you are recieveing poor reviews from users on one type of device and want to see how large that userbase is. It also helps to find out where your game is more popular, so you can spend more time on that demographic.

Finally, you can go through the ratings and reviews of your app across each reagon you have released your game in (Figure 6-30).

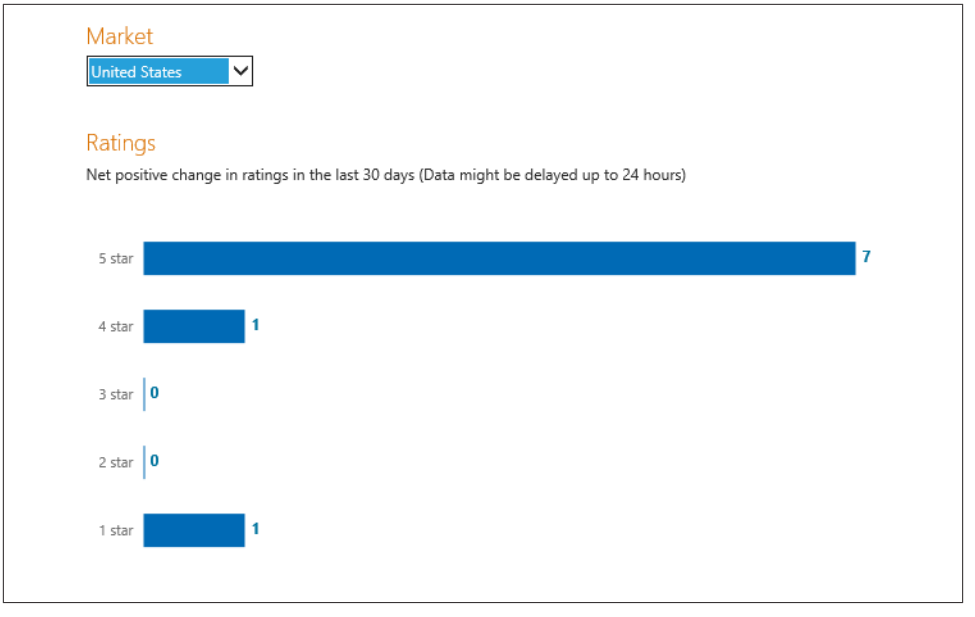


Figure 6-30. Here are reviews of my game for the United States.

All in all, you will find a great deal of information in these reports that may allow you to skip adding your own tracking in your game. Most of this kind of data wouldn't be tracked by third-party stats, so use this to your advantage.

Tips and Tricks for Publishing a Game to the Windows Store

Privacy Policy

The privacy policy is probably the most important thing you will need to do when submitting an app. If you fail to supply one you will most likely fail certification for the following reason:

"The app has declared access to network capabilities and no privacy statement was provided in the Description page. The app has declared access to network capabilities and no privacy statement was provided in the Windows Settings Charm."

The sad part is that this is so easy to fix. First, you will need to create a new HTML file. I called mine "privacy.html," and it lives in my pages/settings directory.

From here you will need to add some HTML to your new file. I got this text from Keith Peters, who in turn also failed to pass certification due to a missing privacy policy. He found this online and, if your app is doing serious things, you could consult someone

such as a lawyer on what your privacy policy should say. My app doesn't do anything with its Internet connection so I am comfortable with the following text, but I assume no responsibility if you end up sticking it in your app as is. Here is the template:

```
< !doctype HTML>
<html>
<head>
    <title>App settings flyout</title>
    <script type="text/javascript" src="/js/settings.js"></script>
</head>
<body>
    <!-- BEGINSETTINGSFLYOUT -->
    <div data-win-control="WinJS.UI.SettingsFlyout"
       " id="privacyPolicy"
        data-win-options="{settingsCommandId:'privacyPolicy', width:'narrow'}">

        <div class="win-ui-dark win-header">
                <button type="button" onclick="WinJS.UI.SettingsFlyout.show()"
class="win-backbutton"></button>
            <div class="win-label">Privacy Policy</div>
        </div>
        <div class="win-content">
             <p>This application does not share personal information with third
parties nor does it store any information about you.</p>
             <p>This privacy policy is subject to change without notice and was
last updated on [INSERT DATE]. If you have any questions feel free to contact
me directly at <a href="mailto:[INSERT EMAIL ADDRESS]">[INSERT EMAIL ADDRESS]</
a>.</p>
        </div>
    </div>
    <!-- ENDSETTINGSFLYOUT -->
</body>
</html>
```

Now you just need a quick way to add this to your app. Add the following code in your default.js file inside of the activated event listener:

```
WinJS.Application.onsettings = function (e) {
     e.detail.applicationcommands = { "privacyPolicy": { title: "Privacy Poli-
cy", href: "/pages/settings/privacy.html" } };
    WinJS.UI.SettingsFlyout.populateSettings(e);
};
```

Now when you run the app, you will see a new privacy policy option in the Settings Charm.

And when you click on it you will see the privacy policy text we added to our privacy.html.

As you can see these two steps, adding a privacy HTML file and linking it up to the settings flyout, was all we needed to do in order to satisfy this critical step for getting

an app approved. The last thing you need to make sure you do is supply a URL to an HTML page (hosted on your site) with the same privacy policy in your app's description.

Setup Capabilities

If your game does not use the Internet, you should uncheck the Internet (Client) setting in the .appxmanifest under the Capabilities tab. Most games don't require an Internet connection, and this will save you from having to include a privacy policy. Here (Figure 6-31) is what the panel looks like:

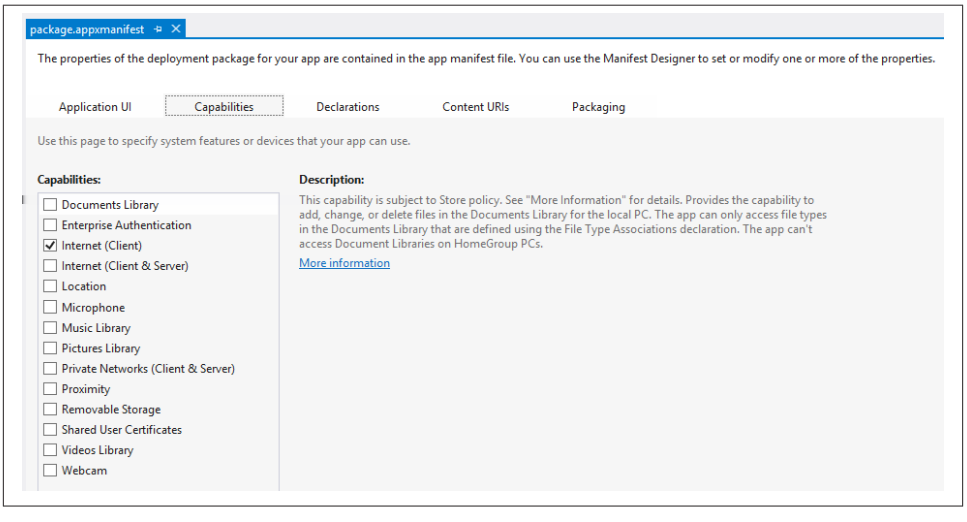


Figure 6-31. This screen alows you to configure your game's capabilties.

If your game does use the Internet or needs access to the local storage or anything else in the list you will want to go over and set that up now before you move onto the chapter on publishing. Each capability has its own requirements and settings, so read through the description to make sure you meet the requirements.

CHAPTER 7
Monetization

The Windows Store monetization is on par with all other major App Store publishing models, which will be very appealing to developers looking to get the most out of their time investment. To better understand some of the ways you can monetize your game on Windows 8, I have decided to highlight 3 of the key ways outside of the standard flat rate for your game:

- **Trial Mode:** On Windows 8, you can add trial mode into your game, which not only allows players to try out the game but also keeps them from having to install a separate app with the un-purchased features. Windows 8 allows you to lock out specific parts of your app based on if it is in trial mode or has been fully paid for, so you can maintain one app with a single codebase.
- **Ad Support:** Finally, an approach that has been falling out of favor over the years is to monetize your game with ads. Since HTML5 games on Windows 8 can run any JavaScript ad SDK, you will find it very easy to quickly set up ads in your own game. We will talk about how to use the Microsoft Advertising SDK for Windows 8 later in the chapter.
- **In-App Purchase (IAP):** This has become one of the most popular ways to monetize your apps on iOS and Android. The good news is that Window 8 apps support IAP as well. Generally, you can offer your game for free and let players buy new levels or open up the entire game and use in-game currency then allow players to buy that currency with real money.

As you can see, you have lots of options to help monetize your hard work. Since Windows 8 is a new platform with a new market, you will have a lot of room to experiment with what works best. To help you monetize your own game, I will talk about each option with some sample code to get it up and running quickly.

Setting a Fixed Price

This is perhaps the most straightforward approach to selling your app. Simply set a price in the store and you are good to go. While setting a fixed price has its advantages in other markets, the Windows Store offers even more creative options around monetizing your game. We'll talk about trial mode, ads, and in-app purchases next, but if you simply want to get your game out there and not deal with adding additional logic to your game, then this is going to be your best option.

For the sake of completeness, here is a quick overview on how to set a price for your game in the developer dashboard (Figure 7-1). Once you have a game set up and are filling out its information, you will want to go into the "Selling details."

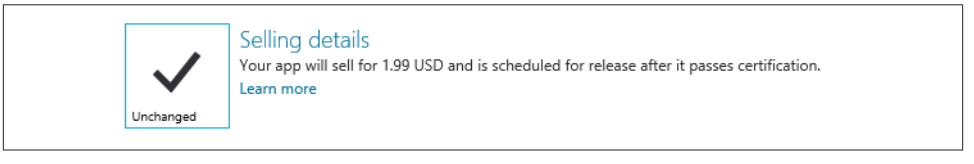


Figure 7-1. You can quickly see the sale price of your game in the developer dashboard.

From here you will be presented with a place to set your price, whether or not there is a trial, and the markets you want to sell your game in.

Once you have selected a price you are comfortable with in Figure 7-2, you should be ready to sell your game in the market. You will also see the price of your game in each market in its native currency.

One thing to keep in mind is that some markets may require you to fill out separate tax information and may even have different limitations on the rating of your game. To avoid failing app certification due to specific country requirements, you may want to start with only a few markets at first. I tend to focus on larger markets, such as the U.S., Canada, the United Kingdom, France, and Germany. From there, I slowly include other locations once the game begins to do well in other markets and I know I have worked out any of the bugs that tend to pop up during a first release.

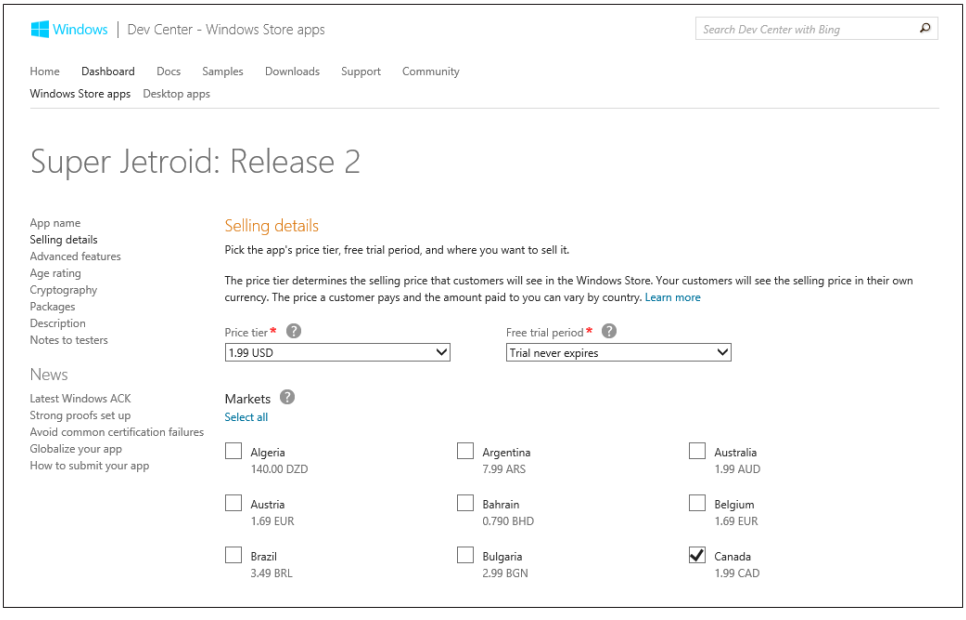


Figure 7-2. When setting up your price you can select if there is a trial and what countries to sell your game in.

As I alluded to earlier, the Windows Store offers a few more options for monetizing your game. Let's talk about one of the more unique features of the Windows Store, which is trial mode.

Trial Mode

If you are familiar with other app stores you may have noticed a huge selection of "lite" and "full" versions of games. This is a byproduct of no clear way to let users try out a game before they buy it. The Windows Store alleviates this issue by offering a simple yet robust trial mode for you to allow people to test out your game before they decide to buy it. One of the best parts of this feature is that you can still maintain the same codebase by simply locking out parts of your game if it is running in trial mode. Let's take a look at how to set this up in your own game.

