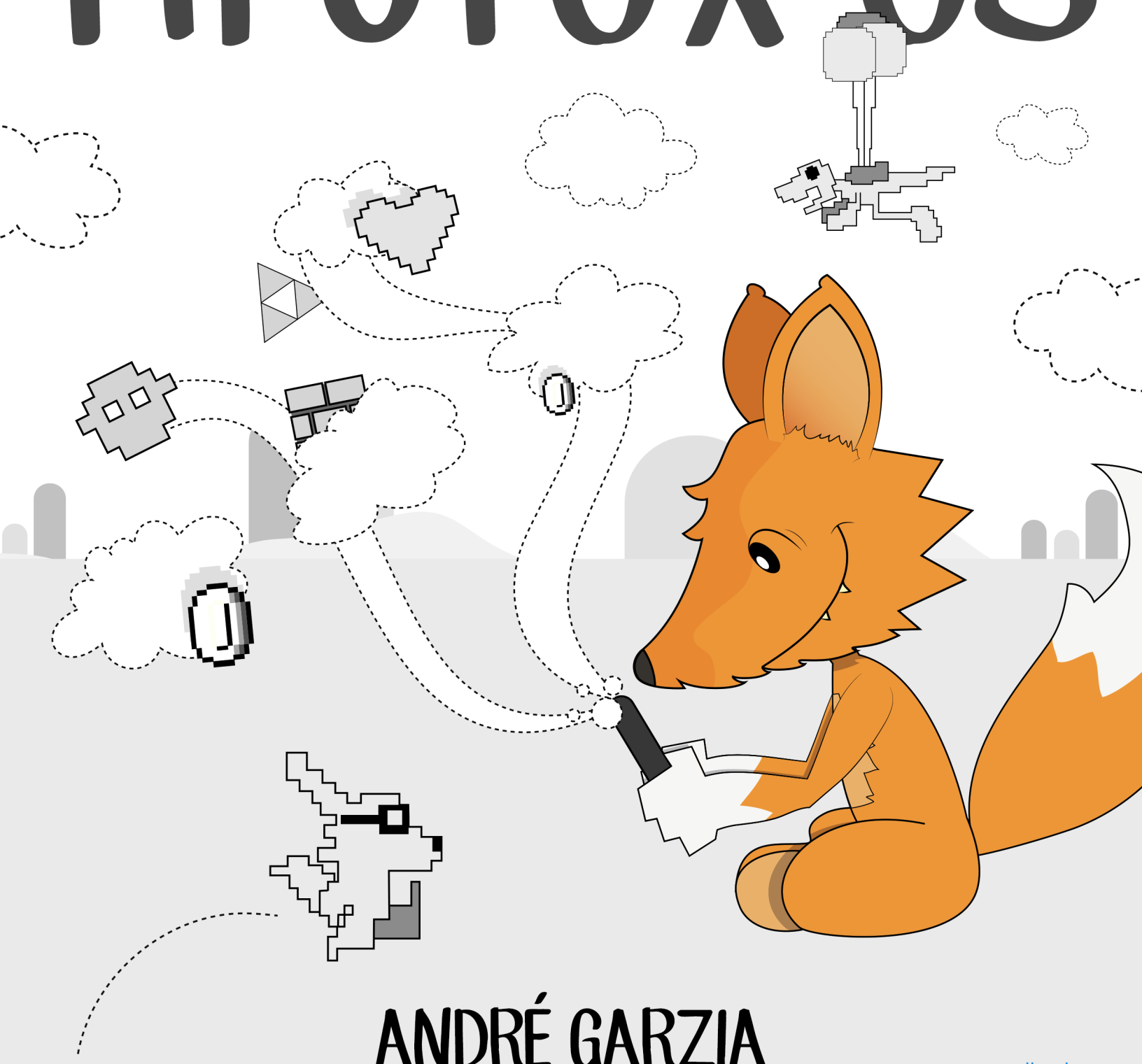
BUILDING GAMES FOR firefox os



ANDRÉ GARZIA

www.dbooks.org

Building Games for Firefox OS

A quick guide about creating casual 2D games with HTML5

Andre Garzia

This book is for sale at http://leanpub.com/buildinggamesforfirefoxos

This version was published on 2014-08-18



This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.



This work is licensed under a Creative Commons Attribution-NonCommercial 3.0 Unported License

Tweet This Book!

Please help Andre Garzia by spreading the word about this book on Twitter!

The suggested hashtag for this book is #FirefoxOS.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#FirefoxOS

www.dbooks.org

Also By Andre Garzia

Guia Rápido de Desenvolvimento para Firefox OS

Quick Guide For Firefox OS App Development

To Elisangela Mendonça de Andrade Garzia

www.dbooks.org

Contents

Warning . . . i
Acknowledgments . . . i
This book is in perpetual beta . . . i
Me, myself, and I . . . ii
Want to know more about Firefox OS Dev? . . . ii
How this book came to be . . . iii
Staying up to date . . . iii
Source code . . . iii
Donations . . . iii
How to contact the author . . . iv
Cover Illustration . . . iv
Who should read this book . . . iv
Feedback & Pull Requests . . . iv
Version history . . . v

Introduction . . . **1**
Firefox OS . . . 1
Fighting Against Proprietary Silos . . . 2
Cross-platform Technologies At Its Heart . . . 2
The Platform That HTML5 Deserves . . . 3
Accessing The Hardware Using The WebAPI . . . 3
Freedom To Develop And Distribute . . . 4
Firefox OS & Game Development . . . 4
Why Develop Games? . . . 5
Summary . . . 5

Setup For Firefox OS Development . . . **6**
The Gecko Engine . . . 6
What applications do you need? . . . 6
App Manager Setup . . . 7
Installing the Firefox OS Simulator . . . 9
Summary . . . 12

The App Manager . . . **13**
Launching The App Manager . . . 14

Adding Apps . . . 14
Running your app . . . 16
Updating your app . . . 16
Debugging . . . 17
Testing apps on a real device . . . 18
Connecting with a Firefox OS device . . . 19
Summary . . . 20

About Games . . . 21
What Are Games Anyway? . . . 21
What Makes a Good Game? . . . 22
Can't You Give Me a Straight Answer? . . . 22
What Should I Do Then?! . . . 23
Keeping a Game Design Journal . . . 23
Summary . . . 24

Our Game . . . 25
Introducing Foxnoid . . . 25
A Good Introductory Game . . . 26
Room For Experimentation . . . 26
A Look At The Big Picture . . . 26
Our Plan . . . 26
Why Phaser . . . 27
Summary . . . 29

Setting Up Our Game . . . 30
Application Types . . . 31
The Application Manifest . . . 31
There Is No Place Like Home . . . 34
Adding The Phaser Framework . . . 35
The CSS . . . 35
The HTML . . . 35
Our Application Manifest . . . 36
Summary . . . 37

Initializing Foxnoid . . . 38
A Bit About Game Lifecycle . . . 38
Game States . . . 38
What Should Our Initialization Handle? . . . 40
Initializing Phaser . . . 40
Summary . . . 43

Managing Assets . . . 44
Defining Atmosphere . . . 44

www.dbooks.org

Planing For Different Form Factors . . . 45
Planning Our Game Assets . . . 45
Building The Preload Game State . . . 49
Summary . . . 52

Gameloop . . . 53
What Is The Essential Part of a Game? . . . 53
Thinking About Our Game Loop . . . 53
Initializing Our World . . . 55
Simulating The World . . . 59
Summary . . . 62

Handling Input . . . 63
The Action That Drives The Game . . . 63
Touch Input Considerations . . . 66
Keyboard Input . . . 67
Gamepad Input . . . 67
Controlling Our Game . . . 67
Summary . . . 70

Winning or Losing . . . 71
Game Over Scenario . . . 71
Game Win Scenario . . . 76
Summary . . . 79

Engaging Players . . . 80
Making The Game A Journey . . . 80
Creating New Levels For Our Game . . . 82
Summary . . . 86

The Firefox Marketplace . . . 87
Checklist before even thinking about sending an app to the marketplace . . . 88
Preparing your app for submission . . . 88
Submitting your app to the marketplace . . . 89
Summary . . . 92

Appendix 1: Useful Links . . . 94

Appendix 2: Javascript Books . . . 95

Appendix 3: Game Development Books . . . 96

Appendix 4: Firefox OS Related Links . . . 97

Appendix 5: Gameloop Chapter Code . . . 98

Appendix 6: Input Chapter Code **102**

Appendix 7: Levels Chapter Code **106**

www.dbooks.org

Warning



Early Releases Change Often

This is a work in progress. You're handling an early release of an unfinished book.

Pick a bucket of LEGO pieces and throw them randomly on the floor. Now, blindfold yourself, spin 20 times and start walking around. This is the kind of danger you're facing here. Content will change and feedback is more than welcome.

This release is so early that its 5:15 AM in here and I am still writing…

You've been warned!

Acknowledgments

To my wife Lili, the best wife in the world!

To Mozilla for always believing in us, for keeping the web open and free and for always placing the user first!

To the Brazilian Mozilla Community for receiving me so well and being just awesome!

To my GSoC mentor Soledad Penades, the Mozilla Apps Team, the Mozilla Tech Evangelists and Dev Engagement teams for being more than awesome!

To Google for the Google Summer of Code 2014! This program is FANTASTIC.

This book is in perpetual beta

My plan is to update this book often, expanding its contents and revising the text as issues are found by readers. Since some APIs are still being implemented by Firefox OS, you will want to make sure

you're reading an up-to-date version of this book. Also as time goes by after the initial release of this book I will be able to explore more aspects of game making and thus make this a more complete resource

Me, myself, and I

In this book you will find many parts where I express my personal opinion and make decisions that may be different from what other programmers would do - particularly if it helps explain an idea more easily. I will always try to make it clear and explain my reasoning when I am giving my opinion. Anyway, if there is an error in what I am saying, I will revise the text and update the book. See the Feedback & Pull Requests section for more information.

Want to know more about Firefox OS Dev?

This book is targeted at Game Development but there are other books and resources for those wanting to know more about Firefox OS Development such as:

- The Quick Guide for Firefox OS Development[1]: a free and open source sibling to this book but focused on app development. If you're reading this book you probably will like that one too.
- Firefox OS Developer Hub[2]: The developer hub on Firefox Marketplace contains a lot of information and resources about creating Firefox OS applications.
- The Firefox OS Zone on Mozilla Developers Network[3]: MDN is probably one of the most important developer resource online focused on web technologies. Its a free and open source treasure trove of documentation, samples and articles. This link is for the Firefox OS zone that aggregates links related to this new platform.
- Beginning Firefox OS HTML5 Game Programming[4]: Bob Thulfram has released this awesome book on Amazon. If you want more resources about Game Development on Firefox OS you need to check this one out.
- Firefox OS Gaming Blog[5]: Bob Thulframs blog about Firefox OS Gaming. After you start creating games you will probably see your creations featured here :-)

There is now a site that lists all known books related to Firefox OS at http://firefoxosbooks.org[6].

[1] https://leanpub.com/quickguidefirefoxosdevelopment/
[2] https://marketplace.firefox.com/developers
[3] https://developer.mozilla.org/en-US/Firefox_OS
[4] http://www.amazon.com/dp/B00IZUYIGO
[5] http://firefoxosgaming.blogspot.co.uk/
[6] http://firefoxosbooks.org

www.dbooks.org

How this book came to be

This book is a sibling to the Quick Guide for Firefox OS Development[7] which is a similar book that cover making apps for Firefox OS. After releasing that book and talking to many people, I though we needed a similar resource targeting game development since that is often one of the overlooked aspects of developer documentation. This book shares some content with the quick guide.

I kept that idea on the back burner for when I had time to implement it but the new GSoC 2014 was upon us and I decided to submit this as a project. Being a GSoC student is wonderful and enables me to dedicate time to this type of effort without the peril of not paying my monthly bills. Its a fantastic program that empowers us students to deliver quality open source material that otherwise would take too long to create.

Staying up to date

This book is distributed for **free** using Leanpub[8].

You can register your email to receive automatic updates when you download this book from its book page at Leanpub[9]. The plan is to update this book monthly. If you got this book from a friend or from some other site, you should consider going to the page above to download and register there thus making sure you will receive the update notices.

Source code

The Foxnoid game source code is available on its Github Repo[10]. You should fetch it.

Donations

Writing a book requires a lot of work and I would like to dedicate more time in my life for this type of activity after the 2014 Google Summer of Code is done. Those that think that this book is useful (or cool) may move the price slider on Leanpub download page from zero to any desired amount and give me some bucks.

Those that would rather donate using PayPal, I can receive donations under the *agarzia@mac.com* account.

I can accept bitcoin donations at:

[7] https://leanpub.com/quickguidefirefoxosdevelopment/
[8] http://leanpub.com
[9] http://leanpub.com/buildinggamesforfirefoxos
[10] https://github.com/firefoxos-gamedev/foxnoid-html5-game

17TNW3pw9iJUD24k2CB4wv6rJaQkZUsGFU[11]

And dogecoin donations (such generosity, much development) at:

DQ4puwF5uFcdt7WZNMoo7V3Y373VD6aHFx[12]

Regardless of donations, you should fill your email on the download form to make sure that once the book is updated you will receive a notice!

How to contact the author

To send comments and feedback please send an email to fxosgamedevguide@andregarzia.com[13]. My website is http://andregarzia.com[14]. My Twitter account is @soapdog[15].

If you want to help improve the content of this book, please see the Feedback & Pull Requests section.

Cover Illustration

The cover page was created by Raphael Eckhardt, a designer and illustrator from Brazil. You can check out his work and contact him (he is a freelancer) at http://raphaeleckhardt.com/[16].

Who should read this book

This book is written for readers with an intermediate knowledge of HTML, CSS and JavaScript who wants to begin their path on the wonderful world of Game Development. Teaching HTML, CSS and JavaScript is beyond the scope of this book. I will give you links for good reference books though.

Feedback & Pull Requests

This is a Free and Open book and I am excited to receive all feedback that you people can give me. All the content of the book is at a GitHub repository[17] and is built using Markdown (with some extensions by Leanpub). To give me feedback, bug fixes and improvements just send me a pull request. Thanks in advance for all contributions.

The Git repository for this book is at https://github.com/firefoxos-gamedev/book[18].

[11] bitcoin:17TNW3pw9iJUD24k2CB4wv6rJaQkZUsGFU
[12] dogecoin:DQ4puwF5uFcdt7WZNMoo7V3Y373VD6aHFx
[13] mailto:fxosgamedevguide@andregarzia.com
[14] http://andregarzia.com
[15] http://twitter.com/soapdog
[16] http://raphaeleckhardt.com/
[17] https://github.com/firefoxos-gamedev/book
[18] https://github.com/firefoxos-gamedev/book

www.dbooks.org

Version history

Version 0.1

This is not a released book!!! This version is a work in progress version. We're releasing early and releasing often.

Introduction

Firefox OS



Firefox OS

Firefox OS[19] is a new mobile platform developed by Mozilla[20] and its partners. Devices running Firefox OS are already available in 22 countries and will reach even more places by the end of 2014. We're already seeing tablets and other form-factor prototypes emerging. The future for Firefox OS is very promissing.

The system is targeted at developing markets in emerging countries. Firefox OS has the mission to bring the next billion people online. To achieve this, Firefox OS devices are built to serve as a *great first smartphone* along with competitive pricing. These devices should not be compared with high-end smartphones such as the Apple iPhone 5S and Samsung Galaxy S5; they are built to be an alternative to feature phones so that people using said devices are able to upgrade to a Firefox OS one at an affordable price and receive the *full smartphone experience*.

[19] http://www.mozilla.org/firefox/os/
[20] http://mozilla.org

www.dbooks.org

In emergent countries such as Brazil and Colombia, smartphones with decent performance are generally too expensive for the average consumer. People are able to buy cheap phones but the platforms used in these devices are intended for high-end devices and due to the phone's entry level hardware tends to underperform thus leading to a bad user experience. Firefox OS is specifically designed to run on limited hardware while providing a decent user experience.

Fighting Against Proprietary Silos

Another differentiating factor of Firefox OS is its openness. Consider that the current mainstream mobile operating systems are proprietary silos where each vendor has the privilege to force his way on the developers and users regardless of their wishes (remember when Apple tried banning languages other than Objective-C from the iTunes App Store?). In those proprietary ecosystems you can only distribute your apps on authorized channels and the vendor usually keeps a significant part of the the money from any purchases made on the device.

Besides locking the developers to proprietary distribution channels, these systems lock you to their software development kits (SDKs). If you want to build a native app for both iOS and Android using the official toolkits you will need to code one app using Objective-C (or Swift soon) and the other with Java. This means that, code-wise, a developer will reuse very little between projects (maybe reuse some media assets). That kind of effort requires that the developer learns two languages and build the same software twice.

Cross-platform Technologies At Its Heart

Firefox OS differentiates itself by using HTML5 as its development platform. HTML5 usually means the ever-evolving collection of Web standards known as HTML 5, CSS 3 and JavaScript. These royalty free standards are implemented by all the major web browsers, and are what make web applications possible. By leveraging the technologies that encompass HTML5, millions of web developers are already able to code for Firefox OS. Apps built for Firefox OS are easy to port to another platform by using wrappers such as Phonegap[21].

> *Cool News:* Cordova 3.5 which is the FOSS project that is used to build Phonegap already supports Firefox OS. You can learn more about Firefox OS support in Cordova by going to Cordova support for Firefox OS on MDN[a] and check the current status of this project at Mozilla Cordova[b].
>
> [a] https://developer.mozilla.org/en-US/Apps/Tools_and_frameworks/Cordova_support_for_Firefox_OS
> [b] http://mozilla-cordova.github.io/status/index.html

[21] http://phonegap.com

The Platform That HTML5 Deserves

The Web is everywhere. Its on your computer, mobile phone, smart TV, and even in your video game consoles. The programming language of the web, JavaScript, is one of the most popular languages in the world. As already mentioned, when people talk about HTML5 they usually mean the collection of three technologies known as HTML, CSS and JavaScript. Recent advances in HTML have brought in a range of new features - advanced form controls, Web sockets, and more semantic markup - when compared to XHTML 1.0 and HTML 4.01. Advances in CSS have also introduced lots of new features, such as Flexbox and CSS Animations, that make it a lot easier to create beautiful responsive layouts. And recent advances in JavaScript have brought significant performance improvements and new capabilities, all while remaining easy to use for both beginners and seasoned developers alike.

Firefox OS is in essence, an extension of the mobile web. By making HTML5 a first-class citizen, Mozilla has opened its platform to millions of web developers. Even if some other browser vendors implement HTML5 in their mobile offerings, Firefox OS goes beyond that by offering a collection of APIs to access the underlying hardware and system using JavaScript. These APIs are collectively known as the WebAPIs.

Accessing The Hardware Using The WebAPI

Some earlier platforms also tried to create operating systems that used web technologies for app creation. For example, when the iPhone was introduced to the world, the only way to create apps was using web technologies. However, those web apps were limited in that they had no hardware or device access - meaning that only a limited range of applications could be built. When Apple then allowed developers to code apps in Objective-C, and also access the device's capabilities, it spurred a huge amount of innovation. Sadly, web apps did not gain access to the device's capabilities, and were thus left as "second-class citizens" - this made them unattractive to both users and developers alike, and unable to compete with native apps in that system.

When we say device capabilities we actually mean accessing hardware and OS level features and services such as updating the address book, creating alarms, accessing the camera and media gallery. On Firefox OS, the WebAPI[22]s are the means by which you will access many of those capabilities.

Another earlier platform, WebOS, also offered hardware access via JavaScript but never tried to standardize its APIs. Mozilla is working with the W3C and other stakeholders to make sure that the WebAPIs are an open standard and that other browsers adopt them too. As these APIs are implemented by other browsers, your apps will require less and less changes to work across different platforms.

It's important to emphasize that the WebAPIs are not exclusive to Firefox OS devices. Mozilla is implementing it for the other platforms on which Firefox runs, such as desktop and Android. This way, you can use your *open web app* in Firefox OS, Firefox on the desktop and Firefox for Android.

[22] https://developer.mozilla.org/en-US/docs/WebAPI

www.dbooks.org

Freedom To Develop And Distribute

Like everything that Mozilla does, Firefox OS is developed in the open and is free. All development can be followed on the Mozilla B2G repository[23] on GitHub. With Firefox OS you have the freedom to follow and contribute with the development of the system and you also have the freedom to distribute your applications on your own channels or on The Firefox Marketplace[24]. What's really awesome is that all the system applications are written in HTML5, so you can check them out and see how the are put together.

The main idea is that you're not locked to Mozilla for anything. If you want to pick the source code for the system and change it for your own needs, so be it. If you need to build apps for internal use on your company, or if you want to distribute your creations only on your own web site, you're free to do it. Usually, in other platforms you're locked into the official app store as the only channel to distribute your apps. Firefox OS has an official market called Firefox Marketplace which has an approval process but you're free to distribute your app outside this store if you want. Just like in the web where you can host your web site anywhere you want, on Firefox OS you can do the same with your applications.

This comes with a small caveat, sadly: some of the WebAPIs are too security sensitive to just allow anyone to use them. To distribute apps that use some of the more "privileged" APIs, you will need to get your applications signed and reviewed by Mozilla's staff.

Firefox OS & Game Development

We've talked a lot about Firefox OS but not much about gaming yet. As explained above, Firefox OS is targeted at a different market than Android and iOS so it is exploring a whole different segment that doesn't yet have established players. Its a great opportunity for new developers to make a name for themselves by creating quality games for a new platform.

A very important point is that since we're using nothing but web technologies you can quickly wrap/adapt your Firefox OS game to run in other mobile platforms and on the web itself like making it a Facebook game or a stand alone offer on your own site or game portal. HTML5 give us the only cross platform development toolkit that can go from video games, to mobile devices to the web itself. Armed with it you can spread your game all over the place! Do not think that since this book is focused on Firefox OS that we want to see you develop exclusive titles for our platform. **Our objective is to enable you to build your game and deliver it everywhere including Firefox OS**.

As you will see, Firefox OS is a great platform that is very easy to develop for. Its very simple to iterate often with your designs and experiment with new stuff.

[23] https://github.com/mozilla-b2g/B2G
[24] https://marketplace.firefox.com/

Why Develop Games?

Imagine all categories of app you can for bit. By category I mean that first level of navigation when you're on your mobile app store where it lists things such as — games, photography & video, social networks, productivity, travel… — did you know that **games account for 50% of app usage on smartphones**. So basically if two people are fiddling with their mobile device then one of them is playing a game and the other is doing something boring.

We all agree that mobile apps are a very good source of income and that many people are making a living out of it. If you plan to make a living out of mobile apps then games are a very tempting option but this alone is not a good reason to develop games in my opinion. When someone asks you why you're developing games then answering about marketshare and possible income is a valid answer but when asking yourself in the inner dungeons of your own mind why you're doing it the only acceptable answer is **I like developing games!!!**. If you start on a path motivated by money alone you will be sad in the long run (or millionaire but that is a different case).

Your main motivation for developing games should be that you have this itchy idea on your mind that you need to explore and bring to the world. Games may be a product but they are also an art form. No one look at a spreadsheet program and speak about aesthetical decisions on its choice of color palette but people regard games as something to be cherished and remembered. Just think about the crowd that E3 attracts versus the crowd that the annual convention on spreadsheet macros attract.

Game developers live in the frontier between product and art. Games can explore emotions, test artistic boundaries, play with atmospheres and be something that players will fondly remember for a long time (I still cry when I think of Aeris dying in Final Fantasy and I know its just a game). Games hold a true potential as an expression form and also as a product to be sold.

If you decide to embark on this adventure you will have a great time, just remember that the creation of great art demands a lot of time and knowledge. Mozart was once a baby babbling nonsense before composing his masterpieces (At least I hope he was). **Take your time to learn and explore and while you're doing it remember to have fun**.

Summary

HTML5 is here to stay and will only get better. Firefox OS is the new open mobile operating system by Mozilla completely based on web technologies. This system is built on the open and offers a robust HTML5 implementation that goes beyond the other platforms by offering the WebAPI which is a collection of APIs to access *hardware and operating system services using JavaScript*. These new APIs are being standardized through the World Wide Web Consortium (W3C) and will hopefully be adopted by other browsers in the future.

In the next chapter we'll setup our machines with the tools we need. Put your gear on because we're starting our game development journey…

www.dbooks.org

Setup For Firefox OS Development

In this chapter we're going to setup our machines to develop for Firefox OS. This setup is the same regardless if you're developing business apps or games.

The Gecko Engine

Browsers use different engines for rendering web pages: Google Chrome and Opera use Blink (a fork of WebKit), Internet Explorer uses Trident, while Safari uses WebKit. Mozilla has its own engine, called Gecko which is used in Firefox desktop, Firefox for Android, and Firefox OS. As these products use the same engine, it is possible to develop for Firefox OS using the Firefox desktop browser (but with some caveats[25]).

What applications do you need?