The first thing you are going to need is a reference to the current app, which contains licensing information among other things. A quick way to do this is by creating a global function that looks like this:

```
initLicense = function() {

    // use for live app
    currentApp = Windows.ApplicationModel.Store.CurrentApp;
```

```
        // use for testing
        //currentApp = Windows.ApplicationModel.Store.CurrentAppSimulator;

        return currentApp.licenseInformation;
    }
```

As you can see, there are two options in this function. The first is what you want to use in the live app you publish to the Windows Store. The second option is used for testing and allows you to get a dummy file defining your game's license information from the CurrentAppSimulator. You can initialize this by simply calling the function and setting it to a variable like this:

```
licenseInfo = initLicense();
```

From here we can check a property called isTrial on the licenseInfo variable and create some conditional logic in our game to limit functionality until the player buys the full game.

Here is a good example of how you would want to use something like this. In one of my games I have ten levels. In trial mode, however, I only want two of them to be playable. I keep a variable called totalLevels, which allows me to loop through and display which levels are unlocked. Since totalLevels is set to 10, because that is the total number of levels I have in the game, the player will never see an error if they beat level 10. What is also good about this setup is that I can also change the totalLevels at any point to limit what levels the player sees. So, in the case of the trial mode of my game, I change the totalLevels variable and display a message that they need to buy the full version of the game (Figure 7-3).

Figure 7-3. In Super Jetroid I encourage the player to buy the full game to unlock all of the levels in trial mode.

Here is what the level limit code looks like:

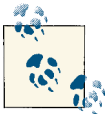
```
if (typeof licenseInfo != "undefined") {
    if (licenseInfo.isTrial) {
        this.totalLevels = 2;
        this.showPurchaseText = true;
    }
}
```

In addition to limiting the number of levels the player sees, I also display a message telling them to purchase the full game. Again, this is very easy to set up. I simply refer back to my license details to see if we are in trial mode. If so, I display the buy message.

One thing that you may want to also include is a link back to the app in the store. So, when people click on the buy now message, they are taken right to the buy page in the Windows Store. Here is a helpful method for handling this:

```
sendToStore = function () {
    var uri = new Windows.Foundation.Uri(currentApp.linkUri.rawUri);
    Windows.System.Launcher.launchUriAsync(uri);
}
```

What is great about this code is that the link to the store will automatically resolve itself when the app is published. This is just part of the application's data that is built in during the publishing process.



It's important to note that this link will not work when you are locally testing. As of now, there is no way to test this during development since it relies on the app actually being published in the store.

As you can see, with only a few lines of code you can simply lock out parts of your game until it's purchased. Likewise, it's just as easy to set up trial mode in the developer dashboard. Simply go to the "Selling details" section of your game and, next to where you set your game's price, you will see a drop down to enable trial mode (Figure 7-4).

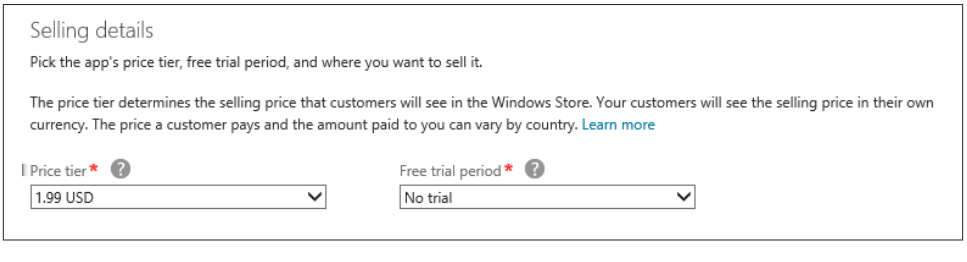


Figure 7-4. The option to set up a trial mode is right next to where you set the price in the developer dashboard.

As you can see in Figure 7-5, you have several time-based options if you want your app to lock the user out after a specific period of time, including an option for the trial to never expire.

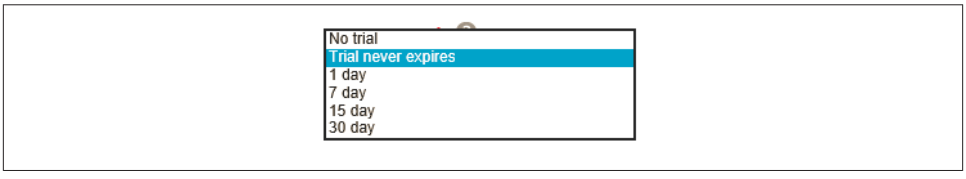


Figure 7-5. Pick the trial duration that makes the most sense for your game.

If you do not want to have a time trial you can simply choose trial never expires. With the previous code in place your game will automatically start working within the constraints you defined for the trial mode.

Incorporating Ads into Your Game

Putting ads into your game is very easy as well. You can use any HTML/JS-based ad SDK or use Microsoft's pubCenter. This means that if you already have ads integrated into your game you can leave them in and continue to make money, assuming the SDK's license allows for that. If you don't have ads in your game, I'll quickly cover how to set up pubCenter. The first thing you are going to need to do is register for an account here. Once you have registered, go into your account and navigate to enable ads for Windows 8.

Once you have your account set up, you will need to register your app and create an ad for a Windows 8 app (Figure 7-6). You can also pick out your ad size and configure other information for your app.

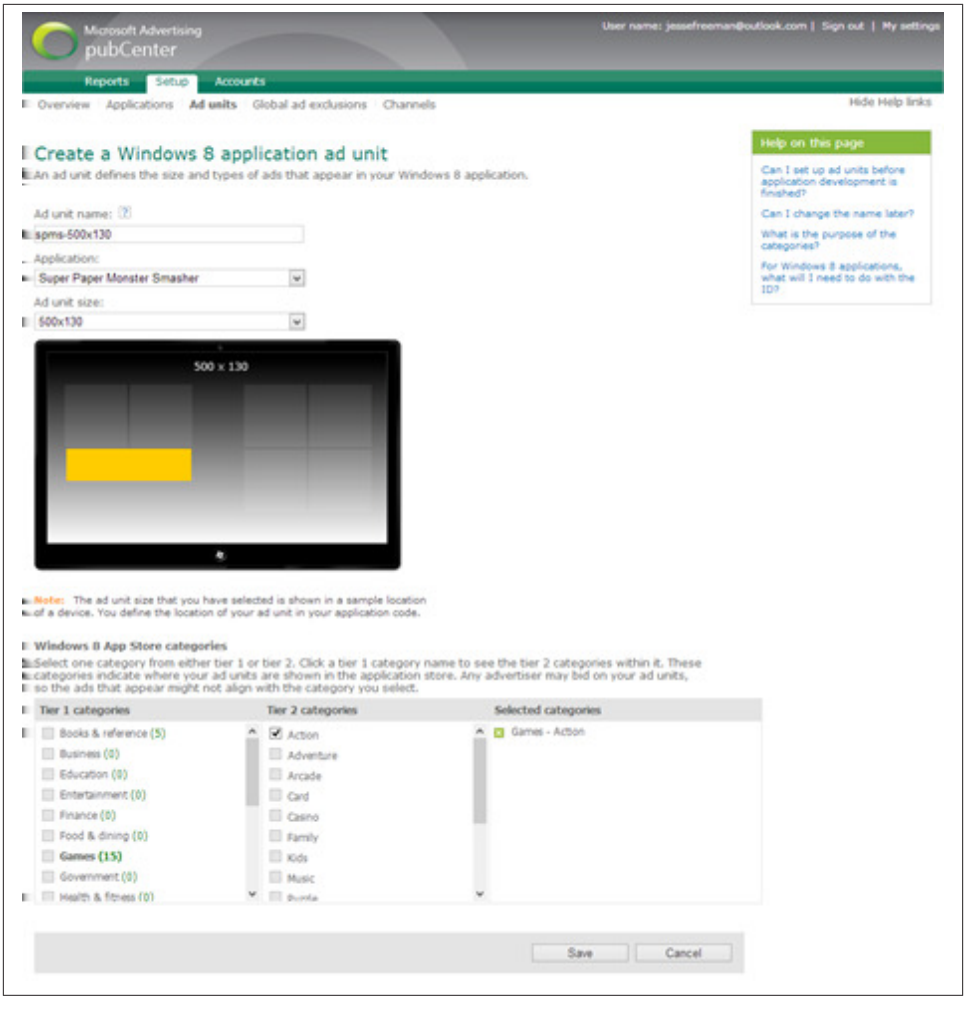


Figure 7-6. Here I am setting up the size of my ad and other properties for it to run inside of a Windows Store app.

Now that you have everything set up, you will get an Application ID and Ad Unit ID (Figure 7-7).

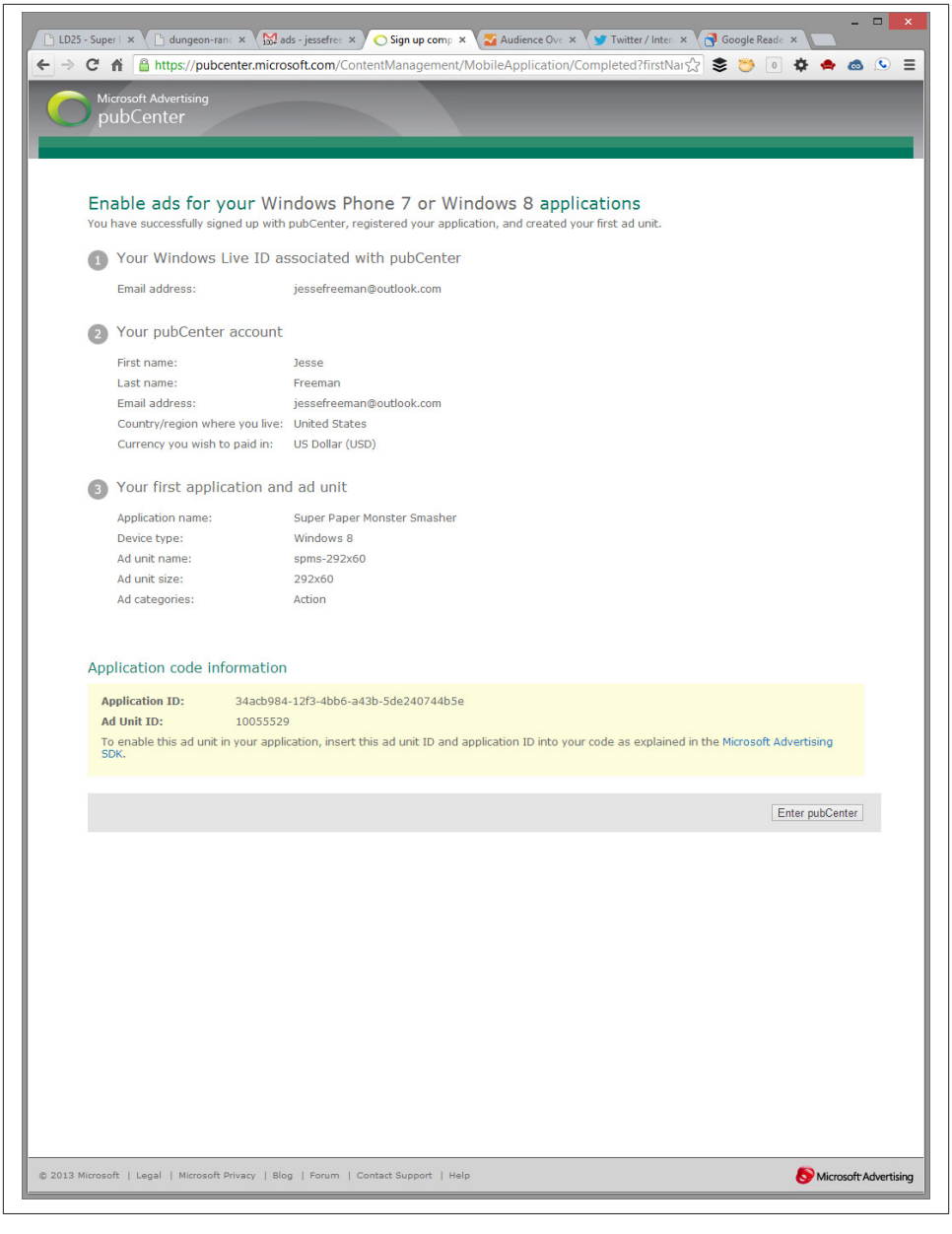


Figure 7-7. Make sure to keep track of your Application ID and Ad Unit ID.

Make sure you save the Application ID since you will need it later on when we put the ad code into our game. Once all of that is done, you can download the ad SDK and

install it into Visual Studio, which you can get here. You can simply install the SDK into Visual Studio by clicking on the downloaded file (Figure 7-8).

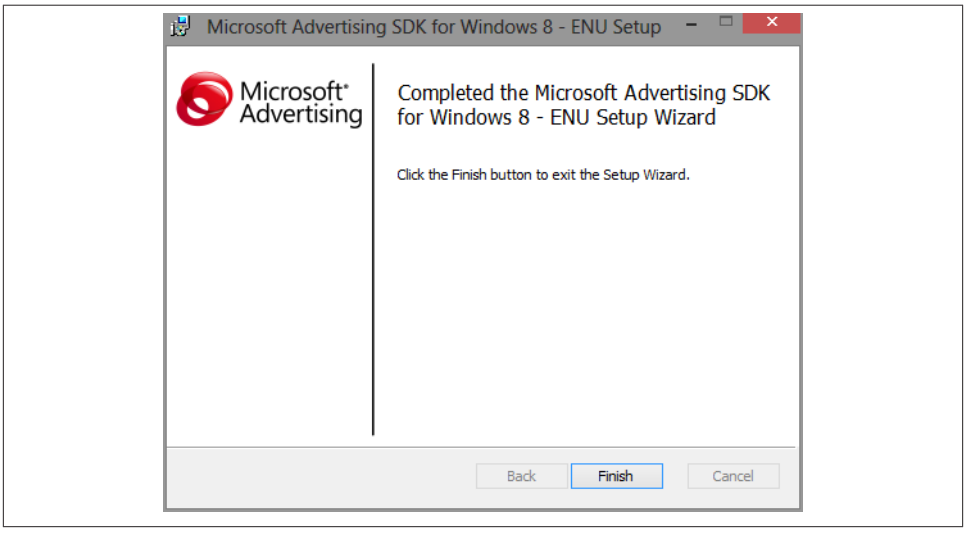


Figure 7-8. The Microsoft Advertising SDK installer.