To develop and test apps made for Firefox OS you will need:

- A recent version of the Firefox Browser[26].
- The Firefox OS Simulator[27] if you're running a device with Firefox OS 1.1.
- A text editor for programming[28].

Since there are a lot of people running Firefox OS 1.1 and most of the devices sold in the retails market are still 1.1 devices we'll keep the **Firefox OS 1.1 Simulator** instructions but we'll duplicate any section related to that simulator and also present how to do the same tasks in the **App Manager**. I will present the **App Manager** content first because thats the current stuff and then present the content for the old simulator.

The **App Manager** is bundled with **Firefox 29 or newer**. We'll describe it in detail on the chapter about App Manager.

[25] Although the same engine is used for all Mozilla products, the version of the engine in Firefox OS is generally behind that of the desktop browser. This is because the release cycle of Firefox OS is currently slower than that of the Desktop browser. In practice, this will mean that some features may not be available (or work as expected) when you try them out in Firefox OS - so always make sure you test your applications on a device that runs Firefox OS. Also, be mindful that users might also be on different versions of Firefox OS, so they might not have all the bleeding edge features. Be sure to always provide a fallback in case where some feature is unavailable. Lucky for us that games don't need many features from the engine so we're pretty much okay.

[26] http://getfirefox.com

[27] https://addons.mozilla.org/en-US/firefox/addon/firefox-os-simulator/

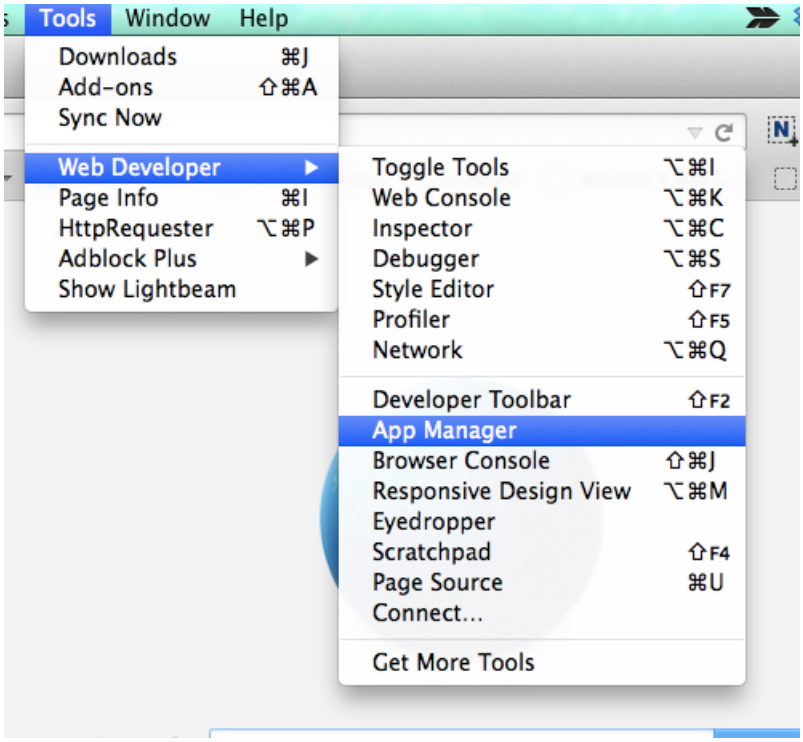
[28] There are many good editors out there with different levels of complexity and features. A very popular one that I recommend for those that don't have a favorite one is SublimeText. Personally, I use WebStorm which is a complete IDE for web app creation.

App Manager Setup

If you're running the current version of Firefox (a.k.a. 29 or newer) then you have the App Manager available to you. Having the App Manager is not enough though. You still need to install the simulators on the App Manager itself to be able to test things without hooking a device to your machine. Mozilla has extensive documentation about the App Manager[29] so if you want to dive a bit deeper check it out.

The App Manager is able to manage multiple Firefox OS versions so you can install simulators for version 1.3, 1.4 and 2.0, remember that the higher the version number the earlier it is. And by earlier I mean buggy but since we can have multiple versions then we should just install them all so that we're able to test our apps with different Firefox OS versions.

Lets take the new App Manager for a spin and install the stuff we'll need for later. To launch the App Manager go to the menu **Tools -> Web Developer -> App Manager**.

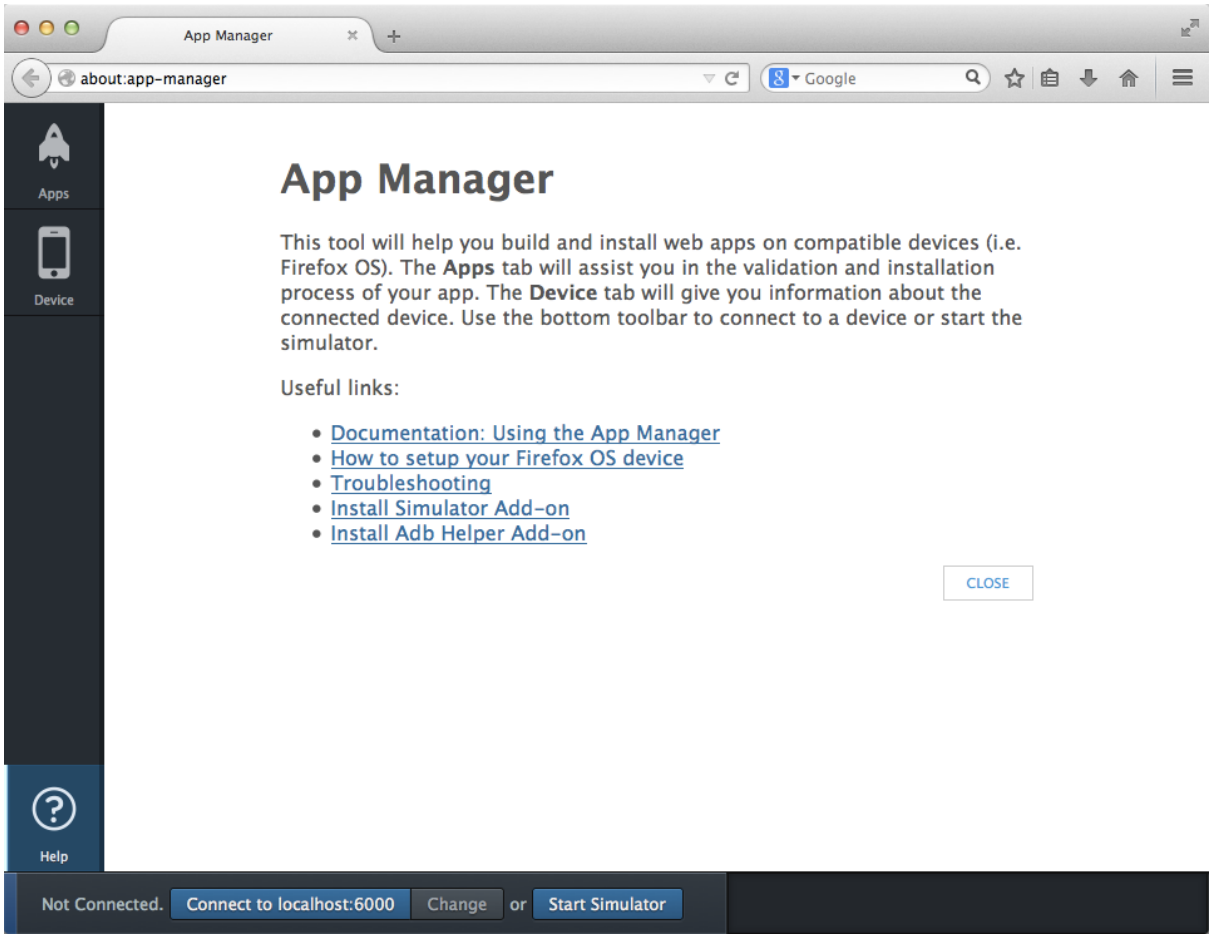


Where you can find the App Manager

After you launch the App Manager you will see a screen like this

[29] https://developer.mozilla.org/en-US/Firefox_OS/Using_the_App_Manager

www.dbooks.org



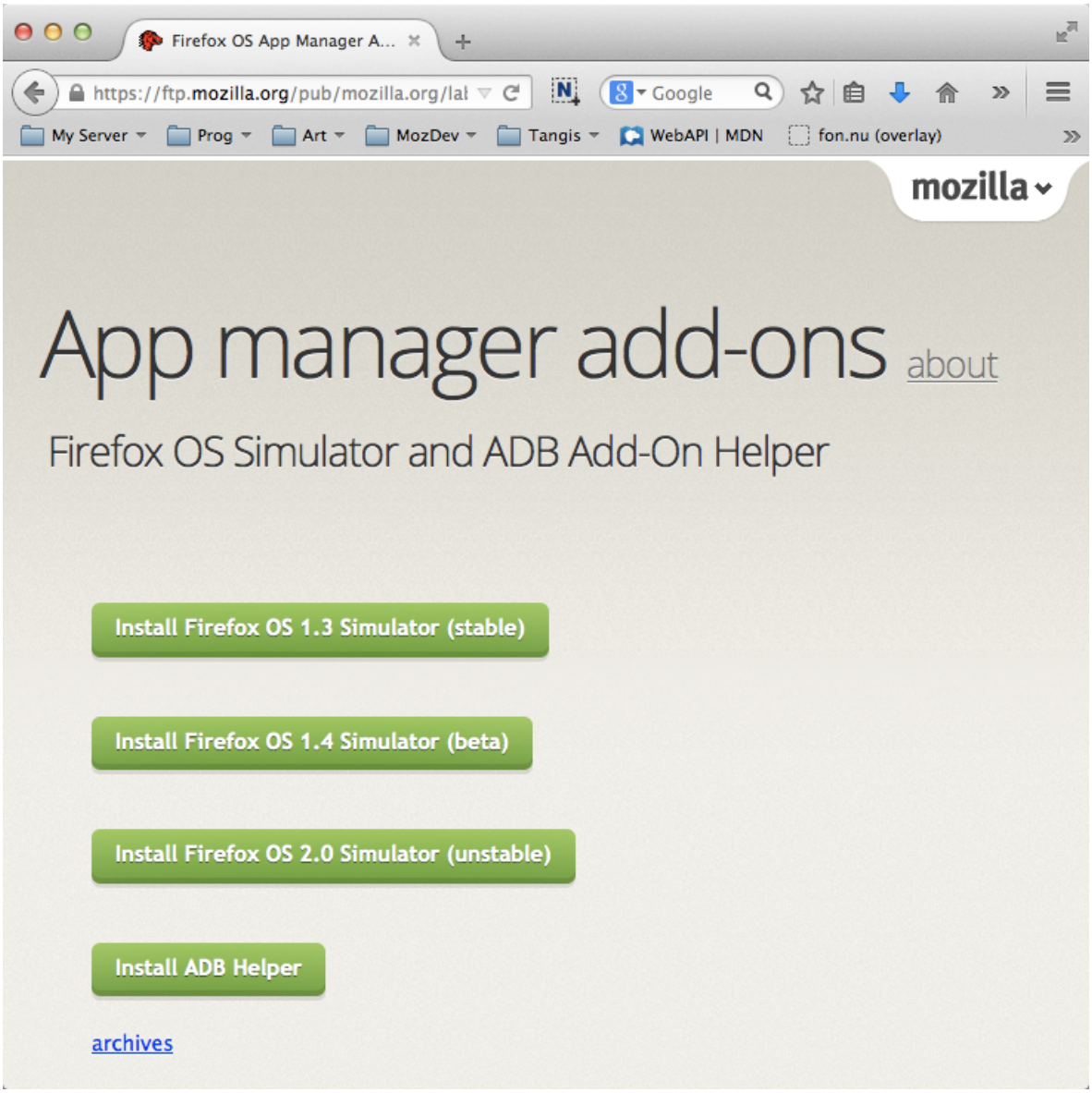
The App Manager Help

You need to click the **Install Simulator** button and select the versions you want to install.

The App Manager uses ADB to communicate with connected devices. It is able to handle all the ADB stuff for you if you install the **ADB Helper Add-on** available at Install the ADB Helper add-on[30].

My recommendation is to install all available versions of the simulator and the ADB Helper.

[30] https://ftp.mozilla.org/pub/mozilla.org/labs/fxos-simulator/



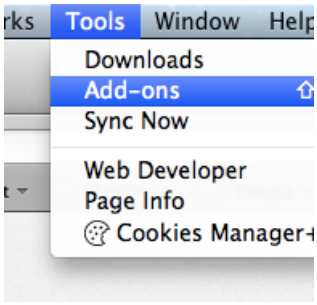
The page to download Simulators and the ADB Helper

Installing the Firefox OS Simulator

If you have a device running **Firefox OS 1.1** then you need to install the **Firefox OS 1.1 Simulator** because your device can't communicate with the new **App Manager**.

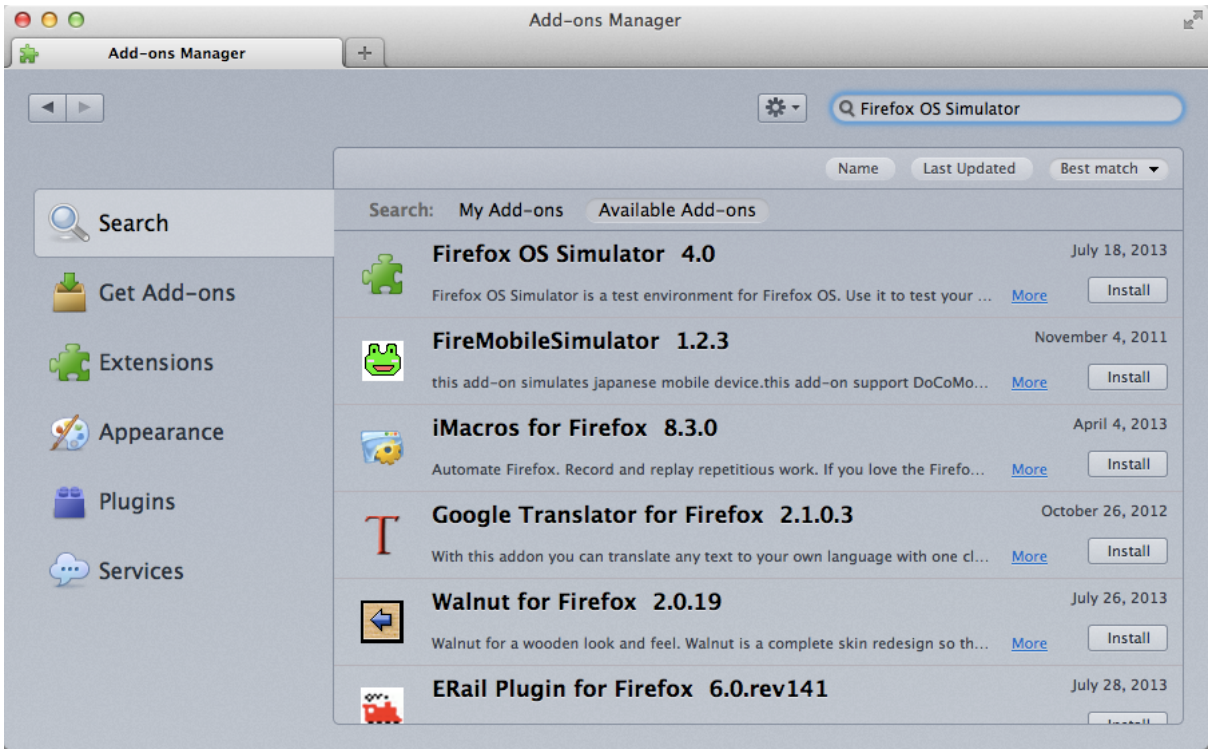
After installing Firefox, the next step is the installation of the Firefox OS Simulator that will be used to test our applications. With Firefox installed and running, go to the **Tools** menu and select **Add-ons**.

www.dbooks.org



***Tools* menu with *Add-ons** menu selected**

Using the search box on the top right corner, search for **Firefox OS Simulator** and install the add-on by clicking the install button.

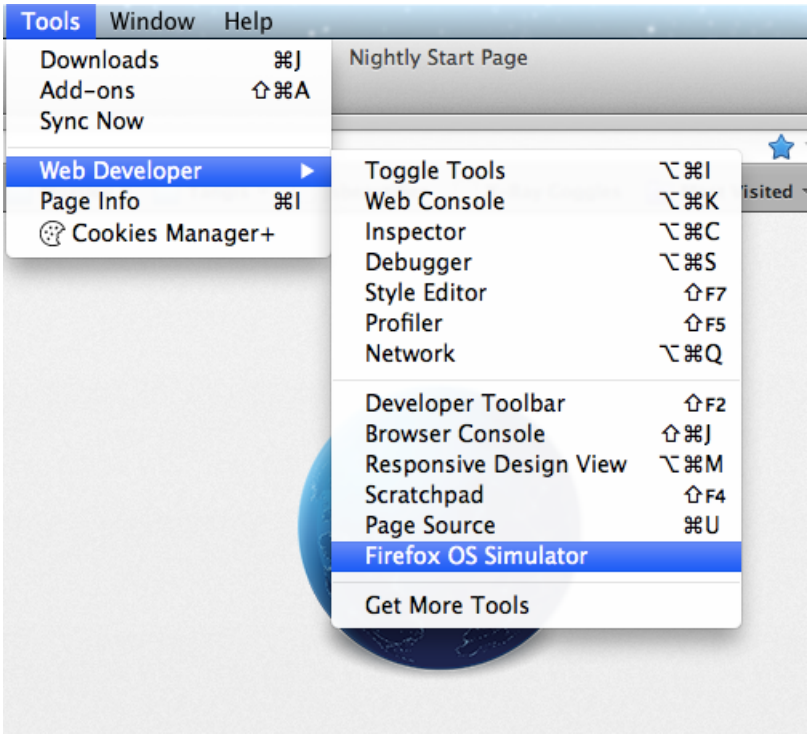


Add-on manager showing the simulator add-on



Attention: If you're running **Firefox 29 or newer** and you have a device running **Firefox OS 1.1 or earlier** then you will need another version of the **Firefox OS 1.1 Simulator** that is not currently listed on the add-ons marketplace. This version is **BETA** but its the best we've got right now. You can fetch it for Mac OS X[31], Linux[32] or Windows[33]. Just drop the xpi file on Firefox and follow the installation instructions. If you want to follow up on the quest of making the **Firefox OS 1.1 simulator** work on **Firefox 29** then check out bug request #1001590 it[34] and add your voice there so that Mozilla knows this is important.

After the installation of the add-on, you will be able to access the simulator by going to the menu **Tools -> Web Developer -> Firefox OS Simulator**.



Where you can find the simulator after is installed

Alternatively, you can navigate to the Firefox OS Simulator[35] addon page, and download the simulator from there.

[31] http://ftp.mozilla.org/pub/mozilla.org/labs/r2d2b2g/r2d2b2g-5.0pre7-mac.xpi
[32] http://ftp.mozilla.org/pub/mozilla.org/labs/r2d2b2g/r2d2b2g-5.0pre7-linux.xpi
[33] http://ftp.mozilla.org/pub/mozilla.org/labs/r2d2b2g/r2d2b2g-5.0pre7-windows.xpi
[34] https://bugzilla.mozilla.org/show_bug.cgi?id=1001590
[35] https://addons.mozilla.org/en-US/firefox/addon/firefox-os-simulator/

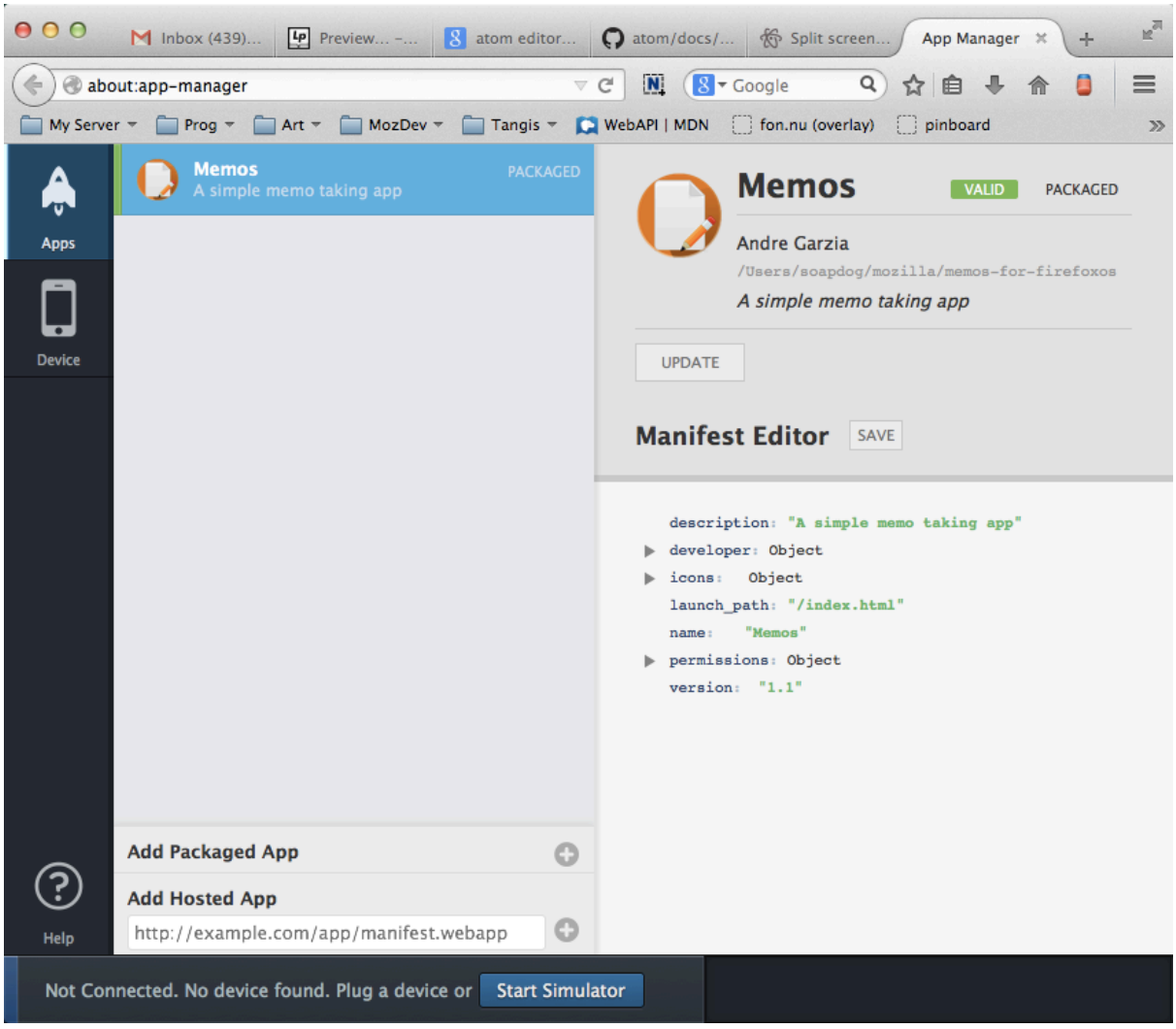
www.dbooks.org

Summary

In this chapter we learned that all we need to develop for Firefox OS is the Firefox browser, the App Manager and Firefox OS Simulators (and a good text editor).

Now that we have setup all the tools we need, its time to dive into the App Manager and learn how it work.

The App Manager



Firefox OS Simulator Dashboard

In this chapter we're going to learn the basics about the App Manager. By the end of it we'll know how to use it to test our apps.

To learn more about it, check out the Firefox OS: Using the App Manager page[36] on MDN.



Remember: that if you are using a device running Firefox OS 1.1 or older then you need to use the Firefox OS 1.1 Simulator extension with it and not the App Manager.

[36] https://developer.mozilla.org/docs/Mozilla/Firefox_OS/Using_the_App_Manager

www.dbooks.org

This chapter is best understood if you take the time to follow the steps with a sample app. Lets use the game we're going to build as a sample app for this task. Download it from the Foxnoid Github repository[37].

Launching The App Manager

To open the App Manager select the **Tools** menu on Firefox and navigate to the **Web Developer** item. Inside that item select the **App Manager**. If you're not seeing the App Manager entry there then you're probably running an old Firefox version and should upgrade.

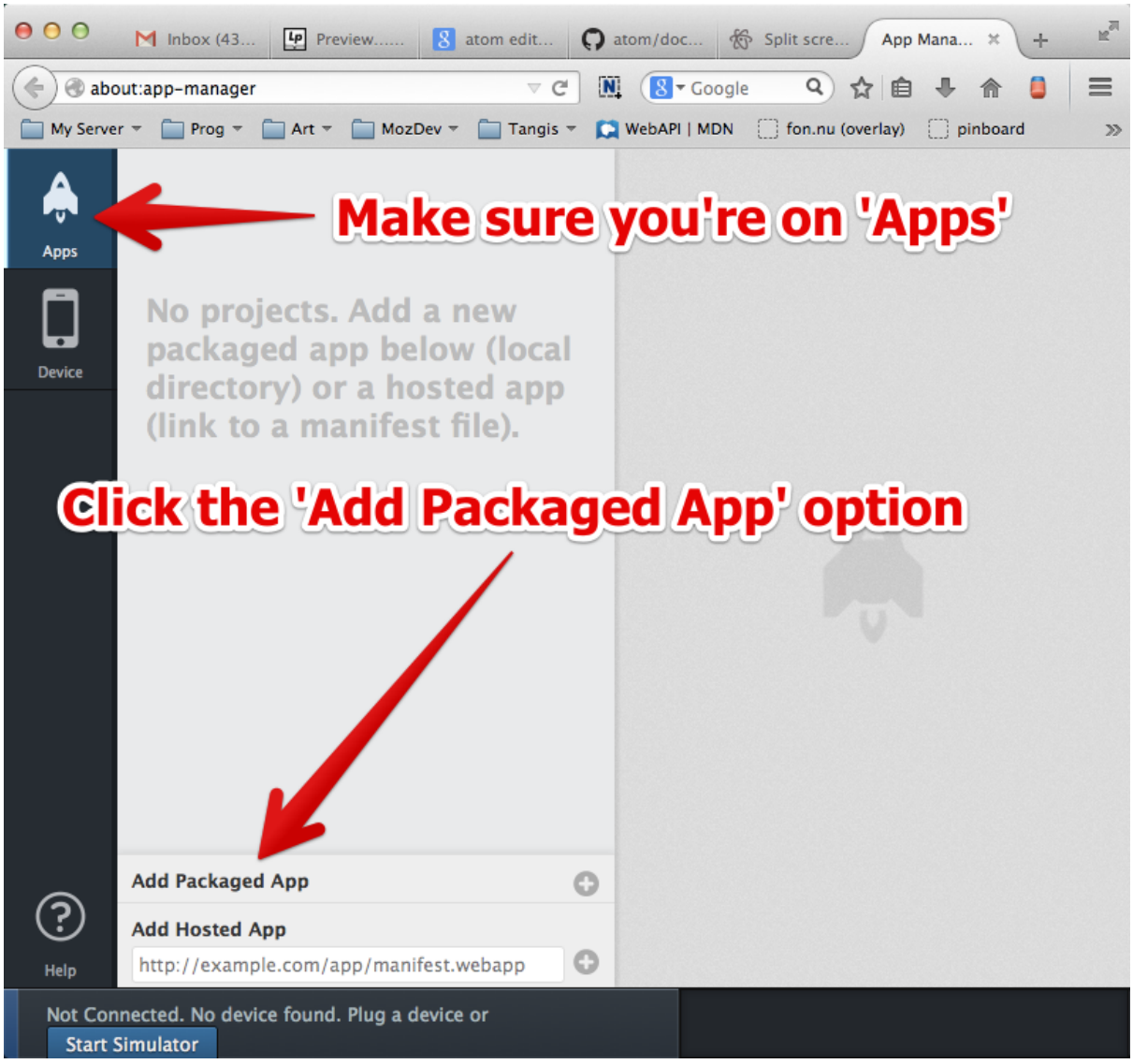
Adding Apps

You can add both hosted and packaged apps to the App Manager. Lets see how to add each type of app:

Adding packaged apps

To add a new packaged application click the **Add Packaged App** button on the **App Manager Dashboard** as shown in the screenshot below.

[37] https://github.com/firefoxos-gamedev/foxnoid-html5-game



Showing the *Add Packaged App* option that adds a packaged app to the App Manager

When you click on the button highlighted on the image, Firefox opens a file selection dialog. You should browse your hard drive and select the **folder that contains the manifest file** for the application that you want to add to the App Manager. If there are no issues with your manifest then your app will be addes to the list on screen.

If you're following with the Foxnoid sample game, then navigate to the folder that contains the game and select the file called **manifest.webapp**.

Adding hosted apps

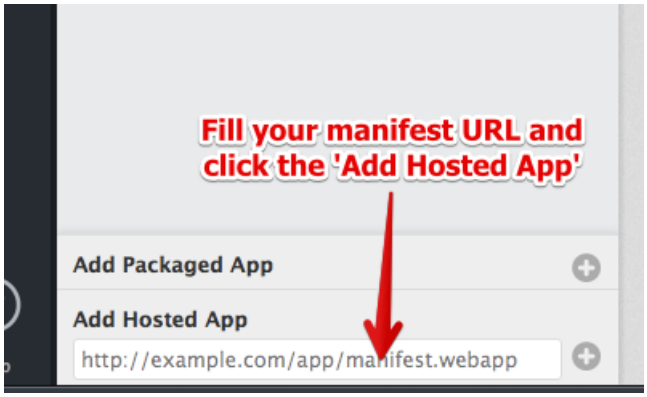
If you're building a hosted app then you should test it by using a web server. Do not try to use the method described above for hosted apps because you may miss some errors that will only happen

www.dbooks.org

on a hosted environment - such as serving the manifest with the wrong *MIME type*. Note that the simulator won't warn you about things like incorrect MIME types, but it's important to get such things right if you submit your app to the Mozilla Marketplace.

Most of the hosted apps are not applications built exclusively for Firefox OS but responsive design based websites that are able to adapt themselves to different devices and resolutions. These web apps usually have a complex backend that needs to be in-place for the application to work and that's why you need to test the app using a real web server running your backend stuff.

To run your app in the simulator, fill the URL of your application in the text entry box and click the **Add Hosted App** button.



Adding a hosted app to the App Manager

After clicking the button, the manifest is verified and if it is correct the application will be added and the App Manager.

Running your app

To run your application click the **Start Simulator** button. After that you will be able to select which of the installed Simulator you want to run. If there is no simulator installed then you will see an option to install one.

Once you have a Simulator running you can click the **Update** button inside your application listing and it will install the app on the running Simulator.

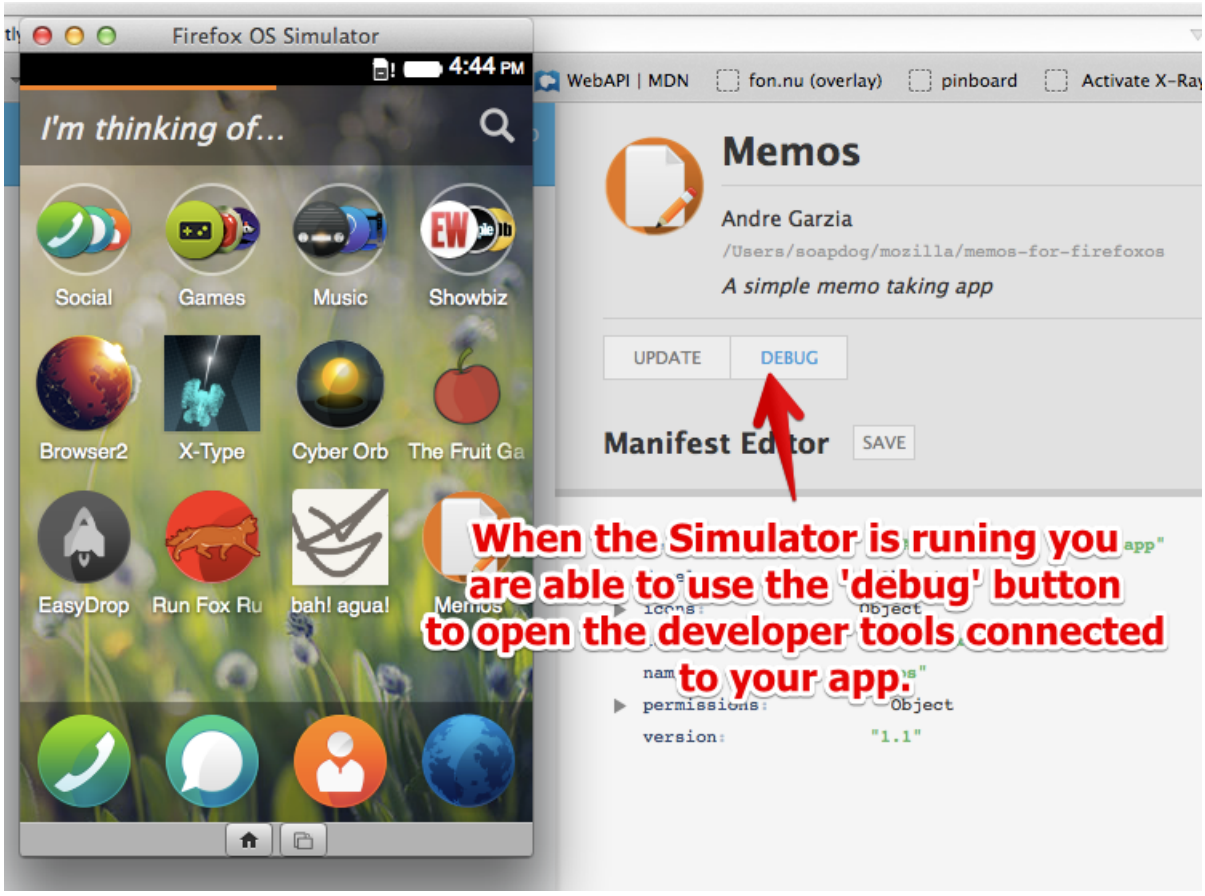
The application icon will appear at the home screen of the Simulator once the installation is complete. You can just click it to run.

Updating your app

Every time you change some of your files and want to test again on the Simulator you need to press the **Update** button to update the installation of your app on the running Simulator.

Debugging

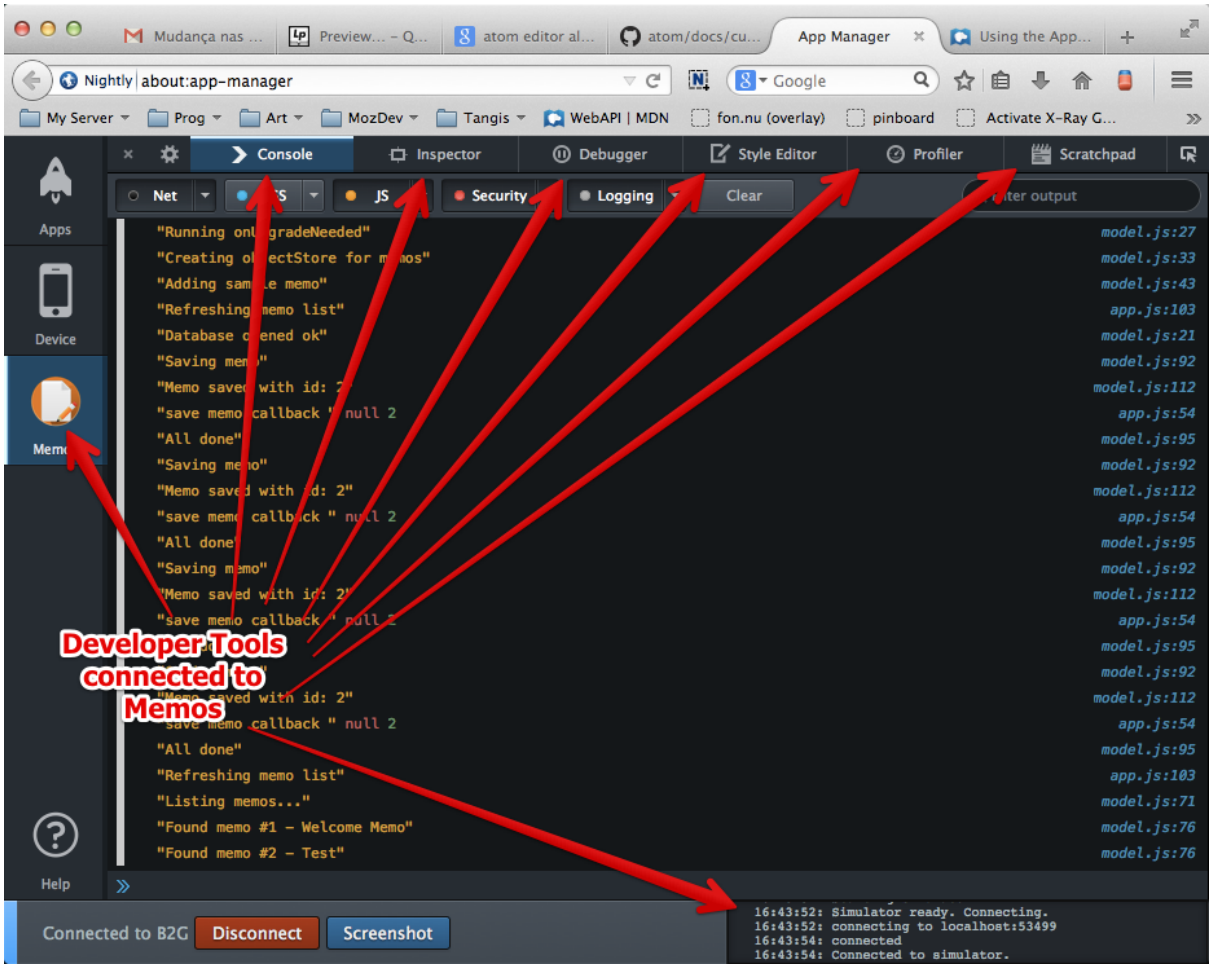
After the application is added to a running Simulator we're able to debug it by clicking the **Debug** button in the application listing. This will launch the your app on the running Simulator and open a **JavaScript Console** connected to your app.



What button to press

After pressing this button you will see a screen like this:

www.dbooks.org



Developer Tools connected to the app running on the simulator

With the tools connected to your app you can test your JavaScript, debug your DOM, edit styles, etc. Like those startup guys like to say: *pivot until your app is good.*

Once your app is running well on the simulator it's time to test on a real device.

Testing apps on a real device

Nothing replaces testing on a real device. On the simulator, you test things by using a mouse and clicking on a computer screen; while on a real device you use your fingers on a touchscreen and by using physical buttons. Its a very different user and development experience.

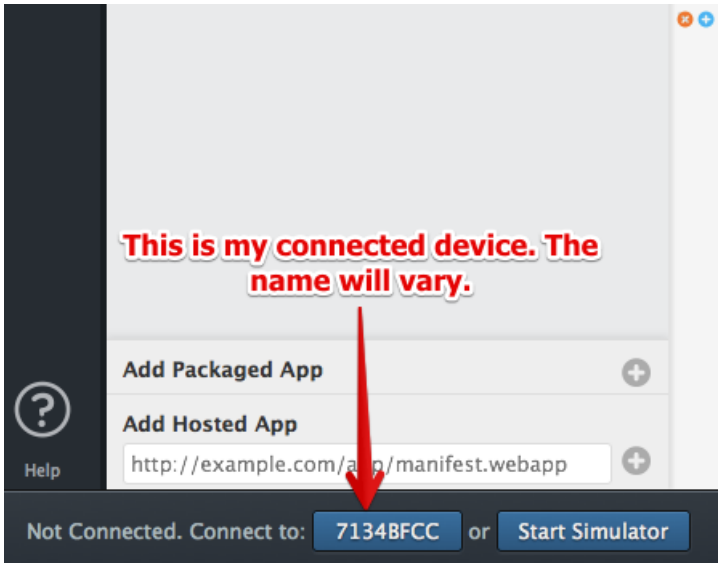
As an example why this type of testing matters, let me tell you a brief story: Some years ago, Raphael Eckhardt (the designer who created the cover of this book) and I were building a puzzle game not that dissimilar to Bejeweled. Our game involved dragging and dropping some pieces on a board and was working pretty well on the simulator.

When we then tested the game on an actual phone we realized our game components were not touch friendly at all. When placing a hand over the screen the board would vanish behind the hand. Even worst, the pieces the users were supposed to drag were smaller than the user's finger tip, so the user couldn't see what they were doing! In summary, our UX sucked very badly. That happened because we kept trying things only on the simulator using a mouse that had a tiny cursor. When we decided to try with our fatter-than-a-cursor fingers we realized that we need to rework our UI.

To avoid having a similarly frustrating experience, always test on a real device... or two, or more if you can get your hands on multiple devices. Test often with simple prototypes. Otherwise, you can waste valuable time and money having to recreate assets.

Connecting with a Firefox OS device

If you have a Firefox OS device (and have any needed drivers installed) then you can push apps directly from the App Manager to the device if the device is connected to your computer. When the App Manager detects that you plugged a Firefox OS phone, it will display the id of the phone on the bottom next to the **Start Simulator** button.



Device Connected!

If you press that button. The phone will ask for permission to establish a debugging connection with the App Manager, you need to grant that. Once that connection is made you will be able to use the **Update** and **Debug** buttons on your app listing to install and debug your application on the connected device just like you did with the running Simulator.

www.dbooks.org

Summary

In summary, the App Manager is fantastic. Its much better than the old Firefox OS 1.1 Simulator Extension since it has better developer tools and can run multiple Firefox OS versions. We can envision the App Manager getting better and better with its built-in manifest editor and more.

Besides feeling awesome and empowered, by this point in the book you hopefully have a good grasp of the workflow for building apps for Firefox OS.

In the next chapter we're going to talk about games! :D

About Games

There are lots of books about game development that doesn't bother to define what a game is because they all assume everyone already knows. If you stop to think about it, there is a great chance that you know what a game is but you can't really define it. The boundaries between activity, play and game are very fuzzy.

Lets think of an activity first. Lots of things are activities. Your job is an activity. Eating and breathing are activities. Some activities are more fun than others. Some people enjoy fishing but is fishing a play? is it a game?

What Are Games Anyway?

What differentiates a playful activity from a game? Some people say that *World of Warcraft* is not a game but a full time job, others are having terrible fun playing it. Defining what is a game is tricky.

Lets pick an activity first, something simple such as throwing rocks into the sea. For it to go from activity to playful activity means I need to be enjoying it. I like skipping stones, I think of it as a playful activity. Now what I would need to do to convert my playful activity of skipping stones into a game?

Games usually have rules. Rules are what make possible to go from a playful activity towards something that you can win or lose. Usually all games have rules and some way to win and or lose. If you think in these terms lots of things become games. If you enjoy your job and it has rules and goals then it can be approached as a game and the **gamification**[38] **of common activities are making a lot of people rich**.

Why are we doing this exercise anyway? Because if we can make the definition of "what is a game?" something less fuzzy then we can understand better our projects. Lots of games these days focus on trivia such as having better and better graphics but very few are exploring new mechanics and new ways to approach gaming.

There are lots of trusted formulas when it comes to game design and you can follow them and make something exceptional such as Angry Birds did. You can also travel thru paths that are not usually visited and create something original and wonderful. No matter what you do, your ability to deliver a good game can be broken if you begin by ignoring the game design process.

Thomas Was Alone[39] has simple graphics and simple mechanics but its story and method of delivery are so original and the theme reinforcement so well crafted that you can't help but be enthralled by the atmosphere of the game. While at a first impression you may think of it as just a blocky puzzle/platformer it is actually a wonderful story about friendship.

[38] http://en.wikipedia.org/wiki/Gamification

[39] http://www.mikebithellgames.com/thomaswasalone/

www.dbooks.org

What Makes a Good Game?

I can't give you a formula for a good game. Not only because I don't have it but because what is a good game for one person may be a bad game for another. There are games and niches for all types of players and what works in with some crowds may not work with others. For example permadeath[40] works pretty well for fans of rogue-like[41] games but not for casual players just looking for a past time on their commute.

Some marvelous games are really hard to define as games. Minecraft[42] is such example, think of it for a bit and question yourself: "What are the rules? How can I win? How can I lose? What is my objective?", answers will vary between players but they will all agree that Minecraft is mankind greatest invention since toilet paper and mountain dew.

One thing appears to be characteristic of great games which is that they provide players with meaningful experiences and memories. Some games are played and forgotten while others linger in your mind. A meaningful experience doesn't need to be something that make you question mankind position in the whole cosmos, it just need to mean something to someone. Bomberman is a great game in my opinion, specially when going multiplayer. I have very fond memories of trapping my friends between bombs while we were all eating pizzas and drinking sodas.

You can create good experiences thru many means. You can make a game with art so beautiful that will make unicorns cry. You can create game mechanics so cruel that will make players keep playing just to prove you wrong. Your story can transport people to other realms. Find your passion and use it.

You will make lots of crap games though… I bet you that the first notebooks Picasso had had terrible stuff in them (or maybe they were all masterpieces and just I suck at drawing). You learn from experience, to learn you must experiment. **To learn game design, you must design games!**

Not everyone can be Golan Trevise[43] and be always right. You will make mistakes but thru them you will grow. T-Rex volleyball was a bad design but T-Rex hot dog eating contest will be awesome.

Can't You Give Me a Straight Answer?

I can't but I know some people that can help you make better questions and come to better conclusions. I have a firm belief that the most important part of a game is its game design. I believe a game live and die by its mechanics and project. I don't really care about shiny graphics or new fads, all I care is about the experience. Some other developers have different opinions, you may have a different opinion but I believe you will agree that having a great design will always make a game better.

[40] a *feature* of some games where when the player dies they are really dead. Like its over. Done. Your fault! Losing is fun. #YOLO

[41] From wikipedia http://en.wikipedia.org/wiki/Roguelike: is a sub-genre of role-playing video games, characterized by procedural level generation, tile-based graphics and permanent death, and typically based on a high fantasy narrative setting.

[42] https://minecraft.net/

[43] http://mdfs.net/Docs/SciFi/Asimov/Timeline

If you want to dive deeper in the wonderful and addictive world of game design (not only video games but also tabletop, card, people running) check out these books:

- The Art of Game Design: A book of lenses[44]: In my personal opinion this is **the best book about game design ever created** and I don't mean this lightly. This book provides you with mental tools and workflows useful in any game related creative process. It teaches you how to organize yourself and think about your game from the perspective of many *lenses*. It is a wonderful experience. This is really recommended.
- Level Up!: The Guide to Great Video Game Design[45]: A great book focused on video games design. Very easy to understand and appreciate. Clear explanations and ideas.

Disclaimer: The links above are affiliated links. I get some bucks if you by the book thru these links and then I can use these bucks to create more open source stuff.

What Should I Do Then?!

First you should not worry. There will be enough time to create your masterpiece later. We're going to create a simple game with a proven mechanic along this book. The game will be instantly recognizable by most readers and easy to grasp for those that are new to it. We'll dive deeper into the game design of our game in the next chapter but for now lets agree to do one thing which I call the **Game Design Journal**.

Keeping a Game Design Journal

As we stated before: you must practice and by practice I don't mean to do as I do when I say that am a practicing athlete when I run just twice every six months (and usually from something or someone). The root of practice is doing something regularly, every six month is not regular enough for this type of activity.

Begin by selecting a **paper notebook that is easy to carry**. Paper trumps digital here because I allows you to go free form and unplugged. Paper notebooks don't run out of batteries, don't interrupt you with notifications and don't go crazy when they lose network connection and power during an update. Some people favor Moleskines but I am partial of Field Notes[46]. My tools of choice are the Field Notes Colors: "EXPEDITION" Edition[47] and the Field Notes Space Pens[48].

[44] http://www.amazon.com/gp/product/0123694965/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0123694965&linkCode=as2&tag=andregacom02-20&linkId=3A6K4FN3GZ3AVW3K

[45] http://www.amazon.com/gp/product/047068867X/ref=as_li_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=047068867X&linkCode=as2&tag=andregacom02-20&linkId=TBDQTMA7QTIX4JOD

[46] http://fieldnotesbrand.com/

[47] http://fieldnotesbrand.com/colors/expedition/

[48] http://fieldnotesbrand.com/spacepen/

www.dbooks.org

The idea is to carry your notebook with you and write down all your game design ideas. You may write a game story, some interesting mechanic, a summary of a game you want to make. Keeping it free-form will allow your more flexibility in your activity. As you make this a regular activity ideas will come more often and most important you will not forget them.

> Cool Tech: The Field Notes Expedition Edition is **waterproof** as shown in TEST 1: WATERPROOF[a]. Now you can take it to the shower and never ever lose a good idea again. How many genius ideas were lost after shower because you could not write them down… never again!
>
> [a] http://vimeo.com/55133541&autoplay=1

I tend to write down the main game idea and then do some really bad sketches. I often write down interesting mechanics. Whenever I see something cool that might be useful later I also write it down.

If you keep this going, you will see that your mind is an endless fountain of ideas. Separating the good ideas from the bad ones is beyond the scope of this book and left as an exercise to the reader (I hated when my math books said something similar…). Anyway, that was just a joke, you should pick some of ideas regardless if they appear to be good or bad and try to implement some proof-of-concept toys just to test them out with friends and random strangers (excuse me, do you have a minute to talk about game design principles?).

Like musicians go from making dogs cry in terror to making that single tear drop from your chin in that perfect concert so will you go from terrible designs to really good stuff. It will just take some time.