Once the SDK is done installing, you will need to include the ad library in your project. Click on the resources folder in the project explorer and right click on it. You can then select "Include new resource." It should look something like Figure 7-9.

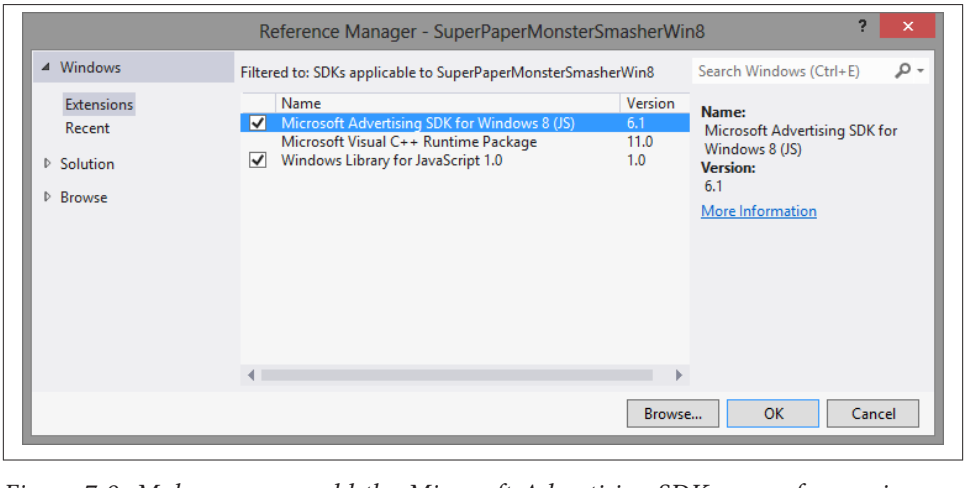


Figure 7-9. Make sure you add the Microsoft Advertising SDK as a reference in your game's project.

That has the link to the SDK download and how to install it into VS and add it to your project.

Now you should have everything linked up and ready to show ads. Add the following HTML to your default.html file:

```
<div style="position: absolute; top: 50px; left: 0px; width: 100%; z-index: 1;">
     <div id="ad"  style="margin-left: auto; margin-right: auto; width: 500px;
height: 130px;" data-win-control="MicrosoftNSJS.Advertising.AdControl" data-win-
options="{applicationId:   '52df6cc7-1ea5-477d-b607-bf7922f81259',   adUnitId:
'10047026'}">
    </div>
</div>
```

This will float the ad above everything else on the pages, assuming you don't have anything higher than a z-index of 1. You can change the inline styles to suit your needs.

This code is only for locally testing. You will need to replace the applicationID with your ID supplied by pubCenter before you publish your game or you won't be able to display your ads. Also, you can't test your ads locally, so use the test ads to make sure they are placed correctly then use your production applicationID so you don't forget to swap it out before publishing.

If you ever need a reference to the actual ad, you can use some JavaScript to get its div ID from the document like this:

```
this.ad = document.getElementById("ad");
```

Once you have a reference of the ad in your code you can use simple CSS to hide and show it by setting this.ad.style.display to "none" or "block." This is useful when you want to hide ads on certain parts of the game or if you tie it into the trial mode and hide it if the player has bought a full version of the game like so:

```
// Remove Ad if the app was paid for
if (!licenseInfo.isTrial) {
    var elm = document.getElementById("ad");
    elm.parentNode.removeChild(elm);
}
```

One of the best ways to capitalize on the trial mode of your game is enabling or disabling ads based on if the player purchases the full version. You can even use both techniques of limiting features and showing ads to entice the player to buy the game. Be creative with how you use advertising in your game. If you make it too annoying people will not like your game; if you don't get users to click on the ads you will not make any money. This is why I suggest coupling ads with trials and helping augment the users who simply play the trial without ever upgrading.

In-App Purchase

In-app purchases (IAP) are another great way to help monetize your game. Crafting a viable and meaningful IAP system into your game takes a lot of thought and planning on your end. The technical side of things is relatively easy, and I provide links to reference material in the resources section of the book. I do want to briefly talk about some strategies for adding IAP into your game.

The first thing you should understand about IAP is that it allows you to make your game free and still find ways to make money by selling additional features. This is an incredibly powerful monetization strategy since it allows your game to show up in the free game listing, which doesn't happen if your game is for sale with a trial mode. That means it is even more likely for people to download and play it. From there it is your job to incentivize them into paying more for missing features.

The reality is that IAP needs to be added to a game from the initial design process so, if you are porting over a game from the Web, chances are you may not have an existing IAP system in place. While you can easily add a basic way to buy more of an in-game currency, I suggest considering the following options for easily adding value to a game that was not originally designed for IAP:

- Additional Levels – Selling additional content or add-on packs is a great way to take advantage of IAP and not make players feel cheated as they play the initial round of levels you supply with the game.
- Additional Characters – Over time you may decide to add new playable characters to your game. These could be to coincide with an event, such as a holiday, or just to add some additional replay value to your existing levels.
- Unlock Features of the Game – You can easily use IAP as a full replacement for trial mode. Simply lock out features of the game, such as multi-player, and allow the player to buy the feature if they want it.
- Remove Ads – Why not use IAP to remove the ads from your game. We discussed the technique earlier in the chapter with trial mode, but taking advantage of IAP would probably be a better solution.

As you can see, there are lots of ways to add on IAP to an existing game. Just be careful to not overdo IAP or that will really turn players off of spending money on your game.

Tips and Tricks for Monetization

I wish I had a surefire way of making money with your games. Every app store is different, and while Windows 8 is still relatively new, don't expect to just put your game in the market and instantly get rich. Here are a few things you should do to help get your game some more exposure and move it up in the ranks.

Promote Your Game

Just like on other platforms, it is critical that you promote your game. There are many ways to go about doing this, from blog posts to getting gaming sites to do reviews. Eventually, if your game gets enough attention, you may be featured on the Windows Store under the Games section (Figure 7-10).

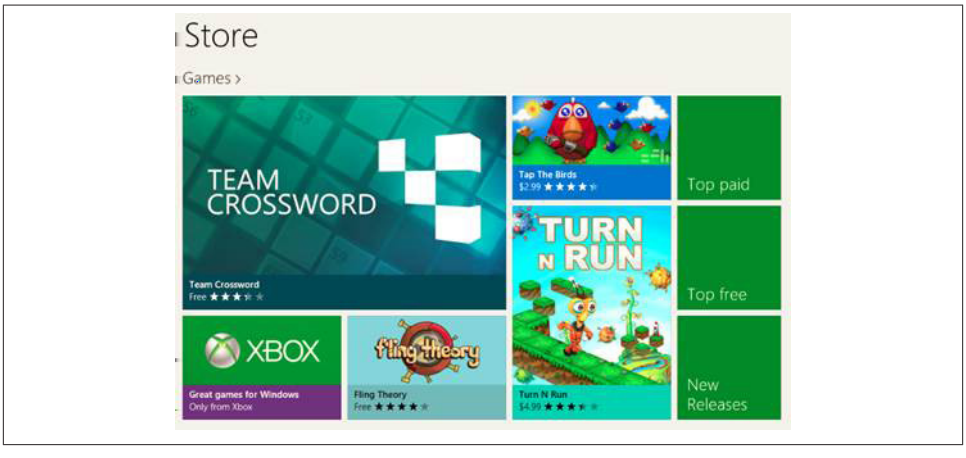


Figure 7-10. The front page of the Windows Store has a dedicated Games section with features, top paid, top free, and new releases.

Being featured will help drive sales of your game up dramatically. Windows also categorizes games as free and paid. Trials fall into the latter, so remember that your game will be competing with the paid category, which usually contains higher profile games, such as Xbox Live titles, which could make it hard to gain ranking. If you only have ads in your game that you disable once a trial has been paid for, you will have a harder time climbing to the top of the charts, so be careful.

Finally, the best thing you can do to help promote your game is to simply have compelling screenshots, icons, and a description. Be clear about what your game does and why people would want to play it. Your app icon and screenshots are the first thing people see, so make them count. And, expect people to only make it through the first two or three screenshots, so put the most compelling ones first.

Incentivize Players to Upgrade from a Trial

If your game has a trial mode, remind people about upgrading. I wouldn't be too pushy about it, but a common technique is to inform people they have earned something like unlocking an achievement or calling out the fact that they can remove the ads in between levels while loading. You can do small messages or full-screen ones, but make sure they open a link back to your game in the Windows Store.

Get People to Rate Your Game

A lot of games and apps ask the player to rate them. You can easily do this by taking advantage of the link back to the game in the Windows Store. Simply pop up a message asking the player to kindly rate the game. Don't be too pushy about this and make sure you always offer a way to opt out, such as a button that says "Don't ask me again." This is especially critical if you have a paid app. No one wants to pay for an app then be pestered to do something else outside of enjoy the game.

Make Compelling IAP Options

While we only briefly touched on the possibilities of how to implement IAP, I highly suggest spending your time designing rewarding IAP for players to buy. While in-game currency is always a safe bet, I would watch out for items that add superficial value to the gameplay. Things like costumes, skins, and other visual customizations tend to only go so far. Likewise, be responsible about what you try to make players buy. If you skew the entire game to force them to use IAP, the player may not be too happy and stop playing/paying, or worse, start giving you a bad review.

Get Reviews

Have friends and family review your game. The more ratings and reviews you have the more likely someone on the fence will be to download or buy your game. When you launch you only have a small window while your game is featured in the new releases category, so try to make the most of it by getting five-star ratings in early to help bump up your overall rating.

CHAPTER 8

Back to the Web

In this chapter we will discuss techniques for making sure your HTML5 game will still run on the Web as well as Windows 8. After spending all of this time getting your game to run on Windows 8, it's important to make sure you can also get the most out of your hard work and take advantage of the ability to run HTML5 games on multiple platforms.

A Web-First Workflow

Throughout this book we focused solely on getting an exciting HTML5 game to run on Windows 8 inside of Visual Studio. While this is a great approach if you want to focus solely on Windows 8, you may want to have the same codebase run on the Web, Windows 8, and other platforms that support HTML5 games. For this, I tend to use what I call a "web-first workflow." The basic idea is that we continue to develop the game in Visual Studio, or your Web editor of choice, but do all of our testing and debugging in the Web browser while continually testing the game out on Windows 8. It's actually very easy to set up, and you may find it to be similar to the way you already work.

Setting Up a Local Web Server

Chances are good that you are already working with some kind of local Web server for testing your own HTML5 game. I tend to use Apache since it's widely used at this point and there are several stable builds you can easily run on Windows 8. Some of the techniques I will describe will also work for other servers, so don't worry if Apache isn't your cup of tea. If you are interested in using Apache, you can use a copy of WAMP (Windows, Apache, MySQL, PHP), which you can get here. It's a one-click Apache solution similar to XAMP, another one-click Apache install solution, if you have used that in the past. Setting it up is easy; I'll quickly walk you through the process.

Simply download the installer and run through the setup wizard (Figure 8-1). It's going to install to the root of your C: drive.

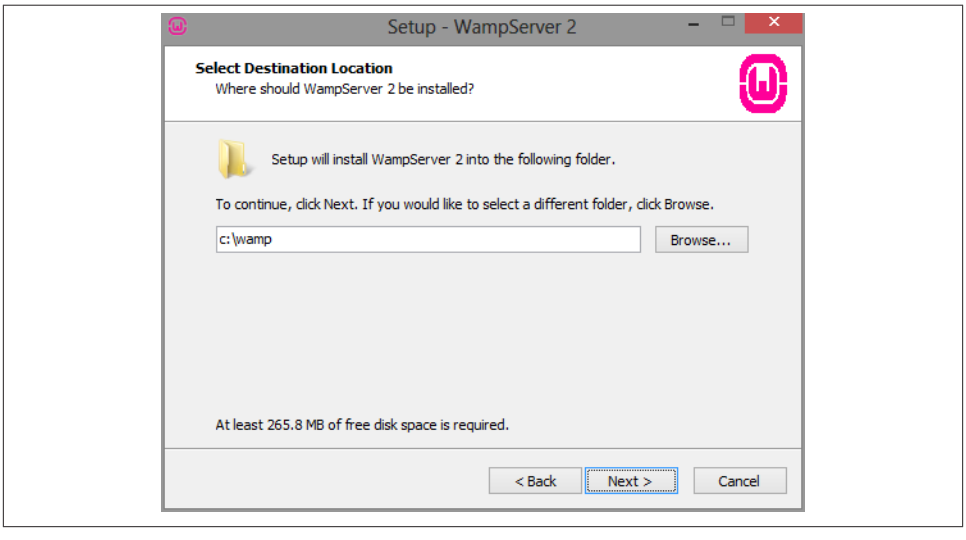


Figure 8-1. By default, WAMP will install to the root of the C: drive.

Once you have WAMP up and running, we are going to want to point the DocumentRoot to where you do your development. On my computer I have a directory called "Development," which I keep inside of my SkyDrive folder (Figure 8-2). Inside of my Development directory I have each platform I work on broken out into its own folders as well. Figure 8-2 is a screenshot of how I have my workspace set up.

The reason I have all my development files organized like this is so that I can set the root of Apache to my Development directory. This way I can run Apache on any project I am working on. Figure 8-3 shows how I set that up by modifying the apache config file.

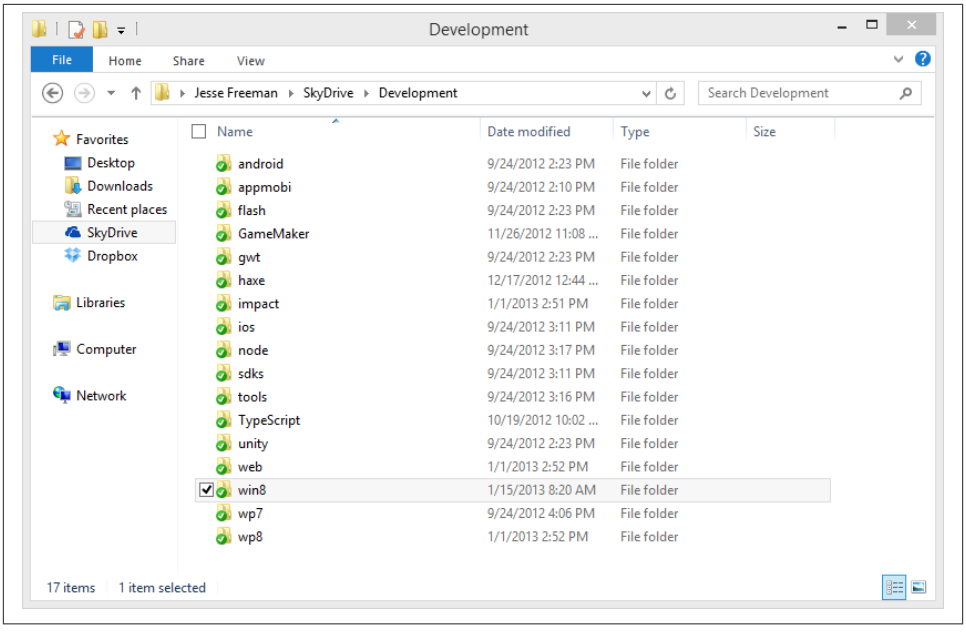


Figure 8-2. My Development directory inside of SkyDrive.

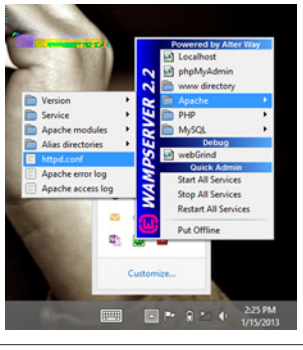


Figure 8-3. You can access the httpd.conf file from the try icon and navigating to the Apache tab.

In the WAMPServer icon in the Windows tray, go to the Apache section and select httpd.conf. This will open Apache's config file in Notepad. You can do the same thing in XAMP as well.

Once we have the file open we are going to want to make a few modifications. The first is to change the port number. I do this in order to avoid any conflicts if you are running

other servers like Node.js or Ruby. I use port 8888, but you can set it up to something else. Search for:

```
Listen 80
```

in the .conf file and change it to:

```
Listen 8888
```

Next, we are going to search for:

```
DocumentRoot "c:/wamp/www/"
```

This is WAMP's default setup. Change it to the path where you plan on doing your own development. On my computer it is going to be:

```
DocumentRoot "C:\Users\Jesse\SkyDrive\Development"
```

Now we need to make one more modification. Search for:

```
<Directory "c:/wamp/www/">
```

Change it to your development path just like we did before. Here is how it looks for my computer:

```
<Directory "C:\Users\Jesse\SkyDrive\Development">
```

After you have made these changes we will need to restart Apache (Figure 8-4).

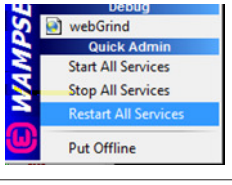


Figure 8-4. Restart Apache from the WAMP menu.

Now if you open your browser and navigate to your localhost you will see all of your projects. As you can see in Figure 8-5, I have navigated to my Win8 project folder in SkyDrive through the browser.

This will be especially helpful when you want to work on your game for the Web and also test it out on Windows 8.

It is important to note that in Windows 8.1 SkyDrive has been integrated into the system and is not its own separate syncing app. During the writing of this book I only had access to the preview copy of Windows 8.1 and found some issues with this approach since SkyDrive now allows you to have all of your files visible locally but stay online unless you explicitly tell their parent directory to sync offline. Just keep this in mind and if you run into issues consider using another cloud syncing service or put all of your code in your documents folder.

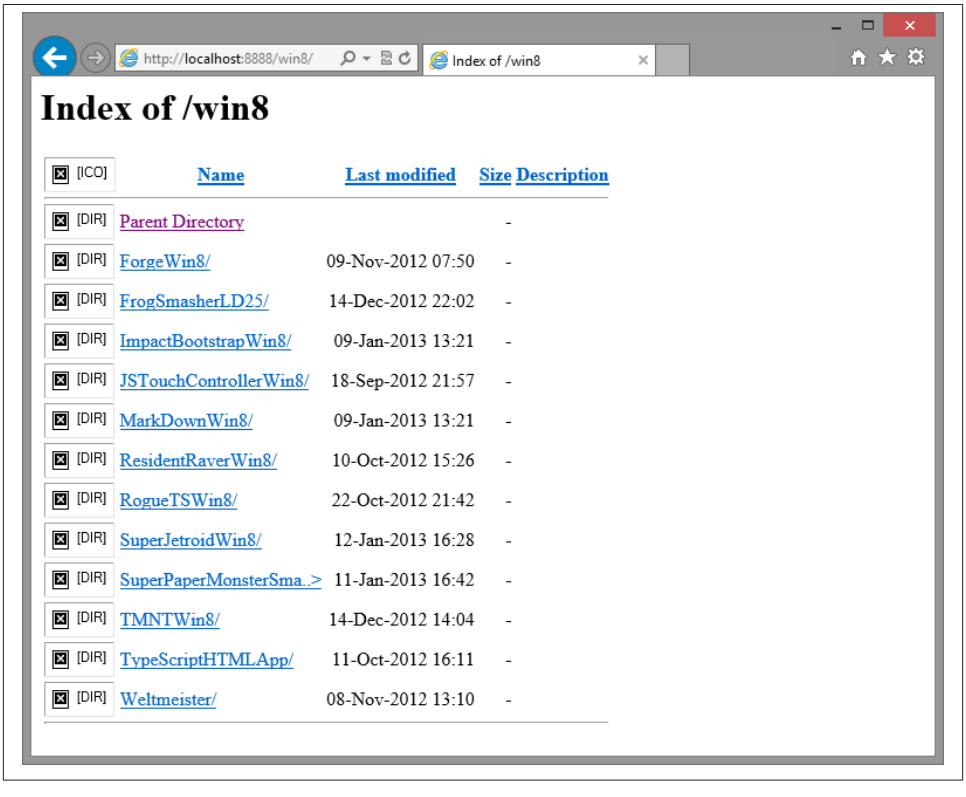


Figure 8-5. Here are all of my Windows 8 projects now loading up through Apache.

Using Node.js

Another great alternative to Apache is Node.js. It offers a lot of advanatges, such as being able to run a Web server from any directory on the fly and being able to automate your project's workflow. Let's take a look at how to quickly set up a local server with Node.js.

Simply visit *http://nodejs.org* and download a copy of the installer for Windows (Figure 8-6).

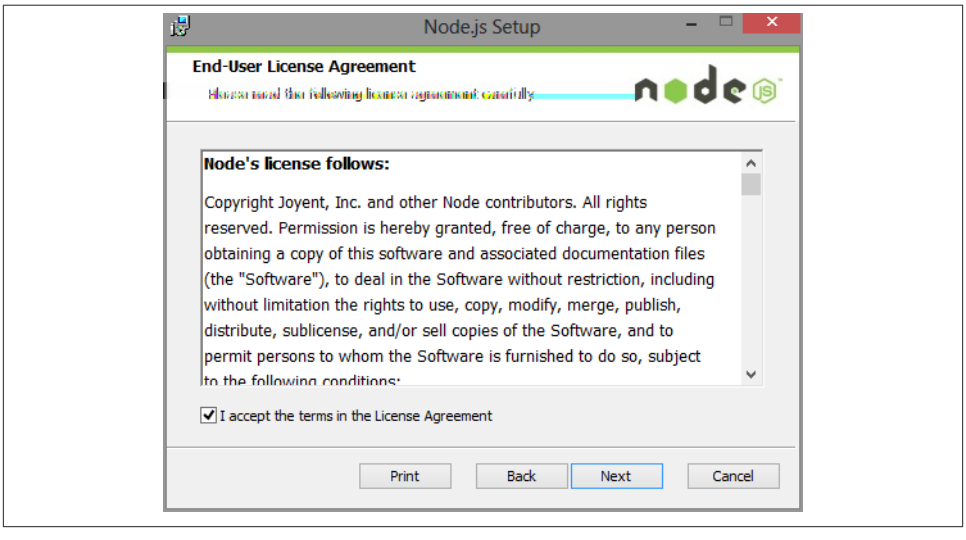


Figure 8-6. The Node.js installer.

Once you install Node.js you will be able to start setting up a simple project workflow. There are a few automation tools out there, but I ended up settling on one called Grunt, which you can learn more about at *http://gruntjs.com*. Grunt is basically the Node.js equivalent of Ant and for the most part is actually better in many ways since you can use your existing JavaScript skills to write automated tasks and when used with NodeJS you can automate and serve your entire project from the same folder.. Another important reason to use Node.js is so we can create a constant working environment for our project. While you can easily just run a server from the command line, Grunt will allow us to do some powerful workflow automation tasks, which I will talk about in the next chapter. In order to install Grunt or any other Node.js modules, you need to do a few things on the command line, so open that up.

To install Node.js modules we will be using a utility that comes with the Node.js install called NPM. This command line-based app will go out and fetch modules for you and install them globally or in your project. To get started, simply type the following to make sure NPM is correctly installed:

```
> npm
```

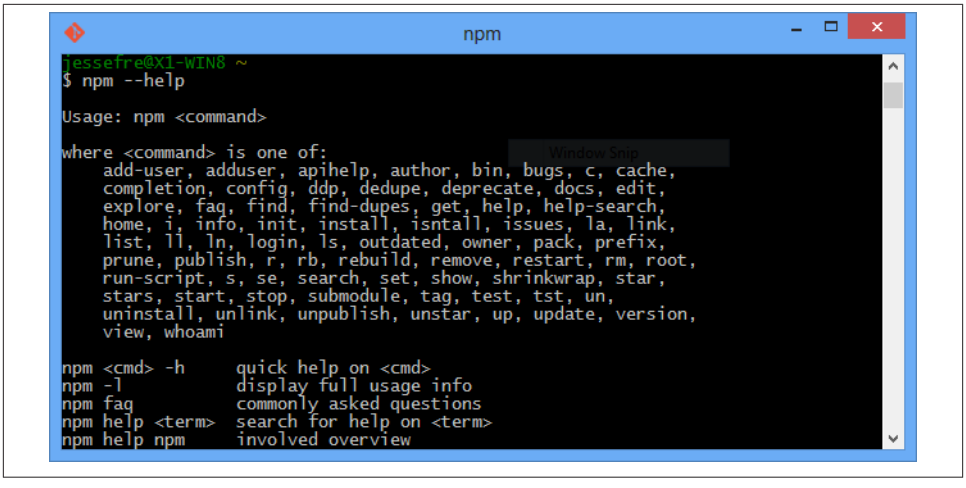


Figure 8-7. Running NPM from the command line will pull up help.

This will run you through all the commands you can use with NPM (Figure 8-7). Now that we see that NPM is working, I'm just going to go over a few of the basic setup commands I used for my own projects.

On a side note, if you are on Windows and have GitBash installed you can use that instead of the command prompt. This is incredibly helpful for people like me who come from a Unix background so you can still continue to use all the commands you are used to.

So, once you are in a project, you are going to want to set up a file to keep track of all the modules your project needs. To do this just CD into your project's directory and type the following:

```
> npm init
```

This will start a wizard to create a configuration file to store all of your modules. Simply walk through the setup questions (Figure 8-8).