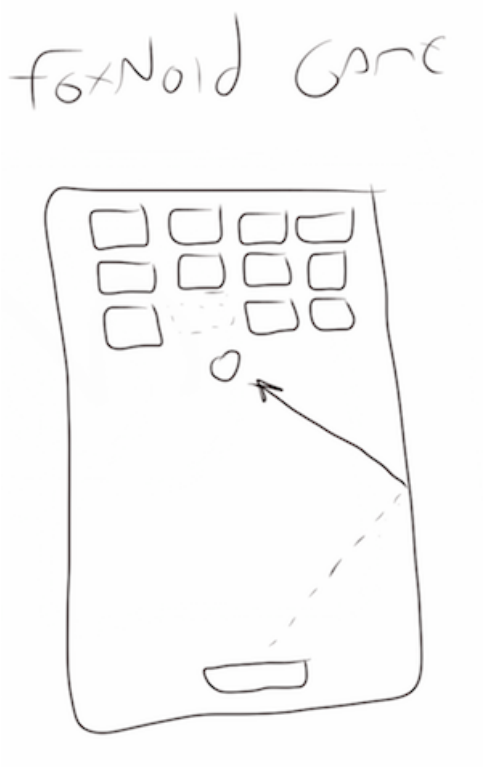
Since this is a small book and we don't have much time lets cheat and use a proven design instead of inventing things.

Summary

The most important part of this chapter are the book references and the game design journal exercise. Forget all the jokes and do yourself a favor by maintaing that journal and investing some good money on those books above. This book you are reading now is free (and open source) so use the money you saved to invest in the good education provided by them.

In the next chapter we'll begin the design and implementation of our game.

Our Game



Foxnoid Sketch

It all starts with a badly designed sketch…

Introducing Foxnoid

Our game will be the classic game where the player controls a paddle and use it to bounce balls at blocks that are floating on the top part of the screen. Once a block is hit by the ball it ceases to exist. When the player clears the screen of blocks he or she will advance to the next level and deal with a new configuration of blocks.

We are calling this game **Foxnoid**.

www.dbooks.org

A Good Introductory Game

I believe this is a great game to learn game design and implementation. The general mechanics are easy to understand and implement. Ever since the advent of Pong[49] we've had this compulsion for bouncing square balls with poorly designed paddles.

Foxnoid will allow us to present a complete game dealing with fast moving graphics, different levels, game states and more. Even though it is a simple game it will allows us to explore many of the common tasks that most game developers face on their own creations.

Room For Experimentation

Another positive point is that we have a lot of room for experimentation since the mechanics are well understood. We can experiment with many different concepts such as changing the art to change the atmosphere, changing the mechanics to make it more difficult or more fun and more.

You can go really devious with mechanics. Suppose you have blocks of two colors and the player had to clear one color before the other. This will make grown people cry. Or imagine a special color block that when hit spawns a new ball on the screen, it could get crowded in there.

As you move along this book and towards the completion of this game project you will probably have many ideas to improve it. Use your game design journal to keep track of them and try them out.

A Look At The Big Picture

Another important aspect on why I chose to implement this specific game besides my love of bouncing stuff is that this game is a **casual 2D arcade** game which is a category well suited for mobile platforms with entry level specs.

The market for mobile action, arcade and puzzle 2D games is the largest fraction of the mobile game market. 3D, simulations, shooters, RPGs they all come after that big chunk and they are way more complex to implement.

Firefox OS devices are mostly entry level devices targeted at emerging markets. This segment is where the 2D casual games shine. If this book was *Developing Games for Sony Playstation 4* we would probably opt for some 3D shiny thing but that is not the case.

Our Plan

The plan is to implement Foxnoid for Firefox OS. To do this we're going to adopt a game development framework called Phaser[50]. Many readers are now probably asking why am I adopting a framework

[49] http://en.wikipedia.org/wiki/Pong
[50] http://phaser.io

instead of working with raw Javascript and the answer to that is that this book is called *Developing Games for Firefox OS* and not *Developing Game Engines for Firefox OS*.

Imagine if you picked up a cooking book and it begun like: *"first find some land with good rainfall and seeds for all the vegetables you will need..."*. Teaching games development thru building game engines is like planning a road trip by first constructing your own car. In the end it might be a delightful adventure but its not what we're aiming for here.

If we opted to go without a framework or library I would need to teach you all about bliting images to the screen and converting keycodes into usable input information and more. We would lose too much time on trivia that has been solved before and once you could do all that you would still need to implement your own game.

Building game engines with good performance and usability is a very difficult job that requires specialist skills. You can create very good games with just raw Javascript calls but most game designers will not always start from a blank sheet. The ones that are not using some framework or library will develop their own and then stick to it.

I am not against developing your own framework but if we did this here then in the end you would be coding your game using my own framework that would not be as good as Phaser because those folks know much more than I do about game engines. Instead of reinventing the wheel and doing a poor job at it when compared it to the current wheels on the market I opted to choose a fine free and open source wheel called Phaser.

Be aware that this is not a book to teach you Phaser. We're just using Phaser to teach you game design concepts and implement a game project. You could easily adapt our content to other frameworks or write your own engine. We are building a shelf with a hammer called Phaser, like all shelfs once it is built no one asks the name of the hammer. Yet, Phaser is the best hammer we found.

Why Phaser

Before we start a flamewar regarding game development frameworks let me explain why I chose Phaser as our tool of choice here.

Batteries Included

This framework has an impressive array of features. Lots of other frameworks include similar features but I could not find a framework that was as easy to use and included the same set. Among all features some deserve mention such as: WebGL & canvas support, preloaders, physics, sprites, input handling, device scaling, tilemaps, mobile browser support, etc.

We'll not even touch half of the features that Phaser offers while building Foxnoid but if this is the first framework you're learning then by the end of the book you will have practical experience and will be able to use it in future games that are more complex and demand more features.

www.dbooks.org

Easy To Use

Yes there are frameworks that include even more batteries but I am yet to find something that is as simple as Phaser. Libraries tend to be simple and not have a lot of features or have too many features and be as complex as a moebius army knife.

Phaser is a good compromise between a very complete feature set and developer friendliness.

Great Community

Another important topic is that Phaser has a great community that hangs out at the HTML5 Game Devs Forum[51]. It has a specific forum at Phaser - HTML5 Game Devs Forum[52].

Its community is very helpful and friendly which is a plus when we're trying to use it to teach new developers.

Momentum

Phaser is Free and Open Source software with a great community and lots of features. It is battle tested and powering lots of games being used world wide. It has a great momentum behind it and its rising to become probably the most used and powerful HTML5 game development framework out there.

Not a Silver Bullet

Even so Phaser is not a silver bullet[53]. Some games are best served with other solutions or even by rolling your own. Lots of new developers tend to think that the tool will dictate what you're capable of doing and that simply by using a great tool you're all of a sudden capable of great things. I've seen people paint realistic landscape portraits with broken crayolas while I have expensive watercolors and can't draw a square with a gradient (lie: I can draw a gradient form that is square-ish).

Like the hammer analogy from above: the hammer isn't all that important. I have an electric screwdriver that has a LED light and is really handy. It helps me a lot but I could do the same job slower with a normal screwdriver and a flashlight. Tools are great when they help you but you can build great things regardless of them. **The most important tool of Game Design is the Game Designer**.

Phaser is indeed a great tool and it will help us a lot just don't think it is the only way forward or that you're doing something wrong if you like something else. Use Phaser for this book, see if you like it. Go fishing for other solutions afterwards and select something that feels right to you (... I bet you will keep using Phaser).

[51] http://www.html5gamedevs.com/

[52] http://www.html5gamedevs.com/forum/14-phaser/

[53] when Java developers face a problem and start throwing more and more XML at it hoping it will resolve itself.

Learning More About Phaser

For the curious we'll list some resources for Phaser below:

- Phaser Official Site[54].
- HTML5 Game Devs Forum[55].
- Phaser - HTML5 Game Devs Forum[56].
- Phaser Documentation[57].
- Phaser Online Samples[58].
- Phaser Source on GitHub[59].
- Discover Phaser Book[60]: A really good book focused on teaching the Phaser Framework by Thomas Palef who is a leading member of the Phaser community. He developed many games and his content is solid. There are package deals available for this book that includes more sample games. I really recommend it.

Summary

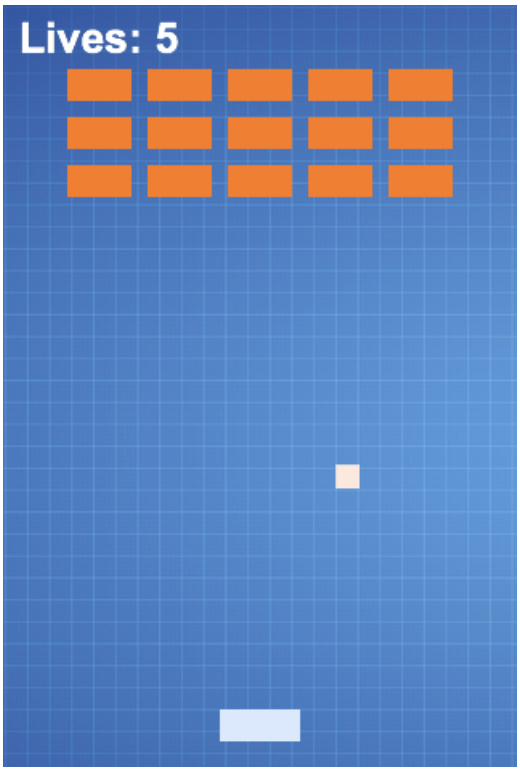
In this chapter you learned about what project we're going to build and the framework we'll use. We talked a bit about hammers as well.

With all that said, lets begin building our game.

[54] http://phaser.io
[55] http://www.html5gamedevs.com/
[56] http://www.html5gamedevs.com/forum/14-phaser/
[57] http://docs,phaser.io
[58] http://examples.phaser.io
[59] https://github.com/photonstorm/phaser
[60] http://www.discoverphaser.com/

www.dbooks.org

Setting Up Our Game



This Foxnoid looks much better than the sketch from previous chapter

Firefox OS applications are a collection of HTML, CSS and Javascript files. While most business applications deal with all these three technologies, mobile web based game development tend to use much more Javascript than HTML and CSS because they tend to be based in Canvas or WebGL due to performance reasons. To build games we're going to use the powerful Canvas API[61] to display 2D graphics and Touch Events API[62] for input. We'll just use some HTML and CSS boilerplate to initialize things and then all the magic happens in Javascriptland.

> **PSA**: In this book we're focusing on the task of building a game but if you'd like to dive deeper into Firefox OS app development then you should download the Free and Open Source book Quick Guide For Firefox OS Development[a] or just read it online, specially the chapter about basic

[61] https://developer.mozilla.org/en-US/docs/Web/HTML/Canvas
[62] https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Touch_events

concepts[b].

[a] https://leanpub.com/quickguidefirefoxosdevelopment/
[b] https://leanpub.com/quickguidefirefoxosdevelopment/read#concepts

Application Types

Firefox OS currently has two types of applications: hosted apps and packaged apps:

- **Hosted Apps**: Are hosted on a web server just like normal websites. This means that when the user launches a hosted app, its content is loaded from the remote server (or from the cache, if available).
- **Packaged Apps**: Are distributed as a zip file and copied to the device when installed. When the user launches a packaged app, its contents are loaded from the zip file instead of a remote server.

There are Pros and Cons to using each type. Games tend to use a lot of static resources such as image and sound files. If you use a hosted app then your user will need to download and cache all these files before playing. If instead you opt for a packaged app then all the static resources are bundled and copied to the device when the application is installed so there won't be a need to download anything after the initial installation.

In this book we'll use a packaged app approach. Its easier to develop since we don't have to deal with cache and cache invalidation and the user doesn't have to wait for things to load over the internet after the installation.

The Application Manifest

The manifest[63] is a JSON[64] file that describes aspects of an app. Usually this file is called **manifest.webapp** and lives next to your main HTML file that is usually called **index.html**.

[63] https://developer.mozilla.org/docs/Apps/Manifest
[64] http://json.org

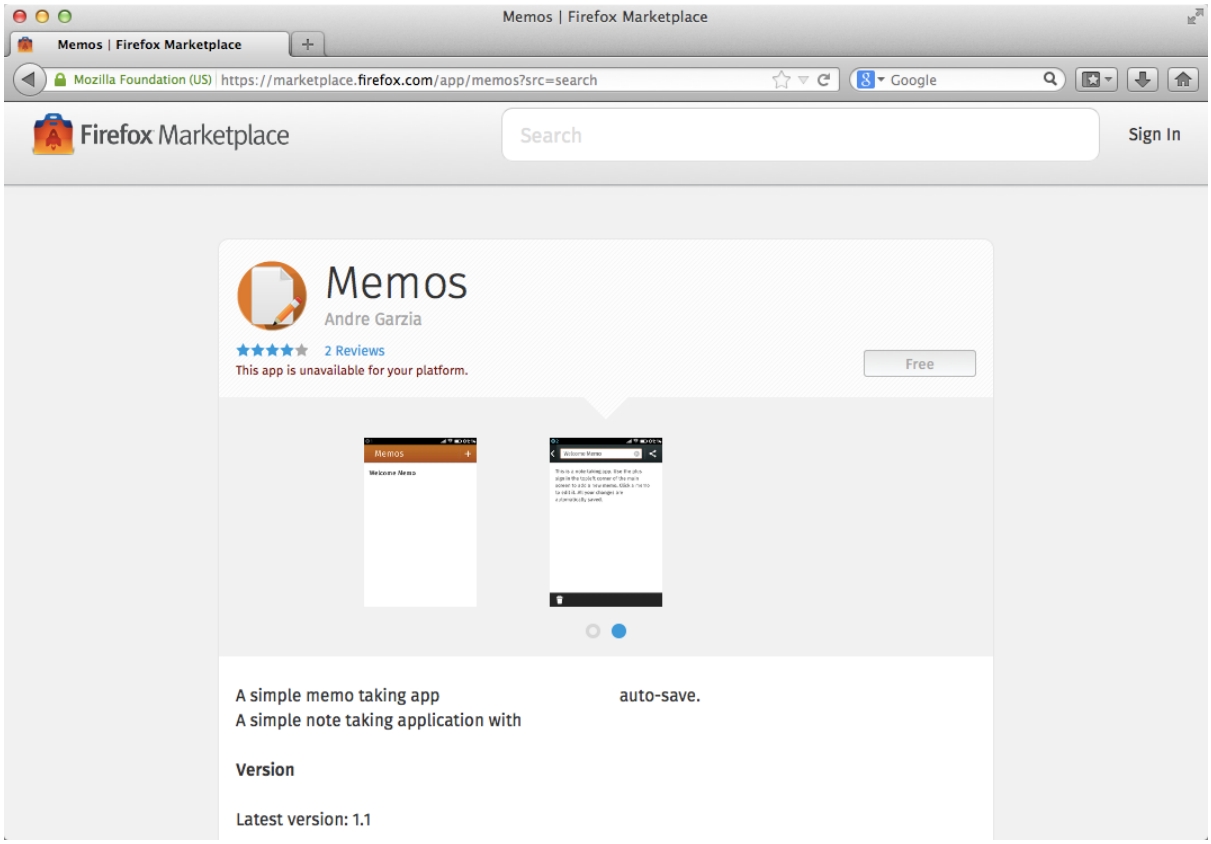
www.dbooks.org

Sample Manifest

```
{
  "name": "Memos",
  "version": "1.1",
  "description": "A simple memo taking app",
  "launch_path": "/index.html",
  "permissions": {
    "storage": {
      "description": "Required for storing and retrieving notes."
      }
  },
  "developer": {
    "name": "Andre Garzia",
    "url": "http://andregarzia.com"
  },
  "icons": {
    "60": "/style/icons/icon_60.png",
    "128": "/style/icons/icon_128.png"
  }
}
```

Above we can see the manifest for an application called memos[65]. Among other things it describes who created the application, which icons are used, what is the name of the app, what file is used to launch the app (in this case it is *index.html*), what hardware access permissions your app requires, etc. This file is used by Firefox OS to add the application to the device's home screen and by the Firefox Marketplace to display the application on the catalog as we can see in the image below.

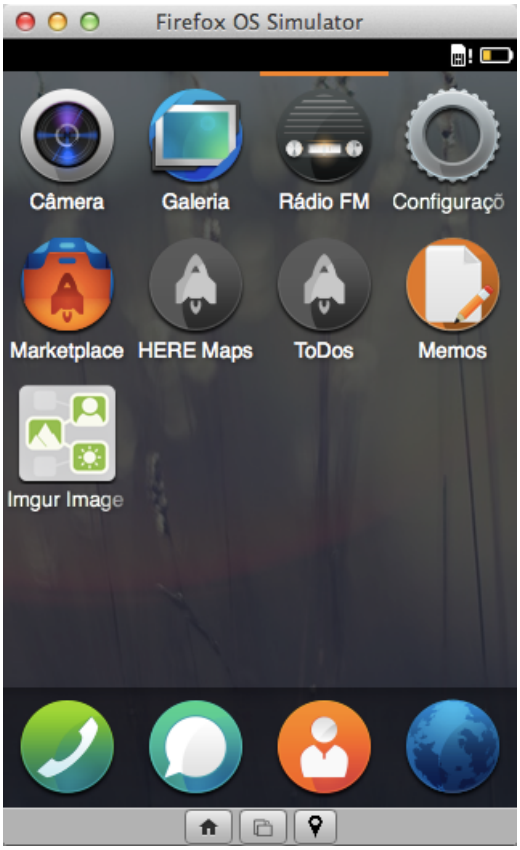
[65]This is a sample app for Firefox OS as seen on the Firefox Marketplace for which the source code is on GitHub.



Memos app shown at the Firefox Marketplace

Note how the information from the manifest is used by the system to add the app to the homescreen, as we can see on the following screenshot.

www.dbooks.org



Memos on the simulator

By gathering your HTML, CSS, JavaScript, and a manifest file you already have an application ready to run on Firefox OS. With this in our mind lets begin building our application.

There Is No Place Like Home

Lets give Foxnoid a nice home in your computer. As we know, we're going to build a collection of HTML, CSS, Javascript and static resource files. All these files will be organized inside a folder on your machine. **Your first task is to create a "foxnoid" folder in your machine, all we do will happen inside this folder**.

Below we'll add some boilerplate just to load up our app to the point where we can focus on the Javascript part of things. Most frameworks include some boring code that you need to use to initialize everything.

Through the remaining of this chapter we're going to write this initialization code. The game implementation itself is written in the following chapters.

Adding The Phaser Framework

Fetch the latest Phaser from phaser.min.js on GitHub[66] and place it in a **js folder inside foxnoid folder**.

This file contains the whole Phaser framework in minifed form. By including this Javascript file in the HTML we'll be able to use its features from our game.

The CSS

Since we'll use Canvas to draw our game display we're not going to rely on CSS to style things. Still we need to reset some values because different browsers assume certain values for margins and paddings. We want to zero these values so that we can fill the screen with the Canvas.

Create a css folder inside foxnoid folder and place a file called style.css inside it with the following content:

CSS Reset

```
1 html, body {
2     background: #000;
3     display: block;
4     margin: 0;
5     padding: 0;
6 }
```

This will set the default background color and reset the needed values.

The HTML

As explained above we're not going to do a lot of CSS and DOM. This game is just Canvas stuff so all our HTML needs is to include all the needed Javascript files and have a single `div` to hold our canvas.

Some games are better written with DOM others we'll do pretty fine with Canvas. You can think of a canvas as a Television where image will be displayed. Programming with canvas is closer to building games for old DOS and Gameboys then structuring hypertext with HTML.

We're going to change this HTML as we move thru the chapters but for now, create a file called **index.html** with the following content:

[66] https://github.com/photonstorm/phaser/releases/download/v2.0.5/phaser.min.js

www.dbooks.org

Minimal index.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>Foxnoid Game</title>
        <link rel="stylesheet" href="css/style.css" />
        <script defer src="js/phaser.min.js"></script>
    </head>
    <body>

        <div id="game"></div>

    </body>
</html>
```

This HTML does nothing besides giving us a place to hold our canvas and including Phaser.

> *Tip:* Using `defer` on the script tags will give us better performance since the JS files will only be executed after the DOM is ready.

Our Application Manifest

Create a file called **manifest.webapp** next to the **index.html** in the foxnoid folder containing the following:

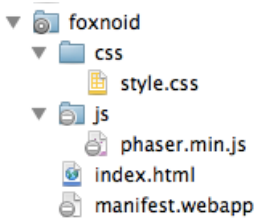
Minimal index.html

```
{
    "version": "0.1",
    "name": "Foxnoid",
    "description": "A game for Firefox OS",
    "launch_path": "/index.html",
    "developer": {
        "name": "Andre Alves Garzia",
        "url": "http://andregarzia.com"
    },
    "installs_allowed_from": ["*"]
}
```

This is the minimum data needed to start our game development process. Be aware that before submitting an app to the Firefox Marketplace[67] we'll need to add some things to this file like icons. We'll talk more about icons and static resources in the chapter about assets.

Summary

Your Foxnoid folder should look like this now:



Foxnoid folder

If it doesn't then you can always fetch the Foxnoid source from the Foxnoid repository on Github[68] just be aware that the version on github is a lot more complete than the one we have now.

Right now you should have a folder with some files that we'll serve as a starting point for our game development. You should also know a bit about application types and manifests.

On the next chapter we'll do some Javascripting to initialize our game.

[67] http://marketplace.firefox.com

[68] https://github.com/firefoxos-gamedev/foxnoid-html5-game

www.dbooks.org

Initializing Foxnoid

In this chapter we're going to learn how to start our game. This may sound simple and silly but this book takes small steps and tries to explain why we're doing things instead of just dumping a bunch of code that you'll use without understanding. I believe that programming is the act of organizing complexity. You can only organize something if you actually understand whats going on. Without understanding, organizing stuff becomes a game of blindfolded sokoban[69].

Before talking about game initialization we need to backpedal a bit and understand about game lifecycles and game states.

A Bit About Game Lifecycle

You can think about the game lifecycle from a birds eye view. In that case a game is something you launch, play and eventually close. So from this broad picture you can think of three stages in the game playing process. The game is launching, playing or closing. Each of these states can be further organized into smaller states.

If we were an eagle soaring high and could use the cheating eyes that nature gave us then we could **zoom** into these game states and see what they are composed of. Since we're not eagles I am going to use an analogy to explain this thing. Lets thing of a game as if it was a dinner out of home.

Game States

Game States as a restaurant analogy

First step is entering the restaurant, being seated and ordering food. This is similar to launching because when this process ends, we're in the state where food has been ordered. In terms of games we can think of this state as your application launching and loading its libraries.

Second step happens in the kitchen. The cook needs to have the ingredients ready so that your order can be prepared. Ingredients are assets that need to be there or the food can't be prepared correctly. Imagine a cook running to and from a super market while your order is prepared, I assure you it will be the last time you have dinner in that place.

In a game, assets are all the resources such as images, sounds and auxiliary files that need to be loaded before the game is playable. We've all seen web pages loading stuff while displaying stuff. You get a lot of relayouts, reflows and repaints operations. We learned not to care much about it but

[69] A transport puzzle game. Learn more about it at its wikipedia page.

imagine if this happened in a game of Super Mario and you're there trying to play while the level and enemies are still loading. That leads to a poor experience.

Third step is preparing your food. With the assets available, the cook can work the magic and deliver a nice plate. This is where your game uses all the needed resources to initialize the game itself. Following our Super Mario example, this is loading the maps and building the level.

Fourth step is having dinner! This is the fun step. (a.k.a.: our game playing step).

Fifth step, dinner is over and you're going home. This is when the user decides its time to go and closes your game.

Cute analogy but can we focus on game dev for a bit

The steps above are the broad steps needed by all games but depending on your needs your states will vary. A huge game such as any recent Zelda series title can't load all its assets at once because there is no such thing as infinite computer memory. When you have a game that is huge and has lots of assets its common to break the game playing state into a loading and playing state where you just load what it is needed to play that part of the game.

Game states are just a way for you to slice and dice your game into manageable bits so that you can focus on one task at a time. You can think of them as an assembly line preparing your game to be ready as the app switches between states. The most important aspect is that the more self contained each state is the easiest it is to maintain your code in the long run. You use game states for the same reason that your wardrobe has drawers, if it was just an empty box that you threw stuff in, then it would be really tricky to get dressed in time for the dinner out.

If you're building a game such as Super Mario you could have a different state for each different screen, so that the title, the options, the map, the level, the win and the game over screens could each be their own state and handle the loading of assets and work on their own. This way its easier to switch bits of the game.

Game states in Phaser

Phaser has game state features as we can see in the documentation for the State Class[70]. Each state can be self contained and handle its own asset loading, working and switching to a different state. Basically each state can be its own tiny assembly line.

A Phaser state is a Javascript object that has methods you can override to handle each of the steps mentioned in our restaurant analogy. For example if you want to load assets you will do it in the `preload()` function of that object. If you want to assemble the game level and build the screen you're displaying you will do it in the `create()` function. The food eating/game playing happens in the `update()` function and so on.

We're going to talk much more about game states as we build Foxnoid.

[70] http://docs.phaser.io/Phaser.State.html

www.dbooks.org

Foxnoid game states

Foxnoid is a game in the tradition of Arkanoid. Its a classic arcade game like Pacman, Pong, Tetris and others. Lets plan the states used in our game.

Foxnoid Game States

Name	Responsible for
Preload	Will load all the assets needed for the game.
Game	The actual game.
GameWin	When the user wins the game
GameOver	When the user loses it

As you can see we're going to have four states in Foxnoid. The first one will load our assets and then switch to the game. The user will play the game and depending on the outcome we'll switch to Game Win or Game Over state.

The remaining of this chapter we'll be about what should happen before we're able to switch to the **Preload** state. There is some initialization code that needs to be done prior to start loading assets.

What Should Our Initialization Handle?

Out initialization code should handle everything that is needed before we can start switching between states. It should load all libraries needed for our game to work (in our case just Phaser) and initialize them. The rest is up to each state.

Initializing Phaser

Lets create a file called **init.js** inside the **js** folder.

init.js: is responsible for initializing Phaser