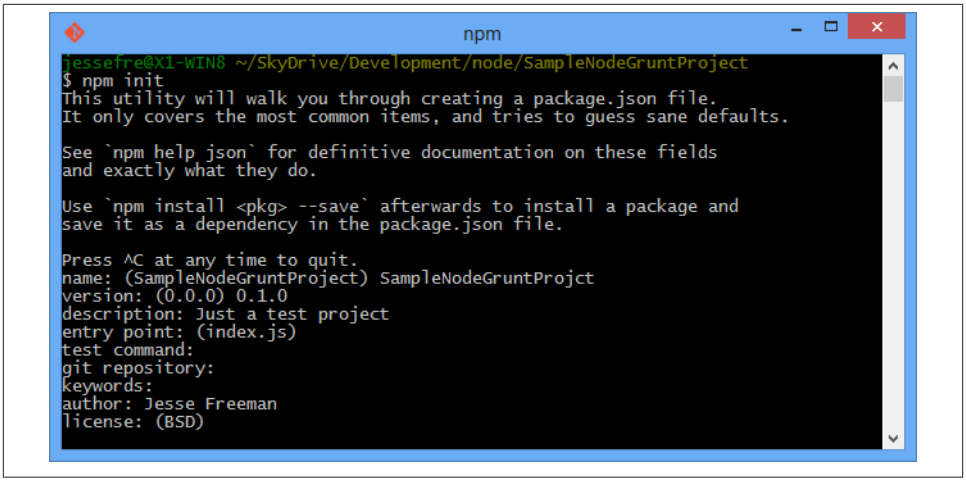


Figure 8-8. The NPM init wizard will walk you through creating a project.json file.

When you are done, you will be asked to save everything as shown in Figure 8-9.

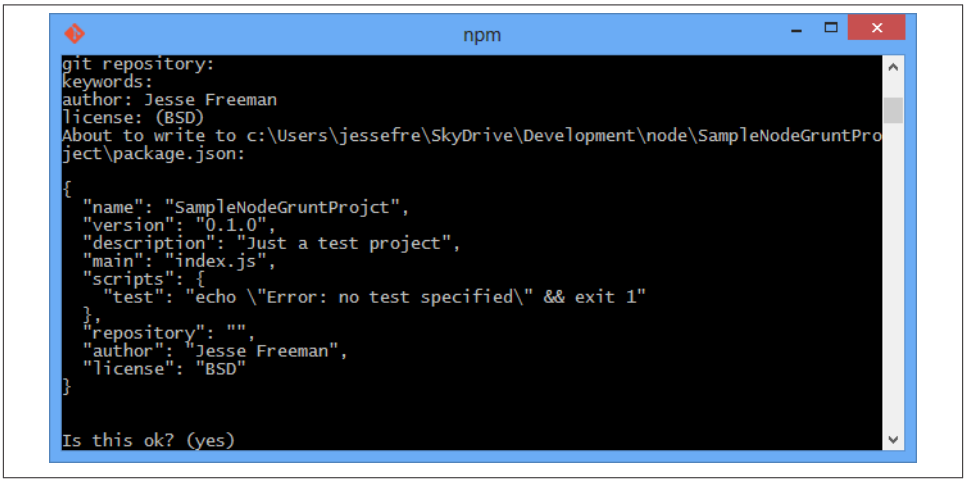


Figure 8-9. Once you fill in all the data you can save out the project.json file.

Now you should have a file named package.json. Next, we are going to want to add some modules to your project. Go back to the command line and type in the following:

```
> npm install -g grunt-cli
```

This command will install Grunt's command line tools. From there you can install Grunt into your project.

```
> npm install grunt -save-dev
```

This will update our package.json with information on the module we just installed. This is critical for sharing your build with others so they can clearly see the dependencies your project may have. If you open up your package.json file you will now see a new object at the end called devDependencies.

```
"devDependencies": {
    "grunt": "~0.4.1"
  }
```

Now you will have everything you need to run Grunt; we just need to install two simple tasks to allow us to run a server in our project.

```
>npm install grunt-contrib-connect -save-dev
```

The last thing we will need is a way to open up a URL.

```
>npm install grunt-open -save-dev
```

By now your devDependencies should look something like this:

```
"devDependencies": {
    "grunt": "~0.4.1",
    "grunt-contrib-connect": "~0.2.0",
    "grunt-open": "~0.2.0"
  }
```

From here we will need to set up a GruntFile. Think of this as our Ant build.xml. Create a new GruntFile.js in your project and add the following to it:

```
module.exports = function (grunt) {
    grunt.loadNpmTasks('grunt-contrib-connect');
    grunt.loadNpmTasks('grunt-open');

    grunt.initConfig({
        pkg: grunt.file.readJSON('package.json'),
        connect: {
            server: {
                options: {
                    port: 8080,
                    base: './'
                }
            }
        },
        open: {
            dev: {
                path: 'http://localhost:8080/index.html'
            }
        }
    });

    grunt.registerTask('default', ['connect', 'open']);

}
```

So, with this file you should be able to CD into your directory where the GruntFile.js is and run it. Let's go through how the file works before we run it.

To start off, we need to import the modules we need, so you will see the following:

```
grunt.loadNpmTasks('grunt-contrib-connect');
grunt.loadNpmTasks('grunt-open');
```

Here we are simply importing the Grunt tasks for connect and open. After this we need to configure each command. Our first one sets up the server.

```
connect: {
    server: {
        options: {
            port: 8080,
            base: './'
        }
    }
}
```

The name of the object, "connect," is how we will reference this task later on when we run it. Each task takes a set of options; in this case we are configuring a server with port 8080 and a base directory to serve as the root of the project. Now all we need is to set up a simple open task.

```
open: {
    dev: {
        path: 'http://localhost:8080/index.html'
    }
}
```

This will allow us to open up our computer's default browser and point it to our localhost.

The last thing we need to do is create some tasks to run this. In order to do this we simply register a task with Grunt.

```
grunt.registerTask('default', ['connect', 'open']);
```

Now that we have gone over how the task works you simply CD into the directory with the GruntFile.js and call grunt.

```
> grunt
```

With everything properly configured to your own project structure, you should see your browser open up when you run the Grunt build. Now you can continue to do your testing in your browser with Node.js running and make changes to your code in Visual Studio.

Project Structure

We have already gone over how a Windows 8 HTML5 project is structured in Visual Studio, so hopefully this will look familiar to you. You may recall that VS creates a

default.html file to host your project. This will actually work to our advantage since you are probably used to using an index.html file to run your game from. The good news is that we can keep both in our project and they will not affect each other. When we want to deploy our game to the Web, we will use the index.html file and VS already should be set up to run the default.html file for our project.

Another great thing about this approach is that we can add Windows 8-specific JavaScript files to the default.html file and ignore them in our Web-based version, which loads up from the index.html file. This allows you to also build logic into your game's core that knows when to load up the Windows 8-specific code or ignore it if it is running on the Web. Here is an easy way to test for that in your code:

```
if (typeof(WinJS) == 'undefined')
    // Execute code for none win8 playback
```

As you can see, if we don't have a reference to WinJS we can assume the game isn't running on Windows 8.

Modifiying the Default JavaScript File

A new Windows 8 HTML5 project in Visual Studio makes use of a default.js file. This contains the boilerplate application lifecycle support; see Chapter 2 for more information on this and on how it works. Ideally, you want your game to start up when the WinJS app calls onActivated(). This works fine in Windows 8 but will fail on the Web since it is only loaded when your project is running inside of Windows 8. You may have something that looks like this from earlier:

```
(function () {
    "use strict";

    WinJS.Binding.optimizeBindingReferences = true;

    var app = WinJS.Application;
    var activation = Windows.ApplicationModel.Activation;

    app.onactivated = function (args) {
        if (args.detail.kind === activation.ActivationKind.launch) {
            if (args.detail.previousExecutionState !== activation.ApplicationEx-
ecutionState.terminated) {
                // Code to run your game for the first time
            } else {
                // Code to run when your game returns from suspension
            }
            args.setPromise(WinJS.UI.processAll());
        }
    };

. . .
```

```
    app.start();
})();
```

Here you can see we have two places to trigger our game. So I wrap the constructor of my games in a startNewGame() function so I can easily call this when the page is ready to load the game. So I would add a call to startNewGame() in my app.onactivated callback when the game loads for the first time. This covers the Windows 8 playback. In the game's main JavaScript file I would have something like:

```
if (typeof(WinJS) == 'undefined') {
    startNewGame(800, 480);
}
```

Here you can see I test if WinJS exists before trying to start my game. If WinJS doesn't exist, which would happen in a Web build of the game, it will automatically start the game.

I also maintain two separate HTML files for my game. One is the default.html that comes with the Visual Studio project; the second is an index.html file that will be used to deploy to a server. The index.html file doesn't have any reference to WinJS, so it works perfectly without any ties to the Windows 8 environment.

There is a lot you can do with this multi-page setup, especially if you plan on deploying your game to other platforms that require their own JS libraries, such as PhoneGap or AppMobi. It also works great for wrapping up a Windows 8 HTML5 game into a Windows Phone app using a browser control.

Tips and Tricks

While we covered a lot in the chapter about supporting dual Web and Windows 8 development, I thought I would highlight a few other tips I found out by using these workflows myself.

Modularize Your Code

While it's easy to test if WinJS exists when your game starts up, you may want to avoid adding in lots of branching logic for each platform. This could easily get out of control if you start to support other platforms like PhoneGap for mobile distribution or online game portals. Try to keep each platform's code in its own module, plugin, or JavaScript file, which you can load based on what platform the game is running on.

Test Regularly

The one thing that can happen is that you tend to ignore doing a Windows 8 build. One of the greatest advantages of JavaScript is the fact that you don't have to compile it. For the most part, this workflow is designed to keep your productivity high by being able

to simply hit refresh in the browser without needing to compile. Theoretically, whatever you can get to run on the Web should run fine on Windows 8, but make sure you do regular builds just to make sure.

Use Automation

With either an Apache setup or a Node.js one, you can easily set up automation scripts to maintain a Windows 8 and Web build of your game. For Apache setups I rely on Ant, and for Node.js I use Grunt. In Chapter 9 I will go over an automation solution I used for a simple HTML5 game that I got running on Windows 8.

CHAPTER 9

Case Study: Heroine Dusk

Up until this part of the book we have talked about porting HTML5 games over to Windows 8 in theory, so I thought I would show off how I took an open-source game called Heroine Dusk and got it running on Windows 8 in only a few minutes. Here is how I did it.

About the Game

Heroine Dusk was created by Clint Bellanger. The game is a retro-style dungeon crawler that was designed to work on the Web and on mobile browsers (see Figure 9-1 and Figure 9-2).

Figure 9-1. The Heroine Dusk start screen.

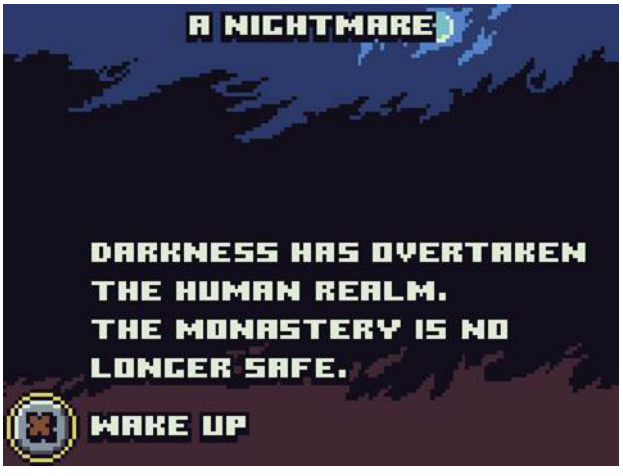


Figure 9-2. Starting a new game in Heroine Dusk.

What makes this game a perfect candidate is that it offers up simple controls that work great with a mouse or on a touchscreen (Figure 9-3).

Figure 9-3. Combat is turn based.

Also, the game offers minimal animations but has a uniquely retro-graphic style that is easily identified by anyone who is a fan of this style of game (Figure 9-4).

Figure 9-4. When you pause the game you get a world map, stats, and can see your character.

While the core mechanics of the game are easy to pick up, the game offers a good amount of depth and, as you will see in the following tutorial, Heroine Dusk made for a perfect candidate to get up and running on Windows 8.

Getting Started

To get started you are going to want to get a copy of Heroine Dusk's source code on GitHub.

Since Clint will be continuing to work on this game long after this book has been published, I forked his code so you can still follow along with the same codebase I started with. You can get my copy of the game here.

From there, simply open up Visual Studio and create a new Window 8 JavaScript project (Figure 9-5).

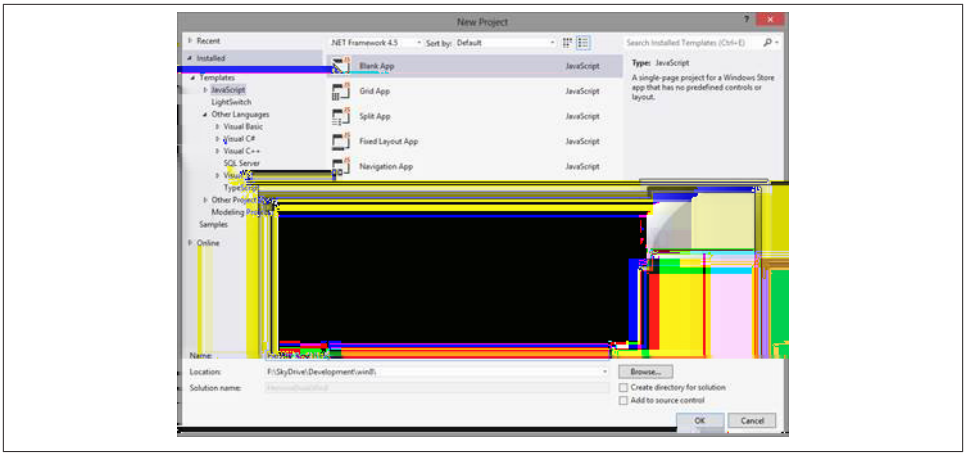


Figure 9-5. Start a new Blank App project in Visual Studio.

Now that we have a Windows 8 project, go to where you downloaded Heroine Dusk's source code and copy over all the contents of the release folder to your Visual Studio project.

Here is a neat trick for bringing existing code into Visual Studio. You can actually just copy files from the explorer and paste them into your Visual Studio solution, which not only copies them over but also includes them for you automatically.

One note, since Heroine Dusk has its own images folder and so does the default Visual Studio project, I simply moved its contents over by hand so I didn't lose the default images our project needs for a Live Tile, store icon, and splash screen.

In Figure 9-6 you can see I now have all the files needed by the game, including its index.html file. Now you need to open up the package.appxmanifest files and change a few settings which you can see in Figure 9-7.

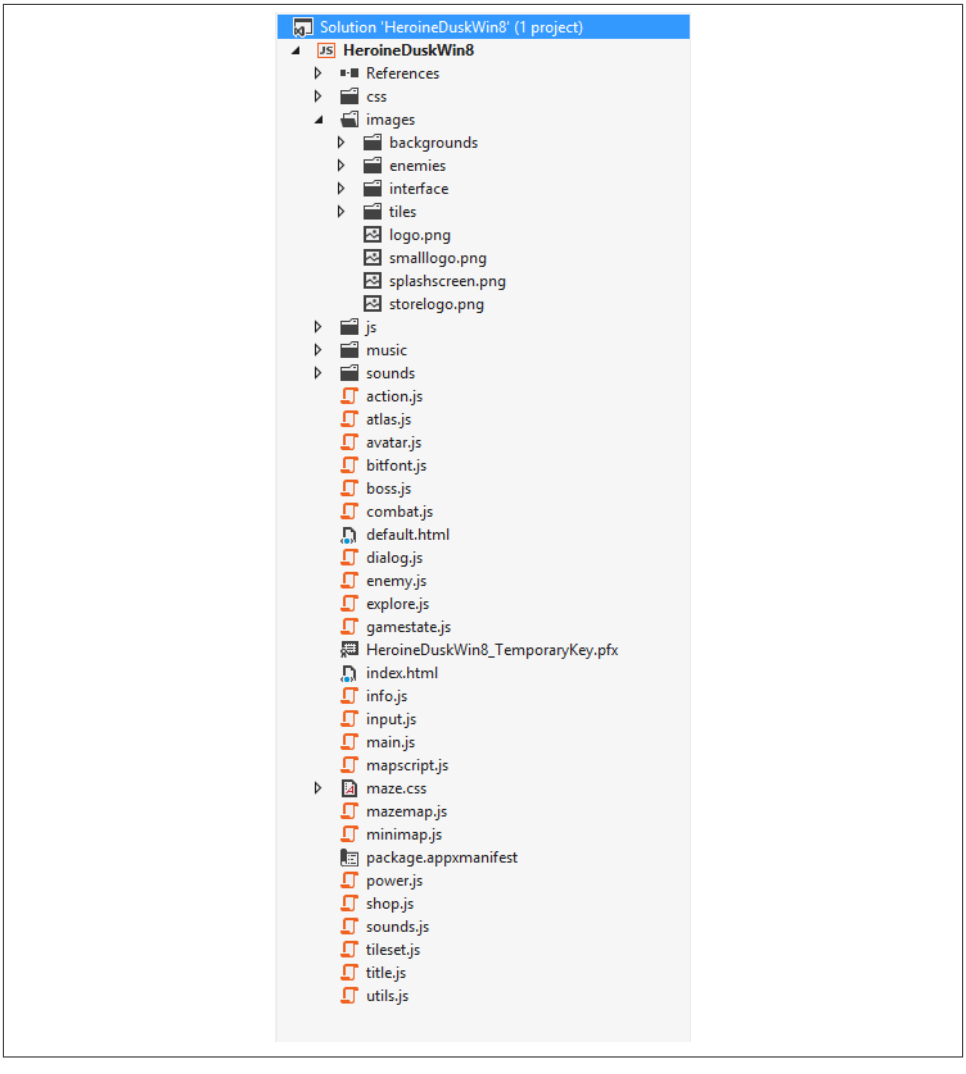


Figure 9-6. All of the files for Heroine Dusk sit in the root of the project folder once we copy them over.

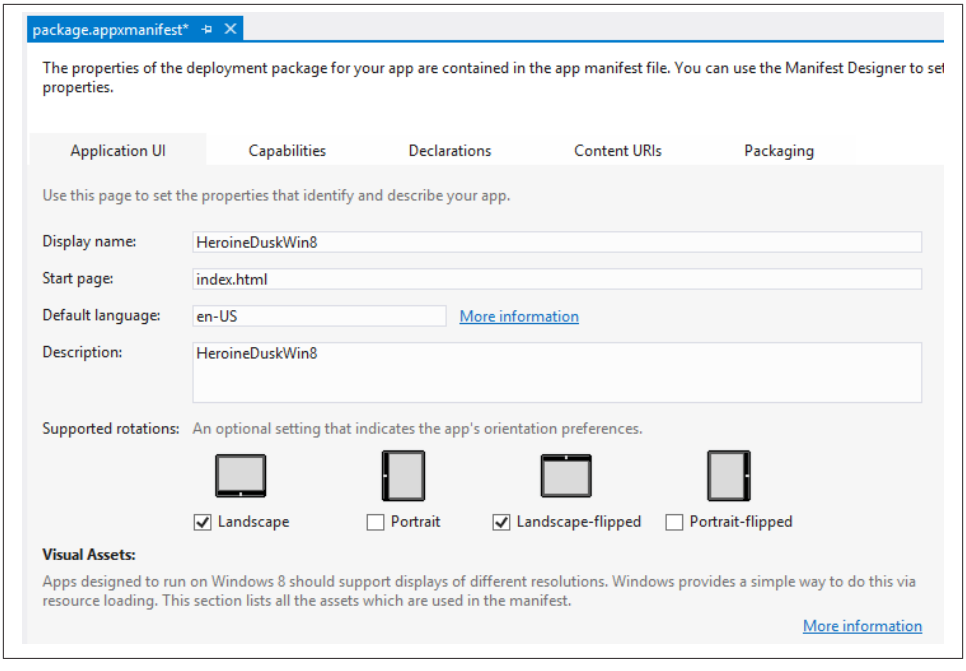


Figure 9-7. In the package.appxmanifest we can change the start page and supported orientations.

All I did here was change the start page to index.html and lock the supported rotations to landscape and landscape-flipped. This is a good technique to simply test that the game will run on Windows 8. When I am done testing this I would eventually integrate the WinJS library back into the game.

Now you are ready to compile the game and give it a try by clicking on the Build button (Figure 9-8).

Figure 9-8. When you are ready, simply click the run button to test the game out locally.

The game runs great and is already perfectly optimized for touchscreens, so there is nothing else you need to do to support Windows 8 tablets. There is one issue, however, and it's one I wanted to bring up for people making pixel art games on Windows 8 who want to get crisp-looking graphics when they scale up the canvas.

If you play the game on Windows 8 via the above instructions, you may notice that the graphics are very blurry and don't look as sharp as what you may see in Chrome or Firefox (Figure 9-9).

Figure 9-9. As you can see, the game runs great on Windows 8 but has some graphics issues since the artwork is all blurry.

The reason is that Clint is using an experimental feature for forcing nearest neighbor scaling, which is not supported by IE. This could be problematic for HTML5 game developers looking to maintain crisp-looking pixel art across multiple resolutions on Windows 8. For my own games I design for a much higher resolution so when my artwork scales up it doesn't look as blurry. I've been chatting with Clint about it and he ended up generating larger artwork to support Windows 8 and other platforms. While not ideal, it's something I wanted to highlight for people planning on making HTML5 games for the Windows Store so you can plan accordingly. I suggest supporting artwork that scales up well to 1024×768 at the bare minimum. My own games are designed for 800×480 and I scale up from there. This allows me to target mobile browsers and also still support larger resolutions on Windows 8.

With the new pre-scaled artwork Clint added to the GitHub project I was able to get a much more impressive-looking game to compile (Figure 9-10).

Figure 9-10. After swapping out the artwork the game now looks much clearer.

You will need to go into the pre-scaled folder and copy the desired image's sizes. For Windows 8, I chose the 960×720 pre-scale six folder, which is close enough to Windows 8's minimum screen resolution of 1024×768. You will also need to modify the config.js file to reflect the artwork size you are using.

So, this really took a few minutes to get up and running, but I didn't stop there. I wanted to highlight a few cool things we can do with Node.js and Grunt to allow you to still maintain a Web build of the game and Windows 8 in the same Visual Studio project. I discussed how to set up Node.js in the previous chapter, so now we can cover how to put it to use.

The first thing we are going to do is clean up all of these JS files all over the place. I am a little OCD about my project organization, so let's put all of the JavaScript files into a folder called src. Normally you would put these into the JS folder, but we are going to use Grunt to actually generate a single JS file for our Windows 8 app to use. Also, move the maze.css file into the css folder. Now you should have a much cleaner-looking project (Figure 9-11).

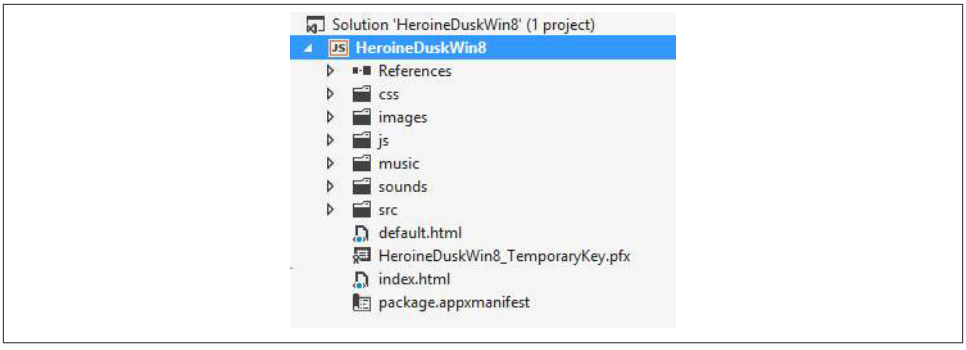


Figure 9-11. Here is what the project structure should look like after we have cleaned up all of the files.

Next we just need to fix a few paths. If you open up the index.html file you will open it up and correct the paths to the css file and all the JS files. We will actually end up replacing all of these JS files with a single minified version once Grunt is up and running, but it's up to you how your own project could work.

Now let's open up the command line and create a package.json file via NPM. Simply CD into the project's directory and run the following command:

```
> npm init
```

Go through the wizard (Figure 9-12) and you should end up with something similar to this:

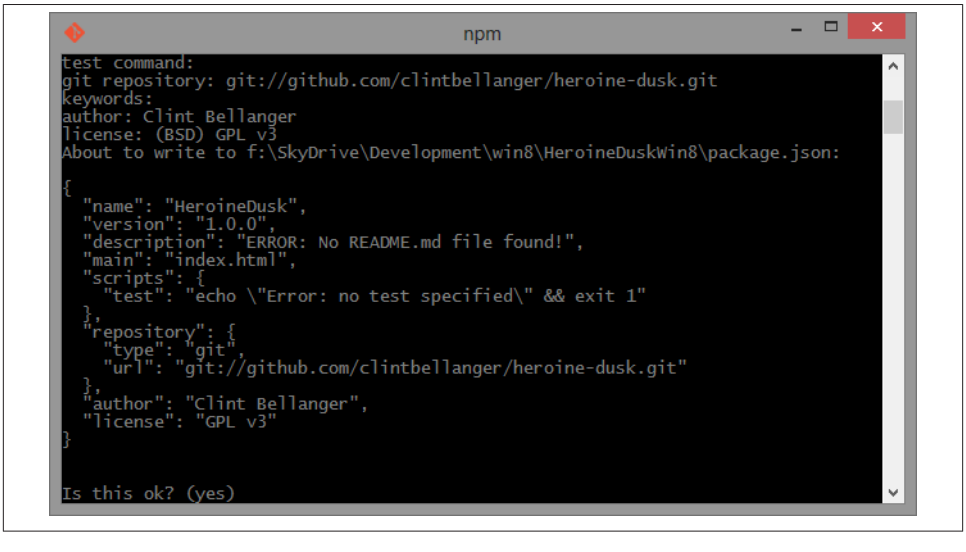


Figure 9-12. We'll start by setting up a project.json file through npm init.