```
1  var GameStates = {}; // <-- Object to hold all our game states.
2
3  document.addEventListener("DOMContentLoaded", function()  {
4
5      // Create your Phaser game and inject it into the game div.
6      // For more information regarding the Phaser Game object refer to:
7      // http://docs.phaser.io/Phaser.Game.html
8
9      // Portrait game orientation.
10
```

```
11      var width = 320;
12      var height = 480;
13
14      var game = new Phaser.Game(width, height, Phaser.CANVAS, "game");
15
16  });
```

On line 01 we create an empty object. This will be used to hold all our game states, this way instead of having four different globals around we're going to have only one.

> **Disclaimer**: Yes, I could use an immediate function and have no globals at all but this is a beginner oriented book and I don't want lose precious pages explaining about closures and nameless functions that invoke themselves.
>
> That being said, those that understand those features can clearly see how easy it would be to use them here. Just surround that init file and be happy. For those that have no clue what I am talking about but would like to know more then check out Appendix 2: Javascript Books for some quality JS reference material.

From line 03 to 17 we have the `DOMContentLoaded` event handler function. This will be executed once all the DOM is loaded. Its inside this handler that our game initialization happens. At this moment, initializing our game is basically loading Phaser and instantiating the the `game` variable. This game variable is the main object that is used to access the features from Phaser. You can learn more about it at the documentation for the Phaser Game Class[71].

We call `Phaser.Game()` on line 14 passing four parameters to it. The first two are the dimensions for the canvas used in our game. We can think of it as the size of our desired screen. Phaser can work with both Canvas and WebGL renderers but the later is not well supported on mobile devices so for casual 2D games I recommend sticking with Canvas. The fourth parameter is the id of the DOM element that will hold the canvas. Once that line is executed Phaser will initialize a canvas object inside the element specified.

Even though we created the **init.js** file, we haven't included it on our **index.html** file yet. Lets do it.

[71] http://docs.phaser.io/Phaser.Game.html

www.dbooks.org

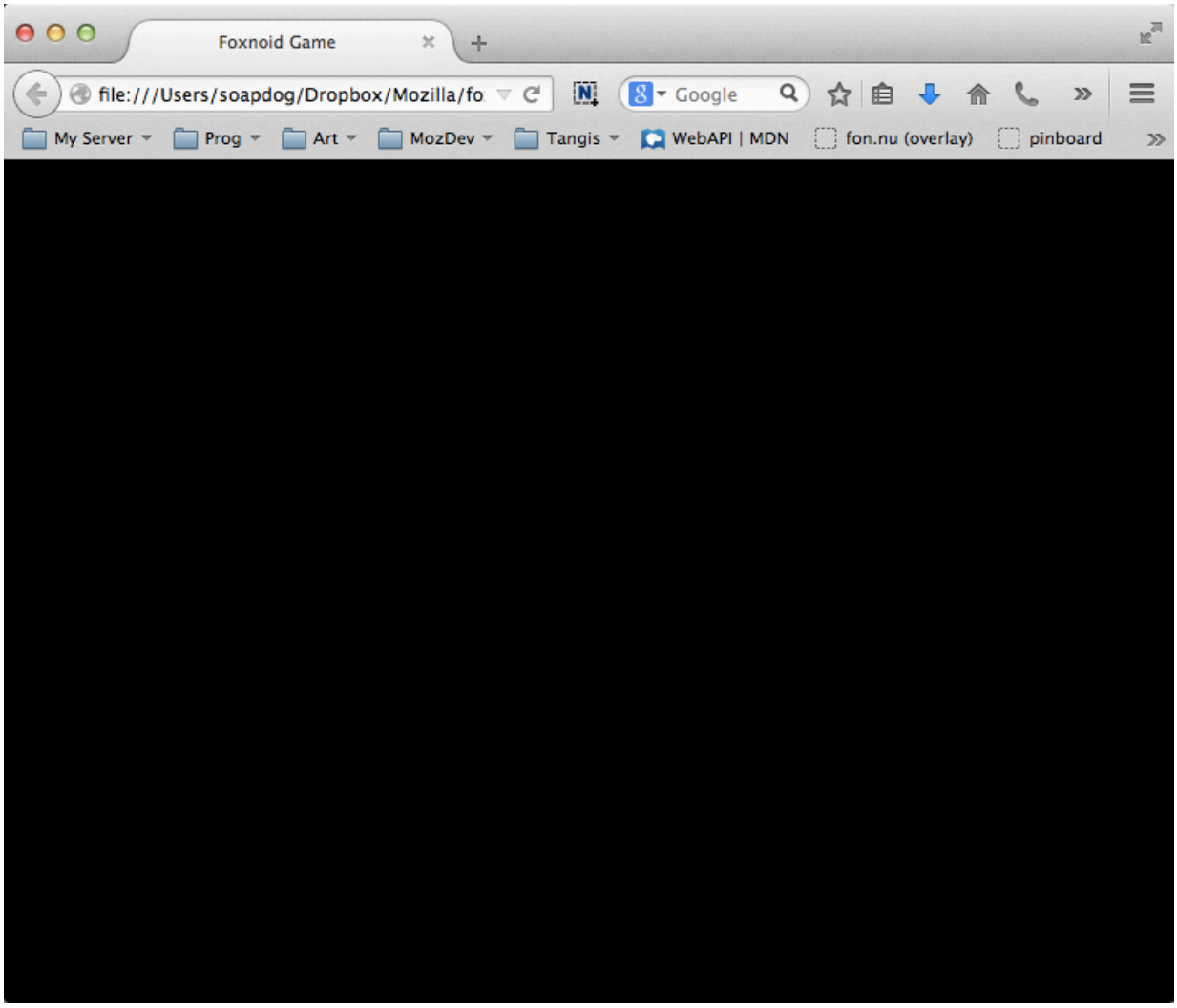
index.html: including our init.js file

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>Foxnoid Game</title>
        <link rel="stylesheet" href="css/style.css" />
        <script defer src="js/phaser.min.js"></script>
        <script src="js/init.js"></script>
    </head>
    <body>

        <div id="game"></div>

    </body>
</html>
```

If we open our **index.html** on our browser now, we'll see a black screen. Thats the canvas working but since it has nothing to do, it stays black.



Black screen means good

The above screenshot is all black because our canvas is black and our background color in the HTML is also black. Now after that initialization we can begin working on our game states.

Summary

In this chapter we explained about game states and how we can use them to organize our game into manageable bits by placing logical stuff together. We decided on building four states for the Foxnoid game and we've reached a point where we have a black screen.

On the next chapter we're going to build our **preload** game state and load our resources.

www.dbooks.org

Managing Assets

In the previous chapter we initialized Phaser and placed a canvas in our screen. In this chapter we're going to load the assets needed by our game.

Before we show the code to create the **Preload** game state we should talk a bit about assets. These days many of the mainstream game makers appear to rely only on having shiny graphics on the screen for the success of their game. Many users are attracted to these type of games where having next generation graphics trumps game mechanics. Building a game that relies only on its assets is a bad move in my humble opinion, also designing state of the art graphics requires a special type of team that many small developers don't have.

The focus of this book is building 2D casual games for smartphones. Normally the games that shine in the mobile market are the ones with the best mechanics (or the ones that are somewhat addictive) and tend not to rely too much on graphics since mobile phones vary so much in performance and power.

If you have a great idea, you can go on coding using placeholder graphics that are used only during development phase and later hire a proper designer to craft your real assets.

Just like tabletop game developers will craft prototypes out of cardboard and crayolas so can you build placeholder graphics and sounds using freely available tools. Any graphic software will work, it doesn't really matter if it is Adobe Photoshop or Microsoft Paint, as long as it can be used to build assets good enough for you to keep building your game we're all set.

Defining Atmosphere

The main aspect of assets, and by assets I mean graphic and sound files used in your game, is that they help set the atmosphere that your game will have. You can use bright primary colors in glossy shapes to build a candy inspired title for kids or you can use different grades of black, red and brown to create some minimalist horror genre. The mechanics in both titles could be the same and the game would differ only by its assets but *Candy Escape* would be a really different game than *Zombies: Run For Your Life*.

Game mechanics reinforce the atmosphere of the game but its visual aspects are key to setting tone and conveying emotions to your player. Some mechanics make sense for some genres and are not good for others. Creating a game for one genre and using assets from a different genre will work if you call yourself indie and there is even a chance that it will make you millionaire (a.k.a. Zombie: Brain Candy Escape will probably work).

An important step is to plan ahead. When defining your rules and planning your game you need to think what type of atmosphere you want and plan your assets accordingly.

A sometimes overlooked part of planning your assets is deciding on what form factors and platforms you will deliver your game.

Planing For Different Form Factors

The focus of this book is Firefox OS but I don't advise you to deliver exclusively to this platform. The cool thing about the Web is how cross-platform it is. The same game you have running on Firefox OS can run with minimal tweaking on Android, iOS, Facebook, OUYA and others.

If you plan from the start to deliver to multiple platforms then you can prepare your assets as needed. When we talk about delivering web stuff to multiple platforms we're usually dealing with different screen sizes, proportions and different input methods. We'll leave the input business to the input chapter and talk about resolution and screen sizes here.

Have you ever pick a thumbnail picture and enlarged it to fill a larger area? It looks pretty bad right? If you have a 16x16 PNG file and you resize it to 2500x1600 it will not look smooth. Its much easier to have a large resolution file and scale it down (which means discarding information) than it having a small file and scaling it up (which means creation of information or pulling rabbits out of theoretical hats).

If you're thinking about delivering your game to be playable on SmartTVs on full HD resolution then you need to have assets that look good in that resolution. You can scale them down to Firefox OS devices as needed and/or create simpler assets for such devices.

By planning ahead it becomes a lot easier to develop your game. If you start by developing a mobile game is a low resolution and then later decides to go to larger screens you may end up with a ugly game or the need to recreate your assets.

At least sound plays the same regardless of screen sizes...

> Anecdote: While I was developing Foxnoid, I decided to use some placeholder graphics until I could find a designer to build real graphics for me. In the end I liked my placeholder stuff so much that I decided to keep it. Anyway, I can keep it just because this is a toy game. I don't recommend that you stick with your placeholder graphics.

Planning Our Game Assets

If we consider the minimum number of assets that we need for our game to playable they would be the background, the paddle, the brick and the ball. With only those four graphics we could assemble our screen and focus on game play.

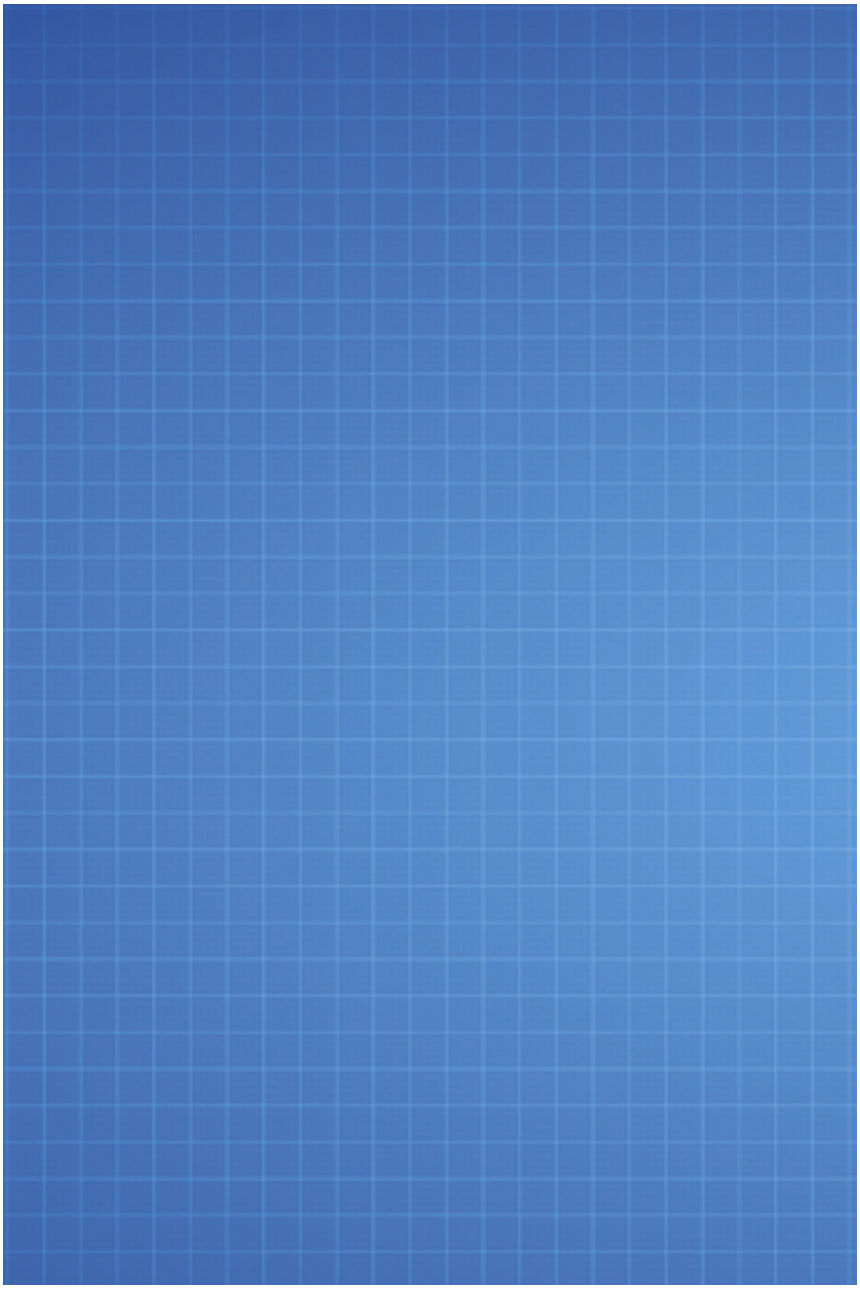
Later we'll build more graphics to improve our game, right now we'll just build these four so that we can keep going.

www.dbooks.org

The background

We all know that I can't draw. So I decided to use a freely available graphic as a placeholder. Mozilla Assets Repository[72] is a collection of assets to help people developing Mozilla related stuff. Firefox OS has some really nice assets that are community oriented including a blue squared technical paper background. I cut a part of that background to use in this game.

[72] https://assets.mozilla.org/



Foxnoid background

The block and the paddle

The blocks are the little bricks that are aligned in the top of the screen and we use for target practice with our paddle and ball. To create our brick I just selected a square in an appropriate size and used flood fill.

www.dbooks.org



The block

Be aware that the background image is a JPEG file because it is a file format that occupies less space but all the other assets are PNG files because this format has transparency support and thus blends better with the background asset.

If we think of the assets as a collection of images one on top of the other you will want to use a file format that allows you to layer stuff on top of each other with the transparent areas bleeding thru.

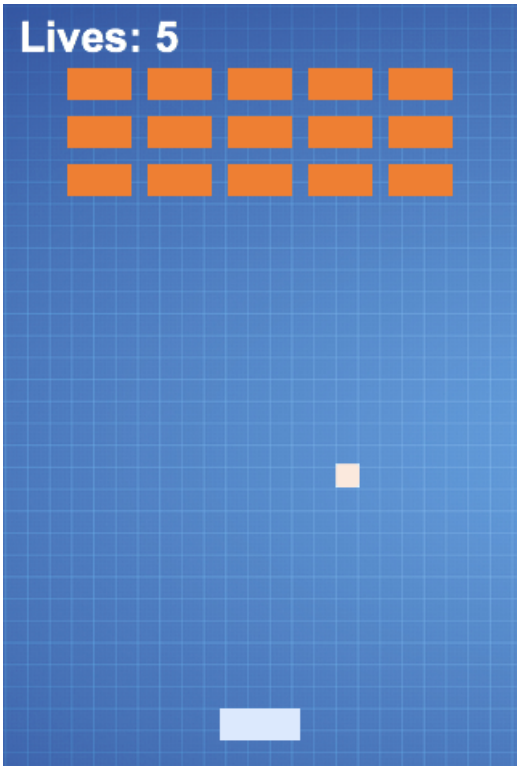
The paddle

The “ball”

Our paddle is created in the same manner as the block and so is the ball. The ball is not a ball, its actually a square. I decided to use a square as our “ball” because it looks more in tune with our graphics and because it makes it a lot easier to detect collisions. Its far easier to detect collisions between rectangles than between other polygons or balls.

How it looks when assembled

Our game looks like this



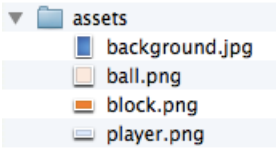
How our game looks

Building The Preload Game State

So we have our assets ready and now, we'll proceed to build the game state that will be responsible for loading them into memory so that they are available to other states in our game.

Preparing assets

To hold all our assets we created an **assets** folder in our **foxnoid** folder. We placed all our graphics files there and we're going to load them from our JS code below.



Our assets folder

You can fetch the assets from the Github Repository[73] and place them in the correct place.

[73] https://github.com/firefoxos-gamedev/foxnoid-html5-game

www.dbooks.org

Coding preload.js

Game states are just Javascript objects with special functions in them. These functions are called by Phaser during the game state lifecycle. Our preloader will load the graphic assets and then switch to the game state called **Game** when ready.

The function that Phaser calls when it is interested in loading stuff is the `preload()` function of the game state object. In this function we'll load our assets. After calling `preload()` it will call `create()` which is where the game state can initialize things. Since after loading the assets there is nothing that our Preloader state needs to do we'll just use the `create()` function call to switch to a different state.

Create a file called **preload.js** inside the **js** folder. Add the code below to that file.

preload.js: the game state to load our assets

```
GameStates.Preloader = {
    preload: function() {
        this.load.image('background', 'assets/background.jpg');
        this.load.image('player', 'assets/player.png');
        this.load.image('ball', 'assets/ball.png');
        this.load.image('block', 'assets/block.png');
    },
    create: function(){
        this.state.start('Game');
    }
};
```

The `preload()` function is where we load our graphics. We use `this.load.image()` calls to load each image passing to it two parameters:

- A key that will be used to refer to the image from now on.
- The path to the image file.

In the `create()` we just switch to the state named **Game** which we'll build in the next chapter.

Changing index.html

Now that we have **preload.js** ready we need to change our **index.html** file to include it.

index.html: now includes our preloader

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>Foxnoid Game</title>
        <link rel="stylesheet" href="css/style.css" />
        <script defer src="js/phaser.min.js"></script>
        <script src="js/init.js"></script>
        <script defer src="js/preload.js"></script>
    </head>
    <body>

        <div id="game"></div>

    </body>
</html>
```

Changing init.js

Remember our initialization file? Well we need to change it too so that we register the **Preloader game state** with it.

init.js: is responsible for initializing Phaser

```
var GameStates = {}; // <-- Object to hold all our game states.

document.addEventListener("DOMContentLoaded", function()  {

    // Portrait game orientation.

    var width = 320;
    var height = 480;

    var game = new Phaser.Game(width, height, Phaser.CANVAS, "game");

    // Add the Game States the game has.
    game.state.add('Preloader', GameStates.Preloader);

    // Now start the Preloader state.
    game.state.start('Preloader');
```

www.dbooks.org

```
17
18 });
```

The change is on line 13 where we use `game.state.add()` to add a game state to our game. The two parameters are:

- A key that we'll use to refer to the given state later.
- The object that holds our state.

Then on line 16 we launch the **Preloader** state. To learn more about the nuts and bolts of game states refer to the Phaser.State documentation[74].

This moves us one step further to having a complete game.

Summary

In this chapter we learn a bit more about assets and how they impact the atmosphere of our game and the experience the player has. We moved on to create out simple graphic assets and load them in a preload game state.

In the next chapter we're going to build our game loop and finally have a little ball bouncing on the screen.

[74] http://docs.phaser.io/Phaser.State.html

Gameloop

This is it! We've finally reached the point where by the end of the chapter we'll actually have stuff on the screen! Before we get too excited about it lets wind down and ponder a minute about…

What Is The Essential Part of a Game?

With our newly understood concepts about game loops and gaming in general we'll do an experiment and cut stuff from a game until we're left with the essence of a game.

Games these days have title screens, intro animations, tutorial levels, infinite option screens, leaderboards and many other things that if we cut out we're not left without a game. If you go back in time into the 90s, you could pick a handheld game not unlike Tetris that when powered on would drop you straight into the game. No fancy options, no tutorial levels, no intro stuff, just the actual game.

This game play part is what we're calling the game loop. Its the piece of your game that picks the input from the user, calculates what happens in the game world and displays it to the player. This continuous loop of picking input, simulating the world and displaying it is the heartbeat of a game. It is what makes it alive. We'll focus on this heartbeat and the other features will follow later.

In this chapter we're going to build part of this beating mechanism. We're going to focus on the simulating the world and displaying it and leave player input to its own chapter. At the end of this chapter, our game will be alive and moving.

Thinking About Our Game Loop

The game loop will be built as its own Game State. In the previous chapter we've built a state to load our assets so that they would be ready to be used by this state that we're going to build now.

Our main objective here is simulating and displaying the world. Lets break down our simulation part and take note on what do we need.

- **Ball moving**: The ball should keep moving.
- **Ball collision with the walls**: If we think the smartphone display has four walls then the ball should collide with them and bounce back so not to leave the screen.
- **Blocks positioning**: Our blocks should be positioned as a grid when the game start.
- **Ball collision with blocks**: The ball should collide with the blocks and once this happens the block should be destroyed.

www.dbooks.org

- **Ball collision with player**: The ball should bounce when it hits the player paddle.

All the stuff about positioning things should happen on the `create()` function of the state which is run once when the state is created. All the moving and collision things should happen on the `update()` function which is the one that is called repeatedly once a game state is active.

Create a file called **game.js** inside the **js folder** and lets begin coding.

Changing index.html

We need to change our **index.html** file to include **game.js**.

index.html: now includes our game state

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>Foxnoid Game</title>
        <link rel="stylesheet" href="css/style.css" />
        <script defer src="js/phaser.min.js"></script>
        <script src="js/init.js"></script>
        <script defer src="js/preload.js"></script>
        <script defer src="js/game.js"></script>
    </head>
    <body>

        <div id="game"></div>

    </body>
</html>
```

After including that file in the HTML we need to register the state in **init.js** as seen here:

init.js: with the new game state

```
var GameStates = {}; // <-- Object to hold all our game states.

document.addEventListener("DOMContentLoaded", function()  {

    // Portrait game orientation.

    var width = 320;
    var height = 480;

    var game = new Phaser.Game(width, height, Phaser.CANVAS, "game");

    // Add the Game States the game has.
    game.state.add('Preloader', GameStates.Preloader);
    game.state.add('Game', GameStates.Game);

    // Now start the Preloader state.
    game.state.start('Preloader');

});
```

Initializing Our World

Phaser Game States[75] as explained in the initialization chapter have different functions that we can implement to suite our game loop workflow. To initialize the world we'll implement the `create()` function that is run once when the state starts. We're not going to implement this initialization as a single continuous block of code because it will be tedious and harder to understand. Instead we're going to build tiny functions that will be called from `create()`.

game.js: is where the fun happens

```
GameStates.Game = {
    initWorld: function() {
        // Some constants
        this.playerSpeed = 250;
        this.ballSpeed = 220;
        this.blocksPerRow = 5;
        this.blockRows = 3;
        this.playerLives = 13;
```

[75] http://docs.phaser.io/Phaser.State.html

www.dbooks.org

```
9
10          // Add the background
11          this.add.sprite(0, 0, 'background');
12      },
13
14      create: function() {
15          this.initWorld();
16      }
17  };
```

Above we can see an example of how our coding works. We created a function called `initWorld()` and then we call it from `create()`. This `initWorld()` is responsible for setting some constants and adding the background image. These constants are used by the functions we'll call on `update()` to simulate the world.

From now on, instead of pasting the whole **game.js** file, I am going to place just the new function that you should add and in the end how the `create()` should look.

Adding the player

Remember that you need to place commas between the methods. Add the method below to the `GameStates.Game` object.

game.js: the addPlayer() function

```
1   addPlayer: function () {
2       // Add the player
3       this.player = this.add.sprite(160, 440, 'player');
4       this.physics.arcade.enable(this.player);
5       this.player.anchor.setTo(0.5, 0);
6       this.player.enableBody = true;
7       this.player.body.immovable = true;
8       this.player.body.collideWorldBounds = true;
9
10
11      // Add the display of player lives
12      this.livesDisplay = this.add.text(10, 8, "Lives: " + this.playerLives, {
13          fill: "white",
14          fontSize: 12
15      });
16  }
```

This function is used to add the player sprite to the world. Sprites are the little things that are moving and interacting with the world, all the rest is background and information display. You can learn more about sprites by checking the Phaser Sprite Class documentation[76].

We're creating a sprite using the image called **player** that was loaded in our **Preload** game state. This sprite is on a specific position (160x440) and we're storing it in the object property called **player**. If you're not confident about the usage of `this` inside an object then check out the MDN documentation about **this**[77]. Basically since the game state is an object and we're storing the player as a property of this object then the player is available to be used in the other functions.

After adding the sprite, we're enabling the physics system to act on the player sprite. When working with mobile devices you don't want to simulate physics on stuff you don't need and waste battery and CPU. Because of this, we need to tell the physics system about all sprites that it should care about.

Phaser comes with three different physics systems ranging from simple arcade like physics to full blown simulations similar to Box2D. Our game doesn't require that amount of simulation and should be just fine with arcade physics. Thats why we're using `this.physics.arcade` and not the other systems. You can check more about the arcade physics at the Phaser Arcade Physics documentation[78].

The next command change the anchor point of the player sprite. Normally sprite anchor coordinates start from the top left corner and its a lot easier to position stuff if we change the anchor to the middle of the sprite. This way if we want to center a sprite on the screen we can position it directly on the center coordinate instead of picking the center and subtracting it by half of the length of the sprite which would be the steps needed if we could not change the anchor point.

The next three statements are related to the physics system. For things to be able to interact with one another in terms of physics they need to have a body, things with no body (and thus no mass) can't interact. We're also setting the body to immovable because if we didn't then each time the ball hits the paddle the paddle would move down and we don't want that. The last statement is to make sure the paddle doesn't leave the screen.

After that we're setting a text display with the number of lives the player still has. The idea is that once the ball hits the bottom of the screen the player loses a life. Once they are all gone its game over.

Adding the ball

[76] http://docs.phaser.io/Phaser.Sprite.html

[77] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

[78] http://docs.phaser.io/Phaser.Physics.Arcade.html

www.dbooks.org

game.js: the addBall() function

```
addBall: function () {
    // Add ball
    this.ball = this.add.sprite(160, 240, 'ball');
    this.physics.arcade.enable(this.ball);
    this.ball.anchor.setTo(0.5, null);
    this.ball.enableBody = true;
    this.ball.body.bounce.setTo(1, 1);
    this.ball.body.velocity.x = this.ballSpeed;
    this.ball.body.velocity.y = this.ballSpeed;
    this.ball.body.collideWorldBounds = true;
}
```

The ball initialization code is very similar to the player code above, the main difference is that the ball is supposed to bounce in all directions. We also give it an initial velocity in both axis so that once the game starts in the `update()` function the ball will be moving.

You can see that we're using the constant `this.ballSpeed` that was set in `initWorld()`. By tweaking these constants you can experiment with the game. There are some hardcoded values in here that could all be abstracted to constants. This would make our life much easier. Maybe in the revision or final version of this text I will change this.

Adding the blocks

game.js: the addBlocks() function

```
addBlocks: function () {
    // Blocks
    this.blocks = this.game.add.group();
    for (var line = 0; line <= this.blockRows; line++) {
        for (var row = 0; row <= this.blocksPerRow; row++) {
            var posY = (line * 30) + 40;
            var posX = (row * 50) + 40;
            console.log("Adding block at: " + posX + "," + posY)
            var temp = this.add.sprite(posX, posY, 'block');
            this.physics.arcade.enable(temp);
            temp.enableBody = true;
            temp.body.immovable = true;

            this.blocks.add(temp);
        }
```

```
16      }
17  }
```

The blocks are a bit different because instead of making each block an independent sprite we're using a group. Groups are useful when you have many of the same entity in a game such as bullets. It will be a lot easier to check collisions with a group because we can do a single check against the group instead of checking each sprite individually.

First we create a new group and store it in a property called blocks. You can learn more about groups at the Phaser Group documentation[79]. We use a nested loop to build our rows and columns (a.k.a. our grid of blocks). In this loop we calculate the position of the given block and create a temporary sprite to hold it. We then add this sprite to the group after setting the needed physics stuff on it.

The create() function

After creating the auxiliary functions above, our create function becomes:

game.js: the create() function

```
create: function() {
    this.initWorld();
    this.addPlayer();
    this.addBall();
    this.addBlocks();
}
```

Easy to follow right? First we initialize the world by setting the constants and adding the background. Then we add the player and the ball and finally the blocks.

With that done, the initialization is ready. If we run our game now we'd see a static screen with everything into the correct place but nothing happening. Our next step is to implement the `update()` function to make things move. We're going to do this in the same way we did the `create()` function which is by doing little functions and calling them from `update()`.

Simulating The World

This is the part where we code the simulation part. In this part we need to take care of moving the ball and colliding with stuff. One thing is that thanks to the physics system we don't need to compute the ball position. In the `addBall()` function we add its initial velocity and bounciness so that in each `update()` call the physics system will calculate its new position and collision with the world bounds on its own, we don't need to do anything. The ball collisions with the blocks and the player need to be implemented by us though.

[79] http://docs.phaser.io/Phaser.Group.html

www.dbooks.org

Colliding with blocks

We're going to use two functions to check collisions with blocks. One is the check itself that will be called from `update()`. The other is a callback that will be called by the physics system if the collision is true. First lets build the check.

game.js: the checkHitWithBlocks() function

```
checkHitWithBlocks: function () {
    this.game.physics.arcade.collide(this.ball, this.blocks, this.ballCollidesWi\
thBlock);
}
```

Colliding stuff is very easy using the Arcade Physics engine in Phaser. You just call `this.game.physics.arcade.coll` passing the sprite you want to check as the first parameter, the what should the sprite be colliding with which can be a group or a sprite, in this case we're checking against all members of the blocks group and the third parameter is the callback that will be triggered if there is a collision.

game.js: the checkHitWithBlocks() function

```
checkHitWithBlocks: function () {
    this.game.physics.arcade.collide(this.ball, this.blocks, this.ballCollidesWi\
thBlock);
}
```

Below we implement the callback.

game.js: the ballCollidesWithBlock() function

```
ballCollidesWithBlock: function(sprite, block) {
    console.log("Collided with block!");
    block.kill();
}
```

The callback receives two parameters which are the sprites that collided. The first one will be our player sprite and the second is the block the player hit. All we do is `kill()` the sprite. There are two functions used to remove a sprite from the game, they are `kill()` and `destroy()`, they have different mechanics. The kill one will really destroy and remove the sprite from the game. The destroy one will remove it from the game but keep it in memory to be reused which is useful when you're also using a sprite pool which will allow you to have many sprites being created and destroyed without performance penalty. This is important for shooter games for example. In our case we don't need this stuff.

Colliding with the player

The collision with the player is a bit different because we don't need a callback since the ball will just bounce away after that collision. There is no consequence from them both colliding.

game.js: the ballCollidesWithBlock() function

```
checkHitWithPlayer: function () {
    this.game.physics.arcade.collide(this.ball, this.player);
}
```

Colliding with the ground

Hitting the ground should cause the player to lose a life and the ball to reset to its original position. We're not going to use the Physics system for this because since we set the `collideWorldBounds = true` the ball automatically bounces on all four walls. We're going to check if the position of the ball is below the player paddle, if it is then the player loses.

First lets create the `resetBall()` routine.

game.js: the resetBall() function

```
resetBall: function() {
    this.ball.reset(160, 240);
    this.ball.body.velocity.x = this.ballSpeed;
    this.ball.body.velocity.y = this.ballSpeed;
}
```

Now lets implement our player losing check:

game.js: the ballCollidesWithBlock() function

```
ballCollidesWithGround: function() {
    if (this.ball.y >= 470) {
        this.playerLives -= 1;
        this.resetBall();
    }

    /*
     Update player life display
     */

    this.livesDisplay.setText("Lives: " + this.playerLives);

```

www.dbooks.org

```
13      if (this.playerLives === 0) {
14          // There will be a game over state here soon...
15      }
16
17  }
```

So if the ball is below the player paddle which is at 440px then we subtract a life from the player and reset the ball to the original position.

The update() function

The `update()` then becomes very easy to follow. At each loop it checks the necessary collisions and the movement physics are handled by Phaser Arcade library.

game.js: the ballCollidesWithBlock() function

```
1  update: function() {
2      this.checkHitWithBlocks();
3      this.checkHitWithPlayer();
4      this.ballCollidesWithGround();
5  }
```

> **Notice:** From this chapter onwards I am placing the final code for the given chapter at an appendix at the end of the book. This will make it easier to check for mistakes and have a clear picture of what we've accomplished.
>
> The code for this chapter is on Appendix 5: Gameloop Chapter Code and also in its own folder with the code package that is bundled with the book.

What about controlling the player?

We're devoting the next chapter to the business of game controls. This was done because this chapter was too long and there is too much to talk about.

Summary

If we open our **index.html** on our web browser now we're going to see the game working and the player loosing lives since you can't control it yet. Lets move along to the next chapter to make this game playable!

Handling Input

In this chapter we're going to make our game playable by adding routines to control the player paddle. Before showing the code to do it we should stop and think a bit about video game controls.

The Action That Drives The Game

Player controls can make or break a game. You should pay special consideration to how you're implementing this in your game and should experiment with different schemes before settling on a solution. The best thing is to prototype your proposals and test them with people.

One common practice is to devote an area of the screen to a virtual gamepad like interface. That type of solution uses a good part of the screen and unlike its real world counterparts (a.k.a. console controllers) virtual gamepads provide really bad tactile feedback.



Virtual gamepad

www.dbooks.org

Another common option is a gamepad like overlay that is placed on top of the game screen with usually almost transparent interface as can be seen below:



Gamepad overlay

In both cases you have specific areas of the screen that need to be hit to control the player actions. In some cases these controls are good. In the examples above one is an popular video game emulator and has no other option besides simulating the real video game controller and the other is a fighting game[80], a genre that involves a lot of quicktime actions and button mashing.

But imagine just for a bit if the popular hit **Angry Birds** used a control scheme such as the ones above. Would you still play it? Would the mechanic of sling shooting poor avians be as rewarding using a virtual gamepad as it is with dragging and releasing that rubber band? It is my firm belief that the more transparent/organic mobile controls are the best it is. Mobile devices have a huge touch screen that can all be used for input. Instead of bolting traditional control schemes into a mobile device we can think of clever usages of the touch screen.

Below we can see a screen shot of the game Craigen[81] which is a kind of infinite runner where you need to jump obstacles and hit monsters.

[80] https://play.google.com/store/apps/details?id=com.jiinfeng3d.bfrdemo

[81] https://marketplace.firefox.com/app/craigen



Craigen screen, jump away and slay monster

This is the type of game that would generally be coded using virtual gamepads or overlays but the author had a better idea as we can see here:

www.dbooks.org



Craigen controls explained

He simply divided the screen in two halves and touching one side means jump and the other means hit. You don't need to touch a specific point, just anywhere on one half or the other. This leads to a very enjoyable experience and happier users.

Touch Input Considerations

When designing games for mobile devices with touch screens you should take really good care to think an unobtrusive way to control your game. Pick top mobile games to see how they are solving this task and plan your game accordingly. Some important things you should consider:

The player hand/finger will move over the screen occasionally hiding part of it from view. Try to keep your interactions closer to the border of the screen. If something happens in the middle regions make it so that the player can touch and remove the finger from the region quickly to avoid blind spots.

Beware of designing small items for the player to touch. The finger tip is not very precise. If you're placing items for the player to tap on the screen try not to place them too close to one another. Just remember how many times you typed the wrong letter in the virtual keyboard of your phone. For example, if you're building a game like Minesweeper[82] try to use a large grid to avoid wrong selections.

[82] http://en.wikipedia.org/wiki/Minesweeper_%28video_game%29

Keyboard Input

Keyboard input is quite straightforward. One thing we should mention is that even when coding a mobile game you should consider having standard keyboard input as well. One advantage is that is helps while development and will allow you to deploy on both mobile devices and desktop-class devices.

Gamepad Input

Gamepads are becoming more popular with casual gamers on desktop computers and mobile devices. There are many mobile device compatible controllers out there and offering support for such controllers on your app is a way to get more players. There is an HTML5 Gamepad API[83] that you can use in your own games.

The main thing about gamepad controllers is that sometimes they are really different from one another regarding button placement. If you're supporting gamepads, you should offer a screen for the player to alter controller configuration such as what button is pressed to fire or jump. As an anecdote let me tell you that I have a cheap Chinese gamepad that was soldered wrong (I believe it was a mistake) and has the directional axis inverted. I can't use that controller at all with games that do not allow me to customize the controls due to it being kinda upside down internally.

Controlling Our Game

For the sake of brevity we're only going to implement touch and keyboard input. Phaser has support for gamepads out of the box but this is out of the scope of the simple game we're building because I don't want to implement the option screen at the moment. If you want to know more about gamepad APIs in Phaser refer to the Gamepad documentation[84].

What are the needs of Foxnoid regarding player control? Basically the player controls a paddle which moves left or right and thats it. Like that Craigen game we're going to divide the screen in two halves and touching any half will cause the paddle to start moving in that direction. We're going to add support for keyboard input with the arrow keys.

Handling Touch

As explained in the Gameloop chapter we should acquire player input at every loop. This means that we're going to create one extra function in our *game.js* and call it from `update()`.

[83] https://developer.mozilla.org/en-US/docs/Web/Guide/API/Gamepad
[84] http://docs.phaser.io/Phaser.Gamepad.html

www.dbooks.org

game.js: the handleTouchInput() function

```
handleTouchInput: function () {
    if (this.input.pointer1.isDown) {
        if (this.input.pointer1.worldX > 160) {
            this.player.body.velocity.x = this.playerSpeed;
        }

        if (this.input.pointer1.worldX <= 160) {
            this.player.body.velocity.x = -1 * this.playerSpeed;
        }
    }
}
```

To handle touch input we're using the Phaser Input routines. To learn more about them refer to Phaser.Input documentation[85].

Basically by default Phaser initialize with support for two fingers input, first finger will be bond to `this.input.pointer1` and if a second finger touch the screen at the same time it will be bond to `this.input.pointer2`. If we wanted extra touch inputs we could create them. In our case we just need a single input touching one side of the screen or the other. Thats why we're just checking `this.input.pointer1`.

The properties `this.input.pointer1.worldX` give us the coordinate in the X axis where the touch happened. Since our game world is 320 pixels wide it is very easy to figure out in which side the player touched. Depending on the side we apply a **velocity to the player paddle in that same direction**. The value of that velocity is in the `this.playerSpeed` constant that is set in the `this.initWorld()` function.

The Arcade Physics system used by Phaser will calculate the paddle position based on that velocity on its own, thats the beauty of using proven frameworks, no more fiddling with physics formulas.

Handling Keyboard

The keyboard code is very similar to the one above in the sense that we just check which arrow key the player is pressing and then apply a velocity to the paddle.

Phaser has a handy method for us to call that setups an default keyboard handler using the arrow keys. We can use that instead of registering each key we want to listen. This could all be done with normal HTML event listeners but since we're using Phaser here we can use its own functions because they tie up with the rest of the `update()` loop nicely.

[85] http://docs.phaser.io/Phaser.Input.html

game.js: the handleKeyboardInput() function

```
handleKeyboardInput: function () {
    if (this.cursors.right.isDown) {
        this.player.body.velocity.x = this.playerSpeed;
    }

    if (this.cursors.left.isDown) {
        this.player.body.velocity.x = -1 * this.playerSpeed;
    }
}
```

To use this code we must add a little bit to the `initWorld()` function. We need to tell Phaser to setup the default arrow keys keyboard manager for us.

game.js: the new initWorld() function

```
initWorld: function() {
    // Some constants
    this.playerSpeed = 250;
    this.ballSpeed = 220;
    this.blocksPerRow = 5;
    this.blockRows = 4;
    this.playerLives = 13;

    // Add the background
    this.add.sprite(0, 0, 'background');

    // Add keyboard input.
    this.cursors = this.input.keyboard.createCursorKeys();
}
```

With that call at the end of `initWorld()` we'll be able to check `this.cursors` in the `update()` function. To learn more about keyboard input check out Phaser.Keyboard documentation[86]. Without that change to `initWorld()` the `this.cursors` would be undefined.

The update() function

This is how the `update()` function looks with the new input routines.

[86] http://docs.phaser.io/Phaser.Keyboard.html

www.dbooks.org

game.js: the new update() function

```
update: function() {
   this.handleTouchInput();
   this.handleKeyboardInput();
   this.checkHitWithBlocks();
   this.checkHitWithPlayer();
   this.ballCollidesWithGround();
}
```

Now our game is playable!

Summary

In this chapter you've learned about input handling and why you should consider your player control scheme carefully before building your game. We've added both keyboard and touch input to Foxnoid. The game is now playable but still we don't have a win and lose scenarios.

The full source for the **game.js** file is at Appendix 6: Input Chapter Code

In the next chapter we're going to add a game win and a game over condition to Foxnoid.

Winning or Losing

Foxnoid is now a playable game but you can't win it (or lose it). In this chapter we're going to solve that by adding both the game win and game over mechanics.

We're going to use two new game states one for each type of ending. Once each situation happens we're going to switch to the needed state. Game states are useful for many things but one special use that they excel at is when the nature of what is happening on your game changes. In the game over situation the player was playing the game but lost it and is no longer able to play, this fundamental difference from the previous situation is easily crafted with game state switches.

Winning and losing are important aspects of games. You can create games with just one of these items. A game with no winning scenario is a game where winning is basically seeing how long you can last before losing. Games without a losing scenario option are common these days, specially on mobile. Games where you can play forever until you win or even in some cases pay-to-win games where you can spend real money to advance a level or buy upgrades.

Flappy Bird is a game without a win scenario (or at least I don't know about it). Basically these type of games work really well if they are hard and use leaderboards where players can compete with each other to have the best score.

Angry Birds is the kind of game where you can't really lose, you just keep stuck on a level until you pass it. That behavior is so common with mobile arcade/puzzle games that is basically the standard thing now.

In Foxnoid we're going to implement both a game win and a game over scenario.

Game Over Scenario

Game over will happen once the player runs out of lives. His initial lives number is set by the `initWorld()` function and updated if needed by the `ballCollidesWithGround()` function. What we're going to do there is check if the player has zero lives and switch to a new game state called **GameOver** if needed. Lets alter that function:

www.dbooks.org

game.js: the ballCollidesWithBlock() function with Game Over switch

```
ballCollidesWithGround: function() {
    if (this.ball.y >= 470) {
        this.playerLives -= 1;
        this.resetBall();
    }

    /*
     Update player life display
     */

    this.livesDisplay.setText("Lives: " + this.playerLives);

    if (this.playerLives === 0) {
        this.state.start("GameOver");
    }

}
```

The important part is `this.state.start("GameOver")` which switches to the new state. Our new state will look like this during the game:



Foxnoid game over screen

We're using the same background as the one in the game with two images for the text parts. We need to add these images to the **Preload** state, so edit the **preload.js** like this:

preload.js: the game state to load our assets

```
GameStates.Preloader = {
    preload: function() {
        this.load.image('background', 'assets/background.jpg');
        this.load.image('player', 'assets/player.png');
        this.load.image('ball', 'assets/ball.png');
        this.load.image('block', 'assets/block.png');
        this.load.image('gameover', 'assets/gameover.png');
        this.load.image('taptoplay', 'assets/taptoplay.png');
    },
    create: function(){
        this.state.start('Game');
    }
};
```

As you can see we simply added two images to the `preload()` function, this way they are available to our new game state once it loads. Talking about game states we need some steps when creating

www.dbooks.org

a new one. First thing we need to create a new file. Lets call it **game_over.js** and place it inside the **js folder**.

We need to include it in **index.html** as shown here

index.html: now includes our game over state

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>Foxnoid Game</title>
        <link rel="stylesheet" href="css/style.css" />
        <script defer src="js/phaser.min.js"></script>
        <script src="js/init.js"></script>
        <script defer src="js/preload.js"></script>
        <script defer src="js/game.js"></script>
        <script defer src="js/game_over.js"></script>
    </head>
    <body>

        <div id="game"></div>

    </body>
</html>
```

After including that file in the HTML we need to register the state in **init.js** as seen here:

init.js: with the new game over state

```
var GameStates = {}; // <-- Object to hold all our game states.

document.addEventListener("DOMContentLoaded", function()  {

    // Portrait game orientation.

    var width = 320;
    var height = 480;

    var game = new Phaser.Game(width, height, Phaser.CANVAS, "game");

    // Add the Game States the game has.
    game.state.add('Preloader', GameStates.Preloader);
    game.state.add('Game', GameStates.Game);
```

```
15      game.state.add('GameOver', GameStates.GameOver);
16
17      // Now start the Preloader state.
18      game.state.start('Preloader');
19
20  });
```

With those steps done we can code **game_over.js**

game_over.js: player lost, how sad

```
1   GameStates.GameOver = {
2       create: function() {
3
4           // Add the background
5           this.add.sprite(0, 0, 'background');
6
7           // Add the message
8           this.add.sprite(20, 30, 'gameover');
9
10          // Add the tap to play button
11          this.add.sprite(20, 300, 'taptoplay');
12      },
13
14      update: function() {
15
16          /**
17           * We just want to detect a tap.
18           * If there is one, we switch back to the game
19           * state and start the game again
20           */
21
22          if (this.input.pointer1.isDown) {
23              this.state.start('Game');
24          }
25
26      }
27
28  };
```

This new game state is pretty simple. On `create()` we build the screen with our messages and on `update()` we keep checking for a touch on the screen. If the player touches the screen we switch to the **game** state which restarts the game.

www.dbooks.org

Game Win Scenario

The game win scenario will be the same as the game over scenario but with different images. As seen here



Player won!

First lets add the new image to the **preload** state:

preload.js: the game state to load our assets

```
GameStates.Preloader = {
    preload: function() {
        this.load.image('background', 'assets/background.jpg');
        this.load.image('player', 'assets/player.png');
        this.load.image('ball', 'assets/ball.png');
        this.load.image('block', 'assets/block.png');
        this.load.image('gameover', 'assets/gameover.png');
        this.load.image('taptoplay', 'assets/taptoplay.png');
        this.load.image('gamewin', 'assets/gamewin.png');
    },
    create: function(){
        this.state.start('Game');
```

```
    }
};
```

And create a file **game_win.js** in the **js folder**. We need to register this new game state in **init.js** and add it to the **index.html**.

index.html: now includes our game win state

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>Foxnoid Game</title>
        <link rel="stylesheet" href="css/style.css" />
        <script defer src="js/phaser.min.js"></script>
        <script src="js/init.js"></script>
        <script defer src="js/preload.js"></script>
        <script defer src="js/game.js"></script>
        <script defer src="js/game_over.js"></script>
        <script defer src="js/game_win.js"></script>
    </head>
    <body>

        <div id="game"></div>

    </body>
</html>
```

After including that file in the HTML we need to register the state in **init.js** as seen here:

init.js: with the new game win state

```
var GameStates = {}; // <-- Object to hold all our game states.

document.addEventListener("DOMContentLoaded", function()  {

    // Portrait game orientation.

    var width = 320;
    var height = 480;

    var game = new Phaser.Game(width, height, Phaser.CANVAS, "game");

```

www.dbooks.org

```
    // Add the Game States the game has.
    game.state.add('Preloader', GameStates.Preloader);
    game.state.add('Game', GameStates.Game);
    game.state.add('GameOver', GameStates.GameOver);
    game.state.add('GameWin', GameStates.GameWin);

    // Now start the Preloader state.
    game.state.start('Preloader');

});
```

With that done we can code the **game_win.js** file.

game_win.js: player won!

```
GameStates.GameWin = {
    create: function() {

        // Add the background
        this.add.sprite(0, 0, 'background');

        // Add the message
        this.add.sprite(20, 30, 'gamewin');

        // Add the tap to play button
        this.add.sprite(20, 300, 'taptoplay');
    },

    update: function() {

        /**
         * We just want to detect a tap.
         * If there is one, we switch back to the game
         * state and start the game again
         */

        if (this.input.pointer1.isDown) {
            this.state.start('Game');
        }

    }

};
```

We've created all files necessary for the game to be winnable but we still need to add that condition to our **game.js** state. The player will win once all the blocks are removed from the game. The best place to check that is during `update()` after the collision check that destroys the blocks. Lets add a function to that file:

game.js: new function to check for game win condition

```
checkGameWin: function () {
    if (this.blocks.countLiving() === 0) {
        this.state.start("GameWin");
    }
}
```

The `this.blocks.countLiving()` call is from `Phaser.Group` and returns the number of sprites alive in a given group. Since each time there is a collision with a block we `kill()` it then that function will return zero once all blocks are cleared. When that happens we switch to the game win scenario that works just like the game over one.