Don't forget that you will need to make sure you have Grunt installed. You can read more about it here and, if you don't have it installed, simply type the following into the command line:

```
> npm install -g grunt-cli
```

Now we just want to install a few Grunt tasks.

```
> npm install grunt-contrib-watch --save-dev
> npm install grunt-contrib-copy --save-dev
> npm install grunt-contrib-concat --save-dev
> npm install grunt-contrib-connect --save-dev
> npm install grunt-open --save-dev
```

You should end up with a package.json that looks like this:

```
{
  "name": "HeroineDusk",
  "version": "1.0.0",
  "description": "ERROR: No README.md file found!",
  "main": "index.html",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "git://github.com/clintbellanger/heroine-dusk.git"
  },
  "author": "Clint Bellanger",
  "license": "GPL v3",
  "devDependencies": {
    "grunt": "~0.4.1",
    "grunt-contrib-watch": "~0.4.0",
    "grunt-contrib-copy": "~0.4.1",
    "grunt-contrib-connect": "~0.3.0",
    "grunt-open": "~0.2.0",
    "grunt-contrib-concat": "~0.3.0"
  }
}
```

Next I created a custom GruntFile that looks like this:

```
module.exports = function (grunt) {
    grunt.loadNpmTasks('grunt-contrib-watch');
    grunt.loadNpmTasks('grunt-contrib-connect');
    grunt.loadNpmTasks('grunt-open');
    grunt.loadNpmTasks('grunt-contrib-concat');
    grunt.loadNpmTasks('grunt-contrib-copy');

    grunt.initConfig({
        pkg: grunt.file.readJSON('package.json'),
        connect: {
            server: {
```

```
            options: {
                port: 8080,
                base: './deploy/web'
            }
        }
    },
    concat: {
        dist: {
            src: [  "src/config.js",
                    "src/utils.js",
                    "src/input.js",
                    "src/bitfont.js",
                    "src/sounds.js",
                    "src/enemy.js",
                    "src/tileset.js",
                    "src/atlas.js",
                    "src/mazemap.js",
                    "src/avatar.js",
                    "src/action.js",
                    "src/power.js",
                    "src/combat.js",
                    "src/minimap.js",
                    "src/info.js",
                    "src/mapscript.js",
                    "src/explore.js",
                    "src/shop.js",
                    "src/dialog.js",
                    "src/boss.js",
                    "src/title.js",
                    "src/gamestate.js",
                    "src/main.js"],
            dest: 'js/<%= pkg.name %>.min.js'
        }
    },
    copy: {
        main: {
            files: [
              {src: ['index.html'], dest: 'deploy/web/', filter: 'isFile'},
              { src: ['css/**', 'images/**', 'js/**'], dest: 'deploy/web/' }
            ]
        }
    },
    watch: {
        files: 'src/**/*.js',
        tasks: ['concat', 'copy']
    },
    open: {
        dev: {
            path: 'http://localhost:8080/index.html'
        }
    }
});
```

```
        grunt.registerTask('default', ['concat', 'copy', 'connect', 'open',
'watch']);

};
```

Basically this just creates a watch target for any changes to JavaScript files in the src folder. Once a file has been changed, I concatenate everything into a single JavaScript file then copy the images, js music, sound, and index.html files over to a deploy/Web folder. Finally, I start a server to host the contents of the Web folder by opening the index.html.

You may have noticed that I manually list each of the JavaScript files in the game to be minified. That is because Heroine Dusk required each file to be loaded in a particular order for the game to work. You may have to do the same for you own games if you have similar requirements.

This workflow allows me to continue to edit the code for the game, maintain my Windows 8 build, and test it out in the browser at the same time. This means I can continue to maintain a Windows Store build of the game and Web version without doing anything extra outside of continuing to edit the game's code. The other thing to keep in mind is that you'll notice I am using Grunt's concat task over uglify. This helps for debugging since you can still read the code if there is an error. If you try debugging uglified code you will not be very happy, but you still run that after everything is said and done if you want even more minification on the file.

Don't forget to include the HeroineDusk.min.js file that gets generated into our js directory. You will need this in order to build for Windows 8. Also, your index.html should include this file, and you can remove the links to each of the individual files. Outside of that everything else should work perfectly. The only thing missing from this game is artwork for a Live Tile, store logo, and splash screen, which are automatically generated for you by Visual Studio when you make the project. And, of course, you would need to support snap view, which is easily done through CSS.

As you can see, it didn't take much to get this game up and running on Windows 8. I was able to apply my own workflow to an existing HTML5 game and get it to seamlessly run inside of Visual Studio with builds for the Web and Windows 8. I can't stress enough how easy it is to port existing HTML5 games to Windows 8, submit them to the Windows Store, and monetize them. I also wanted to thank Clint Bellanger for making his game open source and allowing me the opportunity to talk about porting it over to Windows 8 in this book. Hopefully this helps you better understand the process outlined in each of the chapters leading up to this case study.

CHAPTER 10

Windows 8 Resources

While I tried my best to cover the most critical techniques you will need to know in order to successfully port your HTML5 games over to Windows 8 there are still some additional resources you can use to continue on your journey to publishing HTML5 games to Windows 8. To help you out, I have created this resources section to help offer more detailed examples of code to help take advantage of on Windows 8. Feel free to go through the content and links provided here to further enhance your HTML5 game on Windows 8.

Port to Windows 8 Task List

By now you have seen all of the detailed information you will need to know when porting over you game. To help you keep track of everything, I have created this handy checklist you can walk through in order to help better prepare for the porting process.

Mandatory

1. Run your game in IE10 first.
2. Import code into the Visual Studio project.
3. Connect up code in default.html.
4. Test run to make sure the game works on Windows 8.
5. Add logic to resize the game (support for full-screen, filled, and snap view).
6. Create a basic Live Tile.
7. Add a privacy policy.
8. Test your game and optimize.
9. Publish the game.

Optional steps:

10. Create more engaging Live Tiles.
11. Add support for roaming settings and game stats.
12. Support for game controller
13. Add instructions as a Settings charm.

WinJS App Lifecycle

We only touched the surface of the Windows 8 lifecycle for JavaScript apps. Outside of just tying into the onactivated() call, you may want to better integrate your app into the activate, suspend, and resume states. Here are some resources to better explain this process and how it works.

Windows Store JavaScript Samples

One of the best resources you will find for Windows 8 HTML5 app development is right inside of Visual Studio itself. Simply create a new project and select templates and it will pull down a list of sample projects covering a wide range of Windows 8 development scenarios (Figure 10-1).

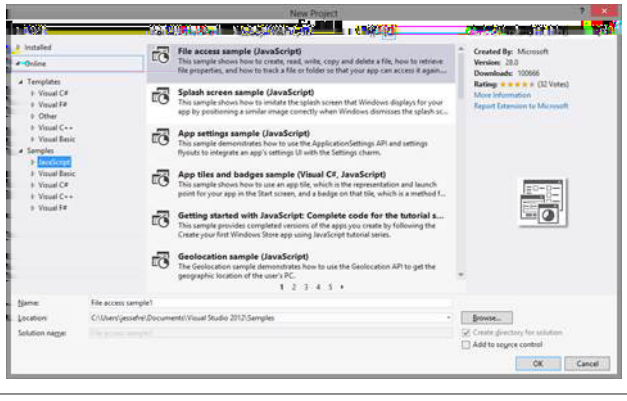


Figure 10-1. You can access a list of sample templates when you create a new project via the Online, Samples dropdown.

This book covered the basics of what you need to know in order to get an existing HTML5 game up and running on the platform. While this is a great way to create something quickly, I highly suggest going through the following Visual Studio code samples I've highlighted what I feel relates to games on Windows 8 along with their

descriptions for you to give a try. I have also added additional links to some of these projects to help along the way.

Live Tiles

You can do a lot of really interesting things with Live Tiles. By default, you get a simple image to display your game's icon, but you can animate this, display stats, and even push events and notifications to the user through the Live Tile. Here are some resources that talk about how to do additional things with the Live Tiles in a Windows 8 application:

Visual Studio Sample

- **App Tiles and Badges Sample** – This sample shows how to use an app tile, which is the representation and launch point for your app in the start screen, and a badge on that tile, which is a method for the app to relay status information to the user when the app is not running.
- **Secondary Tiles Sample** – This sample shows how to pin and use a secondary tile, which is a tile that directly accesses a specific, non-default section or experience within an app, such as a saved game or a specific friend in a social networking app.

Flyout Panels

Flyout panels are a great way to hide more advanced configuration options from the player until they really need to access them. We discuss how to make a simple flyout panel in the publishing section for the privacy policy. These panels are simply HTML pages you can load up on request, so anything you would do with a pop-up window or additional div overlay in your game will be moved over to these developer-defined flyout panels. Here are some more resources about setting them up:

Visual Studio Sample

- **App Settings Sample** – This sample demonstrates how to use the ApplicationSettings API and settings flyout to integrate an app's settings UI with the Settings Charm.
- **Edge Gesture Invocation Sample** – This sample shows how to listen for events that occur in relation to edge-based UI, using the EdgeGesture class.

Loading/Saving to Local File System

Now that we have gone over how to store data locally, you may want to take advantage of the larger storage allocations Windows 8 offers you. One technique I use is to test to see if WinJS exists and switch over to saving to the computer instead of local storage. You can create a similar API like we did in the previous section to manage it. Of course

in Windows 8 we are going to have to deal with some extra code, but let's take a look at how to get that set up.

Visual Studio Sample

- **File Access Sample** – This sample shows how to create, read, write, copy, and delete a file; how to retrieve file properties; and how to track a file or folder so that your app can access it again.
- **File Picker Sample** – This sample shows how to access files and folders by letting the user choose them through the file pickers and how to save a file so that the user can specify the name, file type, and location of a file to save.
- **Reading and Writing Data Sample** – This sample shows how to use the DataReader and DataWriter classes to store and retrieve data.

Accelerometer Support

While the accelerometer is fully supported in HTML5 games on Windows 8, I personally find using it a little difficult on larger tablets. It also doesn't make much sense on a traditional computer, which is why I didn't cover it in the chapter on input. However, adding accelerometer support is very easy. Here is some reference material on how to implement it:

Visual Studio Sample

- **Accelerometer Sensor Sample** – This sample shows how to use the Windows.Devices.Sensors.Accelerometer API.
- **OrientationSensor Sample** – This sample shows how to use the Windows.Devices.Sensors.OrientationSensor API. This sample allows the user to view the rotation matrix and Quaternion values that reflect the current device orientation.

Pen Support

When it comes to pen support, this is also a difficult feature to add in a game that may have been originally designed for the Web and desktop computers. If you remember back to our review of the pointer event in touch, you will remember that we can test the input type from the event. There may be some interesting things you can do when it comes to pen input in your game. While I wouldn't rule it out, just keep in mind that not many form factors are shipping with pens, so again, just like with the accelerometer, think about your ROI and if supporting it is worth the time investment.

Visual Studio Sample

- **Input: Devices Capabilities Sample** – This sample demonstrates how to query the input devices that are connected to the user's device and support the input modes (pointer, touch, pen/stylus, mouse, keyboard) of a Windows Store app.
- **Input: Simplified Ink Sample** – This sample demonstrates how to use ink functionality (capturing, manipulating, and interpreting ink strokes) in Windows Store apps using JavaScript, C#, and C++.

Splash Screen

We talked about how to set up a simple splash screen, which is set up automatically by Visual Studio. There are a lot of things you can do by setting up a better transition from the splash screen into your game. Visual Studio has an excellent example of how to do this.

Visual Studio Sample

- **Splash Screen Sample** – This sample shows how to imitate the splash screen that Windows displays for your app by positioning a similar image correctly when Windows dismisses the splash screen that it displays.

Dialog Boxes

Your game may need to display dialog boxes. WinJS supports a built-in system for creating and showing standard Windows 8-style dialog boxes.

Visual Studio Sample

- **Message Dialog Sample** – This sample demonstrates how to use the MessageDialog for displaying dialogs, settings commands and the actions they will perform, and changing the default button.

Trial and In-App Purchase

We talked about how to monetize your game but, if you are looking for more examples of how to do this on Windows 8, there is a very good sample for it.

Visual Studio Sample

- **Trial App and In-App Purchase Sample** – This sample demonstrates how to use the licensing API provided by the Windows Store to determine the license status of an app or a feature that is enabled by an in-app purchase.

Installing Windows 8 on a Mac

I didn't go over installing Windows 8 on PC hardware since it is very straightforward, but I thought it was important to call out that Windows 8 runs great on Mac hardware, too. For this book, I did testing on a 2012 Mac mini and in the past have used an 11" Air as my day-to-day Mac/Windows 8 development computer. The only downside of doing development on a Mac is the lack of a touchscreen.

To get started you will need to launch Boot Camp on the Mac and get a copy of Windows 8. I used Microsoft's Windows 7 USB/DVD download tool to help turn an ISO of Windows 8 into a bootable USB drive. Boot Camp also has an option to do this for you if you don't have access to another PC, but it may be problematic if you don't have a CD/DVD drive in your Mac, which new ones don't have.

Once you have the USB stick ready with Windows 8 on it, load Boot Camp on your Mac and you will see the screen shown in Figure 10-2.

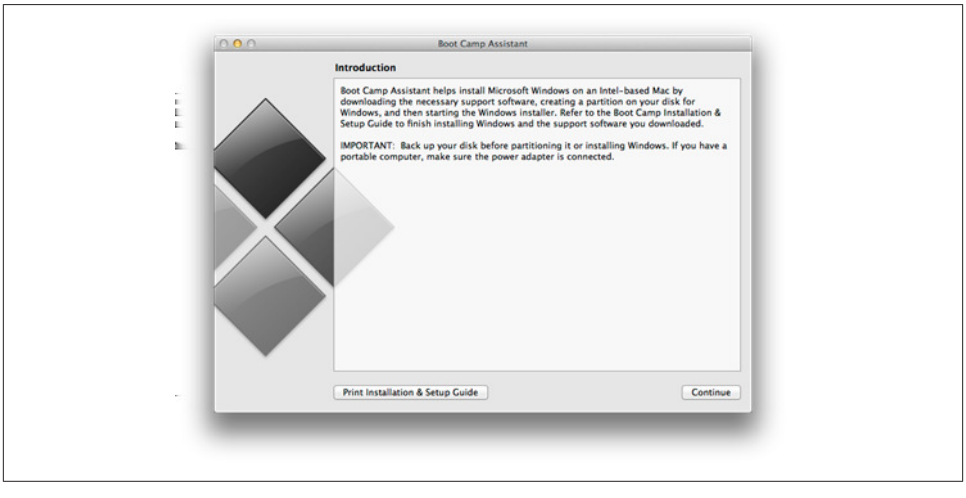


Figure 10-2. Boot Camp introduction screen.

From here you can choose to make a USB install disk of Windows 8 via Boot Camp, download the latest drivers, and install Windows 8 (Figure 10-3). Since I already created my own USB boot stick, I simply chose to download the drivers and go into installing Windows 8.

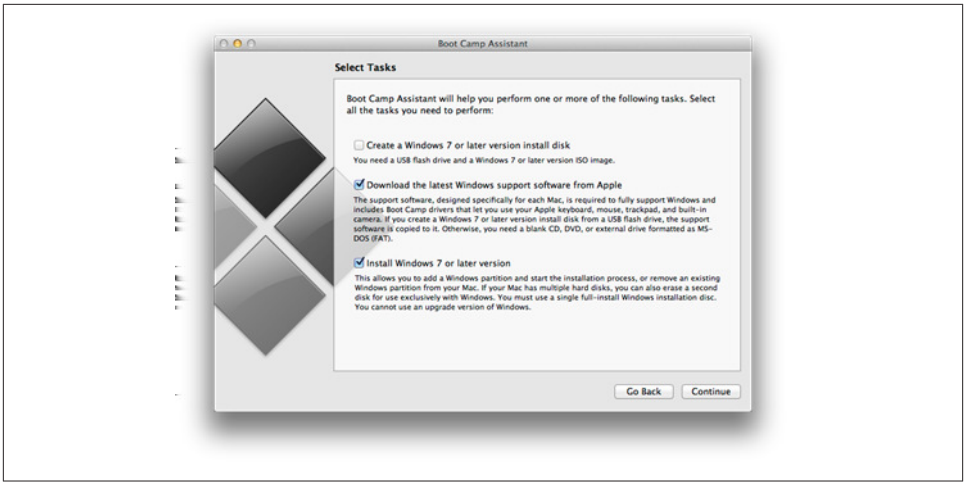


Figure 10-3. Select your options when setting up a Boot Camp installation.

You will be given a choice of where you want to install the drivers (Figure 10-4). Be sure to have an extra USB stick formatted in FAT32 so the Mac can write to it. If you are using a USB bootable installer for Windows 8, it may be formatted in a way that OS X can't write to.

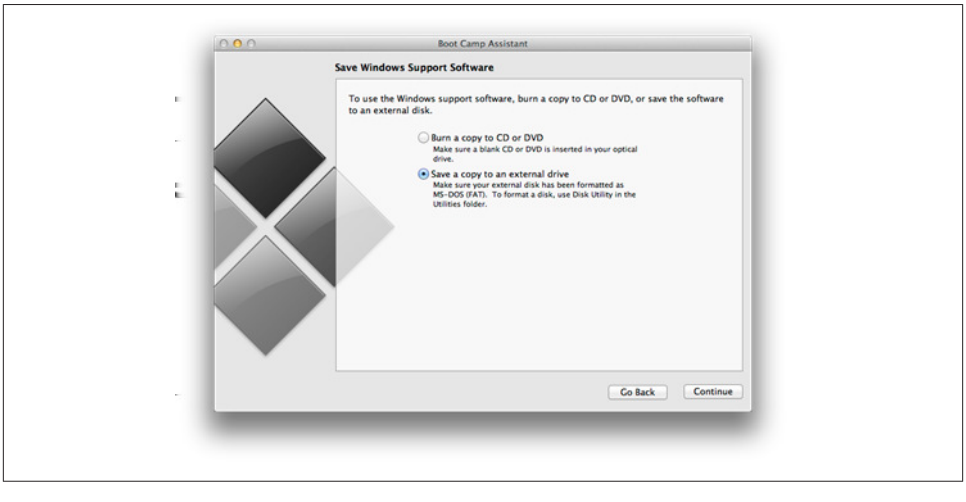


Figure 10-4. Choose a location to save the Windows drivers you'll need to get everything running once Windows is installed.

Once you have downloaded and saved the drivers, you will need to create a new partition (Figure 10-5).

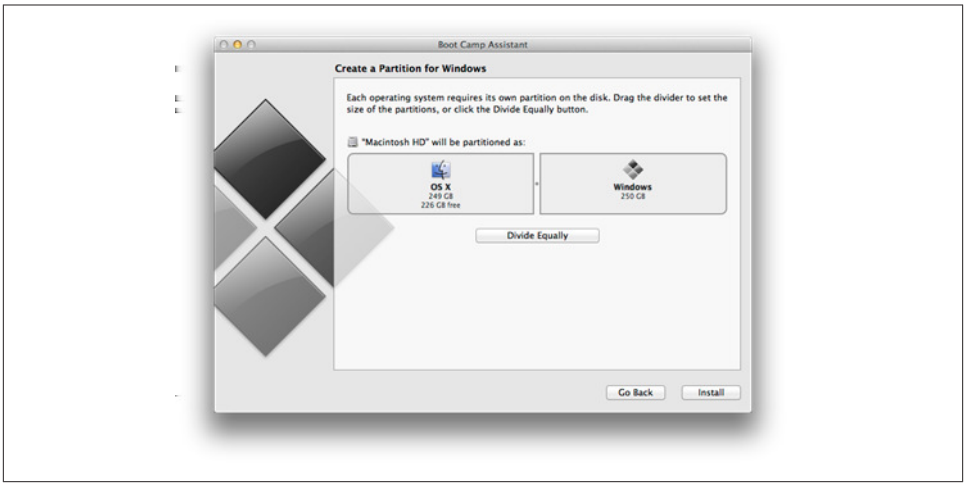


Figure 10-5. Choose the size of your partition.

I usually divide the drive up equally. Depending on what sized drive you have or how much is on it you may not have that luxury. I wouldn't go below 32 gigs. That is generally the hard drive size of the low-end Windows devices, so it should be a decent enough amount of room to work on a few projects. Once you are happy with your partition, click on the Install button and you will be ready to go through the normal Windows installation process.

Important Tip – When installing Windows 8, select the "Custom install" option, then the partition that says BOOTCAMP, and reformat it. You will see that option after clicking the Advanced option. The first time I tried to install Windows 8 on my Mac mini, I reformatted the partition. I got an error that Windows couldn't install because of a Boot Loader problem. I simply restarted the computer, held down option, which brings up the boot drive selection menu, and picked the USB stick with the installer. Then I went back to the custom installation option and tried again. Everything worked. There are lots of resources online for how to do this, so if you run into a problem, it shouldn't be too difficult to find a solution.

Once you have Windows 8 installed, you are going to want to install the Mac drivers. Chances are pretty good that you probably don't have access to WiFi or native resolution. If you remember back to the WindowsSupport folder the BootCamp installer created, you want to make sure that is handy. If you open it up in Windows, you will see all the drivers, a guide, and the setup installer (Figure 10-6).

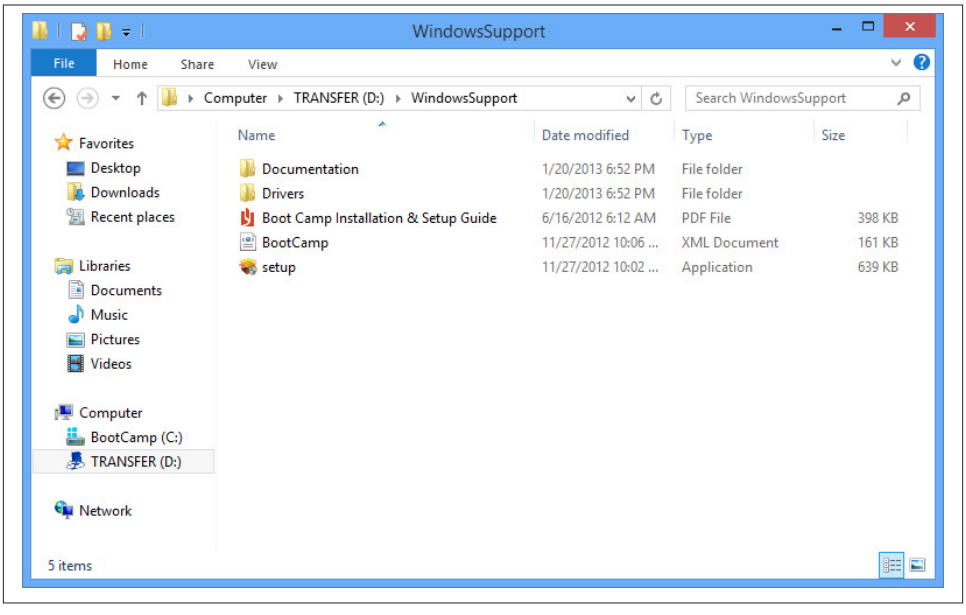


Figure 10-6. The Windows drivers.

Once you run the installer, Boot Camp will walk you through everything you need to get your computer working correctly under Windows 8 (Figure 10-7).

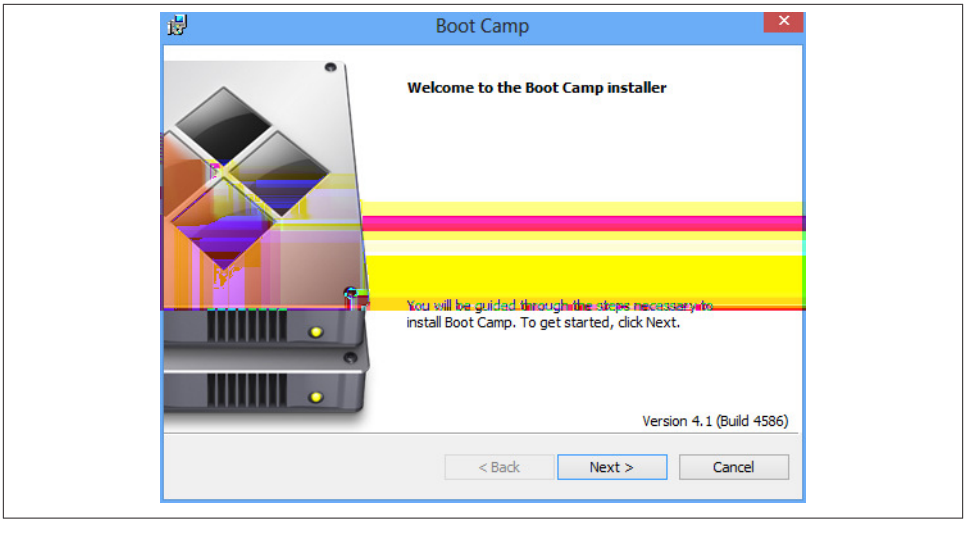


Figure 10-7. Installing the Windows drivers for Boot Camp.

Once everything is done installing, you will be prompted to restart your computer (Figure 10-8).

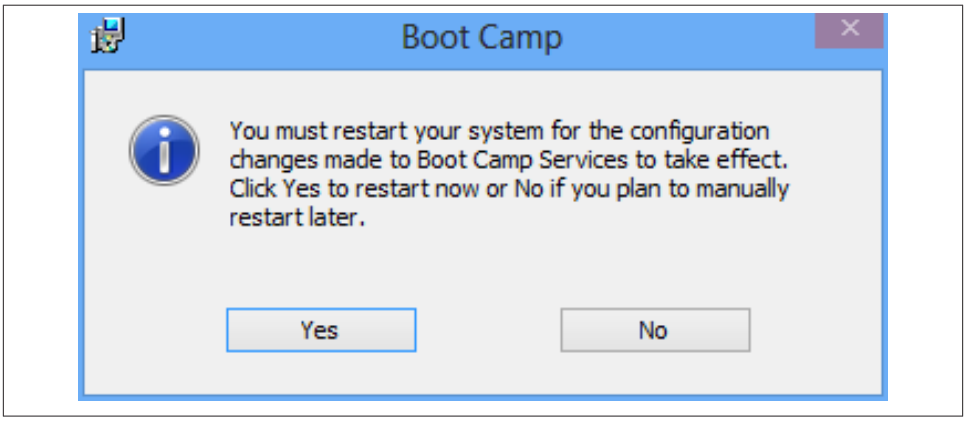
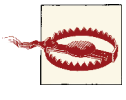


Figure 10-8. Make sure you restart your computer after installing the drivers.

Once everything reloads you should have a fully working Windows 8 install ready to go.



You may have noticed that I didn't really talk about using virtualization. While it is possible to run Windows 8 in a VM on your Mac, you may experience some performance issues that will not offer up a true-to-user experience on the same hardware. I noticed performance problems on my 11″ MacBook Air since it didn't have a dedicated graphics card. I would only suggest using a VM if you really don't have a choice and would suggest getting a dedicated testing device for performance validation and debugging.

About the Author

For more than 13 years, Jesse Freeman has been on the cutting edge of interactive development with a focus on the Web and mobile platforms. As an expert in his field, Jesse has worked for Microsoft, MLB, HBO, the New York Jets, VW, Tommy Hilfiger, Heavy, and many more. In addition to development, Jesse has a background in art with a Masters in Interactive Computer Art from the School of Visual Arts.

Jesse is a seasoned speaker who has presented and keynoted at numerous conferences around the world. In addition to in-person events, Jesse is also incredibly active and well known across social media, including Twitter (@jessefreeman) and his own blog.

Colophon

The animal on the cover of *Releasing HTML5 Games for Windows 8* is an anglerfish.

The cover image is from *Cuvier's Animals*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.