We need to add it to the `update()` function:

game.js: the new update() function

```
update: function() {
   this.handleTouchInput();
   this.handleKeyboardInput();
   this.checkHitWithBlocks();
   this.checkHitWithPlayer();
   this.ballCollidesWithGround();
   this.checkGameWin();
}
```

Now you can play and win or lose the game!

Summary

In this chapter we added mechanics for winning and losing the game. In the next chapter we're going to keep players engaged by adding levels to the game.

www.dbooks.org

Engaging Players

Foxnoid is already a game that you can play and win or lose but we haven't tackled one important aspect of games: **replayability**. Since that word means different thing for different people and it is not in the dictionary, lets define it roughly as **the desire the player has of playing your game again**.

The important felling here is **the desire** to play. We're not focusing on the technical features that allow software to be used again and again. In this chapter we're going to talk about what motivates players to keep playing again and again.

Making The Game A Journey

If you treat your game like a heroes journey where the player has a path of progression with increasing difficulty, then your game becomes much better.

Many mobile casual games explore this mechanic using levels that unlock progressively. One level will only become available after you finish the previous one in a satisfactory way. Angry Birds does that and so do a hundred other games.

Platform games also explore this idea where the player has to move from level to level until success is achieved. Super Mario Bros is an example of that. Every time Toad says "The Princess is in another castle" we know we must keep going.

Players keep playing these types of game because they want to finish the journey. They want to win. They want to see the next level. They may justify their action to keep playing in many ways but it usually boils down to "this is an enjoyable journey and I want to see how it ends".

Replayability

Replayability means different things for different people as I said above. Some popular game development authors don't like the term because it is so vague that it can lead to a lot of confusion. The game development industry is quite young and its jargon is still growing. For example sailing is a pretty old thing and its jargon matured over some centuries. There are lots and lots of ways of expressing concepts regarding sailing. Game development still need some centuries for jargon to evolve and solidify. Still we don't have the patience of old sailors, we game developers are a group of fast people trying to express concepts in this art form while academia and theory are still playing catch up.

Lets talk about two possible aspects of replayability. The first one is the act of returning to the game while the journey is not complete. This is about engaging the player so that they don't forget the

game. Our objective is to evoke that felling that will make them think "just one more level and then I will stop". There are many ways to cause this behavior and some of the most common methods should be known by all.

- Top scores: Games with leaderboards where the player will keep coming back to make sure they beat their own record or top another player. This has become more competitive with the advent of ubiquitous networking.
- Multiple levels: The game progress thru different levels. To win the game the player needs to complete every level. This appeals to the wanderer inside us all. Players will keep playing because they need to finish the journey.
- Collectibles and achievements: Some games offer achievements and collectibles inside it. Players will keep playing to catch everything. If the item above appealed to our inner wanderer this one appeals to our inner hoarder. Achievements combined with social networks are a great thing to engage players.
- Story: Some games are story driven and players will keep playing because they want to know what happens. This is usually combined with the levels approach. Adventure games were in my humble opinion the pinnacle of story driven games. Most of today console 3D action games are story driven. Sometimes the story is not good enough or make any sense but still we all hate to unfinished stories so we keep playing.

These aspects are usually combined for example its common to see networked scoreboards and achievement lists. When designing your game you need to think about how to engage your players and add value to your title.

Another thing people mean when they talk about replayability is how to make players keep playing your game after the journey ended. Or in plain English, how can we make someone play the game again after they win it for the first time. Some games will use the collectible approach and go for things such as "you won the game but you haven't collected all sparkling unicorn eggs yet". Other games, specially the casual or arcade genre, will use the top scores as the main reason for people to keep playing. Others will unlock an "extra hard" difficulty mode after you win so that you can challenge yourself again.

In the end it is important to think about how to engage players before they win your game and also after.

Sense of Progression

One important feeling is the sense of progression. If a player stops to think "why am I playing this anyway?" while playing your game then there is a chance they won't be playing much longer. The level and story mechanics keep the game fresh and give the player this sense of progression. Its the difference between waving a brush randomly and painting a wall. The act of waving the brush is the same but since in the later you have an objective and the fact you can measure your progress makes it much more defined.

www.dbooks.org

Increasing Difficulty

Along with the level mechanic comes the increasing difficulty. A heroes journey is never a stroll in the park. It is more close to climbing a mountain. It starts somewhat easy but it gets more difficult as you are close to the top. The levels will give the player a sense of progress, lets say a road they will travel. Increasing difficulty with the levels will make the road trip more challenging which also means it will be more rewarding as they succeed. The feeling we want to evoke here is first "This game is getting harder" along with "I am getting better at this".

If you fail to engage a player like this they tend to get bored and move on. The hardest game I know is Dwarf Fortress[87] which is a game where you basically simulate a fantasy dwarf colony. The game motto is **loosing is fun**. It has a steep learning curve and you will surely lose it a lot but once you start succeeding a little before failing miserably you start to get hooked (cue: Google for boatsmurdered and dwarf fortress and be amused).

I am not advocating to go full masochist on your player. All I am saying is: don't make it too easy. Make it more like a trail and less like shopping mall.

Creating New Levels For Our Game

Creating networked stuff such as scoreboard and achievements is beyond the scope of this book. Foxnoid doesn't really need a story since it is just a good traditional arcade game. To engage players we're going to implement some levels. To keep things short we're going to implement just a few levels. Feel free to add more on your own.

Defining a level format

Our game has three rows of five blocks each. The easiest way to describe a level is by using an array with 15 elements that can be 0 for empty space or 1 for block. Each level will be an array and the collection of all levels will also be an array.

Create a file called **levels.js** inside the **js folder**

[87] http://www.bay12games.com/dwarves/

levels.js: three different levels for our game

```
var levels = [
    [
        1,1,1,1,1,
        1,0,0,0,1,
        1,1,1,1,1
    ],

    [
        1,0,1,0,1,
        1,0,1,0,1,
        1,0,1,0,1
    ],

    [
        1,1,1,1,1,
        1,1,1,1,1,
        1,1,1,1,1
    ]
];
```

Now lets include this file in our HTML.

index.html: now includes our game win state

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>Foxnoid Game</title>
        <link rel="stylesheet" href="css/style.css" />
        <script defer src="js/phaser.min.js"></script>
        <script src="js/init.js"></script>
        <script defer src="js/preload.js"></script>
        <script defer src="js/game.js"></script>
        <script defer src="js/game_over.js"></script>
        <script defer src="js/game_win.js"></script>
        <script defer src="js/levels.js"></script>
    </head>
    <body>

        <div id="game"></div>
```

www.dbooks.org

```
18
19      </body>
20  </html>
```

> Warning: Yep, you spotted a global called `levels` on the source code above. This is not really good practice. The best way would be for the preloader to load that from a JSON and store it in the `GameStates` variable. In a future revision of this book I might just do that. Foxnoid is a simple game and I am trying to keep the source code easy to follow so that we can go from zero to a complete game. As I said in the introduction, this is an early release and revisions will be done in time. If you think that global offends your tastes let me know.

Displaying The Level

To make use of the levels we'll need to alter our **game.js** file. First thing lets change `initWorld()` to add a constant for the starting level number.

game.js: adding level stuff

```
1   initWorld: function() {
2       // Some constants
3       this.playerSpeed = 250;
4       this.ballSpeed = 220;
5       this.blocksPerRow = 5;
6       this.blockRows = 4;
7       this.playerLives = 13;
8       this.currentLevel = 0;
9
10      // Add the background
11      this.add.sprite(0, 0, 'background');
12
13      // Add keyboard input.
14      // This call creates and returns an object containing 4 hotkeys for Up, Down\
15  , Left and Right.
16      this.cursors = this.input.keyboard.createCursorKeys();
17  }
```

So the game will start from the zeroth (a.k.a. first) element of the levels array. Next we need to be able to display the level marked by `this.currentLevel`. To do that we'll change `addBlocks()`.

game.js: new addBlocks() will display the current level

```
addBlocks: function () {
    var level = levels[this.currentLevel];
    var blockNum = 0;

    // do not create the blocks group
    // if it is already present.
    if (!this.blocks) {
        this.blocks = this.game.add.group();
    }

    for (var line = 0; line <= this.blockRows - 1; line++) {
        for (var row = 0; row <= this.blocksPerRow - 1; row++) {
            var posY = (line * 30) + 40;
            var posX = (row * 50) + 40;

            if (level[blockNum] === 1) {
                var temp = this.add.sprite(posX, posY, 'block');
                this.physics.arcade.enable(temp);
                temp.enableBody = true;
                temp.body.immovable = true;

                this.blocks.add(temp);
            }

            blockNum += 1;
        }
    }
}
```

If we play our game now we'll see that it correctly displays the first level.

Level Progression

To change the way the game works and add the level progression mechanic we just need to change the calculation of the game win scenario. If there are no blocks left but the player current level is not the last level then draw the next level and keep going. To do this we're going to change the `checkGameWin()` function.

www.dbooks.org

game.js: new addBlocks() will display the current level

```
checkGameWin: function () {
    if (this.blocks.countLiving() === 0) {
        if (this.currentLevel === levels.length - 1) {
            this.state.start("GameWin");
        } else {
            this.currentLevel++;
            this.addBlocks();
            this.resetBall();
        }
    }
}
```

We check to see if all blocks have been removed. If there are no more blocks we check if the players current level is the last one. If it is then the game has been won, if it isn't then we advance to the next level, draw it and reset the ball to the original position.

Go ahead and try to win it!

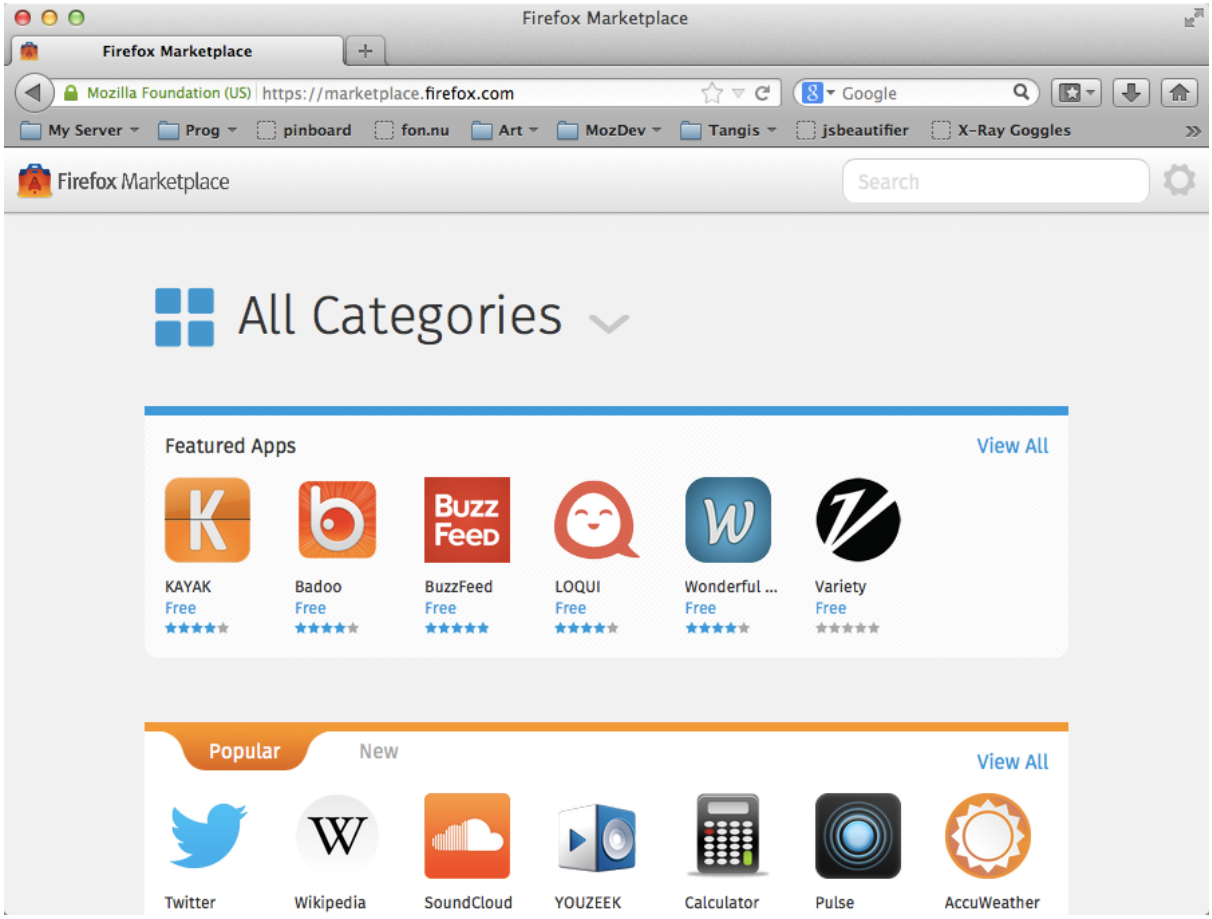
The source code for the **game.js** as seen in this chapter is on Appendix 7: Levels Chapter Code.

Summary

Congratulations!!! You've went from zero to a complete game with a game over and a game win scenario and multiple levels! Be proud of yourself, people seldom complete their game development projects. There are more unfinished games on lost HDs than sand in a beach or something similar.

On the next chapter we're going to publish it to the Firefox Marketplace! Lets move ahead and share our game with the world!

The Firefox Marketplace



Firefox Marketplace

The Firefox Marketplace[88] is the online shop where you can buy or download applications for Firefox OS, Firefox, and Firefox For Android. This is the main channel for distributing Firefox OS applications, but you're not required to use it. If you want to distribute things outside the marketplace, read the previous chapter.

To place your apps on the marketplace you need to be identified via Mozilla Persona[89]. Just click **Sign Up** and follow the instructions. Once you're identified, you will be ready to submit apps to the Firefox Marketplace.

[88] http://marketplace.firefox.com
[89] https://login.persona.org/about

www.dbooks.org

Checklist before even thinking about sending an app to the marketplace

All applications that are submitted to the marketplace go through an approval process (less scary than it sounds!). Hosted web applications go through a lighter process than privileged apps because they use less sensitive APIs. Before sending your application to the marketplace check out the marketplace review criteria[90]. The most important parts are (IMHO):

- Firefox OS devices do not have a **back button** like Android and your desktop browser. If the user navigates to a screen inside your app where there is no way for them to get back to the previous place (i.e. the user gets stuck), your app will be rejected.
- Your app should have a 60x60 icon and clear descriptions.
- Your app should do what the description says. Saying one thing and providing something else will get your app rejected.
- If your app asks for a given permission then you should use it somewhere in your code. Flagging your application as a privileged app and not using any privileged app API will cause your app to be rejected with a request that you submit again as a plain app.
- Your application needs to have a *privacy policy* in place.
- Manifest files should be served with the correct MIME type and come from the same domain as the app for hosted apps.

There are other criteria discussed in the link above - and the rules can change without notice. It will be worth your time to read that page. Getting your application rejected because of small stuff that is easy to fix is a huge waste of time. Better get things right the first time (reviewers love to approve good apps!).

Preparing your app for submission

The steps required to submit your application to the marketplace are different depending on whether it's a hosted or a packaged app. For a hosted app, it just needs to be accessible on the Internet with the correct MIME type and manifest in place. Packaged apps need to be compressed using *zip* and deserve some extra attention.

Many developers make the mistake of selecting the folder containing the application files and zipping it. This causes the zip file to contain a folder and this folder to contain the app. This is not the correct way to zip a Firefox OS application. The correct way is to zip the files and folders needed so that the manifest is on the *root level* of the zip file. On Mac OS X and Linux you can use the terminal to navigate to your application folder and use a command such as `zip -r myapp.zip *` to compress things correctly as shown on the screenshot below.

[90] https://developer.mozilla.org/en-US/docs/Web/Apps/Publishing/Marketplace_review_criteria

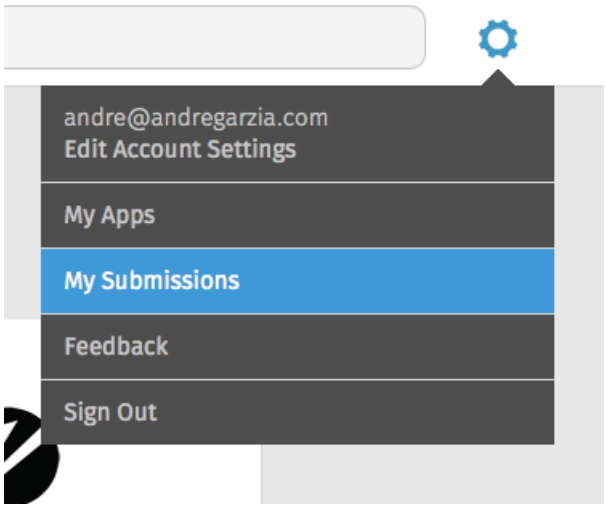
```
ndres-MacBook-Air:firefoxos-sample-app-image-uploader soapdog$ zip -r imgur.zip *
 adding: LICENSE (deflated 67%)
 adding: README.md (deflated 58%)
 adding: app.js (deflated 69%)
 adding: building_blocks/ (stored 0%)
 adding: building_blocks/building_blocks.css (deflated 84%)
 adding: building_blocks/cross_browser.css (deflated 78%)
 adding: building_blocks/fonts/ (stored 0%)
 adding: building_blocks/fonts/FeuraSans-1.0/ (stored 0%)
 adding: building_blocks/fonts/FeuraSans-1.0/FeuraSans-Bold.woff (deflated 0%)
 adding: building_blocks/fonts/FeuraSans-1.0/FeuraSans-Light.woff (deflated 0%)
 adding: building_blocks/fonts/FeuraSans-1.0/FeuraSans-Medium.woff (deflated 0%)
 adding: building_blocks/fonts/FeuraSans-1.0/FeuraSans-Regular.woff (deflated 0%)
 adding: building_blocks/fonts/FeuraSans-1.0/LICENSE (deflated 36%)
 adding: building_blocks/fonts.css (deflated 85%)
 adding: building_blocks/icons/ (stored 0%)
```

Correctly zipping the files

This zip file is what we send to the marketplace.

Submitting your app to the marketplace

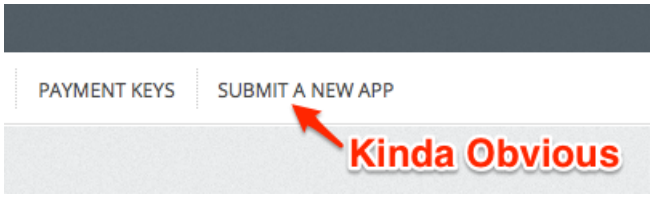
Now with your application ready, and with the firm knowledge that it meets the review criteria, its time we send it to the marketplace. To do so browse to **My Submissions** using the gear button on the top of the marketplace page.



My Submissions

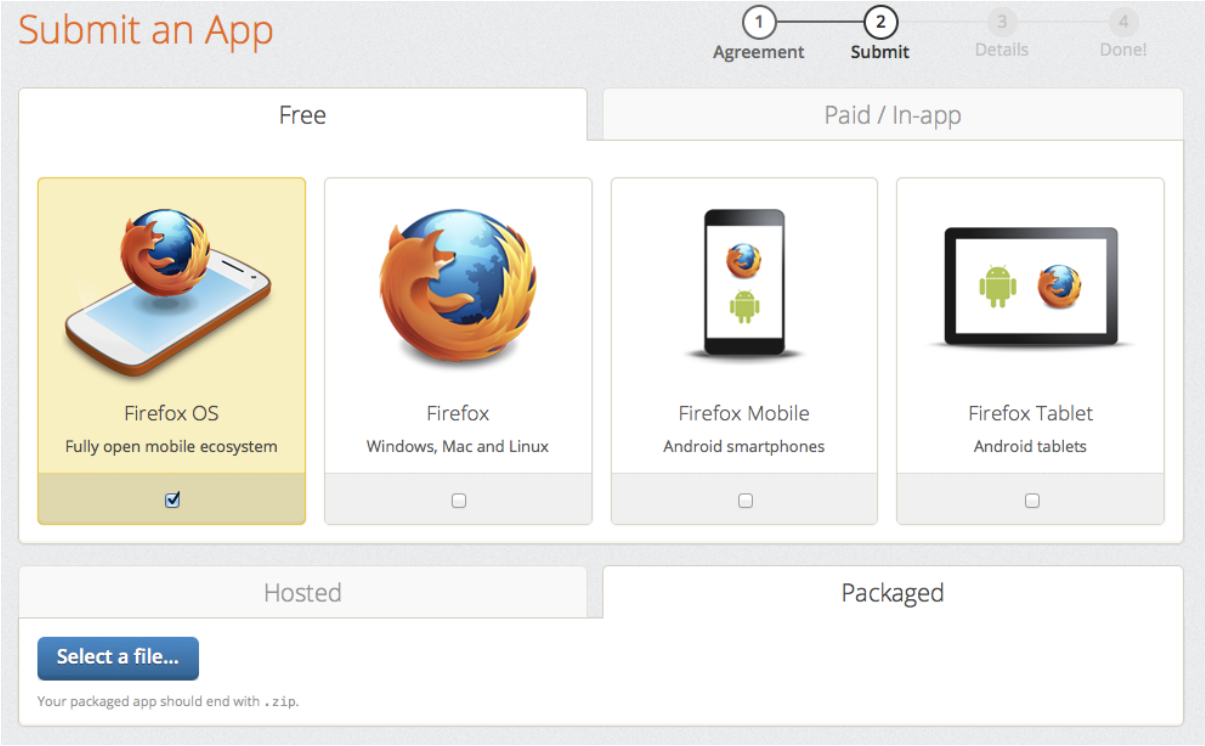
Inside the application management page, you need to click on **Submit An App** on the top menu.

www.dbooks.org



Submit An App

This link will lead you to the form for submitting new apps, as seen in the screenshot below.



Submit New App

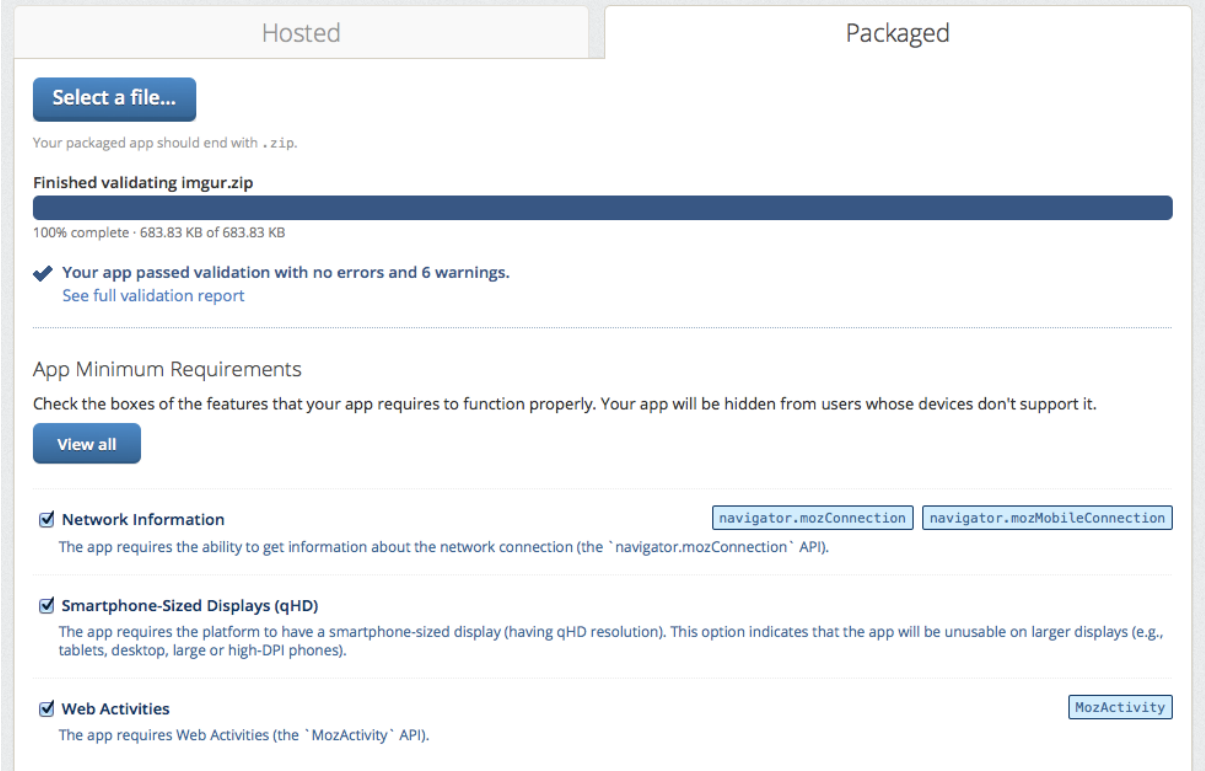
On this screen you will select the following options:

- If the application is hosted or packaged.
- If it is free or paid (or uses *in-app purchases*).
- What type of devices it is available for (Firefox OS, Firefox Desktop, Firefox for Mobile on phones, Firefox for Mobile on Tablets).

After making these choices you're driven to the second screen. On this book we're focusing on packaged apps but hosted apps are similar.

In the remaining text of this chapter we're assuming that we're shipping a free Firefox OS packaged app. In this case we need to upload the zip file we prepared on the previous section.

After uploading the file, it undergoes an automated process and a report is shown with many options.

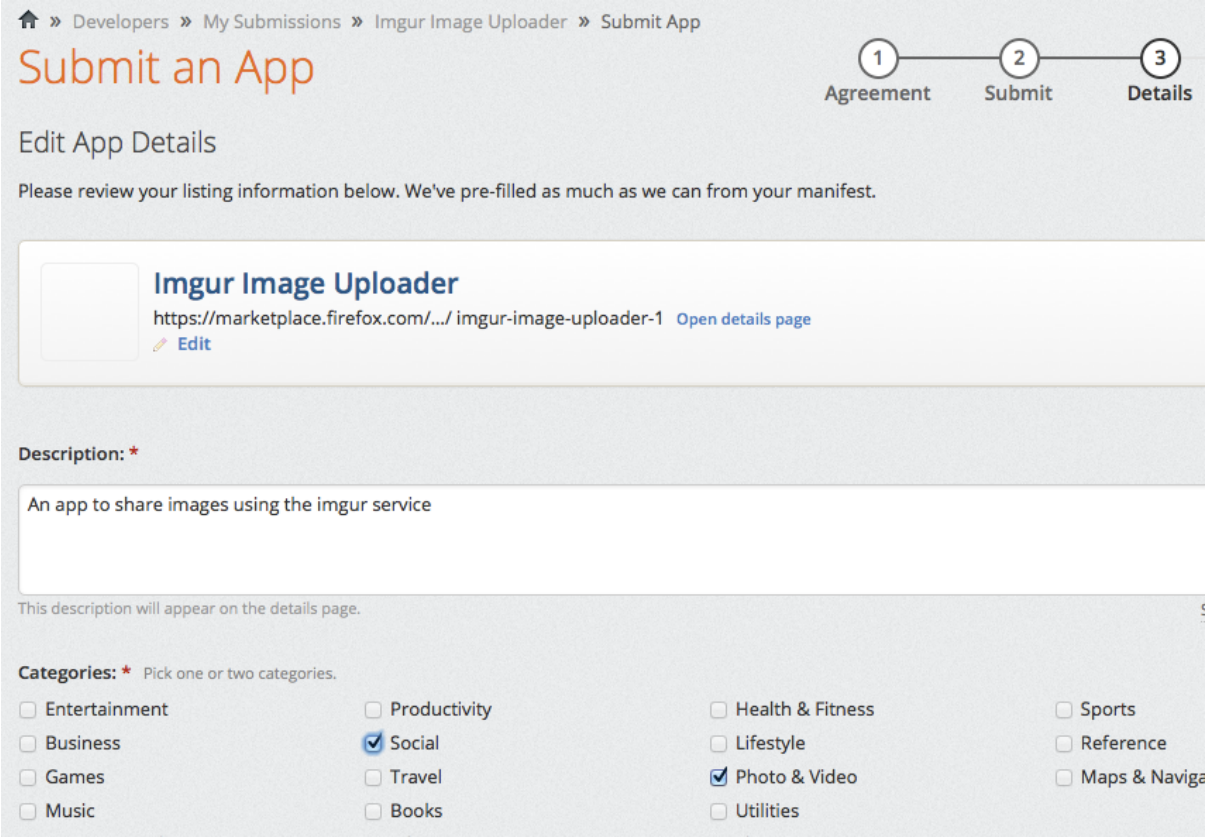


After the zip upload

From the screen shot above we can see that the app I sent to the marketplace has no errors but contains six warnings. Ignoring the warnings for the sake of this text, lets check what the **minimum requirements** for this app are. In this case, the last option *Smartphone-Sized Displays (qHD)* should be unchecked because our application adapts to any screen size.

The next step is called **Step #3: Details** and it is where you fill the information about your application such as category, description, screen captures, etc.

www.dbooks.org



Filling details

After filling-in the details, the submission process is done. Now you just wait for the approval from the marketplace reviewers. Congratulations you shipped a Firefox OS application!!!

On the Application Management page[91] you can check the status of your submissions and alter details if needed.

To learn more about submitting applications to the Firefox Marketplace read this article on the Firefox OS developer hub[92].

Summary

Congratulations!!! You have a new application on the Firefox Marketplace, you're exploring a brand new market!

I hope you enjoyed this quick guide. I plan to update and expand this guide often - so keep your eyes open and register for the updates. If you downloaded this book from Leanpub then its all good because you will receive emails about any updates. If you downloaded it from somewhere else then

[91] https://marketplace.firefox.com/developers/submissions
[92] https://marketplace.firefox.com/developers/docs/submission

please consider fetching it from the official page at Leanpub[93] and registering your email. It's free and, no, you won't get any spam. Promise.

Please send me feedback. I can be reached for feedback on my Twitter account at @soapdog[94] and over email at fxosgameguide@andregarzia.com[95]. My home page is at http://andregarzia.com[96].

Now that you're a part of the group of Firefox OS game creators, come be a part of the greater Mozilla community: Help keep the web free and open made by users for users. Join us at http://www.mozilla.org/contribute/[97] and help Firefox OS grow!

[93] http://leanpub.com/buildinggamesforfirefoxos
[94] http://twitter.com/soapdog
[95] mailto:fxosgameguide@andregarzia.com
[96] http://andregarzia.com
[97] http://www.mozilla.org/en-US/contribute/

www.dbooks.org

Appendix 1: Useful Links

- Mozilla[98]
- Mozilla Brasil Community Page[99]
- Firefox OS Developer Hub[100]
- Mozilla Developers Network[101]: Best documentation ever!
- Firefox OS[102]
- WebAPI on Mozilla Wiki[103]

---

[98] http://mozilla.org
[99] http://mozillabrasi.org.br
[100] http://marketplace.firefox.com/developers
[101] http://developer.mozilla.org/
[102] http://www.mozilla.org/pt-BR/firefox/os/
[103] http://wiki.mozilla.org/WebAPI

Appendix 2: Javascript Books

- JavaScript: The Good Parts[104]: The JavaScript Book.
- JavaScript Patterns[105]: Patterns and best practices.
- JavaScript Enlightenment[106]: Advanced JavaScript techniques.
- Maintainable JavaScript[107]: Writing code that is easy to maintain and work with.

[104] http://shop.oreilly.com/product/9780596517748.do
[105] http://shop.oreilly.com/product/9780596806767.do
[106] http://shop.oreilly.com/product/0636920027713.do
[107] http://shop.oreilly.com/product/0636920027713.do

www.dbooks.org

Appendix 3: Game Development Books

- The Art of Game Design: A book of lenses[108]: In my personal opinion this is **the best book about game design ever created** and I don't mean this lightly. This book provides you with mental tools and workflows useful in any game related creative process. It teaches you how to organize yourself and think about your game from the perspective of many *lenses*. It is a wonderful experience. This is really recommended.
- Level Up!: The Guide to Great Video Game Design[109]: A great book focused on video games design. Very easy to understand and appreciate. Clear explanations and ideas.
- Discover Phaser[110]: A really good book focused on teaching the Phaser Framework by Thomas Palef who is a leading member of the Phaser community. He developed many games and his content is solid. There are package deals available for this book that includes more sample games. I really recommend it.

---

[108] http://www.amazon.com/gp/product/0123694965/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0123694965&linkCode=as2&tag=andregacom02-20&linkId=3A6K4FN3GZ3AVW3K

[109] http://www.amazon.com/gp/product/047068867X/ref=as_li_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=047068867X&linkCode=as2&tag=andregacom02-20&linkId=TBDQTMA7QTIX4JOD

[110] http://www.discoverphaser.com/

Appendix 4: Firefox OS Related Links

- The Quick Guide for Firefox OS Development[111]: a free and open source sibling to this book but focused on app development. If you're reading this book you probably will like that one too.
- Firefox OS Developer Hub[112]: The developer hub on Firefox Marketplace contains a lot of information and resources about creating Firefox OS applications.
- The Firefox OS Zone on Mozilla Developers Network[113]: MDN is probably one of the most important developer resource online focused on web technologies. Its a free and open source treasure trove of documentation, samples and articles. This link is for the Firefox OS zone that aggregates links related to this new platform.
- Beginning Firefox OS HTML5 Game Programming[114]: Bob Thulfram has released this awesome book on Amazon. If you want more resources about Game Development on Firefox OS you need to check this one out.
- Bob Thulfram has released[115]: Bob Thulframs blog about Firefox OS Gaming. After you start creating games you will probably see your creations featured here :-)

[111] https://leanpub.com/quickguidefirefoxosdevelopment/
[112] https://marketplace.firefox.com/developers
[113] https://developer.mozilla.org/en-US/Firefox_OS
[114] http://www.amazon.com/dp/B00IZUYIGO
[115] http://firefoxosgaming.blogspot.co.uk/

www.dbooks.org

Appendix 5: Gameloop Chapter Code

Below is the code for the **game.js** file as created on the Gameloop Chapter.

```
/**
 * This is the Game game state.
 * Its our game loop responsible for the game itself.
 *
 * In the create() function we setup the display.
 * In the update() function we do the gameloop.
 *
 * To learn more about states refer to:
 * Refer to: http://docs.phaser.io/Phaser.State.html
 */

GameStates.Game = {
    resetBall: function() {
        this.ball.reset(160, 240);
        this.ball.body.velocity.x = this.ballSpeed;
        this.ball.body.velocity.y = this.ballSpeed;
    },

    initWorld: function() {
        // Some constants
        this.playerSpeed = 250;
        this.ballSpeed = 220;
        this.blocksPerRow = 5;
        this.blockRows = 4;
        this.playerLives = 13;

        // Add the background
        this.add.sprite(0, 0, 'background');
    },


    addPlayer: function () {
        // Add the player
        this.player = this.add.sprite(160, 440, 'player');
        this.physics.arcade.enable(this.player);
```

```
36          this.player.anchor.setTo(0.5, 0);
37          this.player.enableBody = true;
38          this.player.body.immovable = true;
39          this.player.body.collideWorldBounds = true;
40
41
42          // Add the display of player lives
43          this.livesDisplay = this.add.text(10, 8, "Lives: " +
44              this.playerLives, {
45                  fill: "white",
46                  fontSize: 12
47              });
48      },
49
50      addBall: function () {
51          // Add ball
52          this.ball = this.add.sprite(160, 240, 'ball');
53          this.physics.arcade.enable(this.ball);
54          this.ball.anchor.setTo(0.5, null);
55          this.ball.enableBody = true;
56          this.ball.body.bounce.setTo(1, 1);
57          this.ball.body.velocity.x = this.ballSpeed;
58          this.ball.body.velocity.y = this.ballSpeed;
59          this.ball.body.collideWorldBounds = true;
60      },
61
62      addBlocks: function () {
63          // Blocks
64          this.blocks = this.game.add.group();
65          for (var line = 0; line <= this.blockRows - 1; line++) {
66              for (var row = 0; row <= this.blocksPerRow - 1; row++) {
67                  var posY = (line * 30) + 40;
68                  var posX = (row * 50) + 40;
69                  console.log("Adding block at: " + posX + "," + posY)
70                  var temp = this.add.sprite(posX, posY, 'block');
71                  this.physics.arcade.enable(temp);
72                  temp.enableBody = true;
73                  temp.body.immovable = true;
74
75                  this.blocks.add(temp);
76              }
77          }
```

www.dbooks.org

```
78      },
79
80      create: function() {
81          this.initWorld();
82          this.addPlayer();
83          this.addBall();
84          this.addBlocks();
85      },
86
87      checkHitWithBlocks: function () {
88          this.game.physics.arcade.collide(this.ball, this.blocks, this.ballCollid\
89  esWithBlock);
90      },
91
92      checkHitWithPlayer: function () {
93          this.game.physics.arcade.collide(this.ball, this.player);
94      },
95
96      update: function() {
97          this.checkHitWithBlocks();
98          this.checkHitWithPlayer();
99          this.ballCollidesWithGround();
100     },
101
102     ballCollidesWithBlock: function(sprite, block) {
103         console.log("Collided with block!");
104         block.kill();
105     },
106
107     ballCollidesWithGround: function() {
108         if (this.ball.y >= 470) {
109             this.playerLives -= 1;
110             this.resetBall();
111         }
112
113         /*
114          Update lives display
115          */
116
117         this.livesDisplay.setText("Lives: " + this.playerLives);
118
119         if (this.playerLives === 0) {
```

```
120             // Soon there will be a game over state switch here
121         }
122
123     }
124
125 };
```

www.dbooks.org

Appendix 6: Input Chapter Code

Below is the code for the **game.js** file as created on the Input Chapter.

```
/**
 * This is the Game game state.
 * Its our game loop responsible for the game itself.
 *
 * In the create() function we setup the display.
 * In the update() function we do the gameloop.
 *
 * To learn more about states refer to:
 * Refer to: http://docs.phaser.io/Phaser.State.html
 */

GameStates.Game = {
    resetBall: function() {
        this.ball.reset(160, 240);
        this.ball.body.velocity.x = this.ballSpeed;
        this.ball.body.velocity.y = this.ballSpeed;
    },

    initWorld: function() {
        // Some constants
        this.playerSpeed = 250;
        this.ballSpeed = 220;
        this.blocksPerRow = 5;
        this.blockRows = 4;
        this.playerLives = 13;

        // Add the background
        this.add.sprite(0, 0, 'background');

        // Add keyboard input
        this.cursors = this.input.keyboard.createCursorKeys();
    },


    addPlayer: function () {
```

```
36          // Add the player
37          this.player = this.add.sprite(160, 440, 'player');
38          this.physics.arcade.enable(this.player);
39          this.player.anchor.setTo(0.5, 0);
40          this.player.enableBody = true;
41          this.player.body.immovable = true;
42          this.player.body.collideWorldBounds = true;
43
44
45          // Add the display of player lives
46          this.livesDisplay = this.add.text(10, 8, "Lives: " +
47              this.playerLives, {
48                  fill: "white",
49                  fontSize: 12
50              });
51      },
52
53      addBall: function () {
54          // Add ball
55          this.ball = this.add.sprite(160, 240, 'ball');
56          this.physics.arcade.enable(this.ball);
57          this.ball.anchor.setTo(0.5, null);
58          this.ball.enableBody = true;
59          this.ball.body.bounce.setTo(1, 1);
60          this.ball.body.velocity.x = this.ballSpeed;
61          this.ball.body.velocity.y = this.ballSpeed;
62          this.ball.body.collideWorldBounds = true;
63      },
64
65      addBlocks: function () {
66          // Blocks
67          this.blocks = this.game.add.group();
68          for (var line = 0; line <= this.blockRows - 1; line++) {
69              for (var row = 0; row <= this.blocksPerRow - 1; row++) {
70                  var posY = (line * 30) + 40;
71                  var posX = (row * 50) + 40;
72                  console.log("Adding block at: " + posX + "," + posY)
73                  var temp = this.add.sprite(posX, posY, 'block');
74                  this.physics.arcade.enable(temp);
75                  temp.enableBody = true;
76                  temp.body.immovable = true;
77
```

www.dbooks.org

```
78                 this.blocks.add(temp);
79             }
80         }
81     },
82
83     create: function() {
84         this.initWorld();
85         this.addPlayer();
86         this.addBall();
87         this.addBlocks();
88     },
89
90     checkHitWithBlocks: function () {
91         this.game.physics.arcade.collide(this.ball, this.blocks, this.ballCollid\
92 esWithBlock);
93     },
94
95     checkHitWithPlayer: function () {
96         this.game.physics.arcade.collide(this.ball, this.player);
97     },
98
99     handleTouchInput: function () {
100      if (this.input.pointer1.isDown) {
101          if (this.input.pointer1.worldX > 160) {
102              this.player.body.velocity.x = this.playerSpeed;
103          }
104
105          if (this.input.pointer1.worldX <= 160) {
106              this.player.body.velocity.x = -1 * this.playerSpeed;
107          }
108       }
109     },
110
111     handleKeyboardInput: function () {
112         if (this.cursors.right.isDown) {
113             this.player.body.velocity.x = this.playerSpeed;
114         }
115
116         if (this.cursors.left.isDown) {
117             this.player.body.velocity.x = -1 * this.playerSpeed;
118         }
119     },
```

```
120
121     update: function() {
122         this.handleTouchInput();
123         this.handleKeyboardInput();
124         this.checkHitWithBlocks();
125         this.checkHitWithPlayer();
126         this.ballCollidesWithGround();
127     },
128
129     ballCollidesWithBlock: function(sprite, block) {
130         console.log("Collided with block!");
131         block.kill();
132     },
133
134     ballCollidesWithGround: function() {
135         if (this.ball.y >= 470) {
136             this.playerLives -= 1;
137             this.resetBall();
138         }
139
140         /*
141          Update lives display
142          */
143
144         this.livesDisplay.setText("Lives: " + this.playerLives);
145
146         if (this.playerLives === 0) {
147             // Soon there will be a game over state switch here
148         }
149
150     }
151
152 };
```

www.dbooks.org

Appendix 7: Levels Chapter Code

Below is the code for the **game.js** file as created on the Levels Chapter.

```
/**
 * This is the Game game state. Its our game loop responsible for the game itself.
 *
 * In the create() function we setup the display.
 * In the update() function we do the gameloop.
 *
 * To learn more about states refer to:
 * Refer to: http://docs.phaser.io/Phaser.State.html
 */

GameStates.Game = {
    resetBall: function() {
        this.ball.reset(160, 240);
        this.ball.body.velocity.x = this.ballSpeed;
        this.ball.body.velocity.y = this.ballSpeed;
    },

    initWorld: function() {
        // Some constants
        this.playerSpeed = 250;
        this.ballSpeed = 220;
        this.blocksPerRow = 5;
        this.blockRows = 4;
        this.playerLives = 13;
        this.currentLevel = 0;

        // Add the background
        this.add.sprite(0, 0, 'background');

        // Add keyboard input.
        // This call creates and returns an object containing 4 hotkeys for Up, Down, Left and Right.
        this.cursors = this.input.keyboard.createCursorKeys();
    },
```

```
36
37
38      addPlayer: function () {
39          // Add the player
40          this.player = this.add.sprite(160, 440, 'player');
41          this.physics.arcade.enable(this.player);
42          this.player.anchor.setTo(0.5, 0); //center anchor/origin to the middle o\
43  f the paddle
44          this.player.enableBody = true;
45          this.player.body.immovable = true;
46          this.player.body.collideWorldBounds = true;
47
48
49          // Add the display of player lives
50          this.livesDisplay = this.add.text(10, 8, "Lives: " + this.playerLives, {
51              fill: "white",
52              fontSize: 12
53          });
54      },
55
56      addBall: function () {
57          // Add ball
58          this.ball = this.add.sprite(160, 240, 'ball');
59          this.physics.arcade.enable(this.ball);
60          this.ball.anchor.setTo(0.5, null);
61          this.ball.enableBody = true;
62          this.ball.body.bounce.setTo(1, 1);
63          this.ball.body.velocity.x = this.ballSpeed;
64          this.ball.body.velocity.y = this.ballSpeed;
65          this.ball.body.collideWorldBounds = true;
66      },
67
68      addBlocks: function () {
69          var level = levels[this.currentLevel];
70          var blockNum = 0;
71
72          // do not create the blocks group
73          // if it is already present.
74          if (!this.blocks) {
75              this.blocks = this.game.add.group();
76          }
77
```

www.dbooks.org

```
78          for (var line = 0; line <= this.blockRows - 1; line++) {
79              for (var row = 0; row <= this.blocksPerRow - 1; row++) {
80                  var posY = (line * 30) + 40;
81                  var posX = (row * 50) + 40;
82
83                  if (level[blockNum] === 1) {
84                      var temp = this.add.sprite(posX, posY, 'block');
85                      this.physics.arcade.enable(temp);
86                      temp.enableBody = true;
87                      temp.body.immovable = true;
88
89                      this.blocks.add(temp);
90                  }
91
92                  blockNum += 1;
93              }
94          }
95      },
96
97      create: function() {
98
99          this.initWorld();
100         this.addPlayer();
101         this.addBall();
102         this.addBlocks();
103     },
104
105     checkHitWithBlocks: function () {
106         // Tell physics system to collide the ball and the blocks, if they colli\
107 de we call the method
108         // ballCollidesWithBlock as a callback.
109         this.game.physics.arcade.collide(this.ball, this.blocks, this.ballCollid\
110 esWithBlock);
111     },
112
113     checkHitWithPlayer: function () {
114         // Tell physics system to collide the ball and the player... this check \
115 has no callback because we don't
116         // need to run any routine when the ball hits the player, we just want i\
117 t to bounce and thats handled
118         // automatically by the physics system.
119         this.game.physics.arcade.collide(this.ball, this.player);
```

```
120     },
121
122     handleTouchInput: function () {
123         /*
124          Movement of the player using touch.
125
126          in this case we're just moving the player
127          on one axis.
128
129          To learn more about handling input refer to:
130          http://docs.phaser.io/Phaser.Input.html
131          */
132
133         if (this.input.pointer1.isDown) {
134             if (this.input.pointer1.worldX > 160) {
135                 this.player.body.velocity.x = this.playerSpeed;
136             }
137
138             if (this.input.pointer1.worldX <= 160) {
139                 this.player.body.velocity.x = -1 * this.playerSpeed;
140             }
141         }
142     },
143
144     handleKeyboardInput: function () {
145         /*
146           Moving the player left and right with the arrow keys.
147
148           The move here is continuous because we apply
149           a velocity to the axis. Meaning that the player never stops
150           moving, the paddle is always moving left or right.
151
152           This cursor variable is an instance of the Keyboard Manager.
153           To learn more about the keyboard refer to:
154           http://docs.phaser.io/Phaser.Keyboard.html
155         */
156
157         if (this.cursors.right.isDown) {
158             this.player.body.velocity.x = this.playerSpeed;
159         }
160
161         if (this.cursors.left.isDown) {
```

www.dbooks.org

```
            this.player.body.velocity.x = -1 * this.playerSpeed;
        }
    },

    checkGameWin: function () {
        if (this.blocks.countLiving() === 0) {
            if (this.currentLevel === levels.length - 1) {
                this.state.start("GameWin");
            } else {
                this.currentLevel++;
                this.addBlocks();
                this.resetBall();
            }
        }
    },

    update: function() {
        /*
        This function is called in a loop using requestAnimationFrame(). It is o\
ur game loop, our heartbeat. Every time
        it is called we process the user input and update the display.
         */
        this.handleTouchInput();
        this.handleKeyboardInput();
        /*
        Collision checking

        Phaser has three different types of physics systems depending on your ne\
eds.

        On this game we're using the "Arcade Physics" system which is the simple\
st one. This system
        gives us all we need for our arcade game without being hard on performan\
ce so it is a good
        match for mobile devices with low specs.

        To learn more about the Arcade Physics refer to:
        http://docs.phaser.io/Phaser.Physics.Arcade.html

         */
        this.checkHitWithBlocks();
        this.checkHitWithPlayer();
```

```
204         /*
205         Now lets do some game management. We need to figure out if the player wo\
206 n or lost the game.
207         If he did then we need to switch to the appropriate game state.
208          */
209 
210         // Checking for game win scenario
211         // Player wins the game once all blocks are gone.
212         this.checkGameWin();
213         // Checking for game over scenario
214         // If player has no more lives then its over
215 
216         this.ballCollidesWithGround();
217 
218 
219     },
220 
221     ballCollidesWithBlock: function(sprite, block) {
222         console.log("Collided with block!");
223         block.kill();
224     },
225 
226     ballCollidesWithGround: function() {
227         if (this.ball.y >= 470) {
228             this.playerLives -= 1;
229             this.resetBall();
230         }
231 
232         /*
233          Update lives display
234          */
235 
236         this.livesDisplay.setText("Lives: " + this.playerLives);
237 
238         if (this.playerLives === 0) {
239             this.state.start("GameOver");
240         }
241 
242     }
243 
244 };
```

www.dbooks.